

***Guía para el Examen General
para el Egreso de la Licenciatura
en Ciencias Computacionales (EGEL-COMPU)***

CONTENIDO

PRESENTACIÓN.....	3
PROPÓSITO Y ALCANCE DEL EGEL-COMPU	3
DESTINATARIOS DEL EGEL-COMPU	3
¿Cómo se construye el EGEL-COMPU?	4
¿Qué evalúa el EGEL-COMPU?.....	5
¿Qué tipo de preguntas se incluyen en el examen?	6
1. Preguntas o reactivos de cuestionamiento directo	6
2. Ordenamiento.....	7
3. Elección de elementos	8
4. Relación de columnas	9
PRIMERA PRÁCTICA DE EXAMEN	10
RESPUESTAS DE LA PRIMERA PRÁCTICA DE EXAMEN.....	33
SEGUNDA PRÁCTICA DE EXAMEN	34
RESPUESTAS DE LA SEGUNDA PRÁCTICA DE EXAMEN	62
TERCERA PRÁCTICA DE EXAMEN.....	63
RESPUESTAS DE LA TERCERA PRÁCTICA DE EXAMEN.....	87
CUARTA PRÁCTICA DE EXAMEN	88
RESPUESTAS DE LA CUARTA PRÁCTICA DE EXAMEN	112
QUINTA PRÁCTICA DE EXAMEN	113
RESPUESTAS DE LA QUINTA PRÁCTICA DE EXAMEN	138
ANTOLOGÍA.....	139

PRESENTACIÓN

Esta Guía está dirigida a quienes sustentarán el Examen General para el Egreso de la Licenciatura en Ciencias Computacionales (EGEL-COMPU). Su propósito es ofrecer información que permita a los sustentantes familiarizarse con las principales características del examen, los contenidos que se evalúan, el tipo de preguntas (reactivos) que encontrarán en el examen, así como con algunas sugerencias de estudio y de preparación para presentar el examen.

Se recomienda al sustentante revisar con detenimiento la Guía completa del portal del CENEVAL, y recurrir a ella de manera permanente durante su preparación y para aclarar cualquier duda sobre aspectos académicos, administrativos o logísticos en la presentación del EGEL-COMPU.

PROPÓSITO Y ALCANCE DEL EGEL-COMPU

El propósito del EGEL-COMPU es identificar si los egresados de la licenciatura en Ciencias Computacionales cuentan con los conocimientos y habilidades necesarios para iniciarse eficazmente en el ejercicio de la profesión. La información que ofrece permite al sustentante:

- Conocer el resultado de su formación en relación con un estándar de alcance nacional mediante la aplicación de un examen confiable y válido, probado con egresados de instituciones de educación superior (IES) de todo el país.
- Conocer el resultado de la evaluación en cada área del examen, por lo que puede ubicar aquéllas donde tiene un buen desempeño, así como aquéllas en las que presenta debilidades.
- Beneficiarse curricularmente al contar con un elemento adicional para integrarse al mercado laboral.

DESTINATARIOS DEL EGEL-COMPU

Está dirigido a los egresados de la licenciatura en Ciencias Computacionales que hayan cubierto el 100% de los créditos, estén o no titulados, y en su caso a estudiantes que cursan el último semestre de la carrera, siempre y cuando la institución formadora así lo solicite.

El EGEL-COMPU se redactó en idioma español, por lo que está dirigido a individuos que puedan realizar esta evaluación bajo dicha condición lingüística. Los sustentantes con necesidades físicas especiales serán atendidos en función de su requerimiento especial.

¿Cómo se construye el EGEL-COMPU?

Con el propósito de asegurar pertinencia y validez en los instrumentos de evaluación, el Ceneval se apoya en Consejos Técnicos integrados por expertos en las áreas que conforman la profesión, los cuales pueden representar a diferentes instituciones educativas, colegios o asociaciones de profesionistas, instancias empleadoras del sector público, privado y de carácter independiente. Estos Consejos Técnicos funcionan de acuerdo con un reglamento y se renuevan periódicamente.

El contenido del EGEL-COMPU es el resultado de un complejo proceso metodológico, técnico y de construcción de consensos en el Consejo Técnico y en sus Comités Académicos de apoyo en torno a:

- i) La definición de principales funciones o ámbitos de acción del profesional
- ii) La identificación de las diversas actividades que se relacionan con cada ámbito
- iii) La selección de las tareas indispensables para el desarrollo de cada actividad
- iv) Los conocimientos y habilidades requeridos para la realización de esas tareas profesionales
- v) La inclusión de estos conocimientos y habilidades en los planes y programas de estudio vigentes de la licenciatura en Ciencias Computacionales

Todo esto tiene como referente fundamental la opinión de centenares de profesionistas activos en el campo de la Ciencias Computacionales, formados con planes de estudios diversos y en diferentes instituciones, quienes (en una encuesta nacional) aportaron su punto de vista respecto a:

- i) Las tareas profesionales que se realizan con mayor frecuencia
- ii) El nivel de importancia que estas tareas tienen en el ejercicio de su profesión
- iii) El estudio o no, durante la licenciatura, de los conocimientos y habilidades que son necesarios para la realización de estas tareas

¿Qué evalúa el EGEL-COMPU?

El examen está organizado en áreas, subáreas y temas. Las áreas corresponden a ámbitos profesionales en los que actualmente se organiza la labor del licenciado en ciencias computacionales. Las subáreas comprenden las principales actividades profesionales de cada uno de los ámbitos profesionales referidos. Por último, los temas identifican los conocimientos y habilidades necesarios para realizar tareas específicas relacionadas con cada actividad profesional.

Estructura del EGEL en Ciencias Computacionales por Áreas y Subáreas

Área/Subárea	% en el examen	Número de reactivos	Distribución de reactivos por sesión	
			1 ^a	2 ^a
A. Desarrollo de software de aplicación	37.60	50	50	
1. Análisis, diseño y codificación de software de aplicación	28.57	38	38	
2. Implementación, pruebas y mantenimiento de software de aplicación	9.02	12	12	
B. Desarrollo de software de base para diversos entornos	28.57	38	20	18
1. Modelado de software de base	15.04	20	20	
2. Implementación y prueba de software de base	13.53	18		18
C. Solución a problemas en computación teórica	33.83	45		45
1. Modelado de problemas en computación teórica	12.03	16		16
2. Implementación de técnicas y algoritmos en computación teórica	21.81	29		29
Total de reactivos	100	133	70	63
Estructura aprobada por el Consejo Técnico del EGEL-COMPU el 7 de septiembre de 2015.				
*NOTA: Adicionalmente se incluye un 30% de reactivos piloto que no califican.				

¿Qué tipo de preguntas se incluyen en el examen?

En el examen se utilizan reactivos o preguntas de opción múltiple que contienen fundamentalmente los siguientes dos elementos:

- La base es una pregunta, afirmación, enunciado o gráfico acompañado de una instrucción que plantea un problema explícitamente.
- Las opciones de respuesta son enunciados, palabras, cifras o combinaciones de números y letras que guardan relación con la base del reactivo, donde *sólo una* opción es la correcta. Para todas las preguntas del examen siempre se presentarán cuatro opciones de respuesta.

Durante el examen usted encontrará diferentes formas de preguntar. En algunos casos se le hace una pregunta directa, en otros se le pide completar una información, algunos le solicitan elegir un orden determinado, otros requieren de usted la elección de elementos de una lista dada y otros más le piden relacionar columnas. Comprender estos formatos le permitirá llegar mejor preparado al examen. Con el fin de apoyarlo para facilitar su comprensión, a continuación se presentan algunos ejemplos.

1. Preguntas o reactivos de cuestionamiento directo

En este tipo de reactivos el sustentante tiene que seleccionar una de las cuatro opciones de respuestas a partir del criterio o acción que se solicite en el enunciado, afirmativo o interrogativo, que se presenta en la base del reactivo.

Ejemplo correspondiente al área de Solución a problemas en computación teórica:

El algoritmo de Dijkstra, llamado también algoritmo de caminos mínimos, determina el camino más corto de un vértice origen al resto de vértices en un grafo dirigido, con pesos en cada arista.

Identifique la complejidad del algoritmo si se utiliza un montículo (árbol *heap*), considerando el conjunto de vértices V y el conjunto de aristas E.

```
DIJKTRA (Grafo G, nodo_fuentes)
    for u ∈ V [G] do
        distancia [u] = INFINITO
        padre [u] = NULL
    distancia [s] = 0
    Encolar (color,grafo)
    mientras cola no es vacia do
        u = extraer_minimo (cola)
        for v ∈ adyacencia [u] do
            if distancia [v] > distancia [u] + peso (u,v) do
                distancia [v] = distancia [u] + peso (u,v)
                padre [v] = v
```

- A) $O(|V|^2)$
- B) $O((\log(|V|+|E|))$
- C) $O((|E|+|E| \log |V|))$
- D) $O((|E|+|V|) \log |V|)$

Argumentación de las opciones de respuesta

La opción correcta es la D si en este algoritmo se utiliza una representación dinámica (montículo), se justifica que el número de aristas es mucho menor a N^2 , y de ahí se podría emplear un árbol parcialmente ordenado para organizar los nodos V-S, y por lo tanto se reduciría la complejidad del ciclo más interno un orden logarítmico multiplicado por el número de vértices (ciclo más externo) más las operaciones previas con las aristas.

2. Ordenamiento

Este tipo de reactivos demandan el ordenamiento o jerarquización de un listado de elementos de acuerdo con un criterio determinado. La tarea del sustentante consiste en seleccionar la opción en la que aparezcan los elementos en el orden solicitado.

Ejemplo correspondiente al área de Desarrollo de software de aplicación:

Es el orden de las etapas para la realización del estudio de simulación.

1. Definición del sistema
2. Validación
3. Experimentación
4. Formulación del modelo
5. Documentación
6. Interpretación
7. Colección de datos
8. Implementación del modelo

A) 1, 4, 7, 8, 2, 3, 6, 5

B) 1, 2, 7, 3, 5, 4, 8, 6

C) 2, 3, 4, 5, 6, 7, 8, 1

D) 5, 2, 7, 1, 4, 3, 6, 8

Argumentación de las opciones de respuesta

La opción correcta es la A, es el orden adecuado para realizar un estudio de simulación.

3. Elección de elementos

A partir de un criterio, se seleccionan elementos que forman parte de un conjunto incluido en la base. En las opciones de respuesta se presentan subconjuntos.

Ejemplo correspondiente al área de Desarrollo de software de aplicación:

Con base en el siguiente pseudocódigo, señale los errores de lógica para un problema en el que mediante una cantidad, un interés y un pago solicitados, se determinan los meses que tomará saldar una cantidad por pagar de la que se cobra un interés mensual.

```
inicio
    pedir una cantidad
    guardar en c1
    pedir un pago
    guardar en p1
    pedir un interés (%)
    guardar en int
    int = int / 100
    mes = 0
    mientras la cantidad sea diferente de 0 hacer
        c1 = c1 * int
        c1 = c1 - p1
        mes = mes + 1
    fin mientras
    mostrar mes
fin
```

1. No funciona con algunos valores
2. Puede quedar indefinidamente dentro de algún ciclo
3. Acepta valores no válidos
4. Realiza operaciones innecesarias
5. No da un resultado
6. Da resultados erróneos

- A) 1, 2, 4
- B) 1, 4, 5
- C) 2, 3, 6
- D) 3, 5, 6

Argumentación de las opciones de respuesta

La opción correcta es la C. Cumple con que el código tenga todos los errores elegidos. Puede quedar indefinidamente dentro de algún ciclo, acepta valores no válidos y da resultados erróneos.

4. Relación de columnas

En este tipo de reactivos hay dos columnas, cada una con contenidos distintos, que el sustentante tiene que relacionar de acuerdo con el criterio especificado en la base del reactivo:

Ejemplo correspondiente al área de Desarrollo de software de aplicación:

Un sistema de información que controla el inventario de un almacén debe presentar un manual detallado de las pruebas realizadas. Relacione cada componente del plan de pruebas con su respectiva descripción.

Descripciones	Expresiones aritméticas
1. Elementos probados	a) Determinar las herramientas de <i>software</i> requeridas y la utilización estimada del hardware
2. Registro de pruebas	b) Planificar las pruebas para que todos los requerimientos se prueben individualmente
3. Restricciones	c) Especificar los elementos del proceso del <i>software</i> que deben probarse
4. Trazabilidad de requerimientos	d) Indicar que las pruebas no sólo deben ejecutarse, sino que también deben almacenarse en una bitácora
	e) Se deben anticipar las restricciones que afectan al proceso de pruebas, como la escasez de personal

A) 1a, 2c, 3d, 4b
B) 1b, 2c, 3e, 4a
C) 1c, 2a, 3e, 4d
D) 1e, 2d, 3c, 4b

Argumentación de las opciones de respuesta

La opción correcta es la D, porque cada componente está con su descripción

PRIMERA PRÁCTICA DE EXAMEN

1. El estudio del sistema implica
 - A. estudio de un sistema existente
 - B. Documentar el sistema existente.
 - C. Identificar las deficiencias actuales y establecer nuevas metas.
 - D. Todo lo anterior
 - E. Ninguna de las anteriores

 2. La herramienta principal utilizada en el diseño estructurado es una:
 - A. diagrama de estructura
 - B. diagrama de flujo de datos
 - C. diagrama de flujo del programa
 - D. módulo
 - E. Ninguna de las anteriores

 3. En un _____, un módulo del nuevo sistema de información se activa a la vez.
 - A. Ciclo de vida del desarrollo del sistema
 - B. Herramienta CASE
 - C. Conversión por fases
 - D. factores de éxito
 - E. Ninguna de las anteriores.

 4. En Prototipado.
 - A. BASIC se utiliza
 - B. COBOL se utiliza
 - C. Se utilizan 4GLs.
 - D. el sistema está documentado
 - E. Ninguna de las anteriores

 5. Las instrucciones paso a paso que resuelven un problema se llaman _____.
 - A. un algoritmo
 - B. una lista
 - C. un plan
 - D. Una estructura secuencial.
- Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.*
- EDICT_GUIA_MEX Editorial Guías Mexico*

E. Ninguna de las anteriores

6. El enfoque utilizado en el análisis y diseño de arriba a abajo es
- A. para identificar las funciones de nivel superior mediante la combinación de muchos componentes más pequeños en una sola entidad
 - B. Preparar diagramas de flujo una vez finalizada la programación.
 - C. para identificar una función de nivel superior y luego crear una jerarquía de módulos y componentes de nivel inferior.
 - B. Todo lo anterior
 - E. Ninguna de las anteriores
7. ¿Cuál de los siguientes no es un factor en el fracaso de los proyectos de desarrollo de sistemas?
- A. tamaño de la empresa
 - B. participación inadecuada del usuario
 - C. falla de integración de sistemas
 - D. continuación de un proyecto que debería haber sido cancelado.
 - E. Ninguna de las anteriores
8. Un anillo, se refiere a una cadena de registros, el último de los cuales se refiere al primer registro, en la cadena, se llama a / an
- A. direccionamiento
 - B. ubicación
 - C. puntero
 - D. bucle
 - E. Ninguna de las anteriores
9. La herramienta principal utilizada en el diseño estructurado es una:
- A. diagrama de flujo de datos
 - B. Módulo
 - C. diagrama de estructura
 - D. diagrama de flujo del programa
 - E. Ninguna de las anteriores
10. El _____ de un problema responderá a la pregunta, "¿Qué información necesitará saber la computadora para imprimir o mostrar los tiempos de salida?"
- A. entrada

- D. Propósito
- E. Ninguna de las anteriores

11. ¿Cuál de las siguientes afirmaciones no es cierta para la opción de arrendamiento?

- R. Los cargos de arrendamiento son más bajos que los de alquiler para el mismo período y también son deducibles de impuestos.
- B. El contrato de arrendamiento se puede escribir para mostrar que los pagos más altos son los primeros años para reflejar la disminución en el valor del sistema
- C. Los gastos de seguro, mantenimiento y otros están incluidos en el cargo de alquiler.
- B. Todo lo anterior
- E. Ninguna de las anteriores

12. Las dos clasificaciones de entradas son

- A. energías y mantenimiento
- B. mantenimiento y residuos
- C. mantenimiento y señal
- D. productos y residuos
- E. Ninguna de las anteriores

13. Se prepara la documentación.

- A. en cada etapa
- B. en el diseño del sistema
- C. en el análisis del sistema
- D. en el desarrollo del sistema
- E. Ninguna de las anteriores

14. La fase de implementación del sistema conlleva

- A. Salidas del sistema
- B. carrera piloto
- C. carreras paralelas
- D. Todo lo anterior
- E. Ninguna de las anteriores

- A. El usuario y el personal de sistemas deben trabajar juntos.
- B. Se deben tomar medidas para eliminar el antiguo sistema.
- C. la documentación debe ser enfatizada
- D. Los componentes no mecánicos del sistema deben ser considerados.
- E. Ninguna de las anteriores

16. Gestores potenciales usuarios del SIG.

- A. seleccionar las configuraciones de equipos óptimas
- B. evaluar configuraciones de equipos alternativos
- C. describir las necesidades de información
- D. Todo lo anterior
- E. Ninguna de las anteriores

17. La prueba positiva es

- A. ejecutar el sistema con datos de línea por el usuario real
- B. Asegurarse de que los nuevos programas de hecho procesen ciertas transacciones de acuerdo con las Especificaciones
- C. está verificando la lógica de uno o más programas en el sistema candidato
- D. Pruebas de cambios realizados en un programa nuevo o existente.
- E. Ninguna de las anteriores

18. Lenguaje de definición de datos (DDL)

- A. describe cómo se estructuran los datos en la base de datos
- B. especifica para el DBMS lo que se requiere; Las técnicas utilizadas para procesar datos.
- C. determinar cómo deben estructurarse los datos para producir la vista del usuario
- D. Todo lo anterior

19. En la fase 1 del ciclo de vida del desarrollo del sistema, ¿cuáles de los siguientes aspectos generalmente se analizan?

- A. salidas
- B. entrada (transacciones)
- C. controles
- D. Todo lo anterior
- E. Ninguna de las anteriores

20. Durante la fase de mantenimiento.

- A. Se establecen los requisitos del sistema.
- B. Se lleva a cabo el análisis del sistema.
- C. los programas son probados
- D. Todo lo anterior
- E. Ninguna de las anteriores

21. En análisis y diseño de arriba hacia abajo.

- A. Cada fase subsiguiente es más detallada que la fase anterior.
- B. Cada fase subsiguiente es tan detallada como la fase anterior.
- C. Cada fase subsiguiente es menos detallada que la fase anterior.
- D. Todo lo anterior
- E. Ninguna de las anteriores

22. El símbolo _____ se usa en un diagrama de flujo para representar una tarea de cálculo.

- A. entrada
- B. Salida
- C. proceso
- D. empezar
- E. parar

23. El prototipado del sistema ayuda al diseñador a

- A. Hacer que los programadores entiendan cómo funcionará el sistema.
- B. Comunicarse con el usuario, rápidamente, cómo se verá el sistema cuando se desarrolle, y obtendrá una respuesta.
- C. entregar una demostración del software al administrador del sistema al que informa
- D. tanto (a) como (b)
- E. Ninguna de las anteriores

24. ¿Cuál de los siguientes enfoques de implementación de sistemas debe usarse si desea ejecutar el sistema antiguo y el nuevo al mismo tiempo durante un período específico?

- A. Directo
- B. piloto
- C. paralelo
- D. en fase
- E. Ninguna de las anteriores

25. Característica principal de "anillo" (ring), en la estructura de datos, es

- A. El primer registro apunta solo al último registro
- B. Último registro apunta al primer registro.

E. Ninguna de las anteriores

26. ¿Cuál de los siguientes no se considera una herramienta en la fase de diseño del sistema?

- A. gráfico circular
- B. diagrama de flujo de datos
- C. mesa de decisiones
- D. diagrama de flujo de sistemas
- E. Ninguna de las anteriores

27. Para evitar errores en la transcripción y la transposición, durante la entrada de datos, el analista del sistema debe

- A. proporcionar un dígito de control (check digit)
- B. proporcionar un total de preparado (hash)
- C. proporcionar totales de lote (batch)
- D. Todo lo anterior

28. Las pruebas secuenciales o en serie son

- A. ejecutar el sistema con datos de línea por el usuario real
- B. Asegurarse de que los nuevos programas de hecho procesen ciertas transacciones de acuerdo con las Especificaciones
- C. está verificando la lógica de uno o más programas en el sistema candidato
- D. Pruebas de cambios realizados en un programa nuevo o existente.
- E. Ninguna de las anteriores

29. Una consideración evaluada por la gerencia cuando se planea convertir a un sistema de computadora es:

- A. mantenimiento
- B. software disponible
- C. velocidad del tamaño de la CPU
- D. Todo lo anterior
- E. Ninguna de las anteriores

- A. significa almacenar registros en bloques contiguos de acuerdo con una clave.
- B. Almacena los registros de forma secuencial, pero utiliza un índice para localizar los registros.
- C. utiliza un índice para cada tipo de clave
- D. tiene registros colocados al azar en todo el archivo
- E. Ninguna de las anteriores

31. Una ayuda para el diseño de un sistema

- A. ayudar a analizar tanto los datos como las actividades
- B. Ayuda en la documentación.
- C. generar código
- D. utilizando una interfaz gráfica de usuario
- E. Ninguna de las anteriores

32. La programación estructurada implica

- A. descentralización de la actividad del programa
- B. modularización funcional
- C. localización de errores
- D. Todo lo anterior
- E. Ninguna de las anteriores

33. Para reconstruir un sistema, cuál de los siguientes elementos clave debe ser considerado:

- A. retroalimentación y medio ambiente
- B. Control y procesadores.
- C. salidas y entradas
- D. Todo lo anterior
- E. Ninguna de las anteriores

34. Se llama a las condiciones inmediatamente fuera de un sistema.

- A. el límite
- B. la interfaz
- C. el medio ambiente
- D. los protocolos
- E. Ninguna de las anteriores

- A. nombre del proyecto
- B. descripciones de problemas
- C. alternativa viable
- D. diagramas de flujo de datos
- E. Ninguna de las anteriores

36. ¿Cuál de los siguientes se usa cuando una empresa sale de su organización para desarrollar un nuevo sistema?

- A. diagrama de flujo de datos
- B. diagrama de flujo de sistemas
- C. diccionario del proyecto
- D. solicitud de propuesta
- E. Norte de los anteriores.

37. ¿Cuál de las siguientes no es una característica de un sistema?

- A. opera con algún propósito
- B. tiene componentes homogéneos
- C. tiene componentes que interactúan
- D. opera dentro de un límite
- E. Ninguna de las anteriores

38. ¿Cuál de los siguientes no se considera una herramienta en la fase de diseño del sistema?

- A. Diagrama de flujo de datos
- B. Tabla de decisiones
- C. gráfico circular
- D. Diagrama de flujo del sistema
- E. Ninguna de las anteriores

39. ¿Cuál de los siguientes es una representación gráfica de los módulos en el Sistema y la interconexión entre ellos?

- A. gráfico circular
- B. diagrama de flujo
- C. Cuadro estructural
- D. Tabla de sistema

40. Un grupo de campos relacionados, se conoce como

- A. tuple¹
- B. esquema
- C. registros
- D. archivo
- E. Ninguna de las anteriores

41. Un ejemplo de una estructura de datos jerárquica es

- A. matriz
- B. Lista de enlaces
- C. arbol
- D. anillo
- E. Ninguna de las anteriores

42. Los punteros son útiles en

- A. Localización de un sector particular de un disco magnético
- B. Señalar los errores en los datos de entrada
- C. Atravesando una lista enlazada
- D. Todo lo anterior
- E. Ninguna de las anteriores

43. La entrada de datos en línea es más adecuada en el caso de

- A. actualizando la nómina maestra
- B. ingresar entradas mensuales en el diario
- C. Procesamiento de pago de cheque en un banco
- D. Todo lo anterior
- E. Ninguna de las anteriores

44. Un pseudocódigo es

- A. un código de nivel de máquina
- B. un número al azar
- C. un diagrama de flujo

Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a Editorial_Guia_Mex

En matemáticas, una tupla es una lista (secuencia) ordenada de elementos finitos.

- D. Inglés estructurado para comunicar la lógica de un programa.
- E. Ninguna de las anteriores

45. ¿Cuál de los siguientes tipos de archivo tiene la menor duración?

- A. archivo maestro
- B. archivo de programa
- C. archivo de transacción
- D. archivo de trabajo
- E. Ninguna de las anteriores

46. Un estudio de viabilidad.

- A. Incluye una declaración de los problemas.
- B. considera una solución única.
- C. tanto (a) como (b)
- D. una lista de solución alternativa considerada
- E. Ninguna de las anteriores

47. Se llama un procesador de textos diseñado para programadores.

- A. un formateador
- B. un compilador
- C. un editor
- D. un depurador
- E. Ninguna de las anteriores

48. Una descripción declaración por declaración de un procedimiento se detalla en a;

- A. narrativa escrita
- B. registro de procedimientos
- C. diagrama de flujo de sistemas
- D. diseño de registro
- E. Ninguna de las anteriores

49. Análisis costo-beneficio.

- A. evalúa los factores tangibles y no tangibles
- B. compara el costo, con los beneficios, de introducir un sistema basado en computadora
- C. estima los costos de hardware y software
- D. Todo lo anterior
- E. Ninguna de las anteriores

50. El término usado para referirse a la verificación de los resultados de una computadora, con los documentos de entrada correspondientes, se llama

- A. auditar alrededor de la computadora
- B. Auditoría a través de la computadora.
- C. control de proceso
- D. prueba beta
- E. Ninguna de las anteriores

51. ¿Cuál de los siguientes no es un requisito del diseño estructurado?

- A. Debería estar formado por una jerarquía de módulos.
- B. Debería usar muchas declaraciones GOTO
- C. El código debe ejecutarse de arriba a abajo dentro de cada módulo
- D. Cada módulo debe ser lo más independiente posible de todos los demás módulos, excepto su principal.
- E. Ninguna de las anteriores

52. Un gráfico jerárquico que divide un programa grande en módulos, en los que se diseñan mayores detalles en niveles de programación sucesivamente más bajos, se denomina

- A. esquema
- B. subesquema
- C. estructura
- D. Todo lo anterior
- E. Ninguna de las anteriores

53. ¿Cuál de las siguientes es (son) la (s) característica (s) de un sistema?

- A. organización
- B. interacción
- C. la interdependencia
- D. Todo lo anterior
- E. Ninguna de las anteriores

54. El término ejecución paralela se refiere a

- A. el mismo trabajo se ejecuta en dos computadoras diferentes para probar sus velocidades
- B. El procesamiento de dos trabajos diferentes iniciados desde dos terminales.
- C. la operación concurrente de un sistema existente y un nuevo sistema
- D. Todo lo anterior
- E. Ninguna de las anteriores

55. El diagrama del "cuadro grande" de un sistema es el

- A. diagrama de bloques
- B. diagrama lógico
- C. diagrama de flujo del sistema
- D. diagrama de flujo del programa
- E. Ninguna de las anteriores

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

56. ¿Cuál de los siguientes es el requisito más crítico de un sistema de reserva de pasajeros?

- A. facilidad de programación
- B. tiempo de respuesta
- C. interfaz gráfica de usuario
- D. impresión de entradas
- E. Ninguna de las anteriores

57. El anuncio de la implementación de MIS es hecho por

- A. gerentes de nivel operacional
- B. gerentes de nivel táctico
- C. el presidente
- D. Todo lo anterior
- E. Ninguna de las anteriores

58. El _____ de un problema responderá a la pregunta, "¿Qué quiere ver el usuario impreso o visualizado en la pantalla?"

- A. entrada
- B. Salida
- C. Procesamiento
- D. Propósito
- E. Ninguna de las anteriores

59. Se llama representación gráfica de un sistema de información.

- A. diagrama de flujo
- B. pictograma
- C. diagrama de flujo de datos
- D. histograma
- E. Ninguna de las anteriores

60. El archivo de factura pendiente se debe iniciar en un dispositivo de almacenamiento de acceso de perdón si

- A. la entrada de datos de factura está en línea.
- B. El registro de pago se realiza en un lote de rodeo.
- C. Las consultas sobre pagos deben ser respondidas en línea.
- D. El último registro apunta al primer registro.
- E. Ninguna de las anteriores

61. La instrucción de la receta "Batir hasta que esté suave" es un ejemplo de la estructura ____.

- A. control
- B. Repetición
- C. selección
- D. secuencia
- E. cambio

62. ¿Cuál de los siguientes símbolos no es el Diagrama de flujo de datos (DFD):

- A. un cuadrado
- B. un rectángulo abierto
- C. un circulo
- D. un triangulo
- E. una burbuja

63. El diseño empaquetado incluye todo lo siguiente excepto;

- A. diagramas de flujo del programa
- B. configuración del sistema
- C. descripciones del módulo
- D. Todo lo anterior
- E. Ninguna de las anteriores

64. El analista se entera de las necesidades de información del gerente a través del uso de

- A. encuesta por correo
- B. entrevista en profundidad
- C. experimento controlado
- D. observación
- E. Ninguna de las anteriores

65. En el procesamiento, los controles deben establecerse en un sistema, a fin de

- A. prohibir la manipulación de información por parte de personas no autorizadas
- B. Verificar que todos los datos hayan sido procesados.
- C. bloquear o capturar datos defectuosos desde la entrada al procesamiento
- D. Todo lo anterior
- E. Ninguna de las anteriores

66. Un elemento de datos en un registro, que distingue de manera única al registro, de todos los demás registros, se llama

- A. tiempo de respuesta
- B. llave
- C. Revisión del sistema
- D. datos de prueba
- E. Ninguna de las anteriores

67. Un documento de Turnaround.

- A. se distribuye entre todo el personal de una organización.
- B. se distribuye entre todos los jefes de departamento de una organización.
- C. se emite en un punto que luego vuelve como entrada
- D. Todo lo anterior
- E. Ninguna de las anteriores

68. El primer paso en el proceso de resolución de problemas es _____.

- A. planear el algoritmo
- B. analizar el problema
- C. Escritorio-comprobar el algoritmo
- D. Evaluar y modificar (si es necesario) el programa.
- E. Codificar el algoritmo.

69. La verificación de la calidad del software tanto en entornos simulados como en línea, se conoce como

- A. revisando
- B. usabilidad
- C. validez
- D. validación
- E. Ninguna de las anteriores

70. Algunas empresas han creado una posición de combinación llamada

- A. gerente / analista
- B. programador / operador
- C. analista / programador
- D. gerente / operador
- E. Ninguna de las anteriores

71. El primer paso de la fase de implementación es

- A. planificación de la implementación
- B. selecciona la computadora
- C. Anunciar el proyecto de implementación.
- D. Preparar instalaciones físicas.
- E. Ninguna de las anteriores

72. La ventaja de la Opción de Compra es (son)

- A. No se requiere financiamiento. El riesgo de obsolescencia del sistema se traslada al arrendador.
- B. La flexibilidad de modificar el sistema a voluntad.
- C. Los seguros, el mantenimiento y otros gastos están incluidos en el cargo de alquiler
- D. Todo lo anterior
- E. Ninguna de las anteriores

73. La decisión de la gerencia de alquilar un sistema informático puede basarse en

- A. deseo de evitar un gran pago de una sola vez
- B. ventajas fiscales
- C. flexibilidad operacional en el cambio de hardware
- D. Todo lo anterior
- E. Ninguna de las anteriores

74. Todas las siguientes herramientas se utilizan para descripciones de procesos, excepto:

- A. Inglés estructurado
- B. tablas de decisiones
- C. pseudocódigo
- D. diccionarios de datos
- E. Ninguna de las anteriores

75. ¿Cuál de los siguientes no sería un resultado importante de la fase de análisis de sistemas estructurados?

- A. diccionarios de datos
- B. diagramas de flujo de datos
- C. diagramas de relaciones de entidades
- D. Modelo prototipo
- E. Ninguna de las anteriores

76. Las ayudas al diseño del sistema deberían

- A. ayudar a analizar tanto los datos como las actividades
- B. Ayuda en la documentación.
- C. ayuda en la programación
- D. generar código
- E. Ninguna de las anteriores

77. Todos los datos que se ingresan en la computadora, por primera vez, pasarán por el proceso de

- A. Pruebas del sistema
- B. Pruebas secuenciales
- C. prueba de cuerdas
- D. Validación
- E. Ninguna de las anteriores

78. Los errores cometidos en la etapa de análisis del sistema se muestran en:

- A. implementación
- B. diseño del sistema
- C. desarrollos del sistema
- D. Todo lo anterior
- E. Ninguna de las anteriores

79. La combinación de los enfoques de arriba hacia abajo y de abajo hacia arriba puede denominarse

- A. enfoque integrador
- B. enfoque interpretativo
- C. enfoque interactivo
- D. ambos b y c
- E. Ninguna de las anteriores

80. ¿Cuál de las siguientes afirmaciones no es verdadera?

- A. Los diagramas de relaciones de entidades se utilizan para diseñar archivos
- B. Descripción general de los sistemas puede ser prescrito por los diagramas de flujo de datos
- C. Las fases, en un ciclo de vida de los sistemas, son fijas e invariantes.
- D. Los archivos indexados son más rápidos que los archivos secuenciales en todas las situaciones
- E. Ninguna de las anteriores

81. El método más largo de conversión es

- A. Directo
- B. paralelo
- C. en fase
- D. piloto
- E. Ninguna de las anteriores

82. Los cambios realizados periódicamente en un sistema, después de su implementación, se conocen como sistema

- A. análisis
- B. diseño
- C. desarrollo
- D. mantenimiento
- E. ninguna de las anteriores

83. El primer paso en el ciclo de vida del desarrollo de sistemas es

- A. diseño de base de datos
- B. diseño del sistema
- C. investigación preliminar y análisis
- D. interfaz gráfica de usuario
- E. Ninguna de las anteriores

84. Nassi - Schneiderman cartas

- A. están siendo reemplazados por diagramas de flujo
- B. A menudo describe estructuras de control superpuestas
- C. se componen de cajas dentro de cajas
- D. ambos (a) y (c)
- E. Ninguna de las anteriores

85. ¿Cuál de las siguientes herramientas no se usa para modelar el nuevo sistema?

- A. tablas de decisiones
- B. diccionario de datos
- C. diagramas de flujo de datos
- D. Todo lo anterior
- E. Ninguna de las anteriores

86. Un diseño que consiste en una jerarquía de módulos; Cada módulo tiene una sola entrada y una única subrutina de salida, que se conoce como

- A. estructura de jerarquía
- B. diseño de arriba hacia abajo
- C. Tuple
- D. Turnaround
- E. Ninguna de las anteriores

87. La creación rápida de prototipos demuestra la calidad de un diseño de

- A. tener un programa simular el sistema real
- B. hacer que el analista del sistema presente una descripción general del diseño a los usuarios, programadores y consultores
- C. tener un diagrama de flujo de datos
- D. tanto (a) como (b)
- E. Ninguna de las anteriores

88. El concepto de caja negra:

- A. se invoca al describir un sistema en términos de entradas y salidas, dejando el proceso de transformación en una caja negra.
- B. asume que la caja negra es independiente
- C. asume que las entradas y salidas permanecerán estables
- D. Todo lo anterior
- E. Ninguna de las anteriores

89. Una investigación del sistema puede resultar de

- A. una investigación de análisis
- B. un sistema programado revisado
- C. la solicitud formal de un gerente
- D. Todo lo anterior
- E. Ninguna de las anteriores

90. La representación gráfica de la lógica de control de las funciones de procesamiento o módulos que representan un sistema, se conoce como:

- A. Análisis estructurado
- B. gráfica estructurada
- C. Inglés estructurado

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

RESPUESTAS DE LA PRIMERA PRÁCTICA DE EXAMEN

1	D
2	A
3	C
4	C
5	A
6	C
7	A
8	C
9	C
10	A
11	C
12	C
13	A
14	A
15	C
16	E
17	B
18	A

19	D
20	E
21	A
22	C
23	B
24	C
25	B
26	A
27	D
28	C
29	D
30	D
31	A
32	B
33	D
34	C
35	B
36	D

37	B
38	C
39	C
40	A
41	C
42	C
43	C
44	D
45	D
46	A
47	C
48	A
49	D
50	A
51	B
52	C
53	D
54	C

55	C
56	B
57	D
58	B
59	C
60	C
61	B
62	D
63	B
64	D
65	B
66	C
67	B
68	D
69	C
70	A
71	B
72	D

73	D
74	D
75	A
76	D
77	A
78	A
79	D
80	A
81	D
82	C
83	C
84	D
85	B
86	A
87	E
88	D
89	B
90	B

SEGUNDA PRÁCTICA DE EXAMEN

1. ¿Cuál de los siguientes es un conjunto integrado de datos con una redundancia mínima, de modo que diferentes aplicaciones puedan usar los datos requeridos?

- A. registros
- B. Archivo
- C. DBMS
- D. esquema
- E. Ninguna de las anteriores

2. Cuando uno tiene que procesar todos los registros de datos en un archivo, la mejor organización de tesselas es

- A. indexado
- B. secuencial
- C. acceso directo
- D. acceso aleatorio
- E. Ninguna de las anteriores

3. Un rectángulo abierto.

- A. define una fuente o destino de datos del sistema
- B. identifica el flujo de datos
- C. representa un proceso que transforma los flujos de datos entrantes en flujos de datos salientes
- D. es un almacén de datos de datos en reposo, o un repositorio temporal de datos
- E. Ninguna de las anteriores

4. Las factibilidades estudiadas en la investigación preliminar son (son):

- A. viabilidad técnica
- B. viabilidad económica
- C. viabilidad operacional
- D. Todo lo anterior
- E. Ninguna de las anteriores

5. Los programas de marcas de ramas se caracterizan mejor como

- A. programas de simulador
- B. características (Hallmarking)
- C. Programas actuales del sistema.
- D. Software de proveedor para aplicaciones.
- E. Ninguna de las anteriores

6. El método de conversión en el que los usuarios, al estar acostumbrados a un sistema antiguo, siguen utilizando el sistema antiguo, junto con el nuevo sistema, es

- A. multiprocesamiento
- B. multitarea
- C. carrera paralela
- D. Todo lo anterior
- E. Ninguna de las anteriores

7. Un programador revisa la precisión de un algoritmo por _____.

- A. analizando
- B. Codificación
- C. Control de escritorio
- D. planificación
- E. Ninguna de las anteriores

8. Organización secuencial

- A. significa almacenar registros en bloques contiguos de acuerdo con una clave.
- B. Almacena los registros de forma secuencial, pero utiliza un índice para localizar los registros.
- C. utiliza un índice para cada tipo de clave
- D. tiene registros colocados al azar en todo el archivo
- E. Ninguna de las anteriores

9. ¿Cuál de los siguientes enfoques de implementación de sistemas debería usarse si desea ejecutar el sistema antiguo y el nuevo al mismo tiempo durante un período específico?

- A. Directo
- B. piloto
- C. en fase
- D. paralelo
- E. Ninguna de las anteriores

10. Un _____ es un software diseñado para dibujar diagramas de sistemas de información, escribir especificaciones de procesos y mantener diccionarios de datos.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
A. Ciclo de vida del desarrollo del sistema www.gulaceneval.com.mx Todos los Derechos Reservados.**

- B. Herramienta CASE
- C. Conversión por fases
- D. factores de éxito
- E. Movimiento de los anteriores.

11. Los requisitos también sirven como una lista de verificación de evaluación al final del proyecto de desarrollo, por lo que a veces se les llama _____.

- A. Ciclo de vida del desarrollo del sistema
- B. Herramienta CASE
- C. Conversión por fases
- D. factores de éxito
- E. Movimiento de los anteriores.

12. Una pareja:

- A. es un elemento de datos que se mueve de un módulo a otro
- B. está representado por una línea que une dos módulos.
- C. significa que el módulo superior tiene la capacidad de llamar al módulo inferior
- D. Ninguna de las anteriores
- E. Ninguna de las anteriores

13. ¿Cuál de las siguientes no es una característica del desarrollo de sistemas estructurados?

- A. partición de sistemas en niveles manejables de detalle
- B. Especificación de las interfaces entre módulos.
- C. el uso de herramientas gráficas, como los diagramas de flujo de datos, para modelar el sistema
- D. Todas las anteriores son características.
- E. Ninguna de las anteriores

14. Una prueba de auditoría es

- A. diseñado para permitir el rastreo de cualquier registro de entrada o proceso realizado en un sistema, de vuelta a sus fuentes originales
- B. Post-auditoría del sistema después de su implementación.
- C. Datos de prueba utilizados por el auditor del sistema.
- D. Todo lo anterior
- E. Ninguna de las anteriores

15. ¿Cuál de los siguientes no ocurre en la fase 4 del ciclo de vida del desarrollo del sistema (SDLC)?

- A. conducir entrevistas
- B. entrenar a los usuarios
- C. adquirir hardware y software
- D. probar el nuevo sistema

16. El procedimiento para evaluar el rendimiento relativo de diferentes computadoras se realiza mediante el proceso denominado

- A. procesamiento por lotes
- B. procesamiento secuencial
- C. benchmarking
- D. Todo lo anterior
- E. Ninguna de las anteriores

17. El concepto de caja negra se basa en la suposición (es) de que;

- A. El suprasistema es estable.
- B. Las cajas negras son dependientes o ambientes.
- C. La relación entre las entradas y la salida es estable.
- D. Todo lo anterior
- E. Ninguna de las anteriores

18. ¿Cuál de las siguientes opciones se puede generar como resultado del uso de una herramienta CASE?

- A. código de programación
- B. diagramas de flujo y diagramas de flujo de datos
- C. prototipos y análisis costo / beneficio.
- D. Todo lo anterior
- E. Ninguna de las anteriores

19. Los programadores se refieren a los elementos necesarios para alcanzar la meta de un problema como _____

- A. entrada
- B. salida
- C. Procesamiento
- D. Propósito
- E. Ninguna de las anteriores

20. ¿Cuál de los siguientes no es una ventaja del enfoque estructurado para el desarrollo del sistema?

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
A. Se centra en los aspectos físicos del sistema.**

- B. La especificación del sistema es muy gráfica.
- C. se estudia a fondo el área de usuario.
- D. La documentación es acumulativa.
- E. Ninguna de las anteriores

21. Se utiliza el benchmarking.

- A. para seleccionar sistemas informaticos
- B. para mantener los archivos en condiciones de datos
- C. para la aplicación de proto-mecanografía
- D. para la aceptación del sistema
- E. Ninguna de las anteriores

22. Los primeros elementos definidos para un nuevo sistema son sus:

- A. entradas
- B. salidas
- C. almacenamiento
- D. procesamiento

23. El diagrama de estructura es

- A. un documento de lo que hay que realizar
- B. una declaración de requisito de procesamiento de información
- C. una partición jerárquica del programa
- D. Todo lo anterior
- E. Ninguna de las anteriores

24. ¿Cuál de las siguientes opciones afecta la medida en que está involucrado con el ciclo de vida del desarrollo del sistema en su empresa?

- A. la descripción de su trabajo
- B. tamaño de la organización
- C. tu experiencia relevante
- D. Todo lo anterior
- E. Ninguna de las anteriores

25. El sistema de reservas de los ferrocarriles indios es un ejemplo de

- A. sistema de procesamiento de transacciones
- B. Sistema de apoyo de decisión interactivo
- C. Sistema de control de gestión.
- D. sistema experto
- E. Ninguna de las anteriores

26. La instrucción "Si está lloviendo afuera, entonces toma un paraguas para trabajar" es un ejemplo de la estructura _____,

- A. control
- B. Repetición
- C. selección
- D. secuencia
- E. cambio

27. Al momento del estudio del sistema, los diagramas de flujo se dibujan usando

- A. símbolos no estándar
- B. simbolos generales
- C. símbolos abreviados
- D. símbolos específicos
- E. Ninguna de las anteriores

28. Los programadores usan _____ para organizar y resumir los resultados de su análisis de problemas.

- A. Diagramas de flujo
- B. Gráficos de entrada
- C. Gráficos IPO
- D. Gráficos de salida
- E. Gráficos de proceso

29. ¿Cuál de las siguientes funciones es / son realizadas por el cargador?

- A. Asigne espacio en la memoria para los programas y resuelva las referencias simbólicas entre las cubiertas de objetos
- B. Coloque físicamente las instrucciones y los datos de la máquina en la memoria
- C. Ajuste todas las ubicaciones dependientes de la dirección, como las constantes de dirección, para que correspondan con el espacio asignado
- D. Todo lo anterior

30. Convierta las instrucciones del lenguaje de máquina 11014B en lenguaje ensamblador, asumiendo que no fueron generadas por pseudo-operaciones:

- A. ASRA
- B. LOADA h#OD4E, i
- C. STOREA h#014B, d
- D. ADDA h#01FE, i

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

31. La (s) ventaja (es) de incorporar el procesador de macros en el paso 1 es / son:

- A. Muchas funciones no tienen que ser implementadas dos veces
- B. Las funciones se combinan y no es necesario crear archivos intermedios como salida del procesador de macros y entrada al ensamblador
- C. El programador dispone de más flexibilidad, ya que puede utilizar todas las funciones del ensamblador junto con las macros.
- D. Todo lo anterior

32. ¿En qué modo de direccionamiento, la dirección efectiva del operando se genera al agregar un valor constante al contenido del registro?

- A. modo absoluto
- B. modo indirecto
- C. modo inmediato
- D. modo de índice

33. Un programa de auto-reubicación es aquel que

- A. No se puede hacer que se ejecute en ningún área de almacenamiento que no sea la designada para él en el momento de su codificación o traducción.
- B. consiste en un programa e información relevante para su reubicación.
- C. puede realizar la reubicación de sus partes sensibles a la dirección
- D. Todo lo anterior

34. Si se necesitan formularios especiales para imprimir la salida, el programador especifica estos formularios a través de

- A. JCL
- B. IPL
- C. programas de utilidad
- D. cargar modulos

35. ¿Cuál de los siguientes sistemas de software hace el trabajo de fusionar los registros de dos vuelos en uno?

- A. Sistema de documentación.
- B. Programa de utilidad
- C. software de red
- D. software de seguridad

36. Las declaraciones del lenguaje de control de trabajo (JCL) se utilizan para

- A. Lea la entrada del lector de tarjetas de baja velocidad al disco magnético de alta velocidad
- B. Especifique, para el sistema operativo, el principio y el final de un trabajo en un lote
- C. Asignar la CPU a un trabajo
- D. Todo lo anterior

37. Al analizar la compilación del programa PL / I, el término "Análisis de sintaxis" se asocia con

- A. Reconocimiento de construcciones sintácticas básicas mediante reducciones.
- B. Reconocimiento de elementos básicos y creación de símbolos uniformes.
- C. creación de matriz más opcional
- D. uso del procesador de macros para producir un código de ensamblaje más óptimo

38. Al analizar la compilación del programa PL/I, el término "análisis léxico" se asocia con

- A. Reconocimiento de construcciones sintácticas básicas mediante reducciones.
- B. Reconocimiento de elementos básicos y creación de símbolos uniformes.
- C. creación de matriz más opcional
- D. uso del procesador de macros para producir un código de ensamblaje más óptimo

39. Mesa de terminales (Terminal Table)

- A. una tabla permanente que enumera todas las palabras clave y símbolos especiales del lenguaje en forma simbólica
- B. una tabla permanente de reglas de decisión en forma de patrones para hacer coincidir con la tabla de símbolos uniforme para descubrir la estructura sintáctica
- C. consiste en una lista completa o parcial de los tokens que aparecen en el programa. Creado por el análisis léxico y utilizado para el análisis e interpretación de la sintaxis.
- D. contiene todas las constantes en el programa

40. Un compilador para un lenguaje de alto nivel que se ejecuta en una máquina y produce código para una máquina diferente se llama

- A. optimizando compilador
- B. compilador de una pasada
- C. compilador cruz
- D. compilador multipases

41. Base de datos de código de montaje está asociada con
- A. una tabla permanente que enumera todas las palabras clave y símbolos especiales del lenguaje en forma simbólica
 - B. una tabla permanente de reglas de decisión en forma de patrones para hacer coincidir con la tabla de símbolos uniforme para descubrir la estructura sintáctica
 - C. consiste en una lista completa o parcial o los tokens que aparecen en el programa. Creado por el análisis léxico y utilizado para el análisis e interpretación de la sintaxis.
 - D. versión en lenguaje ensamblador del programa que se crea en la fase de generación de código y se ingresa a la fase de ensamblaje
42. Indica cuál de las siguientes afirmaciones no es cierta acerca de 4GL?
- A. 4GL no admite un alto nivel de interacción de pantalla
 - B. Muchos paquetes de sistemas de gestión de bases de datos soportan 4GLs
 - C. Un 4GL es una herramienta de software que está escrita, posiblemente, en algún lenguaje de tercera generación.
 - D. Todo lo anterior
43. Una estrategia de desarrollo mediante la cual los módulos de control ejecutivo de un sistema se codifican y prueban primero, se conoce como
- A. Desarrollo de abajo hacia arriba
 - B. Desarrollo de arriba hacia abajo.
 - C. Desarrollo de izquierda-derecha
 - D. Todo lo anterior
44. Un programa no reubicable es aquel que
- A. No se puede hacer que se ejecute en ningún área de almacenamiento que no sea la designada para él en el momento de su codificación o traducción.
 - B. consiste en un programa e información relevante para su reubicación.
 - C. puede realizar la reubicación de sus partes sensibles a la dirección
 - D. Todo lo anterior
45. La multiprogramación fue posible por
- A. unidades de entrada / salida que operan independientemente de la CPU
 - B. sistemas operativos
 - C. tanto (a) como (b)
 - D. ni (a) ni (b)

46. Un intérprete es

- A. es un programa que parece ejecutar un programa fuente como si fuera un lenguaje de máquina.
- B. un programa que automatiza la traducción del lenguaje ensamblador al lenguaje de máquina
- C. programa que acepta un programa escrito en un lenguaje de alto nivel y produce un programa objeto
- D. un programa que coloca los programas en la memoria y los prepara para la ejecución

47. El software de sistemas es un programa que dirige el funcionamiento general de la computadora, facilita su uso e interactúa con los usuarios. ¿Cuáles son los diferentes tipos de este software?

- A. sistema operativo
- B. lenguajes
- C. Utilidades
- D. Todo lo anterior

48. Convierta CHARI h#000F, de instrucciones en lenguaje ensamblador en lenguaje de máquina hexadecimal:

- A. 0111EF
- B. 9001E6
- C. DA000F
- D. 40

49. La tabla creada por análisis léxico para describir todos los literales utilizados en el programa fuente es:

- A. mesa de terminales
- B. mesa literal
- C. tabla de identificadores
- D. Reducciones

50. Las ventajas de usar lenguaje ensamblador en lugar de lenguaje de máquina son / son:

- A. Es mnemotécnico y fácil de leer.
- B. Aborda cualquier simbólico, no absoluto.
- C. La introducción de datos al programa es más fácil.
- D. Todo lo anterior

51. La acción de analizar el programa fuente en las clases sintácticas adecuadas se conoce como

- A. análisis de sintaxis
- B. análisis léxico
- C. análisis de interpretación
- D. análisis de sintaxis general

52. Las tareas de la fase de análisis de Lexial son / son:

- A. para analizar el programa fuente en los elementos básicos o tokens del lenguaje
- B. para construir una tabla literal y una tabla de identificadores
- C. para construir una tabla de símbolos uniforme
- D. Todo lo anterior

53. Un formulario de programa de reubicación es el que

- A. No se puede hacer que se ejecute en ningún área de almacenamiento que no sea la designada para él en el momento de su codificación o traducción.
- B. consiste en un programa e información relevante para su reubicación.
- C. puede realizar la reubicación de sus partes sensibles a la dirección
- D. Todo lo anterior

54. ¿De qué manera (s) se puede implementar un procesador de macros para lenguaje ensamblador?

- A. procesador de dos pasos independiente
- B. procesador de una pasada independiente
- C. procesador incorporado en el paso de un ensamblador estándar de dos pasos
- D. Todo lo anterior

55. Un programa de sistema que combina los módulos compilados por separado de un programa en una forma adecuada para la ejecución.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
A. ensamblador**

EDICTI:_GUIA_MEX Editorial Guías México

- B. cargador de enlace
- C. compilador cruz
- D. cargar y listo

56. Un compilador es

- A. un programa que coloca los programas en la memoria y los prepara para la ejecución
- B. un programa que automatiza la traducción del lenguaje ensamblador al lenguaje de máquina
- C. programa que acepta un programa escrito en un lenguaje de alto nivel y produce un programa objeto
- D. es un programa que parece ejecutar un programa fuente como si fuera un lenguaje de máquina.

57. En un esquema de carga absoluta, ¿qué función del cargador realiza el cargador?

- A. reasignación
- B. Asignación
- C. vinculación
- D. cargando

58. El administrador de procesos debe realizar un seguimiento de:

- A. el estado de cada programa
- B. La prioridad de cada programa.
- C. El soporte de gestión de información a un programador que utiliza el sistema.
- D. tanto (a) como (b)

59. Convierte las instrucciones del lenguaje de máquina 080D4E en lenguaje ensamblador, asumiendo que no fueron generadas por pseudo-operaciones:

- C. STOREA h#014B, d
- D. ADDA h#01FE, i

60. ¿Cuál de los siguientes programas del sistema renuncia a la producción de código objeto para generar un código de máquina absoluto y cargarlo en la ubicación de almacenamiento principal física desde la cual se ejecutará inmediatamente después de completar el ensamblaje?

- A. ensamblador de dos pasadas
- B. ensamblador load-and-go
- C. macroprocesador
- D. compilador

61. Convierte las 48 instrucciones de lenguaje de máquina en lenguaje ensamblador, asumiendo que no fueron generadas por pseudo-operaciones:

- A. ASRA
- B. LOADA h#OD4E, i
- C. STOREA h#014B, d
- D. ADDA h#01FE, i

62. El procesador de macros debe realizar

- A. Reconocer definiciones de macro y llamadas de macro
- B. guardar las definiciones de macros
- C. expandir llamadas de macro y sustituir argumentos
- D. Todo lo anterior

63. En un esquema de carga absoluta, la función del cargador se realiza mediante el ensamblador

- A. reasignación
- B. Asignación
- C. vinculación
- D. cargando

- A. 0111EF
- B. 03 16
- C. F8
- D. 42 65 61 72

65. ¿En qué módulo, varias instancias de ejecución producirán el mismo resultado incluso si una instancia no ha terminado antes de que la siguiente haya comenzado?

- A. módulo no reutilizable
- B. reutilizable en serie
- C. módulo reenterable
- D. módulo recursivo

66. Un cargador es

- A. un programa que coloca los programas en la memoria y los prepara para la ejecución
- B. un programa que automatiza la traducción del lenguaje ensamblador al lenguaje de máquina
- C. es un programa que parece ejecutar un programa fuente como si fuera un lenguaje de máquina.
- D. programa que acepta un programa escrito en un lenguaje de alto nivel y produce un programa objeto

67. Bug significa

- A. Un error lógico en un programa.
- B. Un error de sintaxis difícil en un programa.
- C. Documentar programas utilizando una herramienta de documentación eficiente.
- D. Todo lo anterior

68) Esta fase del SDLC se conoce como la "fase continua" en la que el sistema se evalúa y actualiza periódicamente según sea necesario.

- A. investigación preliminar
- B. diseño del sistema
- C. implementación del sistema
- D. mantenimiento del sistema

69) incluye el sistema existente, el sistema propuesto, los diagramas de flujo del sistema, el diseño modular del sistema, los gráficos de diseño de impresión y los diseños de archivos de datos.

- A. Informe de viabilidad
- B. Informe de especificación funcional
- C. Informe de especificación de diseño
- D. Términos de Referencia

70) Después de la implementación del sistema, el mantenimiento del sistema podría hacerse para

- A. Cambios menores en la lógica de procesamiento.
- B. Errores detectados durante el procesamiento.
- C. Revisión de los formatos de los informes.
- D. Todo lo anterior

71) El paso final de la fase de análisis del sistema en el SDLC es

- A. recopilar datos
- B. escribir informe de análisis del sistema
- C. proponer cambios
- D. analizar datos

72) Las diferentes fases para el desarrollo y prueba de los sistemas incluyen:

- i) Desarrollo y prueba de los programas individuales.
 - ii) Desarrollo y prueba de los módulos del sistema como parte de los subsistemas principales
 - iii) Desarrollo y prueba de los subsistemas principales como parte del sistema propuesto
- A. solo i y ii
 - B. Solo ii y iii
 - C. i y iii solamente
 - D. Todos los i, ii y iii.

73) El departamento de procesamiento de datos electrónicos tiene funciones específicas para realizar, que incluyen

- i) Departamento de sistema y programación.
 - ii) Operación del sistema informático
 - iii) Control sobre datos, informes y archivos.
 - iv) preparación de datos
- A. i, ii y iii solamente
 - B. Sólo i, iii y iv
 - C. ii, iii y iv solamente
 - D. Todas las i, ii, iii y iv

74) Se utiliza un estudio de factibilidad para determinar los sistemas propuestos.

- A. requerimientos de recursos
- B. costos y beneficios
- C. disponibilidad de hardware y software
- D. Todo lo anterior

75) La actividad involucrada en es dividir los módulos del sistema en programas más pequeños y asignar estos programas a los miembros del equipo de desarrollo del sistema.

- A. Fase de diseño del sistema
- B. Fase de desarrollo del sistema

76) tiene responsabilidades en todos los aspectos del procesamiento de datos, investigación de operaciones, organización y método, análisis de sistemas e inversión en diseño, etc.

- A. Director de Servicios de Gestión
- B. Gerente de Procesamiento de Datos
- C. analista de sistemas
- D. Analista de Sistemas Senior

77) ¿Durante qué fase del SDLC se capacita a los usuarios para usar el nuevo sistema?

- A. investigación preliminar
- B. implementación de sistemas
- C. desarrollo de sistemas
- D. mantenimiento de sistemas

78) ¿Cuáles de las siguientes son las actividades básicas involucradas en la fase de desarrollo del sistema ...?

- i) Preparar la documentación para cada uno de los programas.
- ii) Recepción de los datos del usuario para las pruebas de aceptación.
- iii) Obtención de la aprobación del usuario después de las pruebas de aceptación.
- iv) Operación y prueba de software y hardware.

- A. i, ii y iv solamente
- B. ii, iii y iv solamente
- C. i, ii y iii solamente
- D. Todas i, ii, iii y iv

79) es responsable del funcionamiento eficiente del departamento y, por lo tanto, debe ser un buen administrador, además de tener algunos conocimientos.

- A. Director de Servicios de Gestión
- B. Gerente de Procesamiento de Datos
- C. analista de sistemas
- D. Analista de Sistemas Senior

80) Usando el Enfoque, se prueba un nuevo sistema en una parte de la organización antes de implementarse en otras.

- C. en fase
- D. piloto

81) Para el desarrollo del sistema propuesto, es importante que se proporcione todo el apoyo posible al equipo de desarrollo. Este soporte incluye la disponibilidad de

- A. espacio de oficina
- B. Datos relevantes
- C. Asistencia Secretarial
- D. Todo lo anterior

82) La evaluación de la efectividad de los procedimientos de mantenimiento de archivos y la idoneidad de los procedimientos de seguridad de archivos son las principales tareas de ...

- A. Director de Servicios de Gestión
- B. Gerente de Procesamiento de Datos
- C. analista de sistemas
- D. Analista de Sistemas Senior

83) Durante una auditoría de sistemas, el rendimiento del sistema se compara con

- A. sistemas similares
- B. sistemas más nuevos
- C. las especificaciones de diseño
- D. sistemas en competencia

84) El resultado final de la fase es un sistema de software completamente desarrollado y probado, junto con documentación completa y resultados de pruebas.

- A. análisis del sistema
- B. diseño del sistema
- C. desarrollo del sistema
- D. implementación del sistema

85) Los analistas de sistemas avanzan a través de tres niveles durante su carrera, el comienzo como "analista" se convierte en "Analista de Sistemas Senior" después de haber adquirido la experiencia necesaria y, finalmente, son nombrados como

- A. analista del sistema principal
- B. director de analista de sistemas
- C. jefe analista de sistemas
- D. analista del sistema gerente

- 86) Indique si las siguientes afirmaciones son verdaderas o falsas.
- i) Un gráfico de cuadrícula muestra la relación entre los documentos de entrada y salida.
 - ii) Una tabla de decisiones muestra las diversas reglas que se aplican a una decisión cuando existen ciertas condiciones.
- A. i-True, ii-True
 - B. i-True, ii-False
 - C. i-falso, ii-verdadero
 - D. i-falso, ii-falso
- 87) La fase de implementación del sistema de SDLC incluye las siguientes actividades
- i) Planificación para la implementación
 - ii) Preparar el cronograma de implementación.
 - iii) Adquisición de hardware
 - iv) Instalación de software
- A. i, ii y iv solamente
 - B. ii, iii y iv solamente
 - C. i, ii y iii solamente
 - D. Todas las i, ii, iii y iv
- 88) El primer paso en el análisis preliminar es
- A. comprar suministros
 - B. contratar consultores
 - C. define el problema
 - D. proponer cambios
- 89) lleva a cabo estudios de viabilidad sobre los sistemas existentes y propuestos para determinar la viabilidad económica del procesamiento informático dentro de ellos.
- A. analista
 - B. Analista jefe de sistemas
 - C. analista de sistemas
 - D. Todo lo anterior
- 90) Comprensión de la naturaleza, función e interrelaciones de varios subsistemas involucrados en ...

- C. desarrollo de sistemas
- D. mantenimiento de sistemas

91) ¿Cuál de las siguientes tareas no forma parte de la fase de diseño del sistema?

- A. diseño de sistemas alternativos
- B. seleccionando el mejor sistema
- C. escribiendo un informe de diseño de sistemas
- D. sugiriendo soluciones alternativas

92) La investigación del sistema existente mediante entrevistas, cuestionarios y observaciones es la principal responsabilidad de ...

- A. analista
- B. Analista jefe de sistemas
- C. analista de sistemas
- D. Todo lo anterior

93) evaluar los recursos requeridos y el costo total de preparar e instalar los procedimientos informáticos y los sistemas manuales de soporte.

- A. analista
- B. Analista jefe de sistemas
- C. analista de sistemas
- D. gerente de procesamiento de datos

94) Determinar si los empleados, gerentes y clientes se resistirán a un nuevo sistema propuesto es parte de este estudio de factibilidad

- A. viabilidad técnica
- B. viabilidad económica
- C. viabilidad organizacional
- D. viabilidad operacional

95) Los proyectos del sistema se inician por las siguientes razones.

- i) capacidad
- ii) control
- iii) comunicación
- iv) costo

- A. i, ii y iv solamente
- B. ii, iii y iv solamente
- C. i, ii y iii solamente

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

96) evalúa los niveles de control requeridos y mantiene el enlace con los auditores para garantizar que el diseño satisfaga sus necesidades.

- A. analista
- B. Analista jefe de sistemas
- C. analista de sistemas
- D. gerente de procesamiento de datos

97) El primer paso para implementar un nuevo sistema es determinar el

- A. requisitos de hardware
- B. requisitos de software
- C. tipo de conversión
- D. mejor alternativa

98) compuesto por gerentes clave de varios departamentos de la organización, así como por miembros del grupo de sistemas de información que es responsable de supervisar la revisión de las propuestas de proyectos.

- A. Comité de dirección
- B. Comité de sistemas de información.
- C. Comité del grupo de usuarios
- D. Todo lo anterior

99) La limitación de es que la documentación de cualquier sistema existente nunca está completa y actualizada.

- A. Revisión de la documentación
- B. Observación de la situación.
- C. Realización de entrevistas
- D. Cuestionario de Administración

100) El El enfoque de implementación se divide en partes más pequeñas que se implementan a lo largo del tiempo.

- A. Directo
- B. paralelo
- C. en fase
- D. piloto

101) El enfoque es generalmente favorecido porque los proyectos de sistemas se consideran inversiones de negocios.

- A. Comité de dirección

D. Comité del Grupo de Desarrolladores

Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.
EDICT:_GUIA_MEX Editorial Guías Mexico

RESPUESTAS DE LA SEGUNDA PRÁCTICA DE EXAMEN

1	C
2	B
3	D
4	D
5	B
6	C
7	C
8	A
9	D
10	B
11	D
12	A
13	D
14	A
15	A
16	C
17	C
18	D
19	A
20	A
21	A

22	B
23	C
24	E
25	A
26	C
27	B
28	C
29	D
30	C
31	D
32	D
33	C
34	A
35	B
36	B
37	A
38	B
39	A
40	C
41	D
42	A

43	B
44	A
45	C
46	A
47	D
48	C
49	B
50	D
51	B
52	D
53	B
54	D
55	B
56	C
57	D
58	D
59	B
60	B
61	A
62	D
63	A

64	C
65	C
66	A
67	A
68	D
69	C
70	D
71	B
72	D
73	D
74	D
75	B
76	A
77	B
78	C
79	B
80	D
81	D
82	B
83	C
84	C

85	C
86	A
87	D
88	C
89	B
90	A
91	D
92	C
93	B
94	D
95	D
96	C
97	C
98	A
99	A
100	C
101	A

TERCERA PRÁCTICA DE EXAMEN

1) tiene una limitación inherente al hecho de que el analista nunca podrá observar las complejidades del sistema.

- A. Revisión de la documentación
- B. Observación de la situación.
- C. Realización de entrevistas
- D. Cuestionario de Administración

2) Un modelo modificable construido antes de que se instale el sistema real se llama a (n)

- A. muestra (sample)
- B. ejemplo (example)
- C. plantilla (template)
- D. prototipo (prototype)

3) aprueba o desaprueba los proyectos y establece prioridades, indicando qué proyectos son los más importantes y deben recibir atención inmediata.

- A. Comité de dirección
- B. Comité de sistemas de información.
- C. Comité del grupo de usuarios
- D. Comité del Grupo de Desarrolladores

4) La limitación de es que el administrador de usuarios puede no ser capaz de explicar el problema en detalle.

- A. Revisión de la documentación
- B. Observación de la situación.
- C. Realización de entrevistas
- D. Cuestionario de Administración

5) Indique si las siguientes afirmaciones son verdaderas o falsas.

i) Los problemas o necesidades de información se discuten durante la fase de diseño de sistemas del SDLC.

ii) El primer paso de la fase de diseño de sistemas del SDLC es seleccionar la mejor alternativa.

- A. i-True, ii-True
- B. i-True, ii-False
- C. i-falso, ii-verdadero
- D. i-falso, ii-falso

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

6) Definir las interfaces entre varios programas y diseñar pruebas para verificar sus interfaces es la actividad básica involucrada en ...

- A. investigación preliminar
- B. implementación de sistemas
- C. desarrollo de sistemas
- D. mantenimiento de sistemas

7) ¿Cuál de los siguientes muestra niveles de gestión y líneas formales de autoridad?

- A. organigrama
- B. mesa de decisiones
- C. diagrama piramidal
- D. cuadro de cuadrícula

8) Las entidades externas pueden ser un

- A. fuente de datos de entrada solamente
- B. Fuente de los datos de entrada o destino de los resultados.
- C. destino de los resultados solamente
- D. repositorio de datos

9) Ventajas de los diagramas de flujo del sistema ...

- A. Comunicación efectiva.
- B. análisis efectivo
- C. Grupo de queasier o relaciones
- D. Todo lo anterior

10) Una lista de preguntas utilizadas en el análisis se llama a (n)

- A. organigrama
- B. guía de entrevista
- C. tabla de cuadrícula
- D. lista de verificación

11) Por una entidad externa, queremos decir una

- A. La unidad fuera del sistema que se está diseñando y que puede ser controlada por un analista.
- B. La unidad fuera del sistema cuyo comportamiento es independiente del sistema está diseñado
- C. Una unidad externa al sistema está diseñada.
- D. Una unidad que no es parte de un DFD

12) Se utiliza un diagrama de contexto.

- A. como el primer paso para desarrollar un DFD detallado de un sistema
- B. En el análisis de sistemas de sistemas muy complejos.
- C. como ayuda al diseño del sistema.
- D. como ayuda al programador.

13) ¿Qué tipo de análisis comienza con el "panorama general" y luego lo divide en partes más pequeñas?

- A. financiero
- B. revertir
- C. de arriba a abajo
- D. ejecutivo

14) Un flujo de datos puede

- A. Sólo un almacén de datos
- B. Solo dejar un almacén de datos.
- C. Entrar o salir de un almacén de datos.
- D. Entrar o salir de un almacén de datos pero no ambos

15) DDS significa

- A. Data Data Systems
- B. Data Digital System
- C. Data Dictionary Systems
- D. Digital Data Service

16) ¿Cuál de los siguientes se usa para mostrar las reglas que se aplican a una decisión cuando se aplican una o más condiciones?

- A. diagrama de flujo del sistema
- B. mesa de decisiones
- C. gráfico de cuadrícula
- D. lista de verificación

17) Los datos no pueden fluir entre dos almacenes de datos porque

- A. no está permitido en DFD
- B. un almacén de datos es un repositorio pasivo de datos
- C. los datos pueden corromperse
- D. se fusionarán

18) Un DFD se nivela normalmente como

- A. Es una buena idea en diseño.
- B. Es recomendado por muchos expertos.
- C. Es fácil hacerlo.
- D. Es más fácil leer y comprender un número de DFD más pequeños que un DFD grande

19) ¿Cuál de las siguientes herramientas muestra los datos o el flujo de información dentro de un sistema de información?

- A. gráfico de cuadrícula
- B. mesa de decisiones
- C. diagrama de flujo del sistema
- D. diagrama de flujo de datos

20) permite definir el flujo de datos a través de una organización o una empresa o una serie de tareas que pueden o no representar un procesamiento computarizado.

- A. proceso del sistema
- B. diagrama de flujo del sistema
- C. diseño del sistema
- D. Sistema estructurado

21) es un método tabular para describir la lógica de las decisiones a tomar.

- A. tablas de decisión
- B. árbol de decisión
- C. Método de decisión
- D. Datos de decisión

22) ¿Cuál de los siguientes no se encuentra en un diagrama de flujo de datos?

- A. entidades
- B. proceso
- C. almacenamiento fuera de línea
- D. archivo

23) Un diagrama de contexto

- A. Describe el contexto de un sistema
- B. es un DFD que ofrece una visión general del sistema
- C. es una descripción detallada de un sistema
- D. no se usa para dibujar un DFD detallado

24) CASE significa

- A. Computer analysis and system engineering
- B. Computer-aided software engineering
- C. Computer-aided system engineering
- D. The computer analyzed system engineering

25) Un rectángulo en un DFD representa

- A. un proceso
- B. un almacén de datos
- C. una entidad externa
- D. una unidad de entrada

26) HIPO significa

- A. Hierarchy input process output
- B. Hierarchy input plus output
- C. Hierarchy plus input process output
- D. Hierarchy input-output process

27) ¿Cuál del diagrama muestra interacciones entre objetos?

- A. diagrama de actividad
- B. diagrama de clase
- C. diagrama de secuencia
- D. diagramas de componentes

28) El diseño del sistema consiste en una investigación preliminar y un estudio de factibilidad, una investigación detallada que consiste en ...

- i) búsqueda de hechos
- ii) análisis y evaluación de datos
- iii) estimando los costos y beneficios
- iv) Elaboración de propuesta de sistema.

- A. i, ii y iii solamente
- B. ii, iii y iv solamente
- C. i, iii y iv solamente
- A. Todos i, ii, iii y iv

29) Es / son las ventajas de utilizar el diagrama de flujo del sistema.

- A. la comunicación
- B. Grupo de queasier o relaciones
- C. análisis efectivo
- D. Todo lo anterior

30) Un diagrama de gráfico de estado describe ...

- A. atributos de los objetos
- B. nodos del sistema
- C. operaciones ejecutadas en un hilo
- D. Eventos desencadenados por un objeto.

31) ¿Cuál de las siguientes herramientas no se utiliza para organizar los proyectos del sistema?

- A. diagrama de flujo del sistema
- B. Tablas de decisión
- C. Árboles del sistema
- D. organigrama

32) es un método tabular para describir la lógica de las decisiones a tomar.

- A. tablas de decisión
- B. tablas del sistema
- C. organigrama
- D. tablas lógicas

33) Los modelos de diagrama de secuencia....

- A. El orden en que se construye el diagrama de clase.
- B. La forma en que los objetos se comunican.
- C. la relación entre estados
- D. Los componentes del sistema.

34) proporciona la definición del flujo de datos a través de una organización o una empresa o una serie de tareas que pueden no representar un procesamiento computarizado.

- A. diagrama de flujo del sistema
- B. Tablas de decisión
- C. Árboles del sistema
- D. organigrama

35) Las cuatro partes de las tablas de decisión son ...

- i) condición del talón
- ii) talón de decisión
- iii) condición de entrada
- iv) talón de acción
- v) entrada de acción

- A. i, ii, iii y iv solamente
- B. ii, iii, iv y v solamente
- C. i, iii, iv y v solamente
- D. Sólo i, ii, iv y v

36) El diagrama de actividad ...

- A. se centra en los flujos impulsados por el procesamiento interno
- B. modela los eventos externos simulando un objeto
- C. se enfoca en las transiciones entre estados de un objeto particular
- D. modela la interacción entre objetos

37) El símbolo en el diagrama de flujo del sistema representa una de las principales funciones de procesamiento.

- A. rectángulo
- B. Squire
- C. circulo
- D. triángulo

38) Que consiste en una lista de todas las condiciones que deben tenerse en cuenta.

- A. condición del Stub
- B. Entrada de condición
- C. Stub de acción
- D. Entrada de acción

39) El diagrama de despliegue muestra ...

- A. objetos de un sistema
- B. Distribución de componentes en los nodos de un sistema.
- C. funciones de un sistema
- D. distribución de nodos

40) El símbolo en el diagrama de flujo del sistema indica la salida de la pantalla.

- A. Rectángulo sesgado
- B. Rectángulo redondeado
- C. triángulo
- D. triángulo redondeado

41) son una representación tabular de la combinación de las condiciones que deben satisfacerse para cada acción particular.

- A. Condición talones
- B. Entradas de condición
- C. talones de acción
- D. Entradas de acción

42) El proceso unificado es una metodología de desarrollo de software que es ...

- A. caso de uso impulsado
- B. componente impulsado
- C. relacionado con la programación extrema
- D. ninguno en una sola iteración

43) El diagrama de flujo del sistema consiste principalmente en ...

i) la fuente a partir de la cual se preparan los datos de entrada y el medio o los dispositivos utilizados.
ii) Los pasos de procesamiento o secuencia de operaciones involucradas.

- A. i-True, ii-False
- B. i-falso, ii-verdadero
- C. i-True, ii-True
- D. i-falso, ii-falso

44) indica las acciones apropiadas que deben tomarse, cuando se cumple una condición.

- A. Condición talones
- B. Entradas de condición
- C. talones de acción
- D. Entradas de acción

45) Un diagrama de interacción debe estar asociado con ...

- A. un caso de uso
- B. un diagrama de transición de estado
- C. un diagrama de actividad
- D. una tarjeta CRC

46) Los resultados intermedios y finales preparados y los medios y dispositivos utilizados para su almacenamiento.

- A. diagrama de flujo del sistema
- B. Tablas de decisión
- C. Árboles de decisión
- D. organigrama

47. El Lenguaje de modelado unificado (UML) se ha convertido en un estándar efectivo para el modelado de software. ¿Cuántas notaciones diferentes tiene?

- a) tres
- b) cuatro
- c) Seis
- d) nueve

48. ¿Qué modelo en el modelado de sistemas describe el comportamiento dinámico del sistema?

- a) Modelo de contexto
- b) Modelo de comportamiento
- c) Modelo de datos
- d) Modelo de objeto

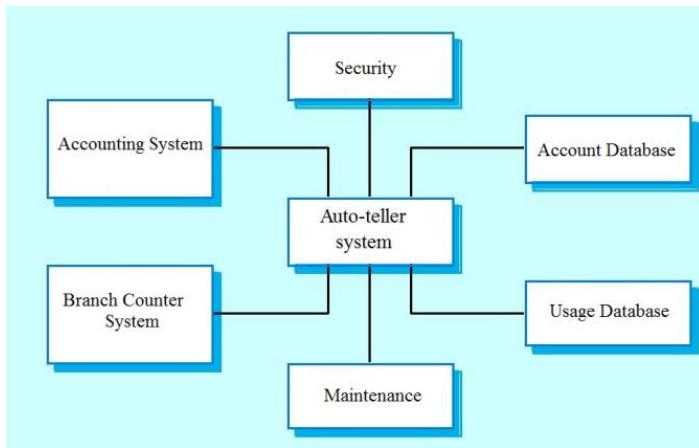
49. ¿Qué modelo en el modelado de sistemas describe la naturaleza estática del sistema?

- a) Modelo de comportamiento
- b) Modelo de contexto
- c) Modelo de datos
- d) Modelo estructural

50. Qué perspectiva en el modelado de sistemas muestra el sistema o la arquitectura de datos.

- a) Perspectiva estructural.
- b) Perspectiva de comportamiento.
- c) perspectiva externa
- d) Todas las mencionadas.

51. ¿Qué modelo de sistema se muestra en las operaciones de cajero automático que se muestran a continuación?



- a) Modelo estructural
- b) modelo de contexto
- c) Modelo de comportamiento.
- d) Modelo de interacción.

52. Los diagramas de actividad se utilizan para modelar el procesamiento de datos.

- a) verdad
- b) falso

53. La ingeniería dirigida por modelos es solo un concepto teórico. No se puede convertir en un código de trabajo / ejecutable.

- a) verdad
- b) falso

54. El UML admite el modelado basado en eventos utilizando diagramas _____.

- a) Despliegue
- b) Colaboración
- c) gráfico de estado
- d) Todas las mencionadas.

55. ¿Cuál de los siguientes diagramas no es compatible con UML considerando el modelado basado en datos?

- a) Actividad
- b) Diagrama de flujo de datos (DFD)
- c) carta del estado
- d) Componente

56. _____ nos permite inferir que diferentes miembros de clases tienen algunas características comunes.

- a) Realización
- b) Agregación
- c) Generalización
- d) dependencia

57. Uno crea modelos de comportamiento de un sistema cuando está discutiendo y diseñando la arquitectura del sistema.

- a) verdad
- b) falso

58. Los diagramas de _____ y _____ de UML representan el modelo de interacción.

- a) Caso de uso, secuencia
- b) Clase, Objeto
- c) Actividad, gráfico de estado
- d) Todas las mencionadas.

59. ¿Qué nivel de Diagrama de relaciones entre entidades (ERD) modela todas las entidades y relaciones?

- a) Nivel 1
- b) Nivel 2
- c) Nivel 3
- d) Nivel 4

60. _____ las clases se usan para crear la interfaz que el usuario ve e interactúa a medida que se usa el software.

- a) controlador
- b) entidad
- c) Límite
- d) negocios

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

61. ¿Cuál de las siguientes afirmaciones es incorrecta con respecto al modelo de Colaborador de responsabilidad de la clase (CRC)?

- a) Todos los escenarios de casos de uso (y los correspondientes diagramas de casos de uso) están organizados en categorías en el modelo CRC
- b) El líder de revisión lee el caso de uso deliberadamente
- c) Solo los desarrolladores en la revisión (del modelo CRC) reciben un subconjunto de las tarjetas de índice del modelo CRC
- d) Todas las mencionadas.

62. Un objeto de datos también puede encapsular procesos y operaciones.

- a) verdad
- b) falso

63. Las dos dimensiones del modelo espiral son

- a) diagonal, angular
- b) radial, perpendicular
- c) radial, angular
- d) diagonal, perpendicular

64. El Modelo Incremental es una combinación de elementos de

- a) Modelo construir y arreglar (Build & FIX) y modelo de cascada
- b) Modelo lineal y modelo RAD
- c) Modelo lineal y modelo de prototipos.
- d) Modelo de cascada y modelo de RAD

65. El modelo preferido para crear aplicaciones cliente / servidor es

- a) Modelo Espiral WINWIN
- b) Modelo en espiral
- c) Modelo concurrente
- d) Modelo incremental

66. Identifique la afirmación correcta con respecto al desarrollo evolutivo:

- a) El desarrollo evolutivo suele tener dos sabores; Desarrollo exploratorio y prototipado desecharable.
- b) Los proyectos muy grandes se realizan generalmente utilizando un enfoque basado en el desarrollo evolutivo
- c) Facilita la gestión fácil de proyectos, a través del alto volumen de documentación que genera.
- d) A veces, la construcción de un prototipo desecharable no es seguida por una reimplementación del sistema de software utilizando un enfoque más estructurado

67. El modelo espiral fue desarrollado por

- a) Victor Bisili
- b) Berry Boehm
- c) Bev Littlewood
- d) Roger Pressman

68. La evolución del software no comprende:

- a) Actividades de desarrollo.
- b) Negociación con el cliente.
- c) Actividades de mantenimiento.
- d) Actividades de reingeniería.

69. Los procesos para la evolución de un producto de software dependen de:

- a) Tipo de software a mantener
- b) Procesos de desarrollo utilizados.
- c) Habilidades y experiencia de las personas involucradas.
- d) Todas las mencionadas.

70. ¿Qué técnica se aplica para garantizar la evolución continua de los sistemas heredados?

- a) ingeniería avanzada
- b) Ingeniería inversa
- c) Reingeniería
- d) Ingeniería inversa y reingeniería

71. La modularización del programa y la traducción del código fuente son actividades de _____

- a) ingeniería avanzada
- b) Ingeniería inversa
- c) Reingeniería
- d) Ingeniería inversa y reingeniería

72. La ingeniería inversa es la última actividad en un proyecto de reingeniería.

- a) verdad
- b) falso

73. El costo de la reingeniería a menudo es significativamente menor que el costo de desarrollar un nuevo software.

- a) verdad
- b) falso

74) ¿Cuál es la característica más importante del modelo en espiral?

- a. Gestión de la calidad
- b. Gestión de riesgos
- c. Gestión del rendimiento
- d. Gestión de la eficiencia

75) El modelo en el que los requisitos se implementan según su categoría es _____.

- a. Modelo de desarrollo evolutivo
- b. Modelo de cascada
- c. Prototipado
- d. Modelo de mejora iterativa

76) ¿Qué software funciona estrictamente de acuerdo con las especificaciones y soluciones definidas?

- a. Tipo estático
- b. Tipo práctico
- c. Tipo incrustado
- d. Ninguna de las anteriores

77) El paradigma de diseño de software es una parte del desarrollo de software e incluye _____.

- a. Diseño, Mantenimiento, Programación.
- b. Codificación, Pruebas, Integración
- c. Recolección de requerimientos, Diseño de software, Programación.
- d. Ninguna de las anteriores

78) Si el proceso del software se basara en conceptos científicos y de ingeniería, sería más fácil recrear un nuevo software que escalar uno existente.

- a. Cierto
- b. Falso

79) ¿Qué aspecto es importante cuando el software se mueve de una plataforma a otra?

- a. Mantenimiento
- b. Operacional
- c. Transicional
- d. Todas las anteriores

80) El software no se considera como una colección de código de programación ejecutable, bibliotecas asociadas y documentación.

- a. Cierto
- b. Falso

81) Las clases se comunican entre sí a través de _____.

- a. Leer sensores
- b. Marcar teléfonos
- c. Mensajes
- d. Ninguna de las anteriores

82) Para el mejor modelo de software adecuado para el proyecto, ¿en cuál de la fase los desarrolladores deciden una hoja de ruta para el plan del proyecto?

- a. Diseño de software
- b. Análisis del sistema
- c. Codificación
- d. Pruebas

83) ¿Qué actividad SDLC inicia el usuario para solicitar un producto de software deseado?

- a. Recolección de requisitos
- b. Implementación
- c. Disposición
- d. Comunicación

84) SDLC no es una secuencia estructurada y bien definida de etapas en ingeniería de software para desarrollar el producto de software previsto.

- a. Ciento
- b. Falso

85) ¿Cuál es la tasa de actividad global efectiva promedio en un sistema de tipo E en evolución que es invariante durante la vida útil del producto?

- a. Autorregulación
- b. Reduciendo la calidad
- c. Sistemas de retroalimentación
- d. Estabilidad organizacional

86) ¿Qué modelo no es adecuado para grandes proyectos de software pero es bueno para aprender y experimentar?

- a. Modelo de Big Bang
- b. Modelo espiral
- c. Modelo iterativo
- d. Modelo de cascada

87) ¿Qué modelo también se conoce como modelo de verificación y validación?

- a. Modelo de cascada
- b. Modelo de Big Bang
- c. Modelo V
- d. Modelo espiral

88) La naturaleza siempre creciente y adaptable del software depende en gran medida del entorno en el que el usuario trabaja en _____.

- a. Costo
- b. Naturaleza dinámica
- c. Gestión de la calidad
- d. Escalabilidad

89) Los modelos SDLC se adoptan según los requisitos del proceso de desarrollo. Puede variar de software a software para garantizar qué modelo es el adecuado.

- a. Cierto
- b. Falso

90) ¿Cuál de las siguientes es la comprensión previa de las limitaciones del producto de software, los problemas relacionados con el sistema de aprendizaje o los cambios que se deben realizar en los sistemas existentes de antemano, identificando y abordando el impacto del proyecto en la organización y el personal, etc.?

- a. Diseño de software
- b. Estudio de factibilidad
- c. Recolección de requisitos

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

91) ¿Cuál es el modelo más simple de paradigma de desarrollo de software?

- a. Modelo espiral
- b. Modelo de Big Bang
- c. Modelo V
- d. Modelo de cascada

92) Las fases del proceso de revisión formal se mencionan a continuación. Colóquelos en el orden correcto.

- i. Planificación
- ii. Reunión de revisión
- iii. Rehacer
- iv. Preparaciones individuales
- v. patada
- vi. Seguir

- a. i, ii, iii, iv, v, vi
- b. vi, i, ii, iii, iv, v
- c. i, v, iv, ii, iii, vi
- d. i, ii, iii, v, iv, vi

93) Cada métrica debe validarse empíricamente en una amplia variedad de contextos antes de publicarse y que se utilizan para tomar decisiones.

- a. Cierto
- b. Falso

94) ¿Qué se ha probado para descubrir errores que indiquen una falta de conformidad con los requisitos del cliente en las dimensiones de la calidad?

- a. Estructura
- b. Función
- c. Usabilidad
- d. Navegación

95) La evaluación de las métricas que dan como resultado la percepción y la calidad de la representación es _____.

- a. Análisis

d. Realimentación

96) ¿Cuál es una agrupación lógica de datos que reside dentro del límite de la aplicación y se mantiene a través de entradas externas?

- a. Número de archivos de interfaz externos
- b. Número de archivos lógicos internos
- c. Número de consultas externas.
- d. Número de entradas externas

97) ¿Cuál es el subsistema de publicación que no requiere ningún procesamiento adicional y se transmite directamente al lado del cliente?

- a. Elementos estáticos
- b. Servicios de publicacion
- c. Servicios externos
- d. Ninguna de las anteriores

98) Object Constraint Language (OCL) es una notación formal desarrollada para que los usuarios de UML puedan agregar más precisión a sus especificaciones.

- a. Cierto
- b. Falso

99) ¿Qué clase proporciona un cambio de contenido o función que corrige un error o mejora el contenido o la funcionalidad local en la gestión de cambios?

- a. Clase 1
- b. Clase 2
- c. Clase 3
- d. Clase 4

100) ¿Qué modelo proporciona la confiabilidad general del sistema que se proyecta y certifica?

- a. Modelo de muestreo
- b. Modelo de componente
- c. Modelo de certificación
- d. Ambos A y B

RESPUESTAS DE LA TERCERA PRÁCTICA DE EXAMEN

1	B
2	D
3	B
4	C
5	D
6	C
7	A
8	B
9	D
10	D
11	C
12	A
13	C
14	C
15	C
16	B
17	D
18	D
19	D
20	B

21	A
22	C
23	B
24	B
25	C
26	A
27	C
28	A
29	D
30	D
31	C
32	A
33	B
34	A
35	C
36	A
37	A
38	A
39	B
40	A

41	B
42	A
43	C
44	D
45	D
46	A
47	D
48	B
49	D
50	A
51	B
52	A
53	B
54	C
55	B
56	C
57	B
58	A
59	B
60	C

61	C
62	B
63	C
64	C
65	C
66	A
67	B
68	B
69	D
70	D
71	C
72	B
73	A
74	B
75	A
76	A
77	A
78	B
79	C
80	B

81	C
82	B
83	D
84	B
85	D
86	A
87	C
88	B
89	B
90	D
91	D
92	C
93	B
94	B
95	C
96	B
97	A
98	A
99	A
100	C

CUARTA PRÁCTICA DE EXAMEN

- 1) Abrevia el término ILFs.
- a. Interface logical files
 - b. Internal logical files
 - c. Input logical files
 - d. Internal logical function
- 2) En el subsistema de administración, ¿cuáles son los elementos funcionales y el flujo de trabajo asociado que admitirán la identificación del objeto de contenido, el control de versiones, la administración de cambios, la auditoría de cambios y los informes?
- a. Base de datos de contenido
 - b. Capacidades de base de datos
 - c. Función de gestión de configuración
 - d. Todo lo mencionado anteriormente
- 3) ¿Cuál es la prueba para garantizar que la aplicación web se interconecte correctamente con otras aplicaciones o bases de datos?
- a. Compatibilidad
 - b. Interoperabilidad
 - c. Actuación
 - d. Seguridad
- 4) Abrevia el término BSS.
- a. Box Structure Specification
 - b. Box Statistical Specification
 - c. Box Statistical System
 - d. Box Structure Sampling
- 5) Qué subsistema implementa un repositorio que abarca los siguientes elementos,
- 1) Base de datos de contenido
 - 2) Capacidades de la base de datos
 - 3) Funciones de gestión de configuración
- a. El subsistema de publicación.
 - b. El subsistema de gestión.
 - c. El subsistema de recogida.
 - d. Ninguna de las anteriores

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

6) ¿Qué condición define las circunstancias para que una operación en particular sea válida?

- a. Postcondición
- b. Condición previa
- c. Invariante
- d. Ninguna de las anteriores

7) Las herramientas CASE son conjuntos de programas de aplicación de software automatizados, que no se utilizan para apoyar, acelerar y suavizar las actividades de SDLC.

- a. Sí
- b. No

8) ¿Qué herramientas se utilizan en la implementación, prueba y mantenimiento?

- a. Herramientas de mayúsculas
- b. Herramientas de caja integradas
- c. Herramientas de minúsculas
- d. Ninguna de las anteriores

9) ¿Qué caja especifica el comportamiento de un sistema o una parte de un sistema?

- a. Caja de estado
- b. Caja clara
- c. Caja negra
- d. Ninguna de las anteriores

10) Dar las desventajas de la modularización.

- a. Los componentes más pequeños son más fáciles de mantener
- b. El programa se puede dividir en función de aspectos funcionales.
- c. El nivel deseado de abstracción se puede traer en el programa
- d. Ninguna de las anteriores

11) ¿Qué proyecto se emprende como consecuencia de una solicitud específica de un cliente?

- a. Proyectos de desarrollo de conceptos
- b. Proyectos de mejora de aplicaciones
- c. Nuevos proyectos de desarrollo de aplicaciones.
- d. Proyectos de mantenimiento de aplicaciones

12. Un conjunto de actividades que aseguran que el software implementa correctamente una función específica.

- a) verificación
- b) pruebas
- c) implementación
- d) validación

13. La validación está basada en computadora.

- a) verdad
- b) falso

14. _____ se realiza en la fase de desarrollo por los depuradores.

- a) Codificación
- b) pruebas
- c) depuración
- d) Implementación

15. La localización o identificación de los errores se conoce como _____

- a) diseño
- b) pruebas
- c) depuración
- d) codificación

16. ¿Cuál define el rol del software?

- a) diseño del sistema
- b) diseño
- c) Ingeniería de sistemas
- d) Implementación

17. ¿A qué se llama prueba de componentes individuales?

- a) prueba del sistema
- b) pruebas unitarias
- c) pruebas de validación
- d) prueba de caja negra

18. Una estrategia de prueba que prueba la aplicación en su conjunto.

- a) Reunión de requisitos
- b) Pruebas de verificación
- c) Pruebas de validación.
- d) Pruebas del sistema

19. Una estrategia de prueba que prueba la aplicación en el contexto de un sistema completo.

- a) sistema
- b) Validación
- c) Unidad
- d) caja gris

20. Se prueba un _____ para asegurar que la información fluye adecuadamente dentro y fuera del sistema.

- a) interfaz del módulo
- b) estructura de datos local
- c) condiciones de contorno
- d) rutas de acceso

21. Una prueba realizada en el sitio del desarrollador bajo pruebas de validación.

- a) alfa
- b) gamma
- c) lambda
- d) unidad

22) Amenaza es la probabilidad de que pueda atacar a un tipo específico y también ocurre dentro de un tiempo determinado.

- a. Sí
- b. No

23) ¿Un plan de gestión de riesgos efectivo deberá abordar cuál de los siguientes problemas?

- a. Evitación de riesgo
- b. Monitoreo de riesgos
- c. Planificación de contingencias
- d. Todo lo mencionado anteriormente

24) ¿Cuál de estas actividades no es una de las actividades recomendadas para ser realizadas por un grupo independiente de SQA?

- a. Servir como el único equipo de prueba para cualquier software producido.
- b. Las herramientas y métodos que soportan las acciones y tareas de SQA.
- c. Procedimientos de gestión de la configuración del software.
- d. Roles organizacionales y responsabilidades relativas a la calidad del producto.

25) ¿Cuáles son las características del riesgo del Software?

- a. Incertidumbre
- b. Pérdida
- c. Ambos A y B
- d. Ninguna de las anteriores

26) ¿Cuál es el grado en que el software realiza su función requerida?

- a. Exactitud
- b. Claridad
- c. Lo completo (Integridad)
- d. Consistencia

27) El análisis de peligros se enfoca en la identificación y evaluación de peligros potenciales que pueden causar _____.

- a. Problemas externos
- b. Problemas internos
- c. Ambos A y B
- d. Ninguna de las anteriores

28) El número total de medidas distintas de ocurrencia de operadores y operandos se utiliza en _____.

- a. Teoria de lawrence
- b. La teoria de halstead
- c. Kyburg, H. E.
- d. Jech, t.

29) El presupuesto no es una sección en los planes estándar de SQA que se recomiendan en IEEE.

- a. Cierto
- b. Falso

30) Construir un producto o sistema excelente para que nadie realmente quiera un riesgo es un _____.

- a. Riesgo tecnico
- b. Riesgo de programa
- c. Riesgo del negocio
- d. Riesgo de rendimiento

31) ¿Qué riesgos identifican los posibles problemas de diseño, implementación, interfaz, verificación y mantenimiento?

- a. Riesgo del proyecto
- b. Riesgo del negocio
- c. Riesgo tecnico
- d. Riesgo de programa

32) ¿Cuál es la secuencia detallada de pasos que describe la interacción entre el usuario y la aplicación?

- a. Guiones de escenarios
- b. Clases de apoyo
- c. Clases claves
- d. Subsistemas

33) Las métricas de cohesión y las métricas de acoplamiento son métricas en qué nivel de diseño?

- a. Diseño de interfaz de usuario
- b. Diseño basado en patrones
- c. Diseño de la arquitectura
- d. Diseño a nivel de componente

34) ¿Cuál es el nivel de proyecto y proceso que proporciona el beneficio de Métrica de calidad?

- a. Amplificación de defectos
- b. Eficacia de eliminación de defectos
- c. Calidad de medición
- d. Todo lo mencionado anteriormente

35) De lo siguiente se dan tres categorías principales de riesgo,

- 1) programa (Schedule) de riesgo
- 2) Riesgo de proyecto
- 3) riesgo técnico
- 4) Riesgo de negocio

- a. 1,2 y 3
- b. 2,3 y 4
- c. 1,2 y 4
- d. 1,3 y 4

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

36) ¿Cuál es la medida más común para la corrección?

- a. Defectos por KLOC
- b. Errores por KLOC
- c. \$ por KLOC
- d. Páginas de documentación por KLOC.

37) ¿Qué métricas se derivan al normalizar las medidas de calidad y / o productividad considerando el tamaño del software que se ha producido?

- a. Tamaño orientado
- b. Orientado a funciones
- c. Orientado a objetos
- d. Uso-caso-orientado

38) ¿Qué modelo de amplificación de defectos se utiliza para ilustrar la generación y detección de errores durante los pasos preliminares de un proceso de ingeniería de software?

- a. Diseño
- b. Diseño detallado
- c. Codificación
- d. Todo lo mencionado anteriormente

39) Una de las técnicas de prueba de la base de falla es _____.

- a. Examen de la unidad
- b. Prueba beta
- c. Pruebas de estrés
- d. Pruebas de mutación

40) La modificación del software para que coincida con los cambios en el entorno siempre cambiante se denomina _____.

- a. Mantenimiento adaptativo
- b. Mantenimiento correctivo
- c. Mantenimiento perfecto
- d. Mantenimiento preventivo

41) Los cambios que se realizan en el sistema para reducir las futuras fallas de falla del sistema se llaman _____.

- a. Mantenimiento preventivo
- b. Mantenimiento adaptativo
- c. Mantenimiento correctivo
- d. Mantenimiento perfecto

42) El mantenimiento preventivo está implementando cambios en los requisitos nuevos o existentes del usuario.

- a. Cierto
- b. Falso

43) El conocimiento del programa de software, diseño y estructura es esencial en _____.

- a. Prueba de caja negra
- b. Prueba de caja blanca
- c. Pruebas de integración
- d. Ninguna de las anteriores

44) ¿Qué es el depósito de componentes de software que los diseñadores refieren para buscar el componente coincidente en función de la funcionalidad y los requisitos de software previstos?

- a. Buscar componentes adecuados
- b. Especificar componentes
- c. Especificación de requerimiento
- d. Incorporar componentes

45) Si los cambios en los costos de mantenimiento a menudo se dejan sin documentar, lo que puede causar más conflictos en el software futuro.

- a. Cierto
- b. Falso

46) El tamaño del sistema es una métrica para el modelo de análisis.

- a. Cierto
- b. Falso

47) En el proceso de Reingeniería, ¿qué conceptos para obtener un software rediseñado?

- a. Aplicar ingeniería avanzada
- b. Realizar
- c. Decidir
- d. Programa de reestructuracion

48) En el mantenimiento del software, la eliminación de errores detectados por los usuarios se conoce como _____.

- a. Adaptado
- b. Correctivo
- c. Perfecto
- d. Preventivo

49) Los nuevos módulos, que deben ser reemplazados o modificados, y también están diseñados contra las especificaciones de requisitos establecidas en la etapa anterior son _____.

- a. Test de aceptación
- b. Pruebas del sistema
- c. Entrega
- d. Diseño

50) A partir de lo siguiente, ¿qué método se adoptará en el proceso de reutilización?

- a. Ya sea manteniendo los mismos requisitos y ajustando los componentes.
- b. Manteniendo los mismos componentes y modificando los requisitos.
- c. Ambos A y B
- d. Ninguna de las anteriores

51) ¿Qué calidad se ocupa del mantenimiento de la calidad del producto de software?

- a. Seguro de calidad
- b. Control de calidad
- c. Eficiencia de calidad
- d. Ninguna de las anteriores

52) La fuerza de explosión, el retroceso y la eliminación de causas son estrategias utilizadas en el arte de la depuración.

- a. Sí
- b. No

53) Los componentes de software proporcionan interfaces, que pueden utilizarse para establecer la comunicación entre diferentes componentes.

- a. Sí
- b. No

54) ¿Qué nivel de subsistema se usa de una aplicación?

- a. Nivel de aplicación
- b. Nivel de componente
- c. Nivel de modulos
- d. Ninguna de las anteriores

55) ¿Qué incluye las modificaciones y actualizaciones realizadas para corregir o solucionar los problemas, que son descubiertos por el usuario o concluidos por los informes de errores del usuario?

- a. Mantenimiento perfecto
- b. Mantenimiento adaptativo
- c. Mantenimiento correctivo
- d. Mantenimiento preventivo

56) La gestión de la configuración es una parte esencial del mantenimiento del sistema. Se facilita con herramientas de control de versiones para controlar versiones, semi-versiones o administración de parches.

- a. Cierto
- b. Falso

57) Cuando el cliente puede solicitar nuevas características o funciones en el software, ¿qué significa en el mantenimiento del software?

- a. Modificaciones de host
- b. Requerimientos del cliente
- c. Condiciones de mercado
- d. Cambios organizativos

58) ¿Qué factores finales del software afectan el costo de mantenimiento?

- a. Estructura del programa de software
- b. Lenguaje de programación
- c. Dependencia del entorno externo.
- d. Todo lo mencionado anteriormente
- e. Ninguna de las anteriores

59) Dar los factores del mundo real que afectan el costo de mantenimiento.

- a. A medida que la tecnología avanza, se vuelve costoso mantener el software antiguo.
- b. La edad estándar de cualquier software se considera hasta 10 a 15 años.
- c. La mayoría de los ingenieros de mantenimiento son novatos y utilizan el método de prueba y error para corregir el problema.
- d. Todo lo mencionado anteriormente

60) IEEE 830-1993 es un estándar recomendado por IEEE para _____.

- a. Especificación de requisitos de software
- b. Diseño de software
- c. Pruebas
- d. Tanto a como B)

61) ¿Cuál no es un paso de Ingeniería de Requisitos?

- a. Elicitación de requisitos
- b. Análisis de requerimientos
- c. Diseño de requerimientos
- d. Documentación de requerimientos

62) SRD significa _____.

- a. Definición de requisitos de software
- b. Definición de requisitos estructurados
- c. Diagrama de requisitos de software
- d. Diagrama de Requerimientos Estructurados

63) El modelado es una representación de las clases orientadas a objetos y las colaboraciones resultantes permitirán que un sistema funcione.

- a. Cierto
- b. Falso

64) Los requisitos pueden ser verificados en las siguientes condiciones.

- 1) Si no se pueden implementar en la práctica.
- 2) Si no son válidos y según la funcionalidad y dominio del software.
- 3) Si no hay ambigüedades.

- a. Cierto
- b. Falso

65) En el análisis de requisitos, ¿qué modelo representa el dominio de información para el problema?

- a. Modelos de datos
- b. Modelos orientados a la clase
- c. Modelos basados en escenarios
- d. Modelos orientados al flujo

66) En el análisis de requisitos, ¿qué modelo describe cómo se comporta el software como consecuencia de eventos externos?

- a. Modelos orientados a la clase
- b. Modelos basados en escenarios
- c. Modelos orientados al flujo
- d. Modelos de comportamiento

67) Los requisitos pueden recopilarse de los usuarios a través de entrevistas, encuestas, análisis de tareas, lluvia de ideas, análisis de dominios, creación de prototipos, estudio de la versión de software utilizable existente y por observación.

- a. Cierto
- b. Falso

68) El proceso para reunir los requisitos de software del Cliente, Analizar y Documentar se conoce como _____.

- a. Proceso de ingeniería de requerimientos
- b. Proceso de elicitation de requisitos.
- c. Requisitos de interfaz de usuario
- d. Analista de sistemas de software

69) Abrevia el término SRS.

- a. Especificación de requisitos de software
- b. Solución de refinamiento de software
- c. Fuente de recursos de software
- d. Ninguna de las anteriores

70) ¿Cuál de estos objetivos primarios deben alcanzarse para el modelo de requisitos?

- a. Describir lo que requiere el cliente.
- b. Establecer una base para la creación de un diseño de software.
- c. Para definir un conjunto de requisitos que pueden ser validados una vez que el software
- d. Todo lo mencionado anteriormente

71) Las entrevistas, que se llevan a cabo entre dos personas en la mesa son _____.

- a. Escrito
- b. No estructurado
- c. Grupo
- d. Uno a uno

72) ¿Los analistas del sistema tienen cuáles de estas responsabilidades siguientes?

- a. Análisis y comprensión de los requisitos del software previsto.
- b. Entendiendo cómo el proyecto contribuirá en los objetivos de la organización.
- c. Identificar fuentes de requerimiento
- d. Todo lo mencionado anteriormente

73) La especificación de requisitos de software debe tener las siguientes características:

- 1) Los requisitos del usuario se expresan en lenguaje natural.
- 2) Los requisitos técnicos se expresan en lenguaje estructurado, que se utiliza dentro de la organización.
- 3) La descripción del diseño debe estar escrita en pseudo código.

- a. Cierto
- b. Falso

74) El proceso para reunir los requisitos de software del Cliente, Analizar y Documentar se conoce como _____.

- a. Proceso de ingeniería de requerimientos
- b. Proceso de obtención (elicitation) de requisitos.
- c. Requisitos de interfaz de usuario
- d. Analista de sistemas de software

75) En la validación de software, los requisitos pueden verificarse en las siguientes condiciones:

- 1) Si se pueden implementar en la práctica.
- 2) Si son válidos y según la funcionalidad y dominio del software.
- 3) Si hay alguna ambigüedad.
- 4) Si están completos

- a. Cierto

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

76) SRS es un documento creado por un analista del sistema después de que se recopilan los requisitos de varias partes interesadas.

- a. Sí
- b. No

77) ¿Qué sistema basado en computadora puede tener un efecto profundo en el diseño que se elija y también se aplicará el enfoque de implementación?

- a. Elementos basados en escenarios
- b. Elementos basados en la clase
- c. Elementos de comportamiento
- d. Elementos orientados al flujo.

78) ¿Cuál está enfocado hacia el objetivo de la organización?

- a. Estudio de factibilidad
- b. Recolección de requisitos
- c. Especificación de requisitos de software
- d. Validación de requisitos de software

79) ¿Qué documento es creado por el analista del sistema después de que se recopilan los requisitos de Varios interesados?

- a. Especificación de requisitos de software
- b. Validación de requisitos de software
- c. Estudio de factibilidad
- d. Recolección de requisitos

80) ¿En qué proceso de obtención los desarrolladores discuten con el cliente y los usuarios finales y conocen sus expectativas del software?

- a. Recolección de requisitos
- b. Requisitos de organizacion
- c. Negociación y discusión
- d. Documentación

81) ¿El proceso de ingeniería de requisitos incluye cuál de estos pasos?

- a. Estudio de factibilidad
- b. Recolección de requisitos
- c. Requisitos de software especificación y validación
- d. Todo lo mencionado anteriormente

82. ¿Cuál de los siguientes términos describe las pruebas?

- a) Encontrando código roto
- b) Evaluar entregable para encontrar errores.
- c) Una etapa de todos los proyectos.
- d) Ninguna de las mencionadas

83. ¿Qué es la complejidad ciclomática?

- a) Prueba de caja negra
- b) Prueba de caja blanca
- c) Prueba de caja amarilla
- d) Prueba de caja verde

84. ¿Límites inferior y superior están presentes en qué gráfico?

- a) Ejecutar gráfico
- b) gráfico de barras
- c) gráfico de control
- d) Ninguna de las mencionadas

85. ¿Las pruebas de mantenimiento se realizan usando la metodología?

- a) Repetir
- b) Pruebas de cordura
- c) Ensayo de amplitud y ensayo de profundidad.
- d) Pruebas de confirmación

86. Las técnicas de caja blanca también se clasifican como

- a) Pruebas basadas en diseño
- b) Ensayos estructurales.
- c) Error al suponer la técnica.
- d) Ninguna de las mencionadas

87. Las pruebas exhaustivas son

- a) siempre posible
- b) prácticamente posible
- c) poco práctico pero posible
- d) impracticable e imposible

88. ¿Cuál de las siguientes es / son la técnica de la caja blanca?

- a) Prueba de declaración
- b) Prueba de decisión
- c) cobertura de la condición
- d) Todas las mencionadas.

89. ¿Cuáles son los diferentes niveles de prueba?

- a) Pruebas unitarias
- b) Pruebas del sistema
- c) Pruebas de integración
- d) Todas las mencionadas.

90. ¿El análisis del valor de la frontera pertenece a?

- a) Prueba de caja blanca
- b) Prueba de caja negra
- c) Prueba de caja blanca y caja negra
- d) Ninguna de las mencionadas

91. La prueba alfa se realiza en

- a) Fin del desarrollador.
- b) Fin del usuario.
- c) Fin del desarrollador y usuario
- d) Ninguna de las mencionadas

92. Las pruebas en las que se verifica el código.

- a) Prueba de caja negra
- b) Prueba de caja blanca
- c) Prueba de caja roja
- d) Prueba de caja verde

93. Pruebas realizadas sin planificación y documentación.

- a) Pruebas unitarias
- b) Pruebas de regresión
- c) Pruebas Adhoc
- d) Ninguna de las mencionadas

94. Las pruebas de aceptación también se conocen como

- a) Prueba de caja gris
- b) Prueba de caja blanca
- c) Prueba alfa
- d) prueba beta

95. ¿Cuál de las siguientes es una prueba no funcional?

- a) Prueba de caja negra
- b) pruebas de rendimiento
- c) Pruebas unitarias
- d) Ninguna de las mencionadas

96. La prueba beta se realiza en

- a) Fin del usuario.
- b) Fin del desarrollador.
- c) Fin de usuario y desarrollador.
- d) Ninguna de las mencionadas

97. SPICE significa

- a) Mejora de procesos de software y determinación de compatibilidad
- b) Mejora de procesos de software y determinación de control
- c) Mejora de procesos de software y determinación de capacidad
- d) Ninguna de las mencionadas

98. La prueba unitaria se realiza por

- a) Usuarios
- b) Desarrolladores
- c) clientes
- d) Ninguna de las mencionadas

99. La prueba de comportamiento es

- a) Prueba de caja blanca
- b) Prueba de caja negra
- c) Prueba de caja gris
- d) Ninguna de las mencionadas

100. ¿Cuál de las siguientes es la prueba de caja negra

- a) Prueba de ruta básica
- b) Análisis del valor límite
- c) Análisis de ruta de código
- d) Ninguna de las mencionadas

101. ¿Cuál de los siguientes no se usa para medir el tamaño del software?

- a) KLOC
- b) Puntos de función

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

RESPUESTAS DE LA CUARTA PRÁCTICA DE EXAMEN

1	B
2	C
3	B
4	A
5	B
6	B
7	B
8	C
9	C
10	D
11	C
12	A
13	A
14	C
15	B
16	C
17	B
18	D
19	A
20	A
21	A

22	SI
23	D
24	A
25	C
26	A.
27	A
28	B
29	A
30	C
31	C
32	A
33	D
34	B
35	B
36	A
37	A
38	D
39	D
40	A
41	A
42	B

43	B
44	A
45	A
46	A
47	A
48	B
49	D
50	C
51	B
52	A
53	A
54	B
55	C
56	A
57	B
58	D
59	D
60	A
61	C
62	B
63	A

64	B
65	A
66	D
67	A
68	A
69	A
70	D
71	D
72	D
73	A
74	A
75	A
76	A
77	C
78	A
79	A
80	A
81	D
82	B
83	B
84	A

85	C
86	B
87	C
88	D
89	D
90	B
91	A
92	B
93	C
94	D
95	B
96	A
97	C
98	B
99	B
100	B
101	C

QUINTA PRÁCTICA DE EXAMEN

1. La depuración de software es un conjunto de actividades que se pueden planificar por adelantado y realizar de manera sistemática.

- a) verdad
- b) falso

2. ¿Cuál de los siguientes no es un software que prueba características genéricas?

- a) Diferentes técnicas de prueba son apropiadas en diferentes puntos en el tiempo
- b) La prueba la realiza el desarrollador del software o un grupo de prueba independiente
- c) Las pruebas y la depuración son actividades diferentes, pero la depuración debe incluirse en cualquier estrategia de prueba.
- d) Ninguna de las mencionadas

3. ITG significa

- a) grupo de prueba instantáneo
- b) grupo de pruebas de integración
- c) grupo de prueba individual
- d) grupo de prueba independiente

4. Al recopilar _____ durante las pruebas de software, es posible desarrollar pautas significativas para detener el proceso de prueba.

- a) Intensidad de falla
- b) tiempo de prueba
- c) Métricas
- d) Todas las mencionadas.

5. ¿Cuál de los siguientes problemas debe abordarse para implementar una estrategia de prueba de software exitosa?

- a) Utilizar revisiones técnicas formales efectivas como filtro antes de la prueba
- b) Desarrollar un plan de pruebas que haga hincapié en las "pruebas rápidas de ciclo".
- c) Objetivos de las pruebas estatales explícitamente
- d) Todas las mencionadas.

6. Los casos de prueba deberían descubrir errores como

- a) Terminación de bucle inexistente
- b) Comparación de diferentes tipos de datos
- c) Operadores lógicos incorrectos o precedencia
- d) Todas las mencionadas.

7. ¿Cuál de los siguientes errores no se debe probar cuando se evalúa el manejo de errores?

- a) La descripción del error es ininteligible.
- b) El error señalado no corresponde al error encontrado
- c) La condición de error causa la intervención del sistema antes del manejo de errores
- d) La descripción del error proporciona información suficiente para ayudar en la ubicación de la causa del error

8. Lo que normalmente se considera como un complemento de la etapa de codificación

- a) Pruebas de integración
- b) pruebas unitarias
- c) Finalización de Pruebas
- d) Pruebas de regresión

9. ¿Cuál de los siguientes no es un caso de prueba de regresión?

- a) Una muestra representativa de pruebas que ejercerán todas las funciones del software.
- b) Pruebas adicionales que se centran en funciones de software que pueden verse afectadas por el cambio
- c) Pruebas que se centran en los componentes de software que se han cambiado
- d) Los componentes de bajo nivel se combinan en grupos que realizan una subfunción de software específica

10. ¿Qué prueba es un enfoque de prueba de integración que se usa comúnmente cuando se están desarrollando productos de software de "envoltura retráctil"?

- a) Pruebas de regresión
- b) Pruebas de integración
- c) pruebas de humo
- d) Pruebas de validación

11. La arquitectura del software orientado a objetos da como resultado una serie de subsistemas en capas que encapsulan clases colaboradoras.

- a) verdad
- b) falso

12. La construcción de software orientado a objetos comienza con la creación de

- a) modelo de diseño
- b) modelo de análisis
- c) niveles de código
- d) Modelo de diseño y análisis.

13. ¿Qué pruebas integran el conjunto de clases requeridas para responder a una entrada o evento para el sistema?

- a) pruebas de grupo (cluster)
- b) pruebas basadas en hilos
- c) pruebas basadas en el uso
- d) ninguno de los mencionados

14. ¿Cuál de los siguientes es uno de los pasos en las pruebas de integración del software OO?

- a) pruebas de grupo
- b) pruebas basadas en hilos
- c) pruebas basadas en el uso
- d) ninguno de los mencionados

15. Se pueden usar métodos _____ para conducir pruebas de validación

- a) Prueba de caja amarilla
- b) Prueba de caja negra
- c) Prueba de caja blanca
- d) Todas las mencionadas.

16. ¿Cuál de las siguientes opciones forma parte del código OO de prueba?

- a) Pruebas de validación.
- b) Pruebas de integración.
- c) pruebas de clase
- d) Pruebas del sistema.

17. El objetivo de _____ dentro de un sistema OO es diseñar pruebas que tengan una alta probabilidad de descubrir errores plausibles.

- a) Pruebas basadas en fallas
- b) Pruebas de integración
- c) Pruebas basadas en el uso
- d) Pruebas basadas en escenarios

18. ¿Qué se refiere a la estructura observable externamente de un programa OO?

- a) Estructura profunda
- b) estructura de la superficie
- c) Estructura del núcleo
- d) Todas las mencionadas.

19. _____ clasifica las operaciones de clase en función de la función genérica que realiza cada uno.

- a) Particionamiento basado en categorías
- b) Particionamiento basado en atributos
- c) Partición basada en el estado
- d) Ninguna de las mencionadas

20. ¿Cuál de los siguientes está orientado a la caja negra y se puede lograr aplicando los mismos métodos de caja negra discutidos para el software convencional?

- a) Pruebas convencionales
- b) Pruebas de validación del sistema OO
- c) Diseño de caso de prueba
- d) Pruebas convencionales y pruebas de validación del sistema OO

21. ¿En cuál de las siguientes estrategias de prueba, una unidad comprobable más pequeña es la clase u objeto encapsulado?

- a) Pruebas unitarias
- b) Pruebas de integración
- c) Pruebas del sistema
- d) Ninguna de las mencionadas

22. ¿Cuál de los siguientes tipos de prueba no es parte de la prueba del sistema?

- a) Pruebas de recuperación
- b) pruebas de estrés
- c) Pruebas del sistema
- d) Pruebas aleatorias

23. ¿Cuál es el primer objetivo del proceso de prueba?

- a) Prevención de errores
- b) pruebas
- c) Ejecución
- d) análisis

24. Los errores de software durante la codificación se conocen como

- a) errores
- b) fallas
- c) bichos (bugs)
- d) defectos

25. Nombre una técnica de evaluación para evaluar la calidad de los casos de prueba.

- a) Análisis de mutaciones.
- b) Validación
- c) Verificación
- d) Análisis de rendimiento

26. La prueba debe realizarse para cada posible

- a) datos
- b) caso
- c) variable
- d) todas las mencionadas

27. ¿Cuál de los siguientes no forma parte del informe de errores?

- a) Caso de prueba
- b) Salida
- c) Versión de software
- d) LOC

28. ¿Cuál de las siguientes opciones no forma parte del Flujo de ejecución durante la depuración?

- a) Paso sobre
- b) entrar en
- c) intensificar
- d) Salir

29. El método de Complejidad ciclomática viene bajo el método de prueba.

- a) caja amarilla
- b) caja blanca
- c) caja gris
- d) caja negra

30. ¿Cuál es una técnica de prueba de caja negra apropiada para todos los niveles de prueba?

- a) Pruebas de aceptación
- b) Pruebas de regresión
- c) partición de equivalencia
- d) Aseguramiento de la calidad

31. ¿Cuál de las siguientes es la manera de asegurar que las pruebas estén realmente probando el código?

- a) Pruebas de estructura de control
- b) Prueba de trayectoria compleja
- c) cobertura del código
- d) Aseguramiento de la calidad del software.

32. Las pruebas efectivas reducirán el costo de _____.

- a) mantenimiento
- b) diseño
- c) codificación
- d) documentación

33. ¿Cuál de los siguientes es un problema común?

- a) Errores de intercambio de datos
- b) Accediendo a elementos de datos del tipo equivocado.
- c) Intentar usar áreas de memoria después de liberarlas.
- d) Todas las mencionadas.

34. Standard Enforcer es un

- a) Herramienta de prueba estática
- b) Pruebas dinámicas
- c) Pruebas estáticas y dinámicas
- d) Ninguna de las mencionadas

35. Muchas aplicaciones que usan análisis estático encuentran 0.1-0.2% de NCSS. NCSS significa

- a) Declaración de fuente no codificada
- b) Oraciones de fuente sin comentarios
- c) Declaración de fuente sin comentarios
- d) Todas las mencionadas.

36. ¿Qué herramienta de prueba hace un trabajo simple de hacer cumplir los estándares de manera uniforme en muchos programas?

- a) Analizador estático
- b) inspector de código
- c) Enforcer estándar
- d) Ambos, el inspector de código y el ejecutor estándar

37. Las pruebas de software con datos reales en un entorno real se conocen como

- a) prueba alfa
- b) pruebas beta
- c) pruebas de regresión
- d) ninguno de los mencionados

38. ¿Cuál de las siguientes herramientas de prueba examina el programa de forma sistemática y automática?

- a) inspector de código
- b) analizador estático
- c) Enforcer estándar
- d) analizador de cobertura

39. ¿Qué herramienta de prueba es responsable de documentar los programas?

- a) Generador de pruebas / archivos
- b) Sistema de prueba de arnés
- c) Sistemas de prueba de archivo
- d) analizador de cobertura

40. La prueba beta se realiza por

- a) Desarrolladores
- b) probadores
- c) Usuarios
- d) Todas las mencionadas.

41. La herramienta ejecutora estándar analiza todo el programa.

- a) verdad
- b) falso

42. Programa de depuración es un programa que se ejecuta simultáneamente con el programa bajo prueba y proporciona comandos para

- a) examinar la memoria y los registros
- b) detener la ejecución en un punto particular
- c) búsqueda de referencias para variables particulares, constantes y registros
- d) todas las mencionadas

43. Execution Verifier es una herramienta dinámica que también se conoce como

- a) Generador de archivos de prueba
- b) analizador de cobertura
- c) Comparador de salida
- d) Sistema de prueba de arnés

44. ¿Por qué es difícil construir el software?

- a) Cambios controlados.
- b) Falta de reutilización
- c) Falta de seguimiento
- d) Todas las mencionadas.

45. ¿Cuál de los siguientes no es un conflicto en el equipo de desarrollo de software?

- a) Actualizaciones simultáneas
- b) Código compartido y común.
- c) Versiones
- d) Problemas gráficos

46. ¿Cuál de los siguientes dura la duración del proyecto y cubre el proceso de desarrollo?

- a) Monitoreo de todos los parámetros clave como costo, cronograma, riesgos
- b) Tomar acciones correctivas cuando sea necesario.
- c) Proporcionar información sobre el proceso de desarrollo en términos de métricas.
- d) Todas las mencionadas.

47. ¿Cuál de los siguientes no es un entorno típico en la facilitación de la comunicación?

- a) equipos multiples
- b) Grupos de usuarios múltiples
- c) festivales múltiples
- d) Ubicaciones múltiples

48. ¿Cuál de los siguientes es un proceso de software?

- a) Análisis y diseño.
- b) Configuración y gestión.
- c) modelado de negocios
- d) Todas las mencionadas.

49. ¿Cuál de los siguientes no está incluido en las Reuniones de Temas?

- a) Temas recogidos el día anterior.
- b) Horario regular de reunión.
- c) Discusión con los negocios.
- d) Asistencia

50. ¿Cuál de los siguientes no forma parte de los conceptos básicos de la gestión de la configuración del software?

- a) Identificación
- b) Versión
- c) Auditoría y revisión
- d) Contabilidad de estado (Status)

51. ¿Qué es una colección de elementos de software tratados como una unidad para los propósitos de SCM?

- a) Elemento de configuración del software
- b) Línea de base
- c) Configuración
- d) Panel de control de configuración

52. ¿Cuál es uno o más elementos de configuración de software que se han revisado y acordado formalmente y que sirven como base para un mayor desarrollo?

- a) Configuración
- b) Línea de base
- c) software
- d) Todas las mencionadas.

53. ¿Qué es validar la integridad de un producto?

- a) Identificación
- b) software
- c) Auditoría y revisión
- d) Contabilidad de estado

54. ¿Cuál es el grupo con la responsabilidad de revisar y aprobar cambios a las líneas de base?

- a) Elemento de configuración del software
- b) Línea de base
- c) Configuración
- d) Panel de control de configuración

55. En muchas configuraciones, PM es un centro de comunicación.

- a) verdad
- b) falso

56. ¿Qué es una instancia específica de una línea de base o un elemento de configuración?

- a) software
- b) Configuración
- c) Versión
- d) Contabilidad de estado

57. SCM significa

- a) Software Control Management
- b) Software Configuration Management
- c) Software Concept Management
- d) Ninguna de las mencionadas

58. Cuando el código se pone a disposición de otros, va en un / a

- a) disco duro
- b) biblioteca de acceso controlado
- c) servidores
- d) control de acceso

59. ¿Cuál de las siguientes no es una fase principal en el Proceso de administración de la configuración (CM)?

- a) Planificación de CM
- b) Ejecutando el proceso CM
- c) auditorias de MC
- d) Ninguna de las mencionadas

60. CM se trata de administrar los diferentes artículos en el producto y los cambios en ellos.

- a) verdad
- b) falso

61. ¿Qué permite que diferentes proyectos utilicen los mismos archivos de origen al mismo tiempo?

- a) Control de versiones
- b) control de acceso
- c) Proceso CM
- d) Control de versiones y control de acceso.

62. ¿Cuál de los siguientes no es un proceso de gestión de cambios?

- a) Registrar los cambios
- b) Estimar impacto en esfuerzo y cronograma.
- c) Revisar el impacto con los grupos de interés.
- d) Ninguna de las mencionadas

63. La administración de la configuración (CM) es necesaria para entregar el producto al cliente

- a) verdad
- b) falso

64. ¿Cuál es uno o más elementos de configuración de software que se han revisado y acordado formalmente y que sirven como base para un mayor desarrollo?

- a) Línea de base
- b) Cambios acumulativos.
- c) CM
- d) Control de cambio

65. ¿Cómo se verifican las líneas de base?

- a) por opiniones
- b) Por inspecciones.
- c) A prueba de código
- d) Todas las mencionadas.

66. ¿Cuál de los siguientes es un ejemplo de elementos de configuración?

- a) procedimientos de SCM
- b) código fuente
- c) Descripciones de diseño de software
- d) Todas las mencionadas.

67. SCM controla solo los productos del proceso de desarrollo.

- a) verdad
- b) falso

68. CCB significa

- a) Cambio de tablero de control
- b) Cambio de línea base de control
- c) Cambios acumulados en la línea de base
- d) Ninguna de las mencionadas

69. ¿Qué información se requiere para procesar un cambio a una línea de base?

- a) Razones para realizar los cambios.
- b) Una descripción de los cambios propuestos.
- c) Lista de otros elementos afectados por los cambios.
- d) Todas las mencionadas.

70: Se corrige la relación jerárquica entre el lenguaje libre de contexto, el derecho lineal y el sensible al contexto.

- A. context-free \subset right-linear \subset context-sensitive
- B. context-free \subset context-sensitive \subset right-linear
- C. context-sensitive \subset right-linear \subset context-free
- D. right-linear \subset context-free \subset context-sensitive

71: En la siguiente gramática:

$$\begin{aligned}x:: &= x \oplus y \mid 4 \\y:: &= z^* y \mid 2 \\z:: &= \text{id}\end{aligned}$$

¿Cuál de las siguientes es verdad?

- A. \oplus es asociativa izquierda mientras que $*$ es asociativa derecha
- B. Tanto \oplus como $*$ son asociativos izquierda
- C. \oplus es asociativa derecha mientras que $*$ es asociativa izquierda
- D. ninguno de estos

72: ¿Cuál de los siguientes CFG no puede ser simulado por un FSM?

- A. $S \rightarrow Sa \mid \text{segundo}$
- B. $S \rightarrow aSb \mid ab$
- C. $S \rightarrow abX, X \rightarrow cY, Y \rightarrow d \mid aX$
- D. ninguno de estos

73: ADG se dice que está en forma de Chomsky (CNF), si todas las producciones son de la forma $A \rightarrow BC$ o $A \rightarrow a$. Sea G un CFG en CNF. Para derivar una cadena de terminales de longitud x , el número de producciones a utilizar es

- A. $2x - 1$
- B. $2x$
- C. $2x + 1$
- D. ninguno de estos

74: ¿Cuál de las siguientes afirmaciones es correcta?

- A. $A = \{ \text{Si } un bn \mid n = 0, 1, 2, 3 \dots \}$ es un lenguaje regular
- B. El conjunto B de todas las cadenas de igual número de a y b 's define un lenguaje regular
- C. $L(A^* B^*) \cap B$ da el conjunto A
- D. ninguno de estos

75: P, Q, R son tres idiomas, si P y R son regulares y si $PQ = R$, entonces

- A. Q tiene que ser regular
- B. Q no puede ser regular

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

76: Una clase de lenguaje que se cierra bajo

- A. La unión y la complementación deben cerrarse bajo la intersección
- B. La intersección y el complemento deben cerrarse bajo unión.
- C. La unión y la intersección deben cerrarse bajo complementación.
- D. tanto (A) como (B)

77: Las producciones.

$$\begin{aligned}E &\rightarrow E + E \\E &\rightarrow E - E \\E &\rightarrow E * E \\E &\rightarrow E / E \\E &\rightarrow \text{id}\end{aligned}$$

- A. generar un lenguaje intrínsecamente ambiguo
- B. generar un lenguaje ambiguo pero no inherentemente
- C. son inequívocos
- D. puede generar todos los cálculos válidos de longitud fija posibles para realizar sumas, restas, multiplicaciones y divisiones, que se pueden expresar en una expresión

78: ¿Cuál de las siguientes definiciones genera el mismo lenguaje que L, donde
 $L = \{x^n y^n \text{ such that } n >= 1\}$?

- I. $E \rightarrow xEy \mid xy$
- II. $xy \mid (x+y)^+$
- III. $.x+y+$

- A. Solo I
- B. I y II
- C. II y III
- D. Sólo II

79: Siguiendo la gramática libre del contexto.

$$\begin{aligned}S &\rightarrow aB \mid bA \\A &\rightarrow b \mid aS \mid bAA \\B &\rightarrow b \mid bS \mid aBB\end{aligned}$$

genera cadenas de terminales que tienen

- A. igual numero de a y b
- B. Número impar de a y número impar b
- C. número par de a y número par de b
- D. número impar de a y número par de a

80: Definir para el lenguaje libre de contexto. $L < \{0;1\}$

Si $L = \{w \mid w \text{ no es vacío y tiene un número igual de 0's y 1's}\}$, entonces init (L) está configurado de todas las cadenas binarias

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
A. con números desiguales de 0's y 1's. www.gulaceneval.com.mx Todos los Derechos Reservados.**
ESTA GUÍA DE ESTUDIO, NO PERTENECE A LA EDITORIAL GUIAS MEXICO

- B. incluyendo la cadena nula.
- C. Ambos (a) y (b)
- D. ninguno de estos

81: ¿Cuál de los siguientes CFG no puede ser simulado por un FSM?

- A. $s \rightarrow sa \mid a$
- B. $s \rightarrow abX, X \rightarrow cY, Y \rightarrow a \mid axY$
- C. $s \rightarrow a sb \mid ab$
- D. ninguno de estos

82: La limitación básica de FSM es que

- A. no puede recordar una gran cantidad de información arbitraria
- B. A veces no reconoce las gramáticas que son regulares
- C. A veces reconoce que las gramáticas no son regulares.
- D. ninguno de estos

83: ¿Cuál de los siguientes no es posible algorítmicamente?

- A. Gramática regular a gramática libre de contexto.
- B. FSA no determinista a FSA determinista
- C. PDA no determinista a PDA determinista.
- D. ninguno de estos

84: El CFG

$s \rightarrow as \mid bs \mid a \mid b$
es equivalente a la expresión regular

- A. $(a + b)$
- B. $(a + b)(a + b)^*$
- C. $(a + b)(a + b)$
- D. ninguno de estos

85: Considera la gramática:

$S \rightarrow ABCc \mid Abc$
 $BA \rightarrow AB$

*Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.
EDICTI_GUIA_MEX Editorial Guías México*

Aa → aa

¿Cuál de las siguientes oraciones se puede derivar de esta gramática?

- A. abc
- B. aab
- C. abcc
- D. abbb

86: Lema de bombeo se utiliza generalmente para probar que

- A. La gramática dada es regular.
- B. La gramática dada no es regular.
- C. si dos expresiones regulares dadas son equivalentes o no
- D. ninguno de estos

87: el lenguaje de todas las palabras con al menos 2 a puede describirse mediante la expresión regular

- A. $(ab)^*a$ and $a(ba)^*$
- B. $(a + b)^* ab^* a (a + b)^*$
- C. $b^* ab^* a (a + b)^*$
- D. todos estos

88: Cualquier cadena de terminales que pueda generar el siguiente CFG es

$$\begin{aligned} S &\rightarrow XY \\ X &\rightarrow aX \mid bX \mid a \\ Y &\rightarrow Ya \mid Yb \mid a \end{aligned}$$

- A. tiene al menos una 'b'
- B. debe terminar en una 'a'
- C. no tiene a o b consecutivas
- D. tiene al menos dos a

89: $L = (an bn an \mid n = 1,2,3)$ es un ejemplo de un lenguaje que es

- A. contexto libre
- B. no libre de contexto
- C. no libre de contexto pero cuyo complemento es la FQ
- D. ambos (b) y (c)

90: Si $\Sigma = \{0, 1\}$, $L = \Sigma^*$ and $R = \{0^n 1^n \mid n \in \mathbb{N} \text{ such that } n > 0\}$
entonces los idiomas $L \cup R$ y R respectivamente son

- A. Regular, Regular
- B. Regular, no regular
- C. No regular, no regular
- D. ninguno de estos

91: FSM puede reconocer

- A. cualquier gramática
- B. solo CG
- C. Ambos (a) y (b)
- D. solo gramática regular

92: El conjunto de idiomas regulares sobre un conjunto de alfabeto dado se cierra debajo de

- Una union
- B. complementación
- C. intersección
- D. Todos estos

93: ¿Cuál de las siguientes afirmaciones es correcta?

- A. Todos los lenguajes no pueden ser generados por CFG
- B. Cualquier lenguaje regular tiene un CFG equivalente
- C. Algunos lenguajes no regulares no pueden ser generados por CFG
- D. ambos (b) y (c)

94: Dado $A = \{0,1\}$ and $L = A^*$. If $R = \{0^n 1^n, n > 0\}$, entonces el lenguaje $L \cup R$ y R son respectivamente

- A. regular
- B. no regular, regular
- C. regular, no regular
- D. contexto libre, no regular

95: Definir para un lenguaje libre de contexto.

$$L \leq \{0 ; 1\} \text{ init } (L) = \{u/uv \in L \text{ for some } v \text{ in } \{0,1\}\}$$

(en otras palabras, init (L) es el conjunto de prefijos de L)

Deje que $L \setminus w/w$ sea noempty y tenga un número igual de 0 y 1)

Entonces init (L) es

- A. conjunto de todas las cadenas binarias con un número desigual de 0 y 1
- B. conjunto de todas las cadenas binarias, incluida la cadena nula
- C. conjunto de todas las cadenas binarias con exactamente un 0 más que el número de 1 o 1 más que el número de 0
- D. ninguno de estos

96: Si L_1 y L_2 son lenguajes sin contexto y R es un conjunto regular, ¿cuál de los siguientes idiomas no es necesariamente un idioma sin contexto?

- A. $L_1 L_2$
- B. $L_1 \cap L_2$
- C. $L_1 \cap R$
- D. $L_1 \cup L_2$

97: Consideremos una gramática con las siguientes producciones.

$S \rightarrow aab \mid bac \mid aB$
 $S \rightarrow \alpha S \mid b$
 $S \rightarrow \alpha b b \mid ab$
 $S\alpha \rightarrow bdb \mid b$

La gramática anterior es

- A. Contexto libre
- B. regular
- C. sensible al contexto
- D. LR (k)

98: ¿Qué se puede decir acerca de un lenguaje regular L sobre $\{a\}$ cuya automatización de estados finitos mínimos tiene dos estados?

- A. L debe ser $\{a^n \mid n \text{ es impar}\}$
- B. L debe ser $\{a^n \mid n \text{ es par (even)}\}$
- C. L debe ser $\{a^n \mid n > 0\}$
- D. O L debe ser $\{a^n \mid n \text{ es impar}\}$, o L debe ser $\{a^n \mid n \text{ es par (even)}\}$

99: la gramática libre de contexto no está cerrada bajo

- A. producto
- B. union
- C. complementación
- D. kleen star

100: Si L es un lenguaje reconocible por un autómata finito, entonces el frente del idioma $\{L\} = \{w \text{ such that } w \text{ is prefix of } v \text{ where } v \in L\}$, es un

- A. lenguaje regular
- B. lenguaje libre de contexto
- C. lenguaje sensible al contexto
- D. lenguaje de enumeración recursiva

101: ¿Para cuál de las siguientes aplicaciones, las expresiones regulares no se pueden usar?

- A. Diseñar computadoras
- B. Diseñar compiladores
- C. Ambos (a) y (b)
- D. Desarrollar computadoras

102: Considera la siguiente gramática.

$$\begin{aligned}S &\rightarrow Ax / By \\A &\rightarrow By/Cw \\B &\rightarrow x / Bw \\C &\rightarrow y\end{aligned}$$

¿Cuál de las expresiones regulares describe el mismo conjunto de cadenas que la gramática?

- A. $xw^*y + xw^*yx + ywx$
- B. $xwy + xw^*xy + ywx$
- C. $xw^*y + xw^*x yx + ywx$
- D. $xw xy + xww^*y + ywx$

103: ¿Cuál de las siguientes afirmaciones es (son) correcta?

- A. Los lenguajes recursivos están cerrados por complementación.
- B. Si una lengua y su complemento son regulares, la lengua es recursiva
- C. Conjunto de lenguaje recursivamente enumerable está cerrado bajo unión
- D. Todos estos

104: ¿Cuál de las siguientes afirmaciones es incorrecta?

- A. Cualquier lenguaje regular tiene una gramática libre de contexto equivalente.
- B. Algunos lenguajes no regulares no pueden ser generados por ninguna gramática sin contexto
- C. Intersección entre el lenguaje libre de contexto y un lenguaje regular es siempre libre de contexto
- D. Todos los lenguajes pueden ser generados por gramática libre de contexto.

105: Consideremos una gramática:

$$G = (\{x, y\}, \{s, x, y\}, p, s)$$

donde elementos de análisis:

$$\begin{aligned}S &\rightarrow x \ y \\S &\rightarrow y \ x \\x &\rightarrow x \ z \\x &\rightarrow x \\y &\rightarrow y \\z &\rightarrow z\end{aligned}$$

El lenguaje L generado por G con mayor precisión se llama

- A. Chomsky tipo 0
- B. Chomsky tipo 1
- C. Chomsky tipo 2
- D. Chomsky tipo 3

106: Consideremos una gramática:

$$G = (\{S\}, \{0, 1\}, p, s)$$

donde los elementos de p son:

$$\begin{aligned}S &\rightarrow ss \\S &\rightarrow 0S1 \\S &\rightarrow 1S0 \\S &\rightarrow \text{empty}\end{aligned}$$

La gramática generará.

- A. lenguaje regular
- B. lenguaje libre de contexto
- C. lenguaje sensible al contexto
- D. lenguaje recursivo enumerable

107: Se llama una gramática que produce más de un árbol de análisis para alguna oración

- A. ambiguos
- B. no ambiguo
- C. regular
- D. ninguno de estos

108: Dada una gramática G se llama una producción de G con un punto en alguna posición del lado derecho

- A. LR (0) item of G
- B. LR (1) item of G
- C. tanto (a) como (b)
- D. ninguno de estos

RESPUESTAS DE LA QUINTA PRÁCTICA DE EXAMEN

1	B
2	A
3	D
4	C
5	D
6	A
7	A
8	B
9	D
10	C
11	A
12	D
13	B
14	A
15	B
16	C
17	A
18	B
19	A
20	D
21	A
22	D

23	A
24	C
25	A
26	D
27	D
28	C
29	B
30	C
31	C
32	A
33	D
34	A
35	C
36	D
37	B
38	B
39	C
40	C
41	B
42	D
43	B
44	C

45	D
46	A
47	C
48	D
49	C
50	B
51	A
52	B
53	C
54	D
55	A
56	C
57	B
58	B
59	D
60	A
61	A
62	D
63	A
64	A
65	C
66	D

67	A
68	A
69	D
70	D
71	A
72	B
73	A
74	C
75	C
76	D
77	B
78	A
79	A
80	B
81	C
82	A
83	C
84	B
85	A
86	B
87	D
88	D

89	D
90	B
91	D
92	D
93	D
94	D
95	B
96	B
97	C
98	B
99	C
100	A
101	C
102	A
103	D
104	D
105	D
106	A
107	A
108	A

ANTOLOGÍA

ANTOLOGÍA

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**
EDICT:_GUIA_MEX_ Editorial Guías Mexico

CONTENIDO

- Lectura 1. Proceso de desarrollo de aplicaciones software
- Lectura 2. Introducción al desarrollo del software
- Lectura 3. Ingeniería de software, análisis y diseño
- Lectura 4. Estándares de desarrollo de sistemas software
- Lectura 5. Análisis, diseño y mantenimiento del software
- Lectura 6. Metodologías para desarrollar software seguro
- Lectura 7. Entornos de programación, concepto, funciones y tipos
- Lectura 8. Pruebas e implementación de la aplicación
- Lectura 9. Ingeniería de software I. Manual de prácticas
- Lectura 10. Metodología de Ingeniería del Software para el desarrollo y mantenimiento de sistemas de información
- Lectura 11. El software de base sistemas operativos y lenguajes
- Lectura 12. Algoritmos y estructuras de datos
- Lectura 13. Modelos de Computación I
- Lectura 14. Un Nuevo Modelo para Procesos de Computación con Palabras en Toma de Decisión Lingüística
- Lectura 15. Informática teórica. Elementos propedéuticos.
- Lectura 16. Metodología de la programación (Ejercicios)
- Lectura 17. Problemario de algoritmos resueltos con diagramas de flujo y pseudocódigo
- Lectura 18. Panorámica de la teoría computacional: Desarrollo y problemas abiertos.
- Lectura 19. Algoritmos y Estructuras de Datos

Lectura 1. Proceso de desarrollo de aplicaciones software

Programación orientada a objetos: Languages, Metodologías y Herramientas Master de Computación

Proceso de desarrollo de aplicaciones software

 José M. Drake
Computadores y Tiempo Real

Santander, 2008



Proceso de desarrollo de aplicaciones software.

- Un proceso de desarrollo de software es la descripción de una secuencia de **actividades** que deben ser seguida por un equipo de **trabajadores** para generar un conjunto coherente de **productos**, uno de los cuales en el programa del sistema deseado.
- El objetivo básico del proceso es hacer **predecible** el trabajo que se requiere:
 - Predecir el costo.
 - Mantener un nivel de calidad
 - Predecir el tiempo de desarrollo

El objetivo de un proceso de desarrollo de programas es la formalización de las actividades relacionadas con el desarrollo del software de un sistema informático.

La mayoría de los proyectos que se desarrollan, finalizan tarde, cuesta mucho mas de lo estimado. ¿Por qué ocurre esto?. El software se encuadra entre los artefactos mas complejos que es capaz de desarrollar el hombre, y además dado que no tiene límites físicos por su carácter inmaterial, su dimensión se puede imaginar ilimitada.



Naturaleza de las aplicaciones software

- # No existe un proceso de desarrollo universal. Debe configurarse de acuerdo con la naturaleza del producto y de la experiencia de la empresa.
- # Tipos de aplicaciones:
 - Aplicaciones **Monoprocesadoras**: Se ejecutan en un solo computador. No se comunica con otras aplicaciones. Ej. Procesador de texto.
 - Aplicaciones **Embebidas**: Se ejecuta en un entorno computarizado especial. Requiere codiseño hardware/software. Ej: Teléfono móvil.
 - Aplicaciones de **Tiempo Real**: Tiene entre sus especificaciones requerimiento temporales. Naturaleza reactiva. Ej: Software de radar.
 - Aplicaciones **Distribuidas**: Se ejecuta en múltiples procesadores. Requiere intercomunicación a través de la red. Ej: Aplicaciones de red.

La adopción por una empresa de un proceso de desarrollo contrastado, le permite producir aplicaciones software con plazos y costos predecibles y con calidad constante. En esta sección se estudia un marco de desarrollo basados en criterios genéricos, y que cada empresa debe configurar y refinar de acuerdo con las características de la empresa y del producto.

No existe un proceso único aplicable al desarrollo de cualquier tipo de aplicación, adoptable por cualquier empresa y valido para cualquier cultura productiva



Objetivos de un proceso de desarrollo

- Proporcionar una guía de ejecución del proyecto que defina para los técnicos la secuencia de tareas que se requieren y los productos que deben generar.
- Mejorar la calidad del producto en:
 - Disminuir el número de fallos
 - Bajar la severidad de los defectos
 - Mejorar la reusabilidad
 - Mejorar la estabilidad del desarrollo y el costo de mantenibilidad
- Mejorar la predecibilidad del proyecto en:
 - La cantidad de trabajo que requiere
 - El tiempo de desarrollo que se necesita
- Generar la información adecuada a los diferentes responsables de forma que ellos puedan hacer un seguimiento efectivo.

La razón básica por la que se requiere disponer de un proceso de desarrollo es mejorar la seguridad de trabajo eliminando riesgos innecesarios y conseguir un producto de la máxima calidad.

Específicamente un proceso de desarrollo debe conseguir:

•Proporcionar una plantilla de desarrollo del proyecto en el que quede definido lo que cada trabajador que interviene debe realizar y los productos que debe generar a lo largo de él.

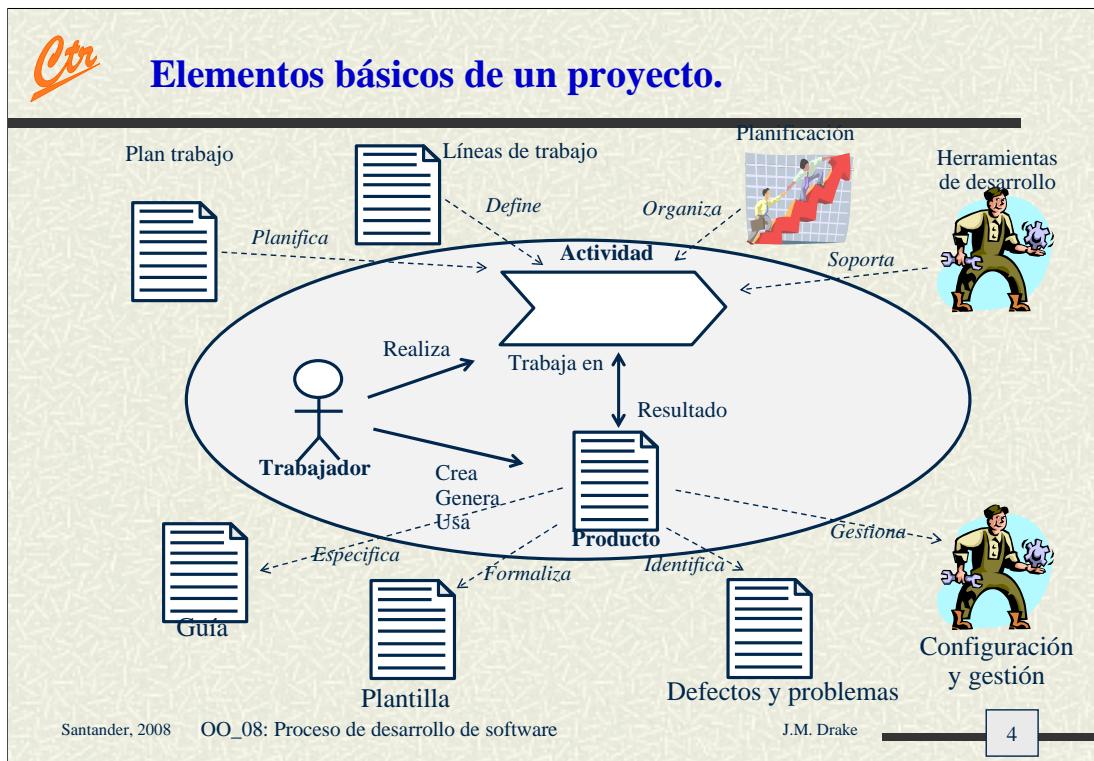
•Mejorar la calidad del producto que se genera en función de:

- Disminuir el número de defectos que se producen y que deben ser corregidos.
- Disminuir la severidad de los defectos residuales que al final pueden permanecer en el producto final.
- Mejorar la reusabilidad, de forma que gran parte del trabajo que se realiza pueda ser reutilizado en próximos proyectos.
- Mejorar la estabilidad del proceso de forma que se minimicen las reelaboraciones del producto.
- Generar un producto que sea de fácil mantenimiento posterior.

•Mejorar la predecibilidad del proyecto en función de:

- La cantidad de esfuerzo humano y de recursos que requiera.
- Disminuir los plazos de desarrollo y llegada al mercado.

•Generar a lo largo del desarrollo de la información adecuada y diferenciada para que los diferentes responsables del proyecto puedan hacer su seguimiento de forma efectiva.



Los elementos básicos de un proceso de desarrollo de software es definir los papeles que juegan los trabajadores, las actividades que desarrollan y los productos que deben generarse. En un plan de desarrollo cada trabajador debe tener su papel dentro de él, lo que define las actividades que debe realizar y los productos que debe generar.

Las actividades son las tareas que deben realizar los trabajadores para cumplir sus obligaciones. A alto nivel, estas actividades son concebidas como las fases del proceso (especificación, análisis, ..), mientras que a mas bajo nivel son tareas mas concretas (crear cierto diagramas, escribir código,..).

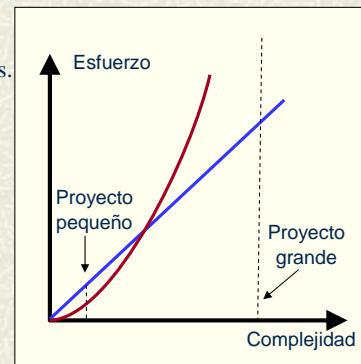
Los productos son los documentos o información que debe ser creada como consecuencia de la actividad que se desarrolla. El producto ultimo es el sistema que se desarrolla, pero en las fases intermedias deben generarse una amplia gama de documentos intermedios. Cada actividad debe tener siempre como principal objetivo generar ciertos productos bien definidos y especificados.

Los procesos deben estar condicionados por el tipo de producto que se desarrolla y por la tradición y experiencia de la empresa que lo desarrolla.



Escalabilidad.

- La escalabilidad es una propiedad importante de un proceso, ya que la dimensión de los proyectos software son muy variables.
- Describe, si el esfuerzo que se requiere en el desarrollo de un proyecto varía suavemente (linealmente) con su complejidad.
- Cuando la complejidad de un proyecto crece:
 - Aumentan los niveles de abstracción que se usan.
 - Se incrementan la intercomunicación entre los miembros.
 - Es mas difícil localizar los errores.Hay que buscar que el esfuerzo crezca linealmente y no exponencialmente.
- Formas de conseguir la escalabilidad:
 - Disponer de diferentes escalas temporal para generar las actividades.
 - Hacer que las guías y plantillas tengan optatividad de acuerdo con las características del proyecto.



Una de las propiedades que deben ser exigidas a un proceso de desarrollo de aplicaciones software es la escalabilidad, lo que hace posible que sea aplicable tanto a sistemas complejos como a sistemas sencillos.

En general la propiedad de escalabilidad representa que si para desarrollar un proyecto de complejidad (y) es necesario realizar un esfuerzo (x), para desarrollar un proyecto de complejidad (100y) se requiere un esfuerzo (100cx) (donde c es una constante).

Cuando un proyecto crece, se produce que:

- Sus objetivos se hacen menos concretas y mas globales.
- El sistema gana en niveles de abstracción (tales como subsistemas, subsistemas, y así).
- Líneas potenciales de comunicación crece exponencialmente con el número de miembros del proyecto.
- El costo de los errores que hay que corregir se incrementa ya que aumentan las posibilidades de interferencias.

En general, existen dos soluciones para conseguir la escalabilidad de un proceso:

- El proceso es visto desde diferentes escalas de tiempo: macro, micro y nano escala, y en función de que el proyecto crezca mas relevancia adquieren la escala macro a fin de organizar y gestionar el proceso de desarrollo mas global.
- Muchas de las fase y mecanismos del proceso se hacen opcionales en función de que la complejidad de la aplicación se requiera.



Existen un conjunto de tecnología y criterios que facilitan los procesos de desarrollo:

- **Modelado Visual:** Facilita la capacidad de apreciar los diferentes elementos e interacciones del sistema en los diferentes niveles de abstracción.
- **Modelos Ejecutables:** La mejor gestión de los errores que inevitablemente se introducen a lo largo del proceso de desarrollo consiste en detectarlos y corregirlos tan pronto como se cometan. Para que esto se pueda realizar eficientemente conviene tener capacidad de realizar las pruebas directamente desde los modelos, bien mediante debugger a nivel de modelo, o bien, mediante la generación automática de prototipos que los hagan ejecutables.
- **Relación biunívoca entre modelos y códigos:** El proceso de desarrollo de las aplicaciones se basa en modelos que se desarrollan para que los diseños sean comprensibles y gestionables, y en código que es el producto final y es necesario que ambos se mantengan en todas las fases sintonizados. Esto se consigue si se automatiza el tránsito entre ambos.
- **Automatización de las pruebas a partir de las especificaciones:** El número de prueba que hay que realizar para detectar errores se incrementa de forma acumulativa, como consecuencia de que no solo hay que verificar los nuevos elementos sino su interferencia con lo ya probado.
- **Frameworks:** Los frameworks son aplicaciones parcialmente desarrolladas que se utilizan como plantillas para el desarrollo de nuevas aplicaciones.
- **Desarrollo incremental e iterativo:** Los sistemas de desarrollo deben basarse en la generación iterativas de prototipos utilizable que vayan aumentando gradualmente su funcionalidad en las sucesivas etapas hasta conseguir que sea plena.



Principales tareas de los procesos software.

- Entender las naturaleza de la aplicación.
- Establecer el plan de trabajo
- Generar y gestionar la documentación.
- Captura de los requerimientos.
- Diseñar y construir el producto.
- Probar y validar el producto.
- Entregar y mantener el producto.

La siguiente es una secuencia común de las actividades para un proyecto software:

1. Es necesario comprender la naturaleza del proyecto. Esto parece obvio, pero casi siempre lleva tiempo entender lo que desean los clientes, en especial cuando ellos mismos no saben por completo que quieren.
2. Los proyectos requieren documentación desde el principio; es muy probable que esta documentación sufra muchos cambios. Por esta razón, desde el principio debe disponerse de una estrategia para mantener los documentos que se generen. Este proceso es denominado “Gestión de la Configuración”.
3. Hay que reunir los requisitos que ha de cumplir la aplicación. Gran parte de esta actividad es conversar con los “interesados”.
4. Hay que analizar el problema, diseñar la solución y codificar los programas.
5. El producto inicial y final debe probarse en forma exhaustiva en todos sus aspectos.
6. Una vez entregado el producto, entra el modo “mantenimiento, que incluye reparaciones y mejoras. Es una actividad que consume muchos recursos, a veces hasta el 65% de los recursos utilizados en el desarrollo de la aplicación. Ello hace que sea considerado un objetivo fundamental.



Niveles de madurez de los procesos de desarrollo.

- **Primitivo:** No existe.
- **Programado:** Tiene definido una secuencia de etapas y los resultados que deben generar cada una de ellas.
- **Sistemático:** Esta formulado de forma sistemática.
- **Administrado:** Incorpora criterios para cuantificar el rendimiento de cada fase y del proceso.
- **Optimizado:** Dispone de parámetros de control que permite su optimización a las características del proyecto.

Se ha propuesto un procedimiento de evaluación de los procesos de desarrollo, que definen cinco niveles de madurez:

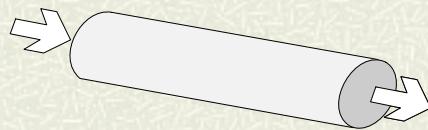
- **Primitivo:** El proceso no está formalizado y el equipo resuelve los problemas sobre la marcha. Sólo reconoce que el equipo es capaz de producir productos software. El éxito de proceso depende en gran medida de la experiencia de las personas que lo desarrollan. Cuando termina un proyecto nada se registra de su costo, tiempo ni calidad.
- **Programado:** El proceso es capaz de definir plazos razonables y verificar el avance del proyecto respecto de los plazos establecidos. Mantiene registros de los costos y tiempos del proyecto- En 1999 se comprobó que solo un 20% de los proyectos alcanzaban este nivel.
- **Sistemático:** El proceso está sistemáticamente definido y se reduce la dependencia de los individuos concretos que la realizan. Es conocido y comprendido por todas las partes que intervienen en el proyecto. Con este nivel se consigue predecir los resultados de proyectos similares a los que previamente desarrollado.
- **Administrado:** Puede predecir los costos y la programación de las tareas. El rendimiento del proceso es medible objetivamente y cuantitativamente.
- **Optimizado:** La ingeniería software está en continua evolución y es necesario disponer de estrategias adaptativas que permitan la adopción de las nuevas tecnologías. Su proceso es un proceso de metas e incluye maneras sistemáticas de evaluar el proceso mismo de la organización.

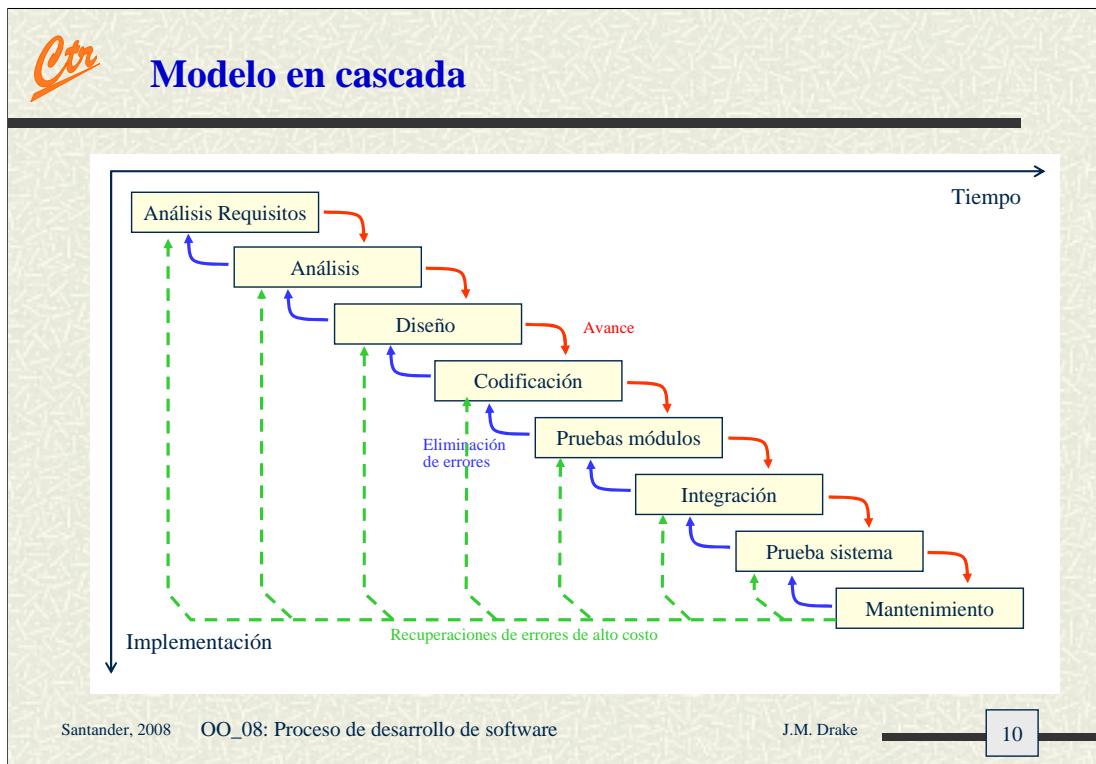


Modelo de procesos lineales.

■ Modelo Túnel:

- Ausencia de modelo
- No hay ningún control
- Sólo válido en proyectos muy pequeños.

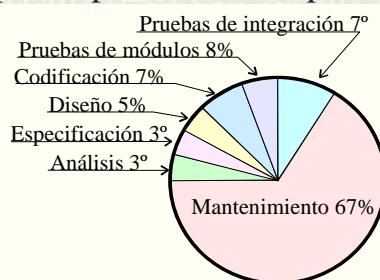






Problemas del proceso en cascada.

- La prueba efectiva solo se hace cuando ya está todo diseñado.
- Presupone que el producto está perfectamente definido antes de iniciar el desarrollo.
- Cuando se descubren problemas en la fase de mantenimiento sólo cabe adaptar el problema a la aplicación, y no al revés.



Costo de las fases del desarrollo de una aplicación

ctr

Proceso en espiral

- La aplicación es desarrollada en sucesivas fases por evolución de sistemas mas simples a sistemas mas complejos.
- En la naturaleza y en la tecnología, los sistemas complejos han sido siempre resultado de la evolución de sistemas mas simples.
- La programación orientada a objetos facilita la programación evolutiva:
 - Se diseñan prototipos con solo algunos objetos.
 - Se diseñan prototipos con objetos con funcionalidad limitada.

Santander, 2008 OO_08: Proceso de desarrollo de software

J.M. Drake

12

El ciclo de vida iterativo se basa en una idea simple: Para comprender, diseñar e implementar un sistema complejo, deben emplearse sucesivas etapas en las que en cada una de ellas se realiza una evolución respecto a la anterior.

Los sistemas complejos que funcionan en la Naturaleza y en el campo de la tecnología han resultado siempre de la evolución de sistemas más simples.

En cada etapa de desarrollo del sistema es necesario conseguir que sea estable frente a los cambios que va a suponer su evolución en la siguiente etapa. Las metodologías orientadas a objetos se construyen alrededor de conceptos como encapsulamiento y modularidad y presentan una mayor robustez frente a cambios. La orientación a objetos favorece el desarrollo de programas según un método iterativo.

Hay varias razones para utilizar el modelo en espiral:

- Con este modelo se pueden reducir riesgos.
- La generación de versiones parciales permite que el cliente se involucre en el proceso de desarrollo de la aplicación.
- Permite reunir a lo largo del proceso métrica del proceso en cada iteración. Y utilizar la información para las siguientes.

El proceso en espiral se ajusta al avance de los proyectos típicos; sin embargo requiere una administración mucho más cuidadosa, ya que hay que conseguir que la documentación sea consistente después de cada iteración. En particular el código debe corresponder al diseño documentado y debe satisfacer los requisitos documentados.



Características del proceso iterativo.

- # El proceso iterativo se basa en producir sucesivos prototipos (sistemas ejecutables) que van evolucionando desde requerimientos muy simples hasta los completos.
- # El desarrollo evolutivo de los prototipos facilita afrontar los problemas de mayor riesgo al principio.
- # A lo largo del desarrollo se van mostrando prototipos reales a los usuarios y clientes:
 - El usuario se enfrenta al producto de forma real.
 - El cliente se hace colaborador del proyecto.
 - El equipo está continuamente motivado por objetivos próximos.
 - La integración de los componentes es gradual.
 - Los progresos se evalúan de forma tangible y no sobre papel.

El proceso se basa en el desarrollo de prototipos ejecutables, y por tanto tangibles y medibles. Las entregas fuerzan al equipo a dar regularmente resultados concretos, lo que evita el síndrome del “90% terminado y 90% por hacer”

El desarrollo regular de las iteraciones facilita que se aborden desde el principio los problemas, y si se presentan, se tenga un tiempo razonable para resolverlos.

A lo largo del desarrollo se muestran prototipo a los clientes o a los usuarios, lo que supone que:

- El usuario se enfrenta a situaciones de uso concreto y le requiere estructurar mejor sus deseos.
- El usuario se constituye en colaborador del proyecto y asume su responsabilidad.
- El equipo está continuamente motivado por objetivos tangibles muy próximos.
- La integración se hace de forma progresiva y desaparece el Bing-Bang de la integración.
- Los progresos se evalúan sobre sistemas demostrables, lo que hace que los gestores responsables se tomen en serio los resultados intermedios.

Ctr N minicascadas.

En el proceso iterativo, cada ciclo reproduce básicamente un ciclo en cascada, solo que con objetivos mas simples.

```
graph TD; subgraph Iteracion [ ]; A[Análisis] --> B[Diseño]; B --> C[Codificación]; C --> D[Prueba]; end; Iteracion -- "N veces" --> Iteracion;
```

Santander, 2008 OO_08: Proceso de desarrollo de software J.M. Drake 14

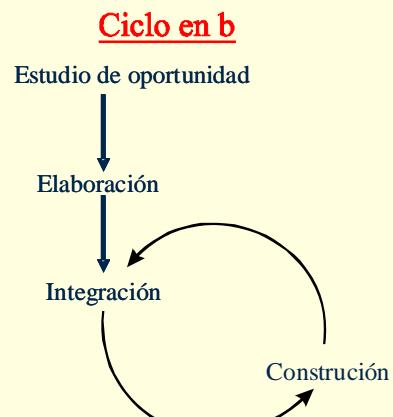
En el ciclo de vida iterativo, cada iteración reproduce el ciclo de vida en cascada, solo que los objetivos son mas simples. Los objetivos de una iteración se establecen en función de la evaluación de las iteraciones precedentes.

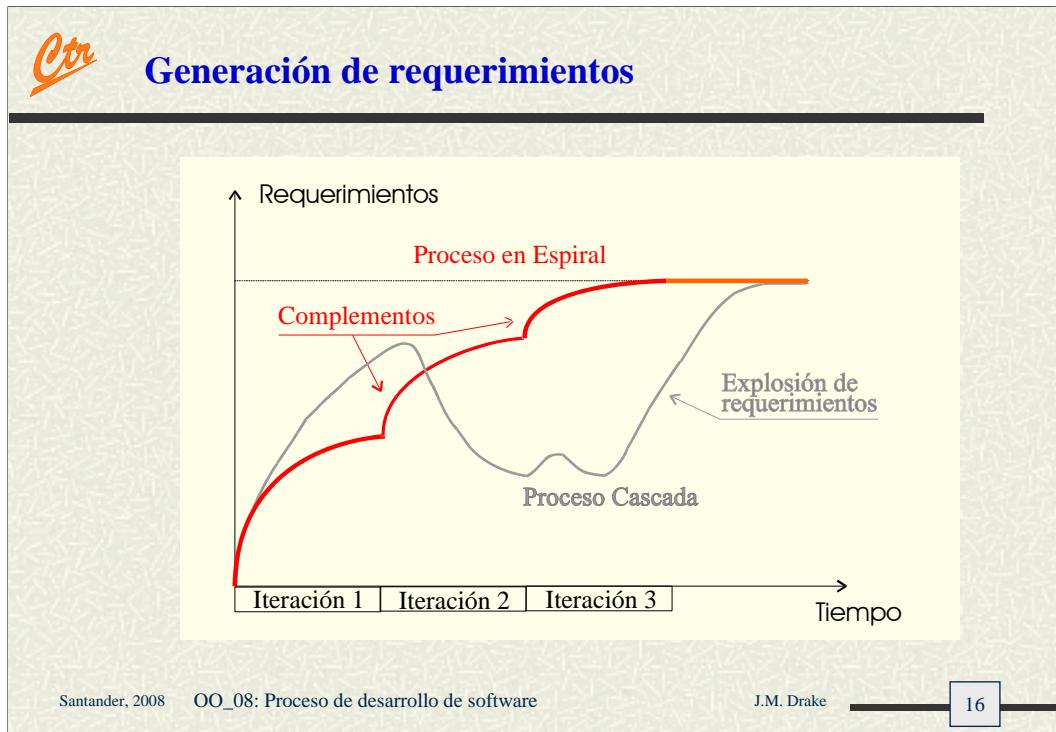
Cada iteración comprende las actividades siguientes:

- **Planificación:** Se elabora en función del estudio de riesgos de los resultados previos.
- **Análisis:** Estudia los casos de uso y los escenarios a realizar. Se descubren nuevas clases y asociaciones.
- **Diseño:** Se estudian las opciones necesarias para realizar la iteración. Si se necesita se retoca la arquitectura.
- **Codificación y pruebas:** Se codifica el nuevo código y se integra con el resultante de iteraciones anteriores.
- **Evaluación del prototipo parcial:** Los resultados se evalúan respecto a los criterios definidos para la iteración.
- **Documentación del prototipo:** Se congela y documenta el conjunto de elementos del prototipo obtenido.



Variantes de ciclo iterativo.



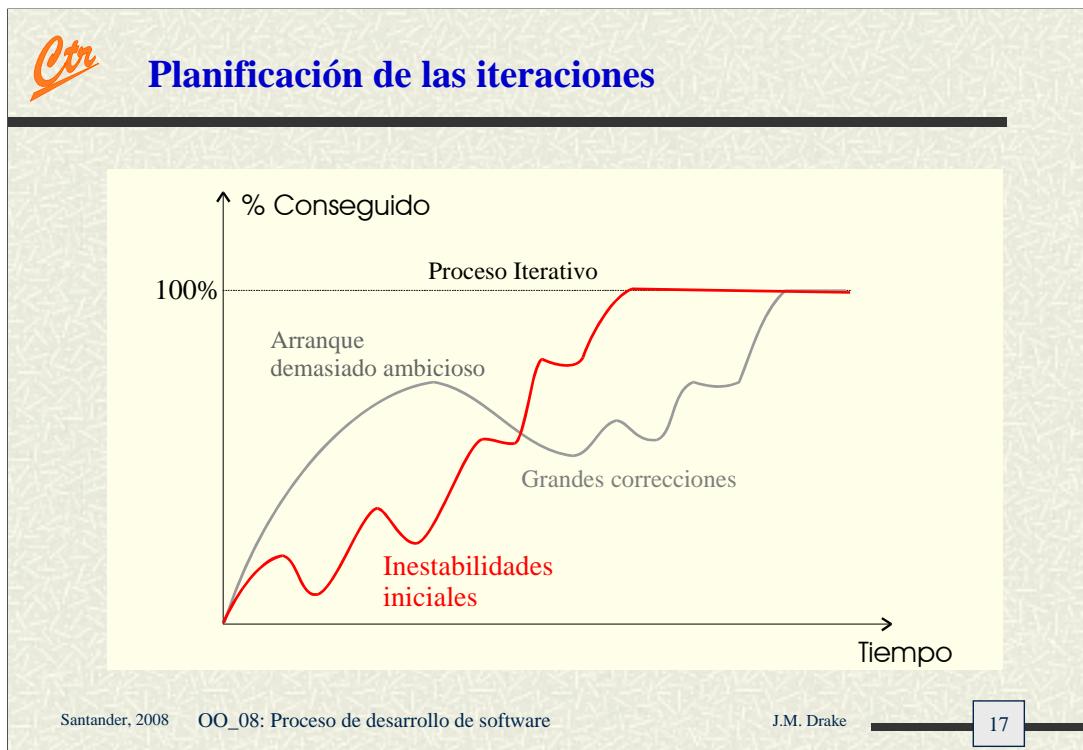


Al método iterativo se le suele atribuir que fomenta la generación ilimitada de requerimientos por parte de los clientes y usuarios.

Este temor no es fundado. Sea cual sea el procedimiento de desarrollo, las necesidades siempre aparecen cuando el cliente o el usuario se enfrenta con la aplicación. Cuanto mejor esté elaborada la fase de especificación, se generarán menos nuevos requerimientos.

En la figura se compara el ritmo de generación de requerimientos en el caso de proceso en cascada y de un proceso iterativo:

- En el proceso en cascada se definen unos requerimientos iniciales como consecuencia de la fase inicial de especificación. Luego el cliente se desentiende y los requerimientos no suben (si acaso bajan porque el programador trata de evitar los problemas que se le presentan). Al final, después de la integración el cliente se enfrenta con el sistema y se produce una explosión de requerimientos cuando el plazo de finalización está muy próximo.
- En el proceso iterativo, la generación de requerimientos iniciales es la misma. En las sucesivas iteraciones, el usuario se enfrenta con los prototipos y generan nuevos requerimientos incrementales. Con el proceso iterativo no se produce la explosión final de requerimientos.



Para un proyecto de unos 18 meses puede haber entre tres y seis iteraciones. Las iteraciones suelen tener una duración similar.

En la gráfica se comparan dos procesos con diferente planificación de iteraciones:

- La curva gris corresponde a una planificación en la que la primera iteración ha sido excesivamente ambiciosa, y presenta los problemas del ciclo en cascada. La integración presenta grandes problemas. Para resolverlos se pierde mucho del trabajo realizado y se incrementa mucho el tiempo de desarrollo.
- La curva roja presenta un proyecto que ha empleado las dos primeras iteraciones en estabilizar la arquitectura, y con perdida de parte del trabajo realizado se consigue eliminar los riesgos identificados. En las siguientes fases el proyecto avanza de forma incremental pero regular.



Proceso de desarrollo de Rational (USPD).

Rational propone un proceso basado en tres criterios:

- # Guiado por “Casos de Uso”.
- # Centrado sobre la “Arquitectura”.
- # Estrategia “Iterativa e Incremental”.

Rational que es una de las promotoras de UML, propone un proceso de desarrollo basado en tres principios:

Controlado por los “Casos de Uso”: Todas las actividades (Especificación, análisis, diseño, verificación y mantenimiento) son guiados por los casos de uso que describen la funcionalidad de la aplicación.

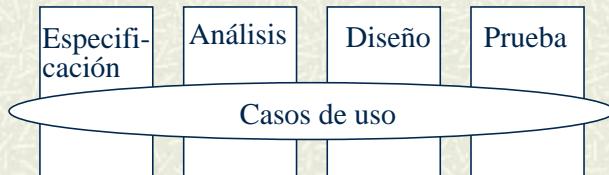
Centrado sobre la Arquitectura: La arquitectura se formula desde el principio del proyecto y se toma como referencia central del proceso. La arquitectura se introduce para satisfacer no solo las necesidades de la funcionalidad, sino también para conseguir flexibilidad frente a la evolución posterior.

Estrategia Iterativa e Incremental: El proceso se divide en pequeñas iteraciones definidas a partir de los casos de uso y de los análisis de riesgos. El desarrollo se realiza por sucesivas iteraciones que proporcionan prototipos incrementales del sistema. Las iteraciones pueden conducirse en paralelo.



Casos de uso

- # Los casos de uso definen la funcionalidad de la aplicación.
- # Los casos de uso constituyen guías transversales que dirigen todas las fases del proceso:
 - Especificación
 - Análisis
 - Diseño
 - Verificación y prueba



Los casos de uso expresan la funcionalidad que los usuarios requieren de la aplicación que se desarrolla y deben tenerse presente como los objetivos que guían las sucesivas actividades del proceso de desarrollo de la misma.

Los casos de uso formulan las necesidades de los usuarios con el lenguaje de los actores. Los modelos de casos de uso describen los servicios que se esperan del sistema, utilizando para ello la forma de interacciones entre los actores y el sistema.

Durante la fase de análisis de los objetos, se comprueba mediante diagramas de secuencias o de colaboración que el conjunto de objetos resultante satisfacen las necesidades requeridas en los casos de uso.

Los casos de uso marcan las necesidades de la aplicación a efectos de la implementación, fundamentalmente en lo que afecta a los requerimientos no funcionales.

Por último, los casos de uso sirven de base para establecer las pruebas funcionales que validan la operatividad de la aplicación.

 Arquitectura

- # La arquitectura trata la estructura global de la aplicación que subyace por debajo de las estructuras de datos y de los algoritmos.
- # La arquitectura se preocupa de la integridad, uniformidad, simplicidad, reusabilidad y estética.
- # La arquitectura ofrece una visión global de la aplicación, formula la estrategia, pero deja las decisiones tácticas para el desarrollo.
- # Características de una buena arquitectura son:
 - Simplicidad ■ Elegancia
 - Inteligibilidad ■ Niveles de abstracción bien definidos
 - Separación clara entre interfaz e implementación a cada nivel.

Santander, 2008 OO_08: Proceso de desarrollo de software J.M. Drake 20

La arquitectura se preocupa de la integridad, uniformidad, simplicidad, reusabilidad y estética de la aplicación.

La arquitectura es la estructura maestra que soporta la aplicación y su formulación estable debe hacerse al principio del proceso de desarrollo.

No existe una arquitectura universal válida para cualquier aplicación, pero si existen arquitecturas reutilizables dentro de un ámbito concreto.

La arquitectura se ha definido como la unión de Componentes, Formas y Motivaciones. Los componentes son las clases y objetos, las formas son las agrupaciones de clases y objetos (patterns), y las motivaciones explican por qué cada agrupación es la adecuada en cada contexto.

Características que debe ofrecer una buena arquitectura son:

- Simplicidad: Entendida como una estructura basada en pocos y bien definidos criterios.
- Elegancia: Su estructura se basa en líneas bien definidas que se mantienen uniformemente a lo largo de toda la aplicación y en la ausencia de atajos y casos especiales que la oscurecen.
- Inteligibilidad: Debe ser fácil de comprender, ya que va a ser utilizada por todos los que intervienen en el desarrollo de la aplicación y que la utilizan como guía de sus actividades.
- Niveles de abstracción bien definidos: La abstracción correcta de los componentes que se propongan a nivel de la arquitectura es la principal herramienta para simplificar la estructura y facilitar su evolución y reusabilidad.
- Separación entre interfaz e implementación: La separación sistemática entre interfaces e implementaciones, ayuda al mantenimiento de la aplicación. Si en el mantenimiento se respetan las interfaces, se pueden modificar las implementaciones de un componente sin efectos secundarios sobre otros.

 **Proceso ROPES** (Rapid Object-Oriented Process for Embedded Systems)

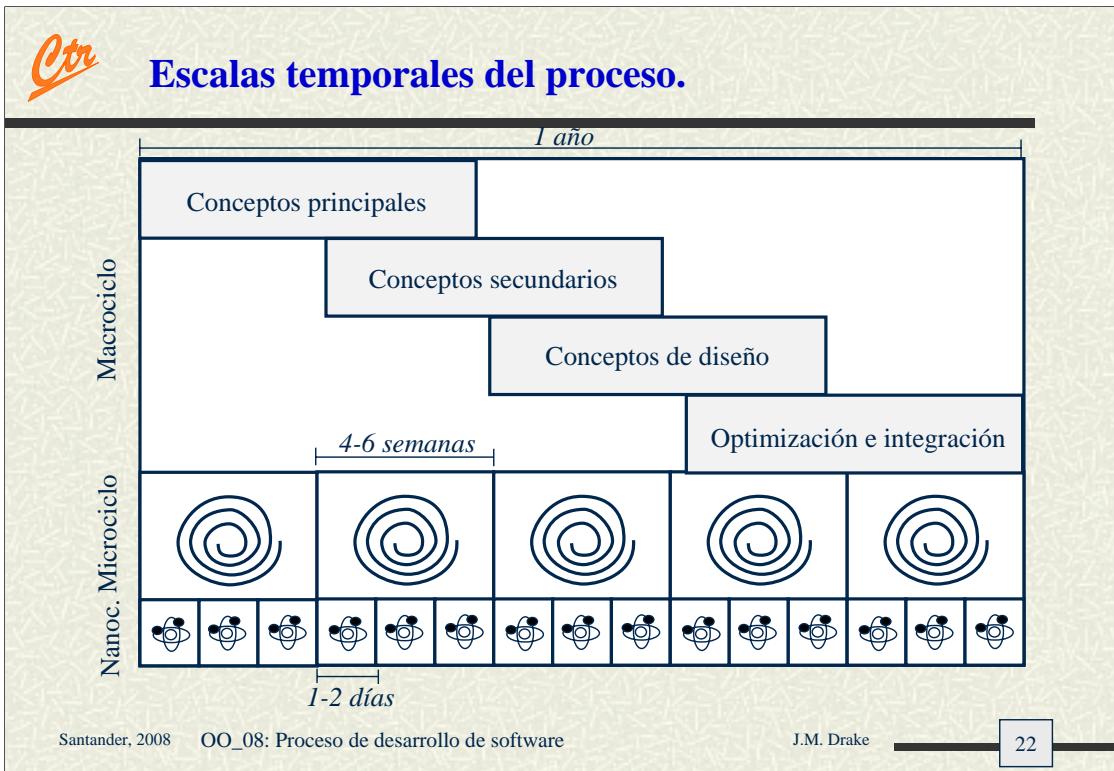
- Es una variante del proceso USPD concebido para desarrollo de sistemas en aplicaciones de tiempo real y embarcadas de tipo medio y grandes.
- Aunque es un proceso de propósito general, enfatiza los aspectos propios de sistemas de tiempo real y embarcados
- El proceso se concibe como una secuencia de actividades ejecutadas por trabajadores con papel diferenciado y que planifica la generación de un conjunto de productos, uno de los cuales es el sistema global.
- Sigue una estrategia de continua de actualización Darwiniana (lo que va bien se mantiene y lo que resulta inútil se mantiene).

Santander, 2008 OO_08: Proceso de desarrollo de software J.M. Drake 21

El proceso ROPES (Rapid Object-Oriented Process for Embedded Systems) es un proceso de desarrollo de aplicaciones software de propósito general (aplicable a cualquier tipo de aplicación) que enfatiza los aspectos de ingeniería software, integración de los sistemas y sus pruebas. Se ha venido utilizando en la empresa Artisan durante los últimos 25 años, aunque ha evolucionado intensamente en función de las tecnologías que van estando disponibles, y siguiendo en la evolución una estrategia darwiniana (lo que va bien se mantiene y lo que resulta inútil se mantiene).

El proceso ROPES es altamente escalable, y ha demostrado su eficacia tanto para proyectos pequeños realizados por dos o tres personas como por proyectos grandes realizados por equipos de cientos de personas..

ROPES se desarrolla con un único objetivo, producir aplicaciones software con un mínimo esfuerzo, sin fallos y con la máxima predecibilidad de su desarrollo.

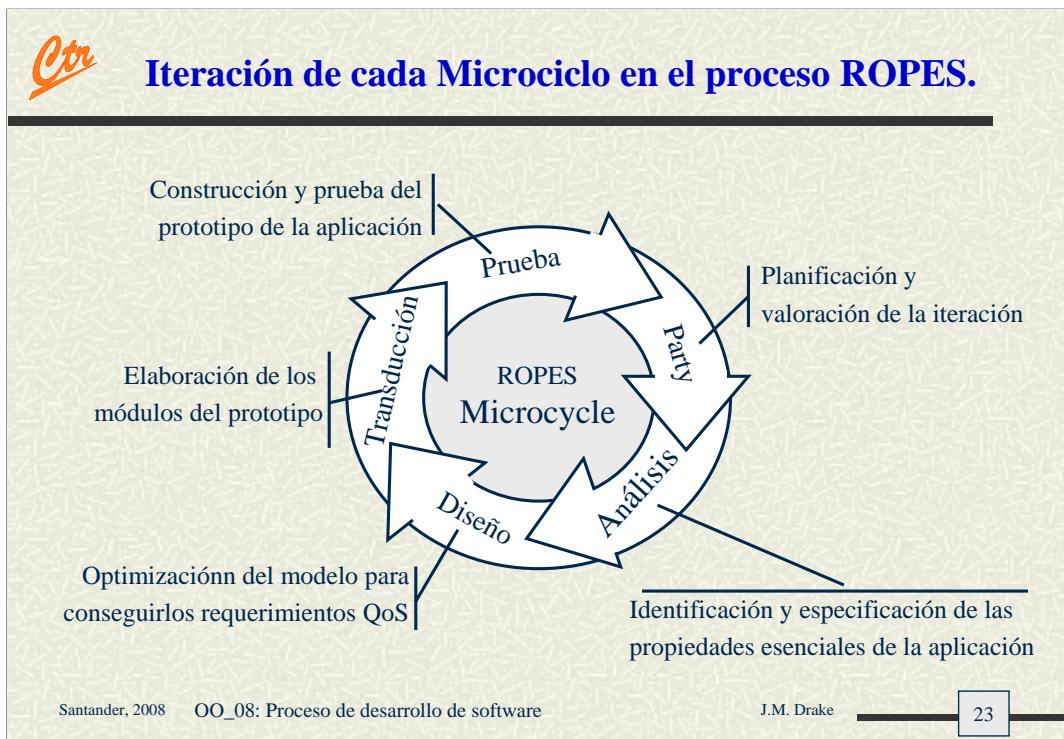


El proceso ROPES utiliza tres diferentes escalas de tiempo simultáneas.

El Macro ciclo hace referencia al desarrollo completo del proceso, desde que se inicia su especificación hasta que se entrega el producto. Dentro de él se desarrollan solapadamente las cuatro fases de un proceso tipo cascada, y su objetivo es dirigir la evolución de los prototipos que se programan, de forma que sucesivamente traten de resolver los problemas sobre conceptos claves, requerimientos, arquitectura y tecnología. En nuestro caso, se toma como referencia un proyecto de un año de duración.

El Micro ciclo tiene un ámbito más reducido y su objetivo es el desarrollo de un nuevo prototipo con una funcionalidad limitada que resuelva actividades consideradas de riesgo en el proceso de desarrollo. La duración adecuada para cada una de estas fases es de entre 4 y 6 meses.

El Nanociclo es el que tiene el ámbito más reducido y su objetivo es modelar ejecutar o establecer ideas que se necesitan. Su duración no debe exceder los 2 días. La razón por la que se introduce esta fase es la conveniencia de probar y estar seguro de forma continuada todas las ideas que se incorporen al proyecto.



En el proceso ROPES el microciclo tiene el objetivo de producir un nuevo prototipo que implemente algunos casos de uso parciales y que resuelva problemas o dudas que introduzcan riesgos.

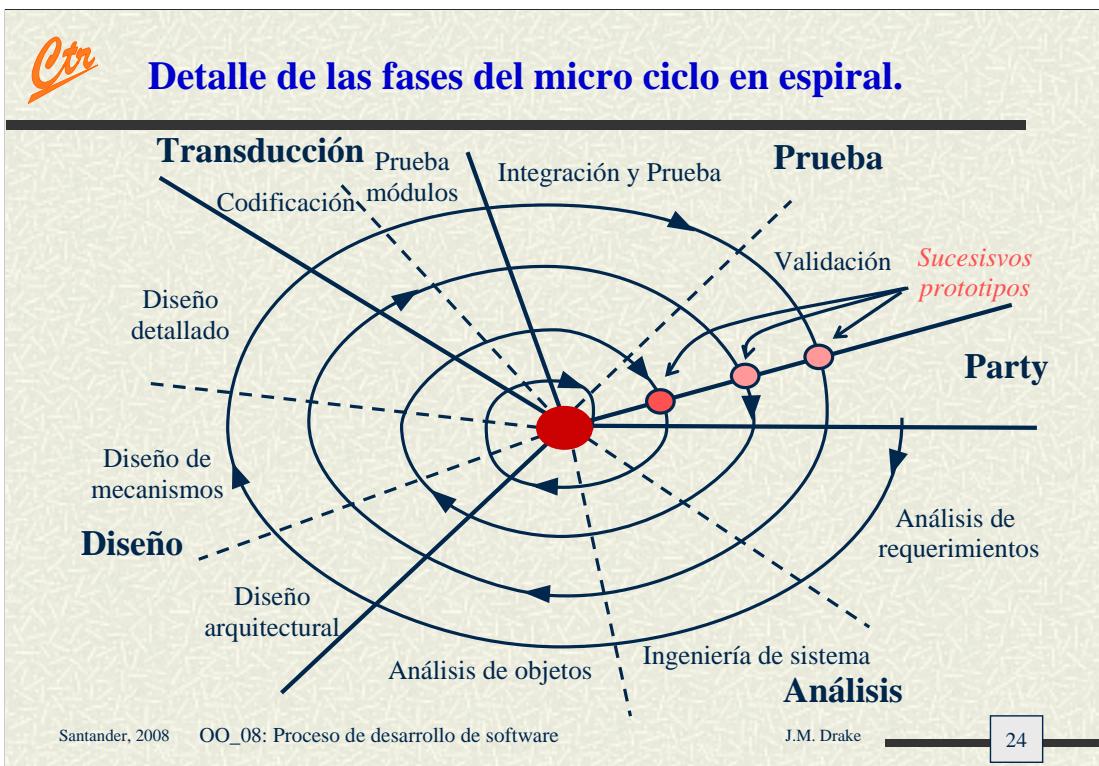
La fase “Party” es donde se establece el objetivo del microciclo y se planifican las actividades a ser realizadas.

La fase “Análisis” define los aspectos esenciales del prototipo que se desarrolla.

La fase “Diseño” optimiza el modelo de análisis eligiendo la arquitectura de diseño adecuada, seleccionando la tecnología y aplicándola al modelo de análisis.

En la fase “Transducción” se genera manualmente o automáticamente el código, se compila y enlaza las nuevas piezas para constituir un prototipo ejecutable.

La fase “Prueba” integra las piezas generadas, aplica los vectores de pruebas previos y los generados específicamente en este microciclo y genera realiza un análisis del estado actual del desarrollo.



Cada iteración del microciclo del proceso en espiral ROPES supone realizar las 4 fases que ya se han descrito. En este gráfico se detallan sus subfases:

- Fase Party: Se evalúan las características del proyecto que convienen abordar, se establece el objetivo del microciclo y se planifican las actividades a ser realizadas.
- Fase Análisis: Define los aspectos esenciales del prototipo que se desarrolla. Por esencial se entienden “el que” es el sistema, esto es, aquellas propiedades que en su ausencia se considera el sistema erróneo o incompleto. Típicamente es una vista de caja negra en la que los detalles sobre sobre la estructura interna detallada es irrelevante.
- Fase Diseño: Se optimiza el modelo de análisis eligiendo la arquitectura de diseño adecuada, seleccionando la tecnología y aplicándola al modelo de análisis. En el diseño se justifica y describe “el como” las características establecidas en el análisis son implementadas. Corresponde a una vista mas detallada, en la que los detalles son relevantes en esta fase. Un diseño es una solución particular de las muchas posibles, y lo mas relevante de esta fase es establecer y satisfacer las características de QoS que deben ser optimizadas y en función de las cuales se justifica la solución.
- Fase Transducción: Concerne a la construcción de la implementación correcta de los elementos estructurales propuestos en el diseño. Se incluye la generación manual o automática del código, se compila y enlaza las nuevas piezas para constituir un prototipo ejecutable.
- Fase Prueba: Se integra el prototipo a partir de las piezas y módulos generados, aplica los vectores de pruebas previos y los generados específicamente en este microciclo y genera realiza un análisis del estado actual del desarrollo.



Fase Party

- # Es la fase de organización y reflexión de cada ciclo de iteración:
 - En el primer ciclo se formulan:
 - La planificación general.
 - El ámbito del proyecto.
 - El plan de gestión de configuraciones.
 - El plan de reuso.
 - El conjunto de casos de usos básicos.
 - En los posteriores ciclos, se organiza la iteración que se inicia, e incluye:
 - La planificación.
 - La propuesta de arquitectura.
 - La secuencia de actividades
 - El objetivo del prototipo que se va a desarrollar.

Uno de los errores mas serios que se pueden cometer en el desarrollo de un proyecto es que no se realice la valoración del estado del proyecto y no se ajuste su ejecución de acuerdo con sus estado. Esto supone dos aspectos igualmente importante: tener capacidad de evaluación del estado, como capacidad de reconducir el proceso de acuerdo con el resultado de la evaluación, a través de reasignar recursos, replanificar actividades, reducir o incrementar el ámbito, etc..

Debido a que la selección e implementación de una arquitectura correcta es de gran importancia para el éxito global del proyecto, es muy importante evaluar al inicio de cada ciclo la correctitud de la arquitectura propuesta y si debe ser cambiada. Para ello es necesario estimar:

- Si la arquitectura permite conseguir las características QoS que se requieren.
- Si la arquitectura soporta razonablemente la evolución y crecimiento de la aplicación.

Si en cada ciclo se requiere continuos cambios estructurales de los elementos ya desarrollados, es razonable pensar que la arquitectura elegida no es la adecuada para la aplicación.



Subfase Análisis de Requerimientos (Análisis)

- # Se identifican y detallan los casos de uso concretos que van a ser implementados en el prototipo actual.
- # Existen dos modos para describir un uso de caso:
 - A través de un conjunto de escenarios que describen las posibles situaciones de interacción entre agentes y sistema.
 - Mediante especificación detallada del caso de uso a través de métodos formales (lenguaje Z, lógica temporal, diagramas de estados UML, o diagramas de actividad UML, etc.) o incluso informales (texto).
- # Productos de esta fase son:
 - Diagramas de clases de uso.
 - Diagramas secuencias.
 - Diagramas de estados.
 - Descripciones textuales.

En la fase de Requirements analysis, los requerimientos del prototipo que se va a desarrollar en la iteración, se identifican y se describen con detalle. Los casos de uso se ha seleccionado en la fase Party, pero hasta esta fase no se detallan en todos sus aspectos.

Hay dos modos básicos de describir un caso de uso:

- Por medio de ejemplos, a través de un conjunto (posiblemente amplio) de escenarios que describen la posibilidades de interacción y todas su variantes. Los escenarios son habitualmente descritos mediante diagramas de secuencias. La ventaja de este método es que es muy accesible a los clientes no técnicos. La principal dificultad es que a menudo se requiere un número excesivo de escenarios para describir de forma completa un caso de uso, y que también es difícil de describir requerimientos negativos.
- Mediante una especificación sistemática utilizando técnicas formales (lenguaje Z, lógica temporal, diagramas de estados, diagramas de actividad, etc) o descripciones textuales. La ventaja de este método es mas preciso y conciso. El inconveniente es que suelen ser difíciles de comprender por los usuarios no técnicos.

La solución mas adecuada es el uso mixto de ambos métodos.

 **Subfase Ingeniería de Sistemas (Análisis).**

- # Consiste en el diseño al mas alto nivel de la arquitectura de la aplicación.
- # Trata de identificar los grandes elementos estructurales (subsistemas y componente) y definir su especificación.
- # Es una fase optativa, requerida si el sistema es complejo, o si va a ser desarrollado por varios grupos.
- # Actividades básicas de esta fase son:
 - Definir la arquitectura de subsistemas.
 - Definir las interfaces de los subsistemas y los protocolos de interacción.
 - Definir como los subsistemas colaboran para realizar al sistema.
 - Descomponer los casos de uso del sistema en casos de usos y requerimientos de los subsistemas.
- # Los principales productos son:
 - Los diagramas de subsistemas.
 - La especificación de requerimientos de los subsistemas.

Santander, 2008 OO_08: Proceso de desarrollo de software J.M. Drake 27

La fase de Ingeniería de sistemas es el diseño arquitectural al máximo nivel del prototipo que se desarrolla en la iteración actual. Es una subfase opcional que solo se ejecuta en sistemas complejos en los que van a ser desarrollados por varios grupos de trabajo. El propósito de la fase es descomponer el sistema en módulos independientemente especificados (subsistemas) que puedan ser desarrollados independientemente por los diferentes equipos.

Las actividades primarias de esta fase son:

- Definir la arquitectura de subsistemas.
- Definir las interfaces y los protocolos de interacción de los subsistemas definidos.
- Definir como los subsistemas colaboran entre sí para implementar los casos de uso del sistema, especificando los papeles que juegan cada subsistema en la colaboración, pero sin detallar su estructura interna.
- Descomponer los casos de uso y los requerimientos del sistema en casos de uso y en requerimientos de los subsistemas.

El producto primario usado en esta fase son los diagramas de subsistema. Estos son diagramas de clase UML en los que los elementos primarios son subsistemas, controladores, interfaces, actores y diagramas de secuencia de alto nivel. Sobre cada subsistema definido en esta fase debe ser aplicado y generar los mismos productos que se desarrollaron en la fase de análisis de requerimiento para el sistema completo.



Subfase Análisis de Objetos (Análisis).

- # Implementa los casos de uso a través de la definición de conjuntos de objetos y de colaboraciones entre ellos.
- # Las clases de objetos se organizan mediante carpetas en función del dominio al que corresponde su semántica.
- # Debe implementarse la funcionalidad esencial y dejar para el diseño los aspectos inducidos por los requerimientos de QoS.
- # El análisis debe ser verificado en nanociclos a través de herramientas de ejecución del modelo o con diagramas de secuencias relativos a la implementación de los casos de uso.
- # Los productos generados en esta fase son diagramas de clases organizados por dominios y diagramas de secuencias que documentan sus colaboraciones.

Un caso de uso es una caja negra que representa la funcionalidad del sistema a través de las interacciones de los actores y el sistema. En esta fase se establece la estructura interna del sistema. Cada caso de uso se concibe como la colaboración de un conjunto de objetos que responden al dominio o dominios de la aplicación, que deben ser identificados y caracterizada sus interfaces en esta fase.

Si la fase previa de ingeniería de sistemas está presente, esta fase debe realizarse separadamente por subsistema.

Debe tenerse cuidado de minimizar la introducción de aspectos relativos al diseño. Solo debe capturarse la funcionalidad esencial, y dejar los aspectos relativos a los requerimientos de QoS para próximas fases.

Debe plantearse si el esquema de objetos que resulta del análisis es correcto, y para ello, en sucesivas fases de nanociclos, deben validarse la consistencia y completitud de la estructura que se propone. Si se disponen herramientas de ejecución del modelo, esto puede automatizarse, en caso contrario, pueden realizarse a través de familias de diagramas de secuencias que ilustren los mecanismos de colaboración para los escenarios contemplados en los casos de uso.

Los productos que se generan en esta fase son los diagramas de clases que modelan los objetos definidos, y los diagramas de secuencias que ilustran sus colaboraciones.



Fase Diseño Arquitectural.

- De acuerdo con las características de la aplicación, se elaboran las vistas que corresponden a los aspectos arquitecturales:
 - Vista de Subsistemas y Componentes.
 - Vista de Concurrencia y Recursos.
 - Vista de Distribución.
 - Vista de Seguridad y Fiabilidad.
 - Vista de Despliegue.
- Esta fase se realiza fundamentalmente incorporando patrones conocidos relativos a los aspectos que se desean optimizar.
- Los productos que se generan en esta fase son diagramas de clases que representan la estructura y diagramas de secuencias que describen las colaboraciones.

Hay muchas formas de definir el término arquitectura. En este curso la entendemos como el conjunto de estrategias y decisiones de diseño que afectan globalmente al sistema. La arquitectura se refiere principalmente a la organización estructural del sistema, y solo implícitamente como objetivo final tiene en cuenta los aspectos de comportamiento y funcionales del sistema.

La arquitectura se puede diseñar a base de estudiar cinco vista complementarias que hacen referencia a aspectos estructurales típicos en los sistemas:

- Vista de subsistemas y componentes: Hace referencia a los grandes módulos (subsistemas y componentes) con que se construyen la aplicación.
- Vista de Concurrencia y Recursos: Hace referencia a los niveles de concurrencia que se deben incorporar en la aplicación y en los recursos necesarios para una operación concurrente segura.
- Vista de Seguridad y Fiabilidad: Hace referencia a los niveles de redundancia necesarios para hacer fiable el sistema y a la gestión de los fallos que puedan producirse en la aplicación.
- Vista de Distribución: Hace referencia a como se establecen diferentes espacios de direcciones, como se distribuyen los módulos software entre ellos, y los mecanismos, formatos y protocolos de colaboración entre ellos.
- Vista de Despliegue: Hace referencia a la forma de mapear los componentes software en los diferentes elementos físicos (procesadores, discos, dispositivos, etc.) que constituyen la plataforma hardware del sistema.

La representación del diseño arquitectural utiliza los tipos de vistas propios de un descripción estructural, esto es, diagramas de clases para definir los objetos y diagramas de secuencia para describir las colaboraciones entre ellas.



Subfase Diseño de Mecanismos

- # Hace referencia a la optimización de colaboraciones entre objetos.
- # Es similar a la fase de diseño arquitectural salvo que su ámbito se reduce a solo un grupo muy reducido de clases.
- # Dado que las posibilidades de colaboración son muy reducidas, debe abordarse buscando patrones ya conocidos.
- # Los productos que genera son también diagramas de clases y diagramas de colaboración. En este nivel suelen ser útiles los diagramas de estados.

La fase de diseño de mecanismos hace referencia a la optimización de colaboraciones en grupos individuales reducidos. Es un tipo de tarea similar a la realizada en la fase de diseño de la arquitectura, salvo que se reduce localmente a un mecanismo de colaboración.

Al igual que en el diseño de la arquitectura, el diseño de los mecanismos debería basarse en patrones de diseño, cuyos beneficios y efectos sean bien conocidos, sin embargo estos patrones son diferentes de los patrones estructurales.

La vista que se genera en esta fase de diseño de mecanismos se compone de diagramas de clases que completan la definición de los objetos que intervienen, así como diagramas de descripción de las interacciones tales como diagramas de secuencias, de colaboración y diagramas de estados. A este nivel, estos dos últimos tipos de diagramas son muy útiles.

Subfase Diseño Detallado.

- # Concerne con la elaboración interna de los los objetos y clases cuya funcionalidad e interfaces han sido definidos en las fases anteriores.
- # Su ámbito se reduce a cada clase de forma independiente.
- # Los aspectos típicos que se diseñan y optimizan, son:
 - Estructuras de datos.
 - Elaboración y descomposición de algoritmos.
 - Optimización de la máquina de estados de la clase.
 - Implementación de las asociaciones.
 - Aspectos relativos a la visibilidad y encapsulación.
 - Garantizar el cumplimiento en fase de ejecución de las precondiciones (en particular de los rangos de las variables y parámetros).
- # Esta fase da lugar a una definición completa de los elementos estructurales de las clases y de sus operaciones y métodos a través de diagramas de estados y actividad.

Santander, 2008 OO_08: Proceso de desarrollo de software J.M. Drake 31

La fase de diseño detallado elabora los aspectos internos de los objetos y clases, y su ámbito se reduce a cada clase de forma independiente.

Los aspectos de la clases que son habitualmente tratados en esta fase son:

- La estructuras de datos internas de las fases.
- La implementación y optimización de los algoritmos y su descomposición cuando son complejos.
- La optimización de la máquina de estados de los objetos.
- La estrategias de elaboración de los objetos.
- Las estructuras con las que se elaboran las asociaciones, en particular cuando son múltiples y requieren mecanismos contenedores.
- La implementación de las visibilidades, en especial cuando son cruzadas y no son directamente implementables, así como los criterios de modularización y encapsulamiento de las clases.
- Estudio de los mecanismos de garantizar en tiempo de ejecución que las precondiciones son satisfechas y como responder cuando no se satisfacen.

Hay muchos criterios y guías para realizar el diseño de las clases, y suele corresponder con las estrategias de programación básica. No suelen basarse en patrones sino en bloques de sentencias proporcionadas por el lenguaje.

El resultado de esta fase son clases completamente definidas, en las que su estructuras de datos internas quedan perfectamente definidas, así como los diagramas de estados y actividades que especifican las operaciones y métodos de su interfaces.

 **Fase Transducción y Elaboración**

- # Hace referencia a la construcción correcta de los elementos que se han diseñado.
- # Incluye las tareas:
 - Generación del código: codificación en lenguaje fuente (ya sea manualmente o automáticamente), compilación, enlazado, e instalación en el entorno de ejecución.
 - La prueba de que el código opera correctamente.
- # Los productos básicos de esta fase son:
 - Código fuente generado de los elementos diseñados.
 - Plan de prueba del funcionamiento del código.
 - Informe de los módulos generados.
 - Componentes compilados y probados.

Santander, 2008 OO_08: Proceso de desarrollo de software J.M. Drake 32

La fase de transducción concierne con la correcta construcción de los elementos arquitecturales que trabajen correctamente.

Esta fase incluye la generación del código (que puede generarse automáticamente por herramientas o manualmente, o por combinación de ambas que es lo mas habitual, la prueba a nivel de código fuente de los algoritmos e interacciones, la compilación y compatibilidad de enlazado de los módulos compilados y la instalación y ejecución de del código ejecutable generado. Así mismo, debe establecerse un plan de verificación de los elementos construidos.

Los principales productos a que da lugar la fase de traducción y elaboración son:

- El código fuente de los elementos definidos en el diseño.
- Un plan de prueba , de las asociaciones, de las operaciones y de las interacciones.
- Un informe sobre los módulos elaborados y su forma de utilización.
- Los componentes de software compilados.

 **Fase Test**

- # Concerne con la construcción del prototipo planificado para la iteración y su validación.
- # Se compone de dos fases:
 - Integración y prueba que hace referencia al acoplamiento de los elementos arquitecturales del prototipo.
 - Validación que hace referencia a la comprobación de que el prototipo satisface (o no) la funcionalidad y características de QoS previstas.
- # Los principales productos que se generan son:
 - Plan de integración e informe de los resultados que se obtienen.
 - Plan de validación e informe de los resultados que se obtienen.
 - Elaboración y prueba de un prototipo ejecutable.
 - Informe de errores y defectos.

Santander, 2008 OO_08: Proceso de desarrollo de software J.M. Drake 33

La fase de Test construye (Integración y prueba) el prototipo con los elementos diseñados y verifica si el prototipo satisface o no (Valida) los requerimiento funcionales y de QoS establecidos para él.

La prueba se limita a demostrar que las interfaces de los elementos estructurales operan correctamente y satisfacen las restricciones establecidas. El proceso suele ser incremental, se van integrando sucesivos elementos estructurales y se va verificando que operan correctamente.

Si se encuentran defectos y problemas durante esta fase, deben ser reportados y analizados ya que son el objetivo final de la iteración, y deben ser considerados como la base de las siguientes iteraciones.



Gestión de un proyecto orientado a objetos

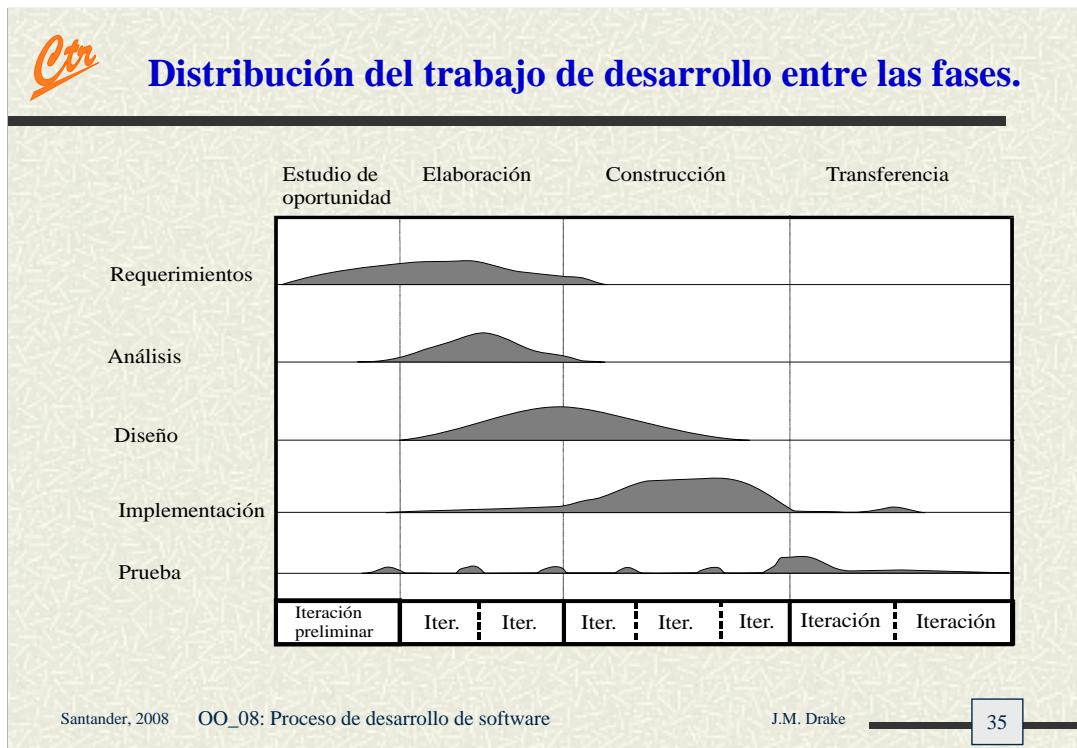
Desde el punto de vista de la gestión, las fases de un proyecto son:

- # Estudio de oportunidad:
 - (Estudio del mercado, especificación del producto y definición del alcance del producto).
- # Fase de Elaboración.
 - (Especificación detallada, Planificación de las actividades, Diseño y Validación de la arquitectura)
- # Fase de Construcción.
 - (Diseño detallado de clases, Codificación e Integración de los componentes)
- # Fase de Transferencia:
 - (Fabricación del prototipo final, fabricación industrial, soporte técnico y mantenimiento)

La gestión del desarrollo de un sistema software trata los aspectos financieros, estratégicos, comerciales y humanos:

Se puede descomponer cuatro fases:

- Estudio de la Oportunidad: que comprende el estudio del mercado, la especificación del producto final y la definición del alcance del proyecto.
- Fase de Elaboración: que corresponde a la especificación de las particularidades del producto, a la planificación de las actividades, a la determinación de los recursos, al diseño, y a la validación de la arquitectura.
- Fase de Construcción: agrupa la realización del producto, la adaptación de la arquitectura, la codificación y la integración)
- Fase de Transferencia: que corresponde a la fabricación del prototipo final, de la fabricación industrial, distribución entre usuarios, soporte técnico y mantenimiento.



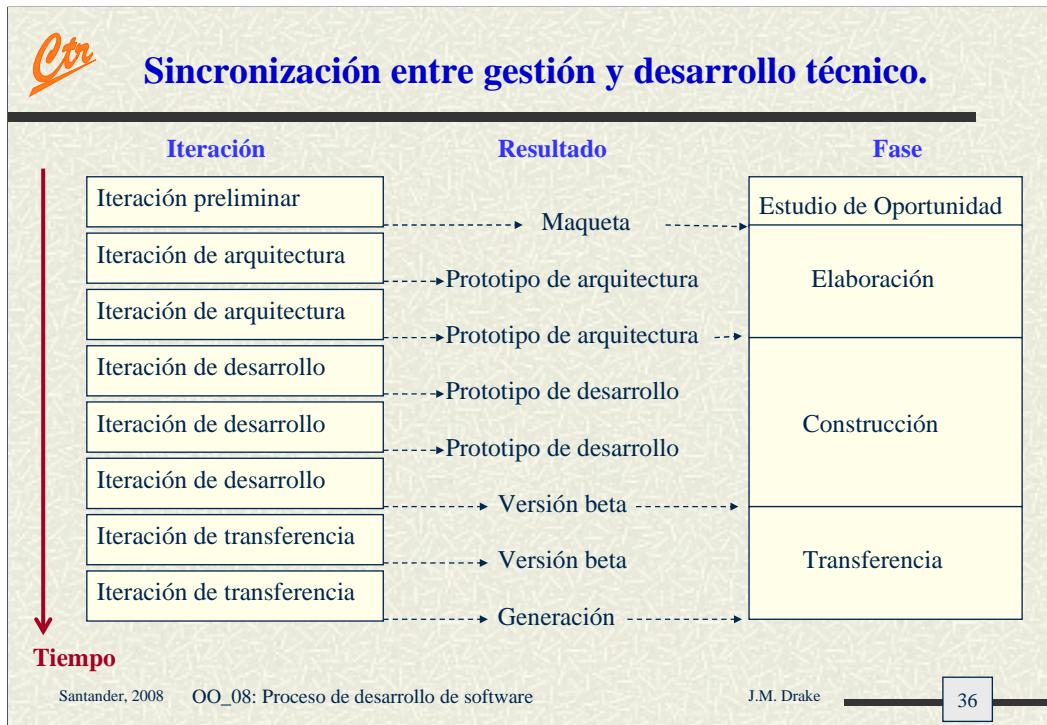
La definición de los requerimientos se realiza básicamente entre las fases de Estudio de oportunidad y de Elaboración. Durante el Estudio de viabilidad se identifican los casos de uso básicos y los restantes secundarios se establecen en la fase de Elaboración.

El análisis está claramente centrado en la fase de Elaboración. El problema se comprende y su estructura se formula mediante un modelo abstracto.

El diseño se realiza entre las fases de Elaboración, en la que se obtienen un modelo concreto de la solución y se formula y estabiliza la arquitectura. Durante la fase de Construcción se definen los detalles de los componentes, se organizan los módulos y se despliegan sobre la plataforma.

La implementación se desarrolla básicamente durante la fase de construcción en la que se codifican e integran la mayoría de los componentes del sistema.

Por último, las pruebas se distribuyen a lo largo de todas las fases coincidiendo con las finalizaciones de las iteraciones.



El proceso de desarrollo técnico y las fases de gestión del proyecto se sincronizan al final de cada fase, sobre el resultado tangible de una iteración.

El estudio de oportunidad puede necesitar un prototipo para estudiar la viabilidad técnica del proyecto y acabar de definir su especificación. La toma de decisión de proseguir o no con el proyecto se toma teniendo en cuenta la evaluación de la maqueta entregada por el equipo de desarrollo.

La fase de Elaboración debe concluir con la definición de una arquitectura estable, y esto suele requerir varios prototipos. La fase de Elaboración concluye con una arquitectura validada por un prototipo.

La fase de Construcción progresó según se van desarrollando los sucesivos prototipos de desarrollo. Cuyos objetivos son afianzar los progresos y garantizar la estabilidad del proyecto. La fase de Construcción termina con el prototipo definitivo en su versión beta, que se entrega a los usuarios para que lo prueben bajo condiciones reales.

La fase de Transferencia comienza con la entrega de la versión beta y prosigue con los prototipos que corregen los defectos detectados por los usuarios. La fase de Transferencia termina cuando el producto llega a la versión de producción.

Lectura 2. Introducción al desarrollo del software

Software libre

Josep Anton Pérez López
Lluís Ribas i Xirgo

XP04/90793/00018



Introducción al desarrollo del software



Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.
Formación de Posgrado EDICT_GUIA_MEX Editorial Guías México

David Megías Jiménez

Coordinador

Ingeniero en Informática por la UAB.
Magíster en Técnicas Avanzadas de Automatización de Procesos por la UAB.
Doctor en Informática por la UAB.
Profesor de los Estudios de Informática y Multimedia de la UOC.

Jordi Mas

Coordinador

Ingeniero de software en la empresa de código abierto Ximian, donde trabaja en la implementación del proyecto libre Mono. Como voluntario, colabora en el desarrollo del procesador de textos Abiword y en la ingeniería de las versiones en catalán del proyecto Mozilla y Gnome. Es también coordinador general de Softcatalà. Como consultor ha trabajado para empresas como Menta, Telépolis, Vodafone, Lotus, eresMas, Amena y Terra España.

Josep Anton Pérez López

Autor

Licenciado en Informática por la Universidad Autónoma de Barcelona.
Magíster en el programa Gráficos, Tratamiento de Imágenes y de Inteligencia Artificial, por la UAB.
Actualmente trabaja como profesor en un centro de educación secundaria.

Lluís Ribas i Xirgo

Autor

Licenciado en Ciencias-Informática y doctorado en Informática por la Universidad Autònoma de Barcelona (UAB).
Profesor titular del Departamento de Informática de la UAB. Consultor de los Estudios de Informática y Multimedia en la Universitat Oberta de Catalunya (UOC).

Primera edición: marzo 2004

© Fundació per a la Universitat Oberta de Catalunya

Av. Tibidabo, 39-43, 08035 Barcelona

Material realizado por Eureka Media, SL

© Autores: Josep Antoni Pérez López y Lluís Ribas i Xirgo

Depósito legal: B-7.600-2004

ISBN: 84-9788-119-2

Se garantiza permiso para copiar, distribuir y modificar este documento según los términos de la *GNU Free Documentation License, Version 1.2o* cualquiera posterior publicada por la Free Software Foundation, sin secciones invariantes ni textos de cubierta delantera o trasera. Se dispone de una copia de la licencia en el apartado "GNU Free Documentation License" de este documento.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

www.GUÍAS_MEXICO.COM

Índice

Agradecimientos	9
1. Introducción a la programación	11
1.1. Introducción	11
1.2. Un poco de historia de C.....	12
1.3. Entorno de programación en GNU/C	16
1.3.1. Un primer programa en C	17
1.3.2. Estructura de un programa simple	21
1.4. La programación imperativa	23
1.4.1. Tipos de datos básicos	24
1.4.2. La asignación y la evaluación de expresiones	28
1.4.3. Instrucciones de selección	33
1.4.4. Funciones estándar de entrada y de salida	35
1.5. Resumen	40
1.6. Ejercicios de autoevaluación	41
1.6.1. Solucionario	42
2. La programación estructurada	45
2.1. Introducción	45
2.2. Principios de la programación estructurada	48
2.3. Instrucciones iterativas	49
2.4. Procesamiento de secuencias de datos	52
2.4.1. Esquemas algorítmicos: recorrido y búsqueda	53
2.4.2. Filtros y tuberías	59
2.5. Depurado de programas	62
2.6. Estructuras de datos	66
2.7. Matrices	67
2.7.1. Declaración	68
2.7.2. Referencia	70
2.7.3. Ejemplos	71
2.8. Estructuras heterogéneas	74
2.8.1. Tuplas	74
2.8.2. Variables de tipo múltiple	77

2.9.	Tipos de datos abstractos	79
2.9.1.	Definición de tipos de datos abstractos	80
2.9.2.	Tipos enumerados	81
2.9.3.	Ejemplo	82
2.10.	Ficheros	84
2.10.1.	Ficheros de flujo de bytes	84
2.10.2.	Funciones estándar para ficheros	85
2.10.3.	Ejemplo	90
2.11.	Principios de la programación modular	92
2.12.	Funciones	92
2.12.1.	Declaración y definición	92
2.12.2.	Ámbito de las variables	96
2.12.3.	Parámetros por valor y por referencia	99
2.12.4.	Ejemplo	101
2.13.	Macros del preprocesador de C	103
2.14.	Resumen	104
2.15.	Ejercicios de autoevaluación	106
2.15.1.	Solucionario	110

3. Programación avanzada en C. Desarrollo

eficiente de aplicaciones	125	
3.1.	Introducción	125
3.2.	Las variables dinámicas	127
3.3.	Los apuntadores	129
3.3.1.	Relación entre apuntadores y vectores	132
3.3.2.	Referencias de funciones	135
3.4.	Creación y destrucción de variables dinámicas	137
3.5.	Tipos de datos dinámicos	139
3.5.1.	Cadenas de caracteres	140
3.5.2.	Listas y colas	145
3.6.	Diseño descendente de programas	156
3.6.1.	Descripción	157
3.6.2.	Ejemplo	158
3.7.	Tipos de datos abstractos y funciones asociadas	159
3.8.	Ficheros de cabecera	166
3.8.1.	Estructura	166
3.8.2.	Ejemplo	169
3.9.	Bibliotecas	172
3.9.1.	Creación	172
3.9.2.	Uso	173
3.9.3.	Ejemplo	174
3.10.	Herramienta make	175
3.10.1.	Fichero makefile	176

3.11. Relación con el sistema operativo. Paso de parámetros a programas	179
3.12. Ejecución de funciones del sistema operativo	181
3.13. Gestión de procesos	183
3.13.1. Definición de proceso	184
3.13.2. Procesos permanentes	185
3.13.3. Procesos concurrentes	188
3.14. Hilos	189
3.14.1. Ejemplo	190
3.15. Procesos	193
3.15.1. Comunicación entre procesos	198
3.16. Resumen	202
3.17. Ejercicios de autoevaluación	204
3.17.1. Solucionario	211
4. Programación orientada a objetos en C++	219
4.1. Introducción	219
4.2. De C a C++	221
4.2.1. El primer programa en C++	221
4.2.2. Entrada y salida de datos	223
4.2.3. Utilizando C++ como C	226
4.2.4. Las instrucciones básicas	227
4.2.5. Los tipos de datos	227
4.2.6. La declaración de variables y constantes	230
4.2.7. La gestión de variables dinámicas	231
4.2.8. Las funciones y sus parámetros	235
4.3. El paradigma de la programación orientada a objetos	240
4.3.1. Clases y objetos	242
4.3.2. Acceso a objetos	246
4.3.3. Constructores y destructores de objetos	250
4.3.4. Organización y uso de bibliotecas en C++	257
4.4. Diseño de programas orientados a objetos	261
4.4.1. La homonimia	262
4.4.2. La herencia simple	267
4.4.3. El polimorfismo	273
4.4.4. Operaciones avanzadas con herencia	279
4.4.5. Orientaciones para el análisis y diseño de programas	281
4.5. Resumen	285
4.6. Ejercicios de autoevaluación	286
4.6.1. Solucionario	287

5. Programación en Java	309
5.1. Introducción	309
5.2. Origen de Java	312
5.3. Características generales de Java	313
5.4. El entorno de desarrollo de Java	317
5.4.1. La plataforma Java	318
5.4.2. Mi primer programa en Java	319
5.4.3. Las instrucciones básicas y los comentarios	320
5.5. Diferencias entre C++ y Java	321
5.5.1. Entrada/salida	321
5.5.2. El preprocesador	324
5.5.3. La declaración de variables y constantes ..	325
5.5.4. Los tipos de datos	325
5.5.5. La gestión de variables dinámicas	326
5.5.6. Las funciones y el paso de parámetros	328
5.6. Las clases en Java	329
5.6.1. Declaración de objetos	330
5.6.2. Acceso a los objetos	331
5.6.3. Destrucción de objetos	332
5.6.4. Constructores de copia	333
5.6.5. Herencia simple y herencia múltiple	333
5.7. Herencia y polimorfismo	334
5.7.1. Las referencias this y super	334
5.7.2. La clase Object	334
5.7.3. Polimorfismo	335
5.7.4. Clases y métodos abstractos	335
5.7.5. Clases y métodos finales	336
5.7.6. Interfaces	337
5.7.7. Paquetes	339
5.7.8. El API (applications programming interface) de Java	340
5.8. El paradigma de la programación orientada a eventos	341
5.8.1. Los eventos en Java	342
5.9. Hilos de ejecución (<i>threads</i>)	344
5.9.1. Creación de hilos de ejecución	345
5.9.2. Ciclo de vida de los hilos de ejecución	348
5.10. Los applets	349
5.10.1. Ciclo de vida de los applets	350
5.10.2. Manera de incluir applets en una página HTML	351
5.10.3. Mi primer applet en Java	352
5.11. Programación de interfaces gráficas en Java	353
5.11.1. Las interfaces de usuario en Java	354

5.11.2. Ejemplo de applet de Swing	355
5.12. Introducción a la información visual	356
5.13. Resumen	357
5.14. Ejercicios de autoevaluación	358
5.14.1. Solucionario	359
Glosario	373
Bibliografia	381

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX • Editorial Guías México

Agradecimientos

Los autores agradecen a la Fundación para la Universitat Oberta de Catalunya (<http://www.uoc.edu>) la financiación de la primera edición de esta obra, enmarcada en el Máster Internacional en Software Libre ofrecido por la citada institución.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX • Editorial Guías México

1. Introducción a la programación

1.1. Introducción

En esta unidad se revisan los fundamentos de la programación y los conceptos básicos del lenguaje C. Se supone que el lector ya tiene conocimientos previos de programación bien en el lenguaje C, bien en algún otro lenguaje.

Por este motivo, se incide especialmente en la metodología de la programación y en los aspectos críticos del lenguaje C en lugar de hacer hincapié en los elementos más básicos de ambos aspectos.

El conocimiento profundo de un lenguaje de programación parte no sólo del entendimiento de su léxico, de su sintaxis y de su semántica, sino que además requiere la comprensión de los objetivos que motivaron su desarrollo. Así pues, en esta unidad se repasa la historia del lenguaje de programación C desde el prisma de la programación de los computadores.

Los programas descritos en un lenguaje de programación como C no pueden ser ejecutados directamente por ninguna máquina. Por tanto, es necesario disponer de herramientas (es decir, programas) que permitan obtener otros programas que estén descritos como una secuencia de órdenes que sí que pueda ejecutar directamente algún computador.

En este sentido, se describirá un entorno de desarrollo de software de libre acceso disponible tanto en plataformas Microsoft como GNU/Linux. Dado que las primeras requieren de un sistema operativo que no se basa en el software libre, la explicación se centrará en las segundas.

El resto de la unidad se ocupará del lenguaje de programación C en lo que queda afectado por el paradigma de la programación imperativa y su modelo de ejecución. El modelo de ejecución trata de la

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

Nota

Una plataforma es, en este contexto, el conjunto formado por un tipo de ordenador y un sistema operativo.

forma como se llevan a cabo las instrucciones indicadas en un programa. En el paradigma de la programación imperativa, las instrucciones son órdenes que se llevan a cabo de forma inmediata para conseguir algún cambio en el estado del procesador y, en particular, para el almacenamiento de los resultados de los cálculos realizados en la ejecución de las instrucciones. Por este motivo, en los últimos apartados se incide en todo aquello que afecta a la evaluación de expresiones (es decir, al cálculo de los resultados de fórmulas), a la selección de la instrucción que hay que llevar a cabo y a la obtención de datos o a la producción de resultados.

En esta unidad, pues, se pretende que el lector alcance los objetivos siguientes:

1. Repasar los conceptos básicos de la programación y el modelo de ejecución de los programas.
2. Entender el paradigma fundamental de la programación imperativa.
3. Adquirir las nociones de C necesarias para el seguimiento del curso.
4. Saber utilizar un entorno de desarrollo de software libre para la programación en C (GNU/Linux y útiles GNU/C).

1.2. Un poco de historia de C

El lenguaje de programación C fue diseñado por Dennis Ritchie en los laboratorios Bell para desarrollar nuevas versiones del sistema operativo Unix, allá por el año 1972. De ahí, la fuerte relación entre el C y la máquina.

Nota

Como curiosidad cabe decir que bastaron unas trece mil líneas de código C (y un millar escaso en ensamblador) para programar el sistema operativo Unix del computador PDP-11.

El lenguaje ensamblador es aquel que tiene una correspondencia directa con el lenguaje máquina que entiende el procesador. En otras

palabras, cada instrucción del lenguaje máquina se corresponde con una instrucción del lenguaje ensamblador.

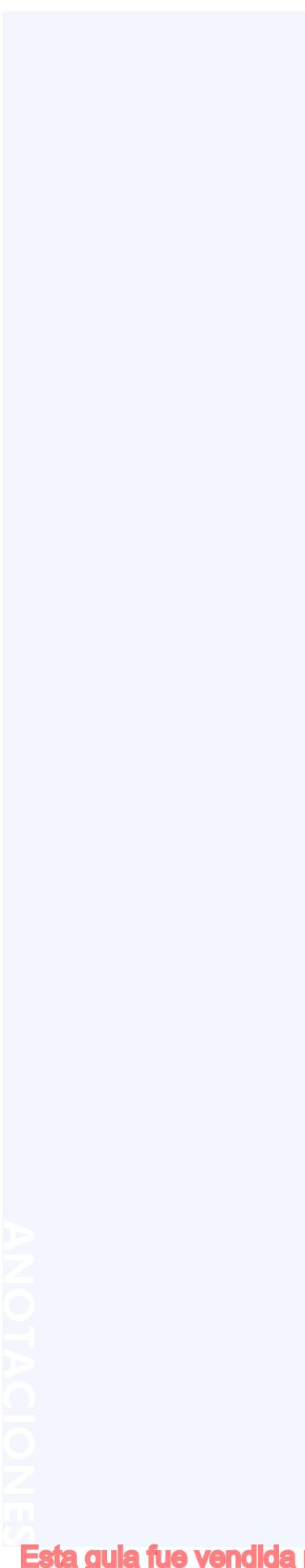
Por el contrario, las instrucciones del lenguaje C pueden equivaler a pequeños programas en el lenguaje máquina, pero de frecuente uso en los algoritmos que se programan en los computadores. Es decir, se trata de un lenguaje en el que se emplean instrucciones que sólo puede procesar una máquina abstracta, que no existe en la realidad (el procesador real sólo entiende el lenguaje máquina). Por ello, se habla del C como lenguaje de alto nivel de abstracción y del ensamblador como lenguaje de bajo nivel.

Esta máquina abstracta se construye parcialmente mediante un conjunto de programas que se ocupan de gestionar la máquina real: el **sistema operativo**. La otra parte se construye empleando un programa de traducción del lenguaje de alto nivel a lenguaje máquina. Estos programas se llaman **compiladores** si generan un código que puede ejecutarse directamente por el computador, o **intérpretes** si es necesaria su ejecución para llevar a cabo el programa descrito en un lenguaje de alto nivel.

Para el caso que nos ocupa, resulta de mucho interés que el código de los programas que constituyen el sistema operativo sea lo más independiente de la máquina posible. Únicamente de esta manera será viable adaptar un sistema operativo a cualquier computador de forma rápida y fiable.

Por otra parte, es necesario que el compilador del lenguaje de alto nivel sea extremadamente eficiente. Para ello, y dado los escasos recursos computacionales (fundamentalmente, capacidad de memoria y velocidad de ejecución) de los ordenadores de aquellos tiempos, se requiere que el lenguaje sea simple y permita una traducción muy ajustada a las características de los procesadores.

Éste fue el motivo de que en el origen de C estuviera el lenguaje denominado B, desarrollado por Ken Thompson para programar Unix para el PDP-7 en 1970. Evidentemente, esta versión del sistema operativo también incluyó una parte programada en ensamblador, pues había operaciones que no podían realizarse sino con la máquina real.



Queda claro que se puede ver el lenguaje C como una versión posterior (y mejorada con inclusión de tipos de datos) del B. Más aún, el lenguaje B estaba basado en el BCPL. Éste fue un lenguaje desarrollado por Martín Richards en 1967, cuyo tipo de datos básico era la palabra *memoria*; es decir, la unidad de información en la que se divide la memoria de los computadores. De hecho, era un lenguaje ensamblador mejorado que requería de un compilador muy simple. A cambio, el programador tenía más control de la máquina real.

A pesar de su nacimiento como lenguaje de programación de sistemas operativos, y, por tanto, con la capacidad de expresar operaciones de bajo nivel, C es un lenguaje de programación de **propósito general**. Esto es, con él es posible programar algoritmos de aplicaciones (conjuntos de programas) de muy distintas características como, por ejemplo, software de contabilidad de empresas, manejo de bases de datos de reservas de aviones, gestión de flotas de transporte de mercancías, cálculos científicos, etcétera.

Bibliografía

La definición de las reglas sintácticas y semánticas del C aparece en la obra siguiente:

B.W. Kernighan; D.M. Ritchie (1978). *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall. Concretamente en el apéndice “C Reference Manual”.



La relativa simplicidad del lenguaje C por su escaso número de instrucciones permite que sus compiladores generen un código en lenguaje máquina muy eficiente y, además, lo convierten en un lenguaje fácilmente transportable de una máquina a otra.

Por otra parte, el repertorio de instrucciones de C posibilita realizar una programación estructurada de alto nivel de abstracción. Cosa que redunda en la programación sistemática, legible y de fácil mantenimiento.

Esta simplicidad, no obstante, ha supuesto la necesidad de disponer para C de un conjunto de funciones muy completas que le confieren una gran potencia para desarrollar aplicaciones de todo tipo. Muchas de estas funciones son estándar y se encuentran disponibles en todos los compiladores de C.

Nota

Una función es una secuencia de instrucciones que se ejecuta de forma unitaria para llevar a cabo alguna tarea concreta. Por lo tanto, a mayor número de funciones ya programadas, menos código habrá que programar.

Las funciones estándar de C se recogen en una biblioteca: la biblioteca estándar de funciones. Así pues, cualquier programa puede emplear todas las funciones que requiera de la misma, puesto que todos los compiladores disponen de ella.

Finalmente, dada su dependencia de las funciones estándar, C promueve una programación modular en la que los propios programadores también pueden preparar funciones específicas en sus programas.

Todas estas características permitieron una gran difusión de C que se normalizó por la ANSI (American National Standards Association) en 1989 a partir de los trabajos de un comité creado en 1983 para "proporcionar una definición no ambigua e independiente de máquina del lenguaje". La segunda edición de Kernighan y Ritchie, que se publicó en 1988, refleja esta versión que se conoce como ANSI-C.

Nota

La versión original de C se conocería, a partir de entonces, como K&R C, es decir, como el C de Kernighan y Ritchie. De esta manera, se distinguen la versión original y la estandarizada por la ANSI.

El resto de la unidad se dedicará a explicar un entorno de desarrollo de programas en C y a repasar la sintaxis y la semántica de este lenguaje.

1.3. Entorno de programación en GNU/C

En el desarrollo de software de libre acceso es importante emplear herramientas (programas) que también lo sean, pues uno de los principios de los programas de libre acceso es que su código pueda ser modificado por los usuarios.

El uso de código que pueda depender de software privado implica que esta posibilidad no exista y, por tanto, que no sea posible considerarlo como libre. Para evitar este problema, es necesario programar mediante software de libre acceso.

GNU (www.gnu.org) es un proyecto de desarrollo de software libre iniciado por Richard Stallman en el año 1984, que cuenta con el apoyo de la Fundación para el Software Libre (FSF).

GNU es un acrónimo recursivo (significa *GNU No es Unix*) para indicar que se trataba del software de acceso libre desarrollado sobre este sistema operativo, pero que no consistía en el sistema operativo. Aunque inicialmente el software desarrollado utilizara una plataforma Unix, que es un sistema operativo de propiedad, no se tardó en incorporar el *kernel* Linux como base de un sistema operativo independiente y completo: el GNU/Linux.

Nota

El *kernel* de un sistema operativo es el software que constituye su núcleo fundamental y, de ahí, su denominación (*kernel* significa, entre otras cosas, 'la parte esencial'). Este núcleo se ocupa, básicamente, de gestionar los recursos de la máquina para los programas que se ejecutan en ella.

El *kernel* Linux, compatible con el de Unix, fue desarrollado por Linus Torvalds en 1991 e incorporado como *kernel* del sistema operativo de GNU un año más tarde.

En todo caso, cabe hacer notar que todas las llamadas *distribuciones* de Linux son, de hecho, versiones del sistema operativo GNU/Linux que, por tanto, cuentan con software de GNU (editor de textos

Emacs, compilador de C, etc.) y también con otro software libre como puede ser el procesador de textos TeX.

Para el desarrollo de programas libres se empleará, pues, un ordenador con el sistema operativo Linux y el compilador de C de GNU (gcc). Aunque se puede emplear cualquier editor de textos ASCII, parece lógico emplear Emacs. Un editor de textos ASCII es aquél que sólo emplea los caracteres definidos en la tabla ASCII para almacenar los textos. (En el Emacs, la representación de los textos se puede ajustar para distinguir fácilmente los comentarios del código del programa en C, por ejemplo.)

Aunque las explicaciones sobre el entorno de desarrollo de software que se realizarán a lo largo de esta unidad y las siguientes hacen referencia al software de GNU, es conveniente remarcar que también es posible emplear herramientas de software libre para Windows, por ejemplo.

En los apartados siguientes se verá cómo emplear el compilador gcc y se revisará muy sucintamente la organización del código de los programas en C.

1.3.1. Un primer programa en C



Un programa es un texto escrito en un lenguaje simple que permite expresar una serie de acciones sobre objetos (instrucciones) de forma no ambigua.

Antes de elaborar un programa, como en todo texto escrito, habremos de conocer las reglas del lenguaje para que aquello que se exprese sea correcto tanto léxica como sintácticamente.

Las normas del lenguaje de programación C se verán progresivamente a lo largo de las unidades correspondientes (las tres primeras).

Además, deberemos procurar que "eso" tenga sentido y exprese exactamente aquello que se desea que haga el programa. Por si no

Nota

El símbolo del dólar (\$) se emplea para indicar que el intérprete de comandos del sistema operativo del ordenador puede aceptar una nueva orden.

Nota

El nombre del fichero que contiene el programa en C tiene extensión ".c" para que sea fácil identificarlo como tal.

Ejemplo

No es lo mismo printf que PrintF.

fuera poco, habremos de cuidar el aspecto del texto de manera que sea posible captar su significado de forma rápida y clara. Aunque algunas veces se indicará alguna norma de estilo, normalmente éste se describirá implícitamente en los ejemplos.

Para escribir un programa en C, es suficiente con ejecutar el emacs:

```
$ emacs hola.c &
```

Ahora ya podemos escribir el siguiente programa en C:

```
#include <stdio.h>
main( )
{
    printf( ""Hola a todos! \n" );
} /* main */
```

Es muy importante tener presente que, en C (y también en C++ y en Java), **se distinguen mayúsculas de minúsculas**. Por lo tanto, el texto del programa tiene que ser exactamente como se ha mostrado a excepción del texto entre comillas y el que está entre los símbolos /* y */.

El editor de textos emacs dispone de menús desplegables en la parte superior para la mayoría de operaciones y, por otra parte, acepta comandos introducidos mediante el teclado. A tal efecto, es necesario teclear también la tecla de control ("CTRL" o "C-") o la de carácter alternativo junto con la del carácter que indica la orden.

A modo de breve resumen, se describen algunos de los comandos más empleados en la siguiente tabla:

Tabla 1.

Comando	Secuencia	Explicación
Files → Find file	C-x, C-f	Abre un fichero. El fichero se copia en un buffer o área temporal para su edición.
Files → Save buffer	C-x, C-s	Guarda el contenido del buffer en el fichero asociado.

Comando	Secuencia	Explicación
Files → Save buffer as	C-x, C-w	Escribe el contenido del buffer en el fichero que se le indique.
Files → Insert file	C-x, C-i	Inserta el contenido del fichero indicado en la posición del texto en el que se encuentre el cursor.
Files → Exit	C-x, C-c	Finaliza la ejecución de emacs.
(cursor movement)	C-a	Sitúa el cursor al principio de la línea.
(cursor movement)	C-e	Sitúa el cursor al final de la línea.
(line killing)	C-k	Elimina la línea (primero el contenido y luego el salto de línea).
Edit → Paste	C-y	Pega el último texto eliminado o copiado.
Edit → Copy	C-y, ..., C-y	Para copiar un trozo de texto, puede eliminarse primero y recuperarlo en la misma posición y, finalmente, en la posición de destino.
Edit → Cut	C-w	Elimina el texto desde la última marca hasta el cursor.
Edit → Undo	C-u	Deshace el último comando.

Nota

Para una mejor comprensión de los comandos se recomienda leer el manual de `emacs` o, en su caso, del editor que se haya elegido para escribir los programas.

Una vez editado y guardado el programa en C, hay que compilarlo para obtener un fichero binario (con ceros y unos) que contenga una versión del programa traducido a lenguaje máquina. Para ello, hay que emplear el compilador gcc:

```
$ gcc -c hola.c
```

Nota

En un tiempo gcc significó compilador de C de GNU, pero, dado que el compilador entiende también otros lenguajes, ha pasado a ser la colección de compiladores de GNU. Por este motivo, es necesario indicar en qué lenguaje se han escrito los programas mediante la extensión del nombre de los ficheros correspondientes. En este caso, con `hola.c`, empleará el compilador de C.

Con ello, se obtendrá un fichero (`hola.o`), denominado fichero objeto. Este archivo contiene ya el programa en lenguaje máquina derivado del programa con el código C, llamado también código fuente. Pero desgraciadamente aún no es posible ejecutar este programa, ya que requiere de una función (`printf`) que se encuentra en la biblioteca de funciones estándar de C.

Para obtener el código ejecutable del programa, será necesario enlazar (en inglés: *link*):

```
$ gcc hola.o -o hola
```

Como la biblioteca de funciones estándar está siempre en un lugar conocido por el compilador, no es necesario indicarlo en la línea de comandos. Eso sí, será necesario indicar en qué fichero se quiere el resultado (el fichero ejecutable) con la opción `-o` seguida del nombre deseado. En caso contrario, el resultado se obtiene en un fichero llamado “`a.out`”.

Habitualmente, el proceso de compilación y enlazado se hacen directamente mediante:

```
$ gcc hola.c -o hola
```

Si el fichero fuente contuviese errores sintácticos, el compilador mostrará los mensajes de error correspondientes y deberemos corregirlos antes de repetir el procedimiento de compilación.

En caso de que todo haya ido bien, dispondremos de un programa ejecutable en un fichero llamado `hola` que nos saludará vehemente cada vez que lo ejecutemos:

```
$ ./hola
Hola a todos!
$
```

Nota

Se debe indicar el camino de acceso al fichero ejecutable para que el intérprete de órdenes pueda localizarlo. Por motivos de seguridad, el directorio de trabajo no se incluye por omisión en el conjunto de caminos de búsqueda de ejecutables del intérprete de comandos.



Este procedimiento se repetirá para cada programa que realicemos en C en un entorno GNU.

1.3.2. Estructura de un programa simple

En general, un programa en C debería estar organizado como sigue:

```
/* Fichero: nombre.c                                     */
/* Contenido: ejemplo de estructura de un programa en C   */
/* Autor: nombre del autor                                */
/* Revisión: preliminar                                  */

/* COMANDOS DEL PREPROCESADOR                           */
/* -inclusión de ficheros de cabeceras                  */
#include <stdio.h>
/* -definición de constantes simbólicas                 */
#define FALSO 0

/* FUNCIONES DEL PROGRAMADOR                            */
main( ) /* Función principal:                         */
{          /* El programa se empieza a ejecutar aquí      */
    ...    /* Cuerpo de la función principal               */
} /* main */
```

En esta organización, las primeras líneas del fichero son comentarios que identifican su contenido, autor y versión. Esto es importante, pues hay que tener presente que el código fuente que creemos debe ser fácilmente utilizable y modificable por otras personas... ¡y por nosotros mismos!

Dada la simplicidad del C, muchas de las operaciones que se realizan en un programa son, en realidad, llamadas a funciones estándar. Así pues, para que el compilador conozca qué parámetros tienen y qué valores devuelven, es necesario incluir en el código de nuestro programa las declaraciones de las funciones que se emplearán. Para ello, se utiliza el comando `#include` del llamado preprocesador de C, que se ocupa de montar un fichero único de entrada para el compilador de C.

Los ficheros que contienen declaraciones de funciones externas a un determinado archivo fuente son llamados *ficheros de cabeceras*

Nota

Los comentarios son textos libres que se escriben entre los dígrafos `/*` y `*/`.

(header files). De ahí que se utilice la extensión ".h" para indicar su contenido.



Las cabeceras, en C, son la parte en la que se declara el nombre de una función, los parámetros que recibe y el tipo de dato que devuelve.

Tanto estos ficheros como el del código fuente de un programa pueden contener definiciones de constantes simbólicas. A continuación presentamos una muestra de estas definiciones:

```
#define VACIO          '\0'      /* El carácter ASCII NUL */
#define NUMERO_OCTAL    0173     /* Un valor octal */
#define MAX_USUARIOS   20
#define CODIGO_HEXA    0xf39b    /* Un valor hexadecimal */
#define PI              3.1416    /* Un número real */
#define PRECISION      1e-10     /* Otro número real */
#define CADENA         "cadena de caracteres"
```

Estas constantes simbólicas son reemplazadas por su valor en el fichero final que se suministra al compilador de C. Es importante recalcar que su uso debe redundar en una mayor legibilidad del código y, por otra parte, en una facilidad de cambio del programa, cuando fuese necesario.

Cabe tener presente que las constantes numéricas enteras descritas en base 8, u octal, deben estar prefijadas por un 0 y las expresadas en base 16, o hexadecimal, por "0x"

Ejemplo

020 no es igual a 20, puesto que este último coincide con el valor veinte en decimal y el primero es un número expresado en base 8, cuya representación binaria es 010000, que equivale a 16, en base 10.

Finalmente, se incluirá el programa en el cuerpo de la función principal. Esta función debe estar presente en todo programa, pues la

primera instrucción que contenga es la que se toma como punto inicial del programa y, por tanto, será la primera en ser ejecutada.

1.4. La programación imperativa

La programación consiste en la traducción de algoritmos a versiones en lenguajes de programación que puedan ser ejecutados directa o indirectamente por un ordenador.

La mayoría de algoritmos consisten en una secuencia de pasos que indican lo que hay que hacer. Estas instrucciones suelen ser de carácter imperativo, es decir, indican lo que hay que hacer de forma incondicional.

La programación de los algoritmos expresados en estos términos se denomina *programación imperativa*. Así pues, en este tipo de programas, cada instrucción implica realizar una determinada acción sobre su entorno, en este caso, en el computador en el que se ejecuta.

Para entender cómo se ejecuta una instrucción, es necesario ver cómo es el entorno en el que se lleva a cabo. La mayoría de los procesadores se organizan de manera que los datos y las instrucciones se encuentran en la memoria principal y la unidad central de procesamiento (CPU, de las siglas en inglés) es la que realiza el siguiente algoritmo para poder ejecutar el programa en memoria:

1. Leer de la memoria la instrucción que hay que ejecutar.
2. Leer de la memoria los datos necesarios para su ejecución.
3. Realizar el cálculo u operación indicada en la instrucción y, según la operación que se realice, grabar el resultado en la memoria.
4. Determinar cuál es la siguiente instrucción que hay que ejecutar.
5. Volver al primer paso.

La CPU hace referencia a las instrucciones y a los datos que pide a la memoria o a los resultados que quiere escribir mediante el número

de posición que ocupan en la misma. Esta posición que ocupan los datos y las instrucciones se conoce como *dirección de memoria*.

En el nivel más bajo, cada dirección distinta de memoria es un único byte y los datos y las instrucciones se identifican por la dirección del primero de sus bytes. En este nivel, la CPU coincide con la CPU física de que dispone el computador.

En el nivel de la máquina abstracta que ejecuta C, se mantiene el hecho de que las referencias a los datos y a las instrucciones sea la dirección de la memoria física del ordenador, pero las instrucciones que puede ejecutar su CPU de alto nivel son más potentes que las que puede llevar a cabo la real.

Independientemente del nivel de abstracción en que se trabaje, la memoria es, de hecho, el entorno de la CPU. Cada instrucción realiza, en este modelo de ejecución, un cambio en el entorno: puede modificar algún dato en memoria y siempre implica determinar cuál es la dirección de la siguiente instrucción a ejecutar.

Dicho de otra manera: la ejecución de una instrucción supone un cambio en el estado del programa. Éste se compone de la dirección de la instrucción que se está ejecutando y del valor de los datos en memoria. Así pues, llevar a cabo una instrucción implica cambiar de estado el programa.

En los próximos apartados se describirán los tipos de datos básicos que puede emplear un programa en C y las instrucciones fundamentales para cambiar su estado: la asignación y la selección condicional de la instrucción siguiente. Finalmente, se verán las funciones estándar para tomar datos del exterior (del teclado) y para mostrarlos al exterior (a través de la pantalla).

1.4.1. Tipos de datos básicos

Los tipos de datos primitivos de un lenguaje son aquéllos cuyo tratamiento se puede realizar con las instrucciones del mismo lenguaje; es decir, que están soportados por el lenguaje de programación correspondiente.

Compatibles con enteros

En C, los tipos de datos primitivos más comunes son los compatibles con enteros. La representación binaria de éstos no es codificada, sino que se corresponde con el valor numérico representado en base 2. Por tanto, se puede calcular su valor numérico en base 10 sumando los productos de los valores intrínsecos (0 o 1) de sus dígitos (bits) por sus valores posicionales ($2^{\text{posición}}$) correspondientes.

Se tratan bien como números naturales, o bien como representaciones de enteros en base 2, si pueden ser negativos. En este último caso, el bit más significativo (el de más a la izquierda) es siempre un 1 y el valor absoluto se obtiene restando el número natural representado del valor máximo representable con el mismo número de bits más 1.

En todo caso, es importante tener presente que el rango de valores de estos datos depende del número de bits que se emplee para su representación. Así pues, en la tabla siguiente se muestran los distintos tipos de datos compatibles con enteros en un computador de 32 bits con un sistema GNU.

Tabla 2.

Especificación	Número de bits	Rango de valores
(signed) char	8 (1 byte)	de -128 a +127
unsigned char	8 (1 byte)	de 0 a 255
(signed) short (int)	16 (2 bytes)	de -32.768 a +32.767
unsigned short (int)	16 (2 bytes)	de 0 a 65.535
(signed) int	32 (4 bytes)	de -2.147.483.648 a +2.147.483.647
unsigned (int)	32 (4 bytes)	de 0 a 4.294.967.295
(signed) long (int)	32 (4 bytes)	de -2.147.483.648 a +2.147.483.647
unsigned long (int)	32 (4 bytes)	de 0 a 4.294.967.295
(signed) long long (int)	64 (8 bytes)	de -2^{63} a $+(2^{63}-1) \approx \pm 9,2 \times 10^{18}$
unsigned long long (int)	64 (8 bytes)	de 0 a $2^{64}-1 \approx 1,8 \times 10^{19}$

Nota

Las palabras de la especificación entre paréntesis son opcionales en las declaraciones de las variables correspondientes. Por otra parte, hay que tener presente que las especificaciones pueden variar levemente con otros compiladores.



Hay que recordar especialmente los distintos rangos de valores que pueden tomar las variables de cada tipo para su correcto uso en los programas. De esta manera, es posible ajustar su tamaño al que realmente sea útil.

Ejemplo

En este estándar, la letra A mayúscula se encuentra en la posición número 65.

El tipo carácter (`char`) es, de hecho, un entero que identifica una posición de la tabla de caracteres ASCII. Para evitar tener que traducir los caracteres a números, éstos se pueden introducir entre comillas simples (por ejemplo: '`A`'). También es posible representar códigos no visibles como el salto de línea ('`\n`') o la tabulación ('`\t`').

Números reales

Este tipo de datos es más complejo que el anterior, pues su representación binaria se encuentra codificada en distintos campos. Así pues, no se corresponde con el valor del número que se podría extraer de los bits que los forman.

Los números reales se representan mediante signo, mantisa y exponente. La mantisa expresa la parte fraccionaria del número y el exponente es el número al que se eleva la base correspondiente:

$$[+/-] \text{mantisa} \times \text{base}^{\text{exponente}}$$

En función del número de bits que se utilicen para representarlos, los valores de la mantisa y del exponente serán mayores o menores. Los distintos tipos de reales y sus rangos aproximados se muestran en la siguiente tabla (válida en sistemas GNU de 32 bits):

Tabla 3.

Especificación	Número de bits	Rango de valores
float	32 (4 bytes)	$\pm 3,4 \times 10^{-38}$
double	64 (8 bytes)	$\pm 1,7 \times 10^{-308}$
long double	96 (12 bytes)	$\pm 1,1 \times 10^{-4.932}$

Como se puede deducir de la tabla anterior, es importante ajustar el tipo de datos real al rango de valores que podrá adquirir una deter-

minada variable para no ocupar memoria innecesariamente. También cabe prever lo contrario: el uso de un tipo de datos que no pueda alcanzar la representación de los valores extremos del rango empleado provocará que éstos no se representen adecuadamente y, como consecuencia, el programa correspondiente puede comportarse de forma errática.

Declaraciones

La declaración de una variable supone manifestar su existencia ante el compilador y que éste planifique una reserva de espacio en la memoria para contener sus datos. La declaración se realiza anteponiendo al nombre de la variable la especificación de su tipo (`char`, `int`, `float`, `double`), que puede, a su vez, ir precedida de uno o varios modificadores (`signed`, `unsigned`, `short`, `long`). El uso de algún modificador hace innecesaria la especificación `int` salvo para `long`. Por ejemplo:

```
unsigned short natural;      /* La variable 'natural' se          */
                             /* declara como un          */
                             /* entero positivo.          */
int                  i, j, k;    /* Variables enteras con signo. */
char                 opcion;   /* Variable de tipo carácter. */
float                percentil; /* Variable de tipo real.        */
```

Por comodidad, es posible dotar a una variable de un contenido inicial determinado. Para ello, basta con añadir a la declaración un signo igual seguido del valor que tendrá al iniciarse la ejecución del programa. Por ejemplo:

```
int      importe = 0;
char     opcion = 'N';
float   angulo = 0.0;
unsigned contador = MAXIMO;
```

El nombre de las variables puede contener cualquier combinación de caracteres alfabéticos (es decir, los del alfabeto inglés, sin '`ñ`', ni '`ç`', ni ningún tipo de carácter acentuado), numéricos y también el símbolo del subrayado (`_`); pero no puede empezar por ningún dígito.



Es conveniente que los nombres con los que “bautizamos” las variables identifiquen su contenido o su utilización en el programa.

1.4.2. La asignación y la evaluación de expresiones

Tal como se ha comentado, en la programación imperativa, la ejecución de las instrucciones implica cambiar el estado del entorno del programa o, lo que es lo mismo, cambiar la referencia de la instrucción a ejecutar y, posiblemente, el contenido de alguna variable. Esto último ocurre cuando la instrucción que se ejecuta es la de asignación:



variable = expresión en términos de variables y valores constantes ;

La potencia (y la posible dificultad de lectura de los programas) de C se encuentra, precisamente en las expresiones.

Nota

Una expresión es cualquier combinación sintácticamente válida de operadores y operandos que pueden ser variables o constantes.

De hecho, cualquier expresión se convierte en una instrucción si se pone un punto y coma al final: todas las instrucciones de C acaban en punto y coma.

Evidentemente, evaluar una expresión no tiene sentido si no se asigna el resultado de su evaluación a alguna variable que pueda almacenarlo para operaciones posteriores. Así pues, el primer operador que se debe aprender es el de asignación:

```
entero = 23;
destino = origen;
```



Es importante no confundir el operador de asignación (=) con el de comparación de igualdad (==); pues en C, ambos son operadores que pueden emplearse entre datos del mismo tipo.

Operadores

A parte de la asignación, los operadores más habituales de C y que aparecen en otros lenguajes de programación se muestran en la tabla siguiente:

Tabla 4.

Clase	Operadores	Significado
Aritmético	+ -	suma y resta
	* /	producto y división
	%	módulo o resto de división entera (sólo para enteros)
Relacional	> >=	"mayor que" y "mayor e igual que"
	< <=	"menor que" y "menor e igual que"
	== !=	"igual a" y "diferente de"
Lógico	!	NO (proposición lógica)
	&&	Y (deben cumplirse todas las partes) y O lógicas



Los operadores aritméticos sirven tanto para los números reales, como para los enteros. Por este motivo, se realizan implícitamente todas las operaciones con el tipo de datos de mayor rango.

A este comportamiento implícito de los operadores se le llama *promoción de tipos de datos* y se realiza cada vez que se opera con datos de tipos distintos.

Por ejemplo, el resultado de una operación con un entero y un real (las constantes reales deben tener un punto decimal o bien contener la letra "e" que separa mantisa de exponente) será siempre un número real. En cambio, si la operación se efectúa sólo con enteros, el resultado será siempre el del tipo de datos entero de mayor rango.
De esta manera:

```
real = 3 / 2 ;
```

tiene como resultado asignar a la variable `real` el valor `1.0`, que es el resultado de la división entera entre `3` y `2` transformado en un real

cuando se hace la operación de asignación. Por eso se escribe 1.0 (con el punto decimal) en lugar de 1.

Aun con esto, el operador de asignación siempre convierte el resultado de la expresión fuente al tipo de datos de la variable receptora. Por ejemplo, la asignación siguiente:

```
entero = 3.0 / 2 + 0.5;
```

asigna el valor 2 a entero. Es decir, se calcula la división real (el número 3.0 es real, ya que lleva un punto decimal) y se suma 0.5 al resultado. Esta suma actúa como factor de redondeo. El resultado es de tipo real y será truncado (su parte decimal será eliminada) al ser asignado a la variable entero.

Coerción de tipo

Para aumentar la legibilidad del código, evitar interpretaciones equivocadas e impedir el uso erróneo de la promoción automática de tipos, resulta conveniente indicar de forma explícita que se hace un cambio de tipo de datos. Para ello, se puede recurrir a la coerción de tipos; esto es: poner entre paréntesis el tipo de datos al que se desea convertir un determinado valor (esté en una variable o sea éste constante):

```
( especificación_de_tipo ) operando
```

Así pues, siguiendo el ejemplo anterior, es posible convertir un número real al entero más próximo mediante un redondeo del siguiente modo:

```
entero = (int) (real + 0.5);
```

En este caso, se indica que la suma se hace entre dos reales y el resultado, que será de tipo real, se convierte explícitamente a entero mediante la coerción de tipos.

Operadores relacionales

Hay que tener presente que en C no hay ningún tipo de datos lógico que se corresponda con 'falso' y 'cierto'. Así pues, cualquier dato

de tipo compatible con entero será indicación de 'falso' si es 0, y 'cierto' si es distinto de 0.

Nota

Esto puede no suceder con los datos de tipo real, pues hay que tener presente que incluso los números infinitesimales cerca del 0 serán tomados como indicación de resultado lógico 'cierto'.

Como consecuencia de ello, los operadores relacionales devuelven 0 para indicar que la relación no se cumple y distinto de 0 en caso afirmativo. Los operadores `&&` y `||` sólo evalúan las expresiones necesarias para determinar, respectivamente, si se cumplen todos los casos o sólo algunos. Así pues, `&&` implica la evaluación de la expresión que constituye su segundo argumento sólo en caso de que el primero haya resultado positivo. Similarmente, `||` sólo ejecutará la evaluación de su segundo argumento si el primero ha resultado 'falso'.

Así pues:

```
(20 > 10) || ( 10.0 / 0.0 < 1.0 )
```

dará como resultado 'cierto' a pesar de que el segundo argumento no se puede evaluar (es una división por cero!).

En el caso anterior, los paréntesis eran innecesarios, pues los operadores relacionales tienen mayor prioridad que los lógicos. Aun así, es conveniente emplear paréntesis en las expresiones para dotarlas de mayor claridad y despejar cualquier duda sobre el orden de evaluación de los distintos operadores que puedan contener.

Otros operadores

Como se ha comentado, C fue un lenguaje inicialmente concebido para la programación de sistemas operativos y, como consecuencia, con un alto grado de relación con la máquina, que se manifiesta en la existencia de un conjunto de operadores orientados a facilitar una traducción muy eficiente a instrucciones del lenguaje máquina.

En particular, dispone de los operadores de autoincremento (`++`) y autodecremento (`--`), que se aplican directamente sobre variables cuyo contenido sea compatible con enteros. Por ejemplo:

```
contador++; /* Equivalente a: contador = contador + 1; */

--descuento; /* Equivalente a: descuento = descuento - 1; */
```

La diferencia entre la forma prefija (es decir, precediendo a la variable) y la forma postfija de los operadores estriba en el momento en que se hace el incremento o decremento: en forma prefija, se hace antes de emplear el contenido de la variable.

Ejemplo

Véase cómo se modifican los contenidos de las variables en el siguiente ejemplo:

```
a = 5;      /* ( a == 5 )                                */
b = ++a;    /* ( a == 6 ) && ( b == 6 )   */
c = b--;    /* ( c == 6 ) && ( b == 5 ) */
```

Por otra parte, también es posible hacer operaciones entre bits. Estas operaciones se hacen entre cada uno de los bits que forma parte de un dato compatible con entero y otro. Así pues, es posible hacer una Y, una O, una O-EX (sólo uno de los dos puede ser cierto) y una negación lógica bit a bit entre los que forman parte de un dato y los de otro. (Un bit a cero representa 'falso' y uno a uno, 'cierto'.)

Los símbolos empleados para estos operadores a nivel de bit son:

- Para la Y lógica: `&` (ampersand)
- Para la O lógica: `|` (barra vertical)
- Para la O exclusiva lógica: `^` (ácento circunflejo)
- Para la negación o complemento lógico: `~` (tilde)

A pesar de que son operaciones válidas entre datos compatibles con enteros, igual que los operadores lógicos, es muy importante tener presente que no dan el mismo resultado. Por ejemplo: `(1 && 2)`

es cierto, pero `(1 & 2)` es falso, puesto que da 0. Para comprobarlo, basta con ver lo que pasa a nivel de bit:

```
1 == 0000 0000 0000 0001
2 == 0000 0000 0000 0010
1 & 2 == 0000 0000 0000 0000
```

La lista de operadores en C no termina aquí. Hay otros que se verán más tarde y algunos que quedarán en el tintero.

1.4.3. Instrucciones de selección

En el modelo de ejecución de los programas, las instrucciones se ejecutan en secuencia, una tras la otra, en el mismo orden en que aparecen en el código. Ciertamente, esta ejecución puramente secuencial no permitiría realizar programas muy complejos, pues siempre realizarían las mismas operaciones. Por ello, es necesario contar con instrucciones que permitan controlar el flujo de ejecución del programa. En otras palabras, disponer de instrucciones que permitan alterar el orden secuencial de su ejecución.

En este apartado se comentan las instrucciones de C que permiten seleccionar entre distintas secuencias de instrucciones. De forma breve, se resumen en la tabla siguiente:

Tabla 5.

Instrucción	Significado
<pre>if(condición) instrucción_si ; else instrucción_no ;</pre>	La condición tiene que ser una expresión cuya evaluación dé como resultado un dato de tipo compatible con entero. Si el resultado es distinto de cero, se considera que la condición se cumple y se ejecuta instrucción_si. En caso contrario, se ejecuta instrucción_no. El else es opcional.
<pre>switch(expresión) { case valor_1 : instrucciones case valor_2 : instrucciones default : instrucciones } /* switch */</pre>	La evaluación de la expresión debe resultar en un dato compatible con entero. Este resultado se compara con los valores indicados en cada case y, de ser igual a alguno de ellos, se ejecutan todas las instrucciones a partir de la primera indicada en ese caso y hasta el final del bloque del switch. Es posible "romper" esta secuencia introduciendo una instrucción break; que finaliza la ejecución de la secuencia de instrucciones. Opcionalmente, es posible indicar un caso por omisión (default) que permite especificar qué instrucciones se ejecutarán si el resultado de la expresión no ha producido ningún dato coincidente con los casos previstos.

En el caso del if es posible ejecutar más de una instrucción, tanto si la condición se cumple como si no, agrupando las instrucciones en

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.³³**

un bloque. Los bloques de instrucciones son instrucciones agrupadas entre llaves:

```
{ instrucción_1; instrucción_2; ... instrucción_N; }
```

En este sentido, es recomendable que todas las instrucciones condicionales agrupen las instrucciones que se deben ejecutar en cada caso:

```
if( condición )
{ instrucciones }
else
{ instrucciones }
/* if */
```

De esta manera se evitan casos confusos como el siguiente:

```
if( a > b )
    mayor = a ;
    menor = b ;
diferencia = mayor - menor;
```

En este caso se asigna `b` a `menor`, independientemente de la condición, pues la única instrucción del `if` es la de asignación a `mayor`.

Como se puede apreciar, las instrucciones que pertenecen a un mismo bloque empiezan siempre en la misma columna. Para facilitar la identificación de los bloques, éstos deben presentar un sangrado a la derecha respecto de la columna inicial de la instrucción que los gobierna (en este caso: `if`, `switch` y `case`).



Por convenio, cada bloque de instrucciones debe presentar un sangrado a la derecha respecto de la instrucción que determina su ejecución.

Dado que resulta frecuente tener que asignar un valor u otro a una variable es función de una condición, es posible, para estos casos,

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

emplear un operador de asignación condicional en lugar de una instrucción if:

```
condición ? expresión_si_cierto : expresión_si_falso
```

Así pues, en lugar de:

```
if( condicion )    var = expresión_si_cierto;
else                  var = expresión_si_falso;
```

Se puede escribir:

```
var = condición ? expresión_si_cierto : expresión_si_falso;
```

Más aun, se puede emplear este operador dentro de cualquier expresión. Por ejemplo:

```
coste = ( km>km_contrato? km-km_contrato : 0 ) * COSTE_KM;
```



Es preferible limitar el uso del operador condicional a los casos en que se facilite la lectura.

1.4.4. Funciones estándar de entrada y de salida

El lenguaje C sólo cuenta con operadores e instrucciones de control de flujo. Cualquier otra operación que se desee realizar hay que programarla o bien emplear las funciones de que se disponga en nuestra biblioteca de programas.

Nota

Ya hemos comentado que una función no es más que una serie de instrucciones que se ejecuta como una unidad para llevar a cabo una tarea concreta. Como idea, puede tomarse la de las funciones matemáticas, que realizan alguna operación con los argumentos dados y devuelven el resultado calculado.

El lenguaje C cuenta con un amplio conjunto de funciones estándar entre las que se encuentran las de entrada y de salida de datos, que veremos en esta sección.

El compilador de C tiene que saber (nombre y tipo de datos de los argumentos y del valor devuelto) qué funciones utilizará nuestro programa para poder generar el código ejecutable de forma correcta. Por tanto, es necesario incluir los ficheros de cabecera que contengan sus declaraciones en el código de nuestro programa. En este caso:

```
#include <stdio.h>
```

Funciones de salida estándar

La salida estándar es el lugar donde se muestran los datos producidos (mensajes, resultados, etcétera) por el programa que se encuentra en ejecución. Normalmente, esta salida es la pantalla del ordenador o una ventana dentro de la pantalla. En este último caso, se trata de la ventana asociada al programa.

```
printf( "formato" [, lista_de_campos ] )
```

Esta función imprime en la pantalla (la salida estándar) el texto contenido en "formato". En este texto se sustituyen los caracteres especiales, que deben de ir precedidos por la barra invertida (\), por su significado en ASCII. Además, se sustituyen los especificadores de campo, que van precedidos por un %, por el valor resultante de la expresión (normalmente, el contenido de una variable) correspondiente indicada en la lista de campos. Este valor se imprime según el formato indicado en el mismo especificador.

La tabla siguiente muestra la correspondencia entre los símbolos de la cadena del formato de impresión y los caracteres ASCII.
n se utiliza para indicar un dígito de un número:

Tabla 6.

Indicación de carácter	Carácter ASCII
\n	new line (salto de línea)
\f	form feed (salto de página)

Indicación de carácter	Carácter ASCII
\b	backspace (retroceso)
\t	tabulator (tabulador)
\nnn	ASCII número nnn
\0nnn	ASCII número nnn (en octal)
\0Xnn	ASCII número nn (en hexadecimal)
\\	backslash (barra invertida)

Los especificadores de campo tienen el formato siguiente:

% [-] [+][anchura [.precisión]]tipo_de_dato

Los corchetes indican que el elemento es opcional. El signo menos se emplea para indicar alineación derecha, cosa habitual en la impresión de números. Para éstos, además, si se especifica el signo más se conseguirá que se muestren precedidos por su signo, sea positivo o negativo. La anchura se utilizará para indicar el número mínimo de caracteres que se utilizarán para mostrar el campo correspondiente y, en el caso particular de los números reales, se puede especificar el número de dígitos que se desea mostrar en la parte decimal mediante la precisión. El tipo de dato que se desea mostrar se debe incluir obligatoriamente y puede ser uno de los siguientes:

Tabla 7.

Enteros		Reales		Otros	
%d	En decimal	%f	En punto flotante	%c	Carácter
%i	En decimal	%e	Con formato exponencial: [+/-]0.000e[+/-]000 con e minúscula o mayúscula (%E)	%s	Cadena de caracteres
%u	En decimal sin signo			%%	El signo de %
%o	En octal sin signo				(listado no completo)
%x	En hexadecimal	%g	En formato e, f, o d.		

Ejemplo

```
printf( "El importe de la factura núm.: %d", num_fact );
printf( "de Sr./-a. %s sube a %.2f\n", cliente, importe );
```

Para los tipos numéricos es posible prefijar el indicador de tipo con una "ele" a la manera que se hace en la declaración de tipos con long. En este caso, el tipo double debe tratarse como un "long float" y, por tanto, como "%lf".

Ejemplo

```
putchar( '\n' );
```

Ejemplo

```
puts( "Hola a todos!\n" );
```

putchar(carácter)

Muestra el carácter indicado por la salida estándar.

puts("cadena de caracteres")

Muestra una cadena de caracteres por la salida estándar.

Funciones de entrada estándar

Se ocupan de obtener datos de la entrada estándar que, habitualmente, se trata del teclado. Devuelven algún valor adicional que informa del resultado del proceso de lectura. El valor devuelto no tiene por qué emplearse si no se necesita, pues muchas veces se conoce que la entrada de datos se puede efectuar sin mayor problema.

scanf("formato" [, lista_de_variables])

Nota

El buffer del teclado es la memoria en la que se almacena lo que se escribe en él.

Lee del buffer del teclado para trasladar su contenido a las variables que tiene como argumentos. La lectura se efectúa según el formato indicado, de forma similar a la especificación de campos empleada para printf.

Para poder depositar los datos leídos en las variables indicadas, esta función requiere que los argumentos de la lista de variables sean las direcciones de memoria en las que se encuentran. Por este motivo, es necesario emplear el operador “dirección de” (&). De esta manera, scanf deja directamente en las zonas de memoria correspondiente la información que haya leído y, naturalmente, la variable afectada verá modificado su contenido con el nuevo dato.



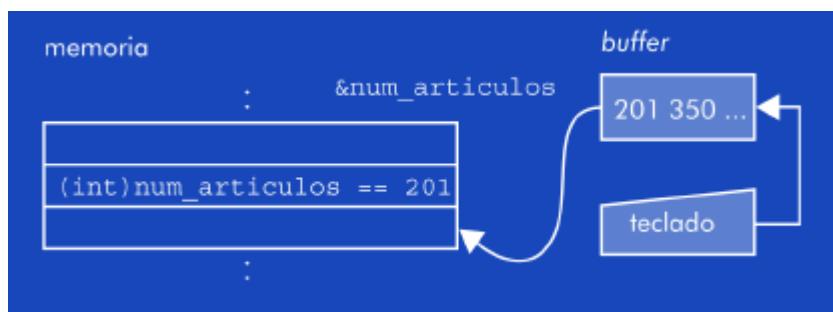
Es importante tener presente que si se especifican menos argumentos que especificadores de campo en el formato, los resultados pueden ser imprevisibles, pues la función cambiará el contenido de alguna zona de memoria totalmente aleatoria.

Ejemplo

```
scanf( "%d", &num_articulos );
```

En el ejemplo anterior, `scanf` lee los caracteres que se tecleen para convertirlos (presumiblemente) a un entero. El valor que se obtenga se colocará en la dirección indicada en su argumento, es decir, en el de la variable `num_articulos`. Esta lectura de caracteres del buffer de memoria del teclado se detiene cuando el carácter leído no se corresponde con un posible carácter del formato especificado. Este carácter se devuelve al buffer para una posible lectura posterior.

Para la entrada que se muestra en la figura siguiente, la función tiene la lectura después del espacio en blanco que separa los números tecleados. Éste y el resto de caracteres se quedan en el buffer para posteriores lecturas.

Figura 1.

La función `scanf` devuelve el número de datos correctamente leídos. Es decir, todos aquellos para los que se ha encontrado algún texto compatible con una representación de su tipo.

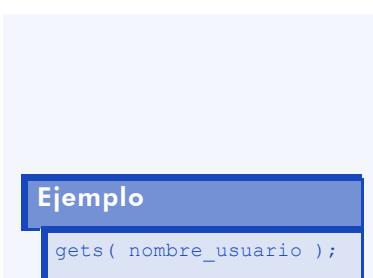
getchar()

Devuelve un carácter leído por la entrada estándar (habitualmente, el buffer del teclado).

En caso de que no pueda leer ningún carácter, devuelve el carácter `EOF`. Esta constante está definida en `stdio.h` y, por tanto, puede emplearse para determinar si la lectura ha tenido éxito o, por lo contrario, se ha producido algún error o se ha llegado al final de los datos de entrada.

Ejemplo

```
opcion = getchar();
```



```
gets( cadena_caracteres )
```

Lee de la entrada estándar toda una serie de caracteres hasta encontrar un final de línea (carácter '`\n`'). Este último carácter es leído pero no se almacena en la cadena de caracteres que tiene por argumento.

De no leer nada devuelve `NULL`, que es una constante definida en `stdio.h` y cuyo valor es 0.

1.5. Resumen

En esta unidad se ha visto el procedimiento de ejecución de los programas en un computador. La unidad que se ocupa de procesar la información (unidad central de procesamiento o CPU) lee una instrucción de la memoria y procede a su ejecución. Esta operación implicará un cambio del estado del entorno del programa, es decir, del contenido de alguna de sus variables y de la dirección de la instrucción siguiente.

Este modelo de ejecución de las instrucciones, que siguen la mayoría de procesadores reales, se replica en el paradigma de la programación imperativa para los lenguajes de alto nivel. En particular, se ha visto cómo esto es cierto en el lenguaje de programación C.

Por este motivo, se han repasado las instrucciones de este lenguaje que permiten modificar el entorno mediante cambios en los datos y cambios en el orden secuencial en que se encuentran las instrucciones en la memoria.

Los cambios que afectan a los datos de los programas son, de hecho, asignaciones a las variables que los contienen. Se ha visto que las variables son espacios en la memoria del computador a los que podemos hacer referencia mediante el nombre con que se declaran y a los que, internamente, se hace referencia mediante la dirección de la primera palabra de la memoria que ocupan.

Se ha hecho hincapié en la forma de evaluación de las expresiones teniendo en cuenta la prioridad entre los operadores y cómo se pue-

de organizar mejor mediante el uso de paréntesis. En este sentido, se ha comentado la conveniencia de emplear coerciones de tipo explícitas para evitar usos equívocos de la promoción automática de los tipos de datos. Esta promoción se debe realizar para que los operadores puedan trabajar siempre con operandos de un mismo tipo, que siempre coincide con el de mayor rango de representación.

En cuanto al control del flujo de ejecución que, normalmente, sigue el orden en que se encuentran las instrucciones en memoria, se han mencionado las instrucciones básicas de selección de secuencias de instrucciones: la instrucción `if` y la instrucción `switch`.

Aprovechando estas explicaciones, se ha introducido la forma especial de considerar los datos lógicos ('falso' y 'cierto') en C. Así pues, cualquier dato puede ser, en un momento dado, empleado como valor lógico. En relación a este tema, se ha comentado la forma especial de evaluación de los operadores Y y O lógicos, que no evalúan las expresiones derechas en caso de que puedan determinar el valor lógico resultante con el primer argumento (el que viene dado por la expresión que les precede).

En el último apartado, se han repasado las funciones estándar básicas de entrada y de salida de datos, de manera que sea posible construir programas para probar no sólo todos los conceptos y elementos de C explicados, sino también el entorno de desarrollo de GNU/C.

Así pues, se puede comprobar en la práctica en qué consiste la compilación y el enlace de los programas. La tarea del compilador de C es la de traducir el programa en C a un programa en lenguaje máquina. El enlazador se ocupa de añadir al código de esta versión el código de las funciones de la biblioteca que se utilicen en el programa. Con este proceso, se obtiene un código ejecutable.

1.6. Ejercicios de autoevaluación

- 1) Editad, compilad (y enlazad), ejecutad y comprobad el funcionamiento del siguiente programa:

```
#include <stdio.h>
main()
```

```
{
    int a, b, suma;
    printf( "Teclea un número entero: " );
    scanf( "%d", &a );
    printf( "Teclea otro número entero: " );
    scanf( "%d", &b );
    suma = a + b;
    printf( "%d + %d = %d\n", a, b, suma );
} /* main */
```

- 2) Haced un programa que, dado un importe en euros y un determinado porcentaje de IVA, calcule el total.
- 3) Haced un programa que calcule cuánto costaría 1Kg o 1 litro de un producto sabiendo el precio de un envase y la cantidad de producto que contiene.
- 4) Modificad el programa anterior para que calcule el precio del producto para la cantidad deseada, que también deberá darse como entrada.
- 5) Haced un programa que calcule el cambio que hay que devolver conociendo el importe total a cobrar y la cantidad recibida como pago. El programa debe advertir si el importe pagado es insuficiente.
- 6) Haced un programa que, dado el número de litros aproximado que hay en el depósito de un coche, su consumo medio cada 100 km y una distancia en kilómetros, indique si es posible recorrerla. En caso negativo, debe indicar cuántos litros habría que añadir al depósito.

1.6.1. Solucionario

- 1) Basta con seguir los pasos indicados. Como ejemplo, si el fichero se llamase "suma.c", esto es lo que debería hacerse después de haberlo creado:

```
$ gcc suma.c -o suma
$ suma
Teclea un número entero: 154
Teclea otro número entero: 703
154 + 703 = 857
$
```

2)

```
#include <stdio.h>
main()
{
    float importe, IVA, total;

    printf( "Importe = " );
    scanf( "%f", &importe );
    printf( "%% IVA = " );
    scanf( "%f", &IVA );
    total = importe * ( 1.0 + IVA / 100.0 );
    printf( "Total = %.2f\n", total );
} /* main */
```

3)

```
#include <stdio.h>
main()
{
    float precio, precio_unit;
    int cantidad;

    printf( "Precio = " );
    scanf( "%f", &precio );
    printf( "Cantidad (gramos o mililitros) = " );
    scanf( "%d", &cantidad );
    precio_unit = precio * 1000.0 / (float) cantidad;
    printf( "Precio por Kg/Litro = %.2f\n", precio_unit );
} /* main */
```

4)

```
#include <stdio.h>
main()
{
    float precio, precio_unit, precio_compra;
    int cantidad, canti_compra;

    printf( "Precio = " );
    scanf( "%f", &precio );
    printf( "Cantidad (gramos o mililitros) = " );
    scanf( "%d", &cantidad );
    printf( "Cantidad a adquirir = " );
    scanf( "%d", &canti_compra );
    precio_unit = precio / (float) cantidad;
```

```

    precio_compra = precio_unit * canti_compra;
    printf( "Precio de compra = %.2f\n", precio_compra );
} /* main */

5)
#include <stdio.h>
main()
{
    float importe, pago;

    printf( "Importe = " );
    scanf( "%f", &importe );
    printf( "Pago = " );
    scanf( "%f", &pago );
    if( pago < importe ) {
        printf( "Importe de pago insuficiente.\n" );
    } else {
        printf( "Cambio=% .2f euros.\n", pago - importe );
    } /* if */
} /* main */

6)
#include <stdio.h>
#define RESERVA 10
main()
{
    int litros, distancia, consumo;
    float consumo_medio;

    printf( "Litros en el depósito = " );
    scanf( "%d", &litros );
    printf( "Consumo medio cada 100Km = " );
    scanf( "%f", &consumo_medio );
    printf( "Distancia a recorrer = " );
    scanf( "%d", &distancia );
    consumo = consumo_medio * (float) distancia / 100.0;
    if( litros < consumo ) {
        printf( "Faltan %d Ltrs.\n", consumo-litros+RESERVA );
    } else {
        printf( "Se puede hacer el recorrido.\n" );
    } /* if */
} /* main */

```

2. La programación estructurada

2.1. Introducción

La programación eficaz es aquella que obtiene un código legible y fácilmente actualizable en un tiempo de desarrollo razonable y cuya ejecución se realiza de forma eficiente.

Afortunadamente, los compiladores e intérpretes de código de programas escritos en lenguajes de alto nivel realizan ciertas optimizaciones para reducir el coste de su ejecución. Sirva como ejemplo el hecho de que muchos compiladores tienen la capacidad de utilizar el mismo espacio de memoria para diversas variables, siempre que éstas no se utilicen simultáneamente, claro está. Además de ésta y otras optimizaciones en el uso de la memoria, también pueden incluir mejoras en el código ejecutable tendentes a disminuir el tiempo final de ejecución. Pueden, por ejemplo, aprovechar los factores comunes de las expresiones para evitar la repetición de cálculos.

Con todo esto, los programadores pueden centrar su tarea en la preparación de programas legibles y de fácil mantenimiento. Por ejemplo, no hay ningún problema en dar nombres significativos a las variables, pues la longitud de los nombres no tiene consecuencias en un código compilado. En este mismo sentido, no resulta lógico emplear trucos de programación orientados a obtener un código ejecutable más eficiente si ello supone disminuir la legibilidad del código fuente. Generalmente, los trucos de programación no suponen una mejora significativa del coste de ejecución y, en cambio, implican dificultad de mantenimiento del programa y dependencia de un determinado entorno de desarrollo y de una determinada máquina.

Por este motivo, en esta unidad se explica cómo organizar el código fuente de un programa. La correcta organización de los programas supone un incremento notable de su legibilidad y, como consecuencia, una disminución de los errores de programación y facilidad de

Nota

Un salto es un cambio en el orden de ejecución de las instrucciones por el que la siguiente instrucción no es la que encuentra a continuación de la que se está ejecutando.

mantenimiento y actualización posterior. Más aún, resultará más fácil aprovechar partes de sus códigos en otros programas.

En el primer apartado se trata de la programación estructurada. Este paradigma de la programación está basado en la programación imperativa, a la que impone restricciones respecto de los saltos que pueden efectuarse durante la ejecución de un programa.

Con estas restricciones se consigue aumentar la legibilidad del código fuente, permitiendo a sus lectores determinar con exactitud el flujo de ejecución de las instrucciones.

Es frecuente, en programación, encontrarse con bloques de instrucciones que deben ejecutarse repetitivamente. Por tanto, es necesario ver cómo disponer el código correspondiente de forma estructurada. En general, estos casos derivan de la necesidad de procesar una serie de datos. Así pues, en el primer apartado, no sólo se verá la programación estructurada, sino también los esquemas algorítmicos para realizar programas que traten con series de datos y, cómo no, las correspondientes instrucciones en C.

Por mucho que la programación estructurada esté encaminada a reducir errores en la programación, éstos no se pueden eliminar. Así pues, se dedica un apartado a la depuración de errores de programación y a las herramientas que nos pueden ayudar a ello: los depuradores.

El segundo apartado se dedica a la organización lógica de los datos de los programas. Hay que tener presente que la información que procesan los computadores y su resultado está constituido habitualmente por una colección variopinta de datos. Estos datos, a su vez, pueden estar constituidos por otros más simples.

En los lenguajes de programación se da soporte a unos tipos básicos de datos, es decir, se incluyen mecanismos (declaraciones, operaciones e instrucciones) para emplearlos en los códigos fuente que se escriben con ellos.

Por tanto, la representación de la información que se maneja en un programa debe hacerse en términos de variables que contengan da-

tos de los tipos fundamentales a los que el lenguaje de programación correspondiente dé soporte. No obstante, resulta conveniente agrupar conjuntos de datos que estén muy relacionados entre sí en la información que representan. Por ejemplo: manejar como una única entidad el número de días de cada uno de los meses del año; el día, el mes y el año de una fecha; o la lista de los días festivos del año.

En el segundo apartado, pues, se tratará de aquellos aspectos de la programación que permiten organizar los datos en estructuras mayores. En particular, se verán las clases de estructuras que existen y cómo utilizar variables que contengan estos datos estructurados. También se verá cómo la definición de tipos de datos a partir de otros tipos, estructurados o no, beneficia a la programación. Dado que estos nuevos tipos no son reconocidos en el lenguaje de programación, son llamados *tipos de datos abstractos*.

El último apartado introduce los principios de la programación modular, que resulta fundamental para entender la programación en C. En este modelo de programación, el código fuente se divide en pequeños programas estructurados que se ocupan de realizar tareas muy específicas dentro del programa global. De hecho, con esto se consigue dividir el programa en subprogramas de más fácil lectura y comprensión. Estos subprogramas constituyen los llamados *módulos*, y de ahí se deriva el nombre de esta técnica de programación.

En C todos los módulos son funciones que suelen realizar acciones muy concretas sobre unas pocas variables del programa. Más aún, cada función suele especializarse en un tipo de datos concreto.

Dada la importancia de este tema, se insistirá en los aspectos de la declaración, definición y uso de las funciones en C. En especial, todo lo referente al mecanismo que se utiliza durante la ejecución de los programas para proporcionar y obtener datos de las funciones que incluyen.

Finalmente, se tratará sobre las macros del preprocesador de C. Estas macros tienen una apariencia similar a la de las llamadas de las funciones en C y pueden llevar a confusiones en la interpretación del código fuente del programa que las utilice.

Nota

Un tipo de datos abstracto es aquel que representa una información no contemplada en el lenguaje de programación empleado. Puede darse el caso de que haya tipos de datos soportados en algún lenguaje de programación que, sin embargo, en otros no lo estén y haya que tratarlos como abstractos.

El objetivo principal de esta unidad es que el lector aprenda a organizar correctamente el código fuente de un programa, pues es un indicador fundamental de la calidad de la programación. De forma más concreta, el estudio de esta unidad pretende que se alcancen los objetivos siguientes:

1. Conocer en qué consiste la programación estructurada.
2. Saber aplicar correctamente los esquemas algorítmicos de tratamiento de secuencias de datos.
3. Identificar los sistemas a emplear para la depuración de errores de un programa.
4. Saber cuáles son las estructuras de datos básicas y cómo emplearlas.
5. Saber en qué consiste la programación modular.
6. Conocer la mecánica de la ejecución de las funciones en C.

2.2. Principios de la programación estructurada

Lecturas complementarias

E.W. Dijkstra (1968). *The goto statement considered harmful*

E.W. Dijkstra (1970). *Notes on structured programming*

La programación estructurada es una técnica de programación que resultó del análisis de las estructuras de control de flujo subyacentes a todo programa de computador. El producto de este estudio reveló que es posible construir cualquier estructura de control de flujo mediante tres estructuras básicas: la secuencial, la condicional y la iterativa.



La programación estructurada consiste en la organización del código de manera que el flujo de ejecución de sus instrucciones resulte evidente a sus lectores.

Un teorema formulado el año 1966 por Böhm y Jacopini dice que todo “programa propio” debería tener un único punto de entrada y un único punto de salida, de manera que toda instrucción entre estos dos puntos es ejecutable y no hay bucles infinitos.

La conjunción de estas propuestas proporciona las bases para la construcción de programas estructurados en los que las estructuras de control de flujo se pueden realizar mediante un conjunto de instrucciones muy reducido.

De hecho, la estructura secuencial no necesita ninguna instrucción adicional, pues los programas se ejecutan normalmente llevando a cabo las instrucciones en el orden en que aparecen en el código fuente.

En la unidad anterior se comentó la instrucción `if`, que permite una ejecución condicional de bloques de instrucciones. Hay que tener presente que, para que sea un programa propio, debe existir la posibilidad de que se ejecuten todos los bloques de instrucciones.

Las estructuras de control de flujo iterativas se comentarán en el apartado siguiente. Vale la pena indicar que, en cuanto a la programación estructurada se refiere, sólo es necesaria una única estructura de control de flujo iterativa. A partir de ésta se pueden construir todas las demás.

2.3. Instrucciones iterativas

Las instrucciones iterativas son instrucciones de control de flujo que permiten repetir la ejecución de un bloque de instrucciones. En la tabla siguiente se muestran las que están presentes en C:

Tabla 8.

instrucción	Significado
<code>while(condición) { instrucciones } /* while */</code>	Se ejecutan todas las instrucciones en el bloque del bucle mientras la expresión de la condición dé como resultado un dato de tipo compatible con entero distinto de cero; es decir, mientras la condición se cumpla. Las instrucciones pueden no ejecutarse nunca.
<code>do { instrucciones } while (condición);</code>	De forma similar al bucle <code>while</code> , se ejecutan todas las instrucciones en el bloque del bucle mientras la expresión de la condición se cumpla. La diferencia estriba en que las instrucciones se ejecutarán, al menos, una vez. (La comprobación de la condición y, por tanto, de la posible repetición de las instrucciones, se realiza al final del bloque.)
<code>for(inicialización ; condición ; continuación) { instrucciones } /* for */</code>	El comportamiento es parecido a un bucle <code>while</code> ; es decir, mientras se cumpla la condición se ejecutan las instrucciones de su bloque. En este caso, sin embargo, es posible indicar qué instrucción o instrucciones quieren ejecutarse de forma previa al inicio del bucle (<code>inicialización</code>) y qué instrucción o instrucciones hay que ejecutar cada vez que finaliza la ejecución de las instrucciones (<code>continuación</code>).

Como se puede apreciar, todos los bucles pueden reducirse a un bucle "mientras". Aun así, hay casos en los que resulta más lógico emplear alguna de sus variaciones.

Hay que tener presente que la estructura del flujo de control de un programa en un lenguaje de alto nivel no refleja lo que realmente hace el procesador (saltos condicionales e incondicionales) en el aspecto del control del flujo de la ejecución de un programa. Aun así, el lenguaje C dispone de instrucciones que nos acercan a la realidad de la máquina, como la de salida forzada de bucle (`break;`) y la de la continuación forzada de bucle (`continue;`). Además, también cuenta con una instrucción de salto incondicional (`goto`) que no debería de emplearse en ningún programa de alto nivel.

Normalmente, la programación de un bucle implica determinar cuál es el bloque de instrucciones que hay que repetir y, sobre todo, bajo qué condiciones hay que realizar su ejecución. En este sentido, es muy importante tener presente que la condición que gobierna un bucle es la que determina la validez de la repetición y, especialmente, su finalización cuando no se cumple. Nótese que debe existir algún caso en el que la evaluación de la expresión de la condición dé como resultado un valor 'falso'. En caso contrario, el bucle se repetiría indefinidamente (esto es lo que se llamaría un caso de "bucle infinito").

Habiendo determinado el bloque iterativo y la condición que lo gobierna, también cabe programar la posible preparación del entorno antes del bucle y las instrucciones que sean necesarias a su conclusión: su inicialización y su finalización.



La instrucción iterativa debe escogerse en función de la condición que gobierna el bucle y de su posible inicialización.

En los casos en que sea posible que no se ejecuten las instrucciones del bucle, es conveniente emplear `while`. Por ejemplo, para calcular cuántos divisores tiene un número entero positivo dado:

```
/* ... */  
/* Inicialización: _____ */  
divisor = 1; /* Candidato a divisor */  
ndiv = 0; /* Número de divisores */  
/* Bucle: _____ */  
while( divisor < numero ) {  
    if( numero % divisor == 0 ) ndiv = ndiv + 1;  
    divisor = divisor + 1;  
} /* while */  
/* Finalización: _____ */  
if( numero > 0 ) ndiv = ndiv + 1;  
/* ... */
```

A veces, la condición que gobierna el bucle depende de alguna variable que se puede tomar como un contador de repeticiones; es decir, su contenido refleja el número de iteraciones realizado. En estos casos, puede considerarse el uso de un bucle `for`. En concreto, se podría interpretar como “iterar el siguiente conjunto de instrucciones para todos los valores de un contador entre uno inicial y uno final dados”. En el ejemplo siguiente, se muestra esta interpretación en código C:

```
/* ... */  
unsigned int contador;  
/* ... */  
for( contador = INICIO ;  
    contador <= FINAL ;  
    contador = contador + INCREMENTO  
) {  
    instrucciones  
} /* for */  
/* ... */
```

A pesar de que el ejemplo anterior es muy común, no es necesario que la variable que actúe de contador tenga que incrementarse, ni que tenga que hacerlo en un paso fijo, ni que la condición sólo deba consistir en comprobar que haya llegado a su valor final y, ni mucho menos, que sea una variable adicional que no se emplee en las instrucciones del cuerpo a iterar. De hecho sería muy útil que fuera alguna variable cuyo contenido se modificara a cada iteración y que, con ello, pudiese emplearse como contador.

Es recomendable evitar el empleo del `for` para los casos en que no haya contadores. En su lugar, es mucho mejor emplear un `while`.

En algunos casos, gran parte de la inicialización coincidiría con el cuerpo del bucle, o bien se hace necesario evidenciar que el bucle se ejecutará al menos una vez. Si esto se da, es conveniente utilizar una estructura `do...while`. Como ejemplo, veamos el código de un programa que realiza la suma de distintos importes hasta que el importe leído sea cero:

```
/* ... */
float total, importe;
/* ... */
total = 0.00;
printf( "SUMA" );
do {
    printf( " + " );
    scanf( "%f", &importe );
    total = total + importe;
} while( importe != 0.00 );
printf( " = %.2f", total );
/* ... */
```

La constante numérica real `0.00` se usa para indicar que sólo son significativos dos dígitos fraccionarios, ya que, a todos los efectos, sería igual optar por escribir `0.0` o, incluso, `0` (en el último caso, el número entero sería convertido a un número real antes de realizar la asignación).

Sea cual sea el caso, la norma de aplicación es la de mantener siempre un código inteligible. Por otra parte, la elección del tipo de instrucción iterativa depende del gusto estético del programador y de su experiencia, sin mayores consecuencias en la eficiencia del programa en cuanto a coste de ejecución.

2.4. Procesamiento de secuencias de datos

Mucha de la información que se trata consta de secuencias de datos que pueden darse explícita o implícitamente.

Ejemplo

En el primer caso, se trataría del procesamiento de información de una serie de datos procedentes del dispositivo de entrada estándar.

Un ejemplo del segundo sería aquel en el que se deben procesar una serie de valores que adquiere una misma variable.

En los dos casos, el tratamiento de la secuencia se puede observar en el código del programa, pues debe realizarse mediante alguna instrucción iterativa. Habitualmente, este bucle se corresponde con un esquema algorítmico determinado. En los siguientes apartados veremos los dos esquemas fundamentales para el tratamiento de secuencias de datos.

2.4.1. Esquemas algorítmicos: recorrido y búsqueda

Los esquemas algorítmicos para el procesado de secuencias de datos son unos patrones que se repiten frecuentemente en muchos algoritmos. Consecuentemente, existen unos patrones equivalentes en los lenguajes de programación como el C. En este apartado veremos los patrones básicos de tratamiento de secuencias: el de recorrido y el de búsqueda.

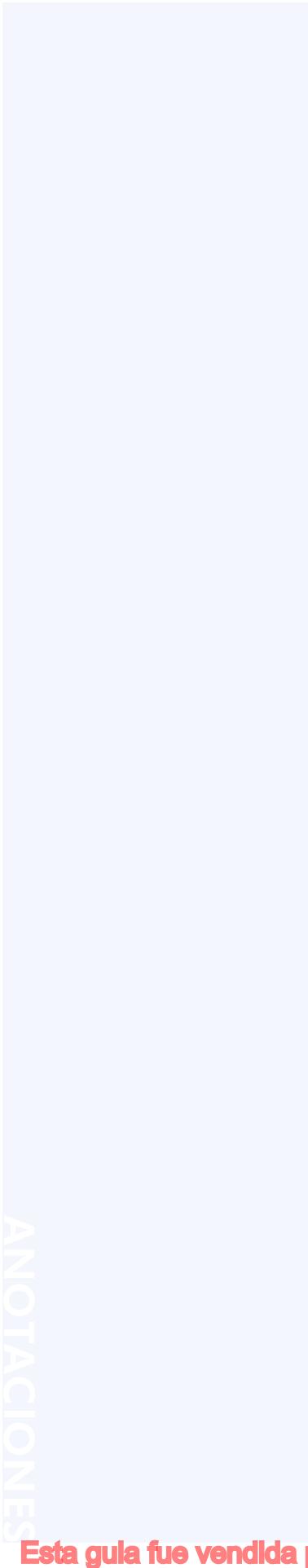
Recorrido



Un recorrido de una secuencia implica realizar un tratamiento idéntico a todos los miembros de la misma.

En otras palabras, supone tratar cada uno de los elementos de la secuencia, desde el primero hasta el último.

Si el número de elementos de que constará la secuencia es conocido *a priori* y la inicialización del bucle es muy simple, entonces puede ser conveniente emplear un bucle `for`. En caso contrario, los bucles



más adecuados son el bucle while o el do...while si se sabe que habrá, al menos, un elemento en la secuencia de datos.

El esquema algorítmico del recorrido de una secuencia sería, en su versión para C, el que se presenta a continuación:

```
/* inicialización para el procesamiento de la secuencia*/
/* (puede incluir el tratamiento del primer elemento) */
while( ! /* final de secuencia */ ) {
    /* tratar el elemento */
    /* avanzar en secuencia */
} /* while */
/* finalización del procesamiento de la secuencia */
/* (puede incluir el tratamiento del último elemento)
```

El patrón anterior podría realizarse con alguna otra instrucción iterativa, si las circunstancias lo aconsejan.

Para ilustrar varios ejemplos de recorrido, supongamos que se desea obtener la temperatura media en la zona de una estación meteorológica. Para ello, procederemos a hacer un programa al que se le suministran las temperaturas registradas a intervalos regulares por el termómetro de la estación y obtenga la media de los valores introducidos.

Así pues, el cuerpo del bucle consiste, simplemente, en acumular la temperatura (tratar el elemento) y en leer una nueva temperatura (avanzar en la secuencia):

```
/* ... */
acumulado = acumulado + temperatura;
cantidad = cantidad + 1;
scanf( "%f", &temperatura );
/* ... */
```

En este bloque iterativo se puede observar que temperatura debe tener un valor determinado antes de poderse acumular en la variable acumulado, la cual, a su vez, también tiene que estar inicializada. Similarmente, cantidad deberá inicializarse a cero.

Por ello, la fase de inicialización y preparación de la secuencia ya está lista:

```
/* ... */  
unsigned int cantidad;  
float acumulado;  
/* ... */  
cantidad = 0;  
acumulado = 0.00;  
scanf( "%f", &temperatura );  
/* bucle ... */
```

Aun queda por resolver el problema de establecer la condición de finalización de la secuencia de datos. En este sentido, puede ser que la secuencia de datos tenga una marca de final de secuencia o que su longitud sea conocida.

En el primero de los casos, la marca de final debe de ser un elemento especial de la secuencia que tenga un valor distinto del que pueda tener cualquier otro dato. En este sentido, como se sabe que una temperatura no puede ser nunca inferior a $-273,16^{\circ}\text{C}$ (y, mucho menos, una temperatura ambiental), se puede emplear este valor como marca de final. Por claridad, esta marca será una constante definida en el preprocesador:

```
#define MIN_TEMP -273.16
```

Cuando se encuentre, no deberá ser procesada y, en cambio, sí que debe hacerse la finalización del recorrido, calculando la media:

```
/* ... */  
float media;  
/* ... fin del bucle */  
if( cantidad > 0 ) {  
    media = acumulado / (float) cantidad;  
} else {  
    media = MIN_TEMP;  
} /* if */  
/* ... */
```

En el cálculo de la media, se comprueba primero que haya algún dato significativo para computarse. En caso contrario, se asigna a media la temperatura de marca de final. Con todo, el código del recorrido sería el mostrado a continuación:

```
/* ... */
cantidad = 0;
acumulado = 0.00;
scanf( "%f", &temperatura );
while( ! ( temperatura == MIN_TEMP ) ) {
    acumulado = acumulado + temperatura;
    cantidad = cantidad + 1;
    scanf( "%f", &temperatura );
} /* while */
if( cantidad > 0 ) {
    media = acumulado / (float) cantidad;
} else {
    media = MIN_TEMP;
} /* if */
/* ... */
```

Si la marca de final de secuencia se proporcionara aparte, la instrucción iterativa debería ser una do..while. En este caso, se supondrá que la secuencia de entrada la forman elementos con dos datos: la temperatura y un valor entero tomado como valor lógico que indica si es el último elemento:

```
/* ... */
cantidad = 0;
acumulado = 0.00;
do {
    scanf( "%f", &temperatura );
    acumulado = acumulado + temperatura;
    cantidad = cantidad + 1;
    scanf( "%u", &es_ultimo );
} while( ! es_ultimo );
if( cantidad > 0 ) {
    media = acumulado / (float) cantidad;
} else {
    media = MIN_TEMP;
} /* if */
/* ... */
```

En caso de que se conociera el número de temperaturas (NTEMP) que se han registrado, bastaría con emplear un bucle de tipo **for**:

```
/* ... */  
acumulado = 0.00;  
for( cant = 1; cant <= NTEMP; cant = cant + 1 ) {  
    scanf( "%f", &temperatura );  
    acumulado = acumulado + temperatura;  
} /* for */  
media = acumulado / (float) NTEMP;  
/* ... */
```

Búsqueda

Las búsquedas consisten en recorridos, mayoritariamente parciales, de secuencias de datos de entrada. Se recorren los datos de una secuencia de entrada hasta encontrar el que satisfaga una determinada condición. Evidentemente, si no se encuentra ningún elemento que satisfaga la condición, se realizará el recorrido completo de la secuencia.



De forma general, la búsqueda consiste en recorrer una secuencia de datos de entrada hasta que se cumpla una determinada condición o se acaben los elementos de la secuencia. No es necesario que la condición afecte a un único elemento.

Siguiendo el ejemplo anterior, es posible hacer una búsqueda que detenga el recorrido cuando la media progresiva se mantenga en un margen de ± 1 °C respecto de la temperatura detectada durante más de 10 registros.

El esquema algorítmico es muy parecido al del recorrido, salvo por el hecho de que se incorpora la condición de búsqueda y que, a la

salida del bucle, es necesario comprobar si la búsqueda se ha resuelto satisfactoriamente o no:

```
/* inicialización para el procesamiento de la secuencia */
/* (puede incluir el tratamiento del primer elemento)      */
encontrado = FALSO;
while( ! /* final de secuencia */ && !encontrado ) {
    /* tratar el elemento */
    if( /* condición de encontrado */ ) {
        encontrado = CIERTO;
    } else {
        /* avanzar en secuencia */
    } /* if */
} /* while */
/* finalización del procesamiento de la secuencia           */
if( encontrado ) {
    /* instrucciones */
} else {
    /* instrucciones */
} /* if */
```

En este esquema se supone que se han definido las constantes FALSO y CIERTO del modo siguiente:

```
#define FALSO 0
#define CIERTO 1
```

Si se aplica el patrón anterior a la búsqueda de una media progresiva estable, el código fuente sería el siguiente:

```
/* ... */
cantidad = 0;
acumulado = 0.00;
scanf( "%f", &temperatura );
seguidos = 0;
encontrado = FALSO;
while( !(temperatura == MIN_TEMP) && !encontrado ) {
    acumulado = acumulado + temperatura;
    cantidad = cantidad + 1;
    media = acumulado / (float) cantidad;
```

```
if( media<=temperatura+1.0 || temperatura-1.0<=media ) {  
    seguidos = seguidos + 1;  
} else {  
    seguidos = 0;  
} /* if */  
if( seguidos == 10 ) {  
    encontrado = CIERTO;  
} else {  
    scanf( "%f", &temperatura );  
} /* if */  
} /* while */  
/* ... */
```

En los casos de búsqueda no suele ser conveniente emplear un `for`, ya que suele ser una instrucción iterativa que emplea un contador que toma una serie de valores desde uno inicial hasta uno final. Es decir, hace un recorrido por la secuencia implícita de todos los valores que toma la variable de conteo.

2.4.2. Filtros y tuberías

Los filtros son programas que generan una secuencia de datos a partir de un recorrido de una secuencia de datos de entrada. Habitualmente, la secuencia de datos de salida contiene los datos procesados de la de entrada.

El nombre de *filtros* se les aplica porque es muy común que la secuencia de salida sea, simplemente, una secuencia de datos como la de entrada en la que se han suprimido algunos de sus elementos.

Un filtro sería, por ejemplo, un programa que tuviera como salida las sumas parciales de los números de entrada:

```
#include <stdio.h>  
main()  
{  
    float suma, sumando;  
    suma = 0.00;  
    while( scanf( "%f", &sumando ) == 1 ) {
```

```

        suma = suma + sumando;
        printf( "%.2f ", suma );
    } /* while */
} /* main */

```

Otro filtro, quizá más útil, podría tratarse de un programa que sustituya los tabuladores por el número de espacios en blanco necesarios hasta la siguiente columna de tabulación:

```

#include <stdio.h>
#define TAB 8
main()
{
    char caracter;
    unsigned short posicion, tabulador;
    posicion = 0;
    caracter = getchar();
    while( caracter != EOF ) {
        switch( caracter ) {
            case '\t':/* avanza hasta siguiente columna*/
                for( tabulador = posicion;
                     tabulador < TAB;
                     tabulador = tabulador + 1 ) {
                    putchar( ' ' );
                } /* for */
                posicion = 0;
                break;
            case '\n':/* nueva línea implica columna 0 */
                putchar( caracter );
                posicion = 0;
                break;
            default:
                putchar( caracter );
                posicion = (posicion + 1) % TAB;
        } /* switch */
        caracter = getchar();
    } /* while */
} /* main */

```

Estos pequeños programas pueden resultar útiles por sí mismos o bien combinados. Así pues, la secuencia de datos de salida de uno puede

constituir la secuencia de entrada de otro, constituyéndose lo que denominamos una **tubería** (pipe, en inglés) la idea visual es que por un extremo de la tubería se introduce un flujo de datos y por el otro se obtiene otro flujo de datos ya procesado. En el camino, la tubería puede incluir uno o más filtros que retienen y/o transforman los datos.

Ejemplo

Un filtro podría convertir una secuencia de datos de entrada consistentes en tres números (código de artículo, precio y cantidad) a una secuencia de datos de salida de dos números (código e importe) y el siguiente podría consistir en un filtro de suma, para recoger el importe total.

Para que esto sea posible, es necesario contar con la ayuda del sistema operativo. Así pues, no es necesario que la entrada de datos se efectúe a través del teclado ni tampoco que la salida de datos sea obligatoriamente por la pantalla, como dispositivos de entrada y salida estándar que son. En Linux (y también en otros SO) se puede redirigir la entrada y la salida estándar de datos mediante los comandos de redirección. De esta manera, se puede conseguir que la entrada estándar de datos sea un fichero determinado y que los datos de salida se almacenen en otro fichero que se emplee como salida estándar.

En el ejemplo anterior, se puede suponer que existe un fichero (`ticket.dat`) con los datos de entrada y queremos obtener el total de la compra. Para ello, podemos emplear un filtro para calcular los importes parciales, cuya salida será la entrada de otro que obtenga el total.

Para aplicar el primer filtro, será necesario que ejecutemos el programa correspondiente (al que llamaremos `calcula_importes`) redirigiendo la entrada estándar al fichero `ticket.dat`, y la salida al fichero `importes.dat`:

```
$ calcula_importes <ticket.dat >importes.dat
```

Con ello, `importes.dat` recogerá la secuencia de pares de datos (código de artículo e importe) que el programa haya generado por

la salida estándar. Las redirecciones se determinan mediante los símbolos “menor que” para la entrada estándar y “mayor que” para la salida estándar.

Si deseamos calcular los importes de otras compras para luego calcular la suma de todas ellas, será conveniente añadir a `importes.dat` todos los importes parciales de todos los boletos de compra. Esto es posible mediante el operador de redirección de salida doblado, cuyo significado podría ser “añadir al fichero la salida estándar del programa”:

```
$ calcula_importes <otro_ticket.dat >>importes.dat
```

Cuando hayamos recogido todos los importes parciales que queramos sumar, podremos proceder a llamar al programa que calcula la suma:

```
$ suma <importes.dat
```

Si sólo nos interesa la suma de un único boleto de compra, podemos montar una tubería en la que la salida del cálculo de los importes parciales sea la entrada de la suma:

```
$ calcula_importes <ticket.dat | suma
```

Como se puede observar en el comando anterior, la tubería se monta con el operador de tubería representado por el símbolo de la barra vertical. Los datos de la salida estándar de la ejecución de lo que le precede los transmite como entrada estándar al programa que tenga a continuación.

2.5. Depurado de programas

El depurado de programas consiste en eliminar los errores que éstos contengan. Los errores pueden ser debidos tanto a la programación como al algoritmo programado. Así pues, el depurado de un programa puede implicar un cambio en el algoritmo correspondiente.

Cuando la causa del error se encuentra en el algoritmo o en su in-

correcta programación se habla de **error de lógica**. En cambio, si el error tiene su razón en la violación de las normas del lenguaje de programación se habla de un **error de sintaxis** (aunque algunos errores tengan una naturaleza léxica o semántica).

Nota

Debug es el término inglés para referirse al depurado de errores de programas de computador. El verbo se puede traducir por “eliminar bichos” y tiene su origen en un reporte de 1945 sobre una prueba del computador Mark II realizada en la Universidad de Harvard. En el informe se registró que se encontró una polilla en un relé que provocaba su mal funcionamiento. Para probar que se había quitado el bicho (y resuelto el error), se incluyó la polilla en el mismo informe. Fue sujetada mediante cinta adhesiva y se añadió un pie que decía “primer caso de una polilla encontrada”. Fue también la primera aparición del verbo debug (quitar bichos) que tomó la acepción actual.

Los errores de sintaxis son detectados por el compilador, ya que le impiden generar código ejecutable. Si el compilador puede generar código a pesar de la posible existencia de un error, el compilador suele emitir un aviso.

Por ejemplo, es posible que una expresión en un `if` contenga un operador de asignación, pero lo habitual es que se trate de una confusión entre operadores de asignación y de comparación:

```
/* ... */  
if( a = 5 ) b = 6;  
/* ... */
```

Más aún, en el código anterior se trata de un error de programación, puesto que la instrucción parece indicar que es posible que `b` pueda no ser 6. Si atendemos a la condición del `if`, se trata de una asignación del valor 5 a la variable `a`, con resultado igual al valor asignado. Así pues, como el valor 5 es distinto de cero, el resultado es siempre afirmativo y, consecuentemente, `b` siempre toma el valor 6.

Por este motivo, es muy recomendable que el compilador nos dé todos los avisos que pueda. Para ello, debe ejecutarse con el argumento siguiente:

```
$ gcc -Wall -o programa programa.c
```

El argumento `-Wall` indica que se dé aviso de la mayoría de casos en los que pueda haber algún error lógico. A pesar de que el argumento parece indicar que se nos avisará sobre cualquier situación, aún hay algunos casos sobre los que no avisa.



Los errores más difíciles de detectar son los errores lógicos que escapan incluso a los avisos del compilador. Estos errores son debidos a una programación indebida del algoritmo correspondiente, o bien, a que el propio algoritmo es incorrecto. En todo caso, después de su detección hay que proceder a su localización en el código fuente.

Para la localización de los errores será necesario determinar en qué estado del entorno se producen; es decir, bajo qué condiciones ocurren. Por lo tanto, es necesario averiguar qué valores de las variables conducen el flujo de ejecución del programa a la instrucción en la que se manifiesta el error.

Desafortunadamente, los errores suelen manifestarse en un punto posterior al del estado en el que realmente se produjo el fallo del comportamiento del programa. Así pues, es necesario poder observar el estado del programa en cualquier momento para seguir su evolución hasta la manifestación del error con el propósito de detectar el fallo que lo causa.

Para aumentar la **observabilidad** de un programa, es habitual introducir testigos (también llamados *chivatos*) en su código de manera que nos muestren el contenido de determinadas variables. De todas maneras, este procedimiento supone la modificación del programa cada vez que se introducen nuevos testigos, se eliminan los que resultan innecesarios o se modifican.

Por otra parte, para una mejor localización del error, es necesario poder tener control sobre el flujo de ejecución. La **controlabilidad** implica la capacidad de modificar el contenido de las variables y de elegir entre distintos flujos de ejecución. Para conseguir un cierto grado de control es necesario introducir cambios significativos en el programa que se examina.

En lugar de todo lo anterior, es mejor emplear una herramienta que nos permita observar y controlar la ejecución de los programas para su depuración. Estas herramientas son los llamados *depuradores* (*debuggers*, en inglés).

Para que un depurador pueda realizar su trabajo, es necesario compilar los programas de manera que el código resultante incluya información relativa al código fuente. Así pues, para depurar un programa, deberemos compilarlo con la opción `-g`:

```
$ gcc -Wall -o programa -g programa.c
```

En GNU/C existe un depurador llamado `gdb` que nos permitirá ejecutar un programa, hacer que se pare en determinadas condiciones, examinar el estado del programa cuando esté parado y, por último, cambiarlo para poder experimentar posibles soluciones.

El depurador se invoca de la siguiente manera:

```
$ gdb programa
```

En la tabla siguiente se muestran algunos de los comandos que podemos indicarle a GDB:

Tabla 8.

Comando	Acción
<code>run</code>	Empieza a ejecutar el programa por su primera instrucción. El programa sólo se detendrá en un punto de parada, cuando el depurador reciba un aviso de parada (es decir, con el tecleo de control y C simultáneamente), o bien cuando espere alguna entrada de datos.
<code>break num_línea</code>	Establece un punto de parada antes de la primera instrucción que se encuentra en la línea indicada del código fuente. Si se omite el número de línea, entonces lo establece en la primera instrucción de la línea actual; es decir, en la que se ha detenido.
<code>clear num_línea</code>	Elimina el punto de parada establecido en la línea indicada o, si se omite, en la línea actual.
<code>c</code>	Continúa la ejecución después de una detención.

Comando	Acción
next	Ejecuta la instrucción siguiente y se detiene.
print expresión	Imprime el resultado de evaluar la expresión indicada. En particular, la expresión puede ser una variable y el resultado de su evaluación, su contenido.
help	Muestra la lista de los comandos.
quit	Finaliza la ejecución de GDB

Los puntos de parada o *breakpoints* son marcas en el código ejecutable que permiten al depurador conocer si debe parar la ejecución del programa en curso o, por el contrario, debe continuar permitiendo su ejecución. Estas marcas se pueden fijar o eliminar a través del propio depurador. Con ello, es posible ejecutar porciones de código de forma unitaria.

En particular, puede ser conveniente introducir un punto de parada en `main` antes de proceder a su ejecución, de manera que se detenga en la primera instrucción y nos permita realizar un mejor seguimiento de la misma. Para tal efecto, basta con el comando `break main`, ya que es posible indicar nombres de funciones como puntos de parada.

De todas maneras, es mucho más práctico emplear algún entorno gráfico en el que pueda verse el código fuente al mismo tiempo que la salida del programa que se ejecuta. Para ello, se puede emplear, por ejemplo, el DDD (*Data Display Debugger*) o el XXGDB. Los dos entornos emplean el GDB como depurador y, por tanto, disponen de las mismas opciones. No obstante, su manejo es más fácil porque la mayoría de comandos están a la vista y, en todo caso, en los menús desplegables de que disponen.

2.6. Estructuras de datos

Los tipos de datos básicos (compatibles con enteros y reales) pueden agruparse en estructuras homogéneas o heterogéneas, de manera que se facilita (y aclara) el acceso a sus componentes dentro de un programa.



Una estructura homogénea es aquella cuyos datos son todos del mismo tipo y una heterogénea puede estar formada por datos de tipo distinto.

En los apartados siguientes se revisarán las principales estructuras de datos en C, aunque existen en todos los lenguajes de programación estructurada. Cada apartado se organiza de manera que se vea cómo se pueden llevar a cabo las siguientes operaciones sobre las variables:

- Declararlas, para que el compilador les reserve el espacio correspondiente.
- Inicializarlas, de manera que el compilador les dé un contenido inicial (que puede cambiar) en el programa ejecutable resultante.
- Referenciarlas, de forma que se pueda acceder a su contenido, tanto para modificarlo como para leerlo.

Como es obligatorio anteponer el tipo de la variable en su declaración, resulta conveniente identificar a los tipos de datos estructurados con un nombre de tipo. Estos nuevos tipos de datos se conocen como *tipos de datos abstractos*. El último apartado estará dedicado a ellos.

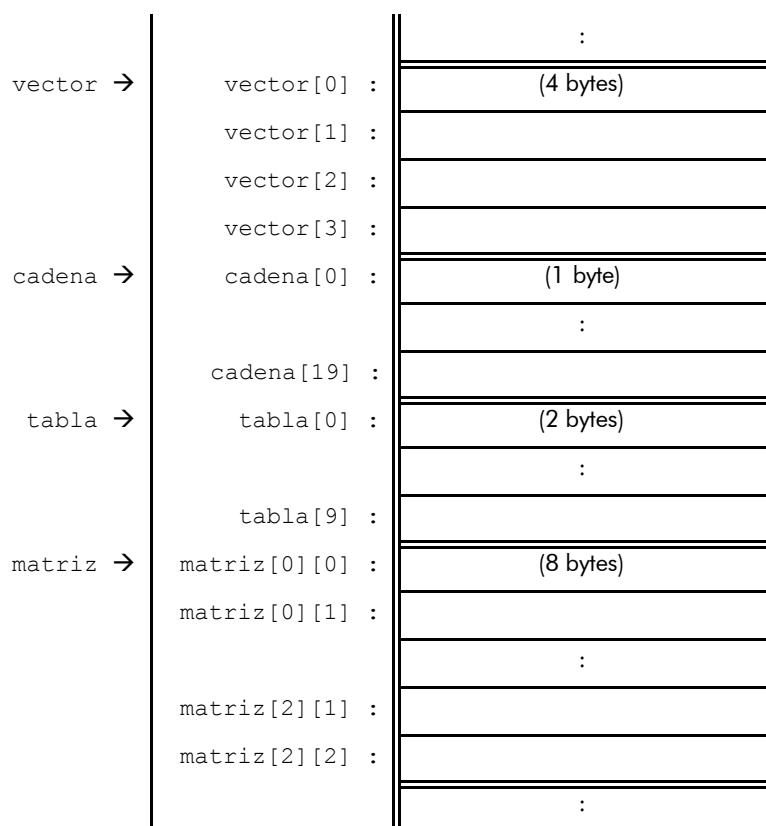
2.7. Matrices

Las matrices son estructuras de datos homogéneas de tamaño fijo. Es decir, se representa siempre una información que emplea un número determinado de datos. También se llaman **arreglos** (una traducción bastante directa del término *array* en inglés), **tablas** (para las de una o dos dimensiones) o **vectores** (si son unidimensionales). En el caso particular de los vectores de caracteres, reciben el nombre de **cadenas de caracteres** o *strings*, en inglés.

2.7.1. Declaración

A continuación se muestran cuatro declaraciones de matrices distintas y un esquema de su distribución en la memoria del computador. El número de bytes de cada división depende del tipo de dato de cada matriz correspondiente. En el esquema ya se avanza el nombre de cada dato dentro de la matriz, en el que se distingue el nombre común de la matriz y una identificación particular del dato, que se corresponde con la posición del elemento en ella. Es muy importante tener presente que las posiciones siempre se numeran desde 0, en C.

Figura 2.



A continuación, se muestran las declaraciones de las variables que conducirían hasta la distribución en memoria que se ha visto:

```

int           vector[4];
char          cadena[20];
unsigned short tabla[10];
double        matriz[3][3];
  
```

La primera declaración prepara un vector de 4 enteros con signo; la segunda, una cadena de 20 caracteres; la tercera, una tabla de 10 enteros positivos, y la última reserva espacio para una matriz de 3×3 números reales de doble precisión.

Nota

La matriz cuadrada se almacena en la memoria por filas; es decir, primero aparece la primera fila, luego la segunda y así hasta la última.

En caso de que hubiera que declarar una matriz con un número mayor de dimensiones, bastaría con añadir su tamaño entre corchetes en la posición deseada entre el nombre de la estructura y el punto y coma final.

Como se ha comentado, las cadenas de caracteres son, realmente, matrices unidimensionales en C. Es decir, que tienen una longitud máxima fijada por el espacio reservado al vector que les corresponde. Aun así, las cadenas de caracteres pueden ser de longitud variable y, por tanto, se utiliza un marcador de final. En este caso, se trata del carácter NUL del código ASCII, cuyo valor numérico es 0. En el ejemplo anterior, la cadena puede ser cualquier texto de hasta 19 caracteres, pues es necesario prever que el último carácter es el de fin de cadena ('\0').

En todos los casos, especialmente cuando se trata de variables que hayan de contener valores constantes, se puede dar un valor inicial a cada uno de los elementos que contienen.

Hay que tener presente que las matrices se guardan en memoria por filas y que es posible no especificar la primera dimensión (la que aparece inmediatamente después del nombre de la variable) de una matriz. En este caso, tomará las dimensiones necesarias para contener los datos presentes en su inicialización. El resto de dimensiones deben de estar fijadas de manera que cada elemento de la primera dimensión tenga una ocupación de memoria conocido.

En los ejemplos siguientes, podemos observar distintas inicializaciones para las declaraciones de las variables anteriores.

```

int          vector[4] = { 0, 1, 2, 3 };
char         cadena[20] = { 'H', 'o', 'l', 'a', '\0' };
unsigned short tabla[10] = { 98, 76, 54, 32, 1, };
double       matriz[3][3] = {{0.0, 0.1, 0.2},
                           {1.0, 1.1, 1.2},
                           {2.0, 2.1, 2.2} };

```

En el caso de la cadena de caracteres, los elementos en posiciones posteriores a la ocupada por '\0' no tendrán ningún valor inicial. Es más, podrán tener cualquier valor. Se trata, pues, de una inicialización incompleta.

Para facilitar la inicialización de las cadenas de caracteres también es posible hacerlo de la manera siguiente:

```
char cadena[20] = "Hola";
```

Si, además, la cadena no ha de cambiar de valor, es posible aprovechar que no es necesario especificar la dimensión, si ésta se puede calcular a través de la inicialización que se hace de la variable correspondiente:

```
char cadena[] = "Hola";
```

En el caso de la tabla, se realiza una inicialización completa al indicar, con la última coma, que todos los elementos posteriores tendrán el mismo valor que el último dado.

2.7.2. Referencia

Para hacer referencia, en alguna expresión, a un elemento de una matriz, basta con indicar su nombre y la posición que ocupa dentro de ella:

$$\text{matriz}[i_0][i_1]\dots[i_n]$$

donde i_k son expresiones el resultado de las cuales debe de ser un valor entero. Habitualmente, las expresiones suelen ser muy simples: una variable o una constante.

Por ejemplo, para leer de la entrada estándar los datos para la matriz de reales dobles de 3×3 que se ha declarado anteriormente, se podría hacer el programa siguiente en el que, por supuesto, las variables fila y columna son enteros positivos:

```
/* ... */  
for( fila = 0; fila < 3; fila = fila + 1 ) {  
    for( columna = 0; columna < 3; columna = columna + 1 ) {  
        printf( "matriz[%u][%u]=? ", fila, columna );  
        scanf( "%lf ", &dato );  
        matriz[fila][columna] = dato;  
    } /* for */  
} /* for */  
/* ... */
```

Es muy importante tener presente que el compilador de C no añade código para comprobar la validez de los índices de las matrices. Por lo tanto, no se comprueban los límites de las matrices y se puede hacer referencia a cualquier elemento, tanto si pertenece a la matriz como si no. ¡Esto es siempre responsabilidad del programador!

Además, en C, los corchetes son operadores de acceso a estructuras homogéneas de datos (es decir, matrices) que calculan la posición de un elemento a partir de la dirección de memoria base en la que se encuentran y el argumento que se les da. Esto implica que es posible, por ejemplo, acceder a una columna de una matriz cuadrada (por ejemplo: `int A[3][3];`) indicando sólo su primer índice (por ejemplo: `pcol = A[0];`). Más aún, es posible que se cometa el error de referirse a un elemento en la forma `A[1, 2]` (común en otros lenguajes de programación). En este caso, el compilador acepta la referencia al tratarse de una forma válida de acceder a la última columna de la matriz, puesto que la coma es un operador de concatenación de expresiones cuyo resultado es el de la última expresión evaluada; es decir, para el ejemplo dado, la referencia `A[1, 2]` sería, en realidad, `A[2]`.

2.7.3. Ejemplos

En este primer ejemplo, el programa comprobará si una palabra o frase corta es un palíndromo; es decir, si se lee igual de izquierda a derecha que de derecha a izquierda.

Ejemplo

Uno de los palíndromos más conocidos en castellano es el siguiente: dábale arroz a la zorra el abad.

```
#include <stdio.h>
#define LONGITUD 81
#define NULO      '\0'
main( )
{
    char         texto[LONGITUD];
    unsigned int longitud, izq, der;
    printf( "Comprobación de palíndromos.\n" );
    printf( "Introduzca texto: " );
    gets( texto );
    longitud = 0;
    while( texto[longitud] != NULO ) {
        longitud = longitud + 1;
    } /* while */
    izq = 0;
    der = longitud;
    while( (texto[izq] == texto[der]) && (izq < der) ) {
        izq = izq + 1;
        der = der - 1;
    } /* while */
    if( izq < der ) {
        printf( "No es palíndromo.\n" );
    } else {
        printf( "¡Es palíndromo!\n" );
    } /* if */
} /* main */
```

Como `gets` toma como argumento la referencia de toda la cadena de caracteres, es decir, la dirección de la posición inicial de memoria que ocupa, no es necesario emplear el operador de “dirección de”.

El siguiente programa que se muestra almacena en un vector los coeficientes de un polinomio para luego evaluarlo en un determinado punto. El polinomio tiene la forma siguiente:

$$P(x) = a_{\text{MAX_GRADO}-1}x^{\text{MAX_GRADO}} + \dots + a_2x^2 + a_1x + a_0$$

Los polinomios serían almacenados en un vector según la correspondencia siguiente:

$$\begin{aligned} a[\text{MAX_GRADO}-1] &= a_{\text{MAX_GRADO}-1} \\ &\vdots \quad : \\ a[2] &= a_2 \\ a[1] &= a_1 \\ a[0] &= a_0 \end{aligned}$$

El programa deberá evaluar el polinomio para una x determinada según el método de Horner, en el cual el polinomio se trata como si estuviera expresado en la forma:

$$P(x) = (\dots (a_{\text{MAX_GRADO}-1}x + a_{\text{MAX_GRADO}-2})x + \dots + a_1)x + a_0$$

De esta manera, el coeficiente de mayor grado se multiplica por x y le suma el coeficiente del grado precedente. El resultado se vuelve a multiplicar por x , siempre que en este proceso no se haya llegado al término independiente. Si así fuera, ya se habría obtenido el resultado final.

Nota

Con este método se reduce el número de operaciones que habrá que realizar, pues no es necesario calcular ninguna potencia de x .

```
#include <stdio.h>
#define MAX_GRADO 16
main( )
{
    double a[MAX_GRADO];
    double x, resultado;
    int     grado, i;

    printf( "Evaluación de polinomios.\n" );
    for( i = 0; i < MAX_GRADO; i = i + 1 ) {
        a[i] = 0.0;
    } /* for */
    printf( "grado máximo del polinomio = ? " );
    scanf( "%d", &grado );
    if( ( 0 ≤ grado ) && ( grado < MAX_GRADO ) ) {
        for( i = 0; i ≤ grado; i = i + 1 ) {
            printf( "a[%d]*x^%d = ? ", i, i );
            scanf( "%lf", &x );
        }
    }
}
```

```

    a[i] = x;
} /* for */
printf( "x = ? " );
scanf( "%lf", &x );
result = 0.0;
for( i = grado; i > 0; i = i - 1 ) {
    resultado = x * resultado + a[i-1];
} /* for */
printf( "P(%g) = %g\n", x, resultado, x );
} else {
    printf( "El grado debe estar entre 0 y %d!\n",
        MAX_GRADO-1
    ); /* printf */
} /* if */
} /* main */
}

```

Es conveniente, ahora, programar estos ejemplos con el fin de adquirir una cierta práctica en la programación con matrices.

2.8. Estructuras heterogéneas

Ejemplo

Las fechas (día, mes y año), los datos personales (nombre, apellidos, dirección, población, etcétera), las entradas de las guías telefónicas (número, propietario, dirección), y tantas otras.

Las estructuras de datos heterogéneas son aquellas capaces de contener datos de distinto tipo. Generalmente, son agrupaciones de datos (tuplas) que forman una unidad lógica respecto de la información que procesan los programas que las usan.

2.8.1. Tuplas

Las tuplas son conjuntos de datos de distinto tipo. Cada elemento dentro de una tupla se identifica con un nombre de campo específico. Estas tuplas, en C, se denominan *estructuras* (*struct*).

Del mismo modo que sucede con las matrices, son útiles para organizar los datos desde un punto de vista lógico. Esta organización lógica supone poder tratar conjuntos de datos fuertemente relacionados entre sí como una única entidad. Es decir, que los programas que las empleen reflejarán su relación y, por tanto, serán mucho más inteligibles y menos propensos a errores.

Nota

Se consigue mucha más claridad si se emplea una tupla para una fecha que si se emplean tres enteros distintos (día, mes y año). Por otra parte, las referencias a los campos de la fecha incluyen una mención a que son parte de la misma; cosa que no sucede si estos datos están contenidos en variables independientes.

Declaración

La declaración de las estructuras heterogéneas o tuplas en C empieza por la palabra clave `struct`, que debe ir seguida de un bloque de declaraciones de las variables que pertenezcan a la estructura y, a continuación, el nombre de la variable o los de una lista de variables que contendrán datos del tipo que se declara.

Dado que el procedimiento que se acaba de describir se debe repetir para declarar otras tuplas idénticas, es conveniente dar un nombre (entre `struct` y el bloque de declaraciones de sus campos) a las estructuras declaradas. Con esto, sólo es necesario incluir la declaración de los campos de la estructura en la de la primera variable de este tipo. Para las demás, será suficiente con especificar el nombre de la estructura.

Los nombres de las estructuras heterogéneas suelen seguir algún convenio para que sea fácil identificarlas. En este caso, se toma uno de los más extendidos: añadir “`_s`” como posijo del nombre.

El ejemplo siguiente describe cómo podría ser una estructura de datos relativa a un avión localizado por un radar de un centro de control de aviación y la variable correspondiente (`avion`). Como se puede observar, no se repite la declaración de los campos de la misma en la posterior declaración de un vector de estas estructuras para contener la información de hasta `MAXNAV` aviones (se supone que es la máxima concentración de aviones posible al alcance de ese punto de control y que ha sido previamente definido):

```
struct avion_s {  
    double radio, angulo;
```

```

        double     altura;
        char      nombre[33];
        unsigned   codigo;
    } avion;
struct avion_s aviones[MAXNAV];

```

También es posible dar valores iniciales a las estructuras empleando una asignación al final de la declaración. Los valores de los distintos campos tienen que separarse mediante comas e ir incluidos entre llaves:

```

struct persona_s {
    char      nombre[ MAXLONG ];
    unsigned   short edad;
} persona = { "Carmen" , 31 };
struct persona_s ganadora = { "desconocida" , 0 };
struct persona_s gente[] = { { "Eva", 43 },
                            { "Pedro", 51 },
                            { "Jesús", 32 },
                            { "Anna", 37 },
                            { "Joaquín", 42 }
}; /* struct persona_s gente */

```

Referencia

Ejemplo

```
ganadora.edad = 25;
inicial = gente[i].nombre[0];
```

La referencia a un campo determinado de una tupla se hace con el nombre del campo después del nombre de la variable que lo contiene, separando los dos mediante un operador de acceso a campo de estructura (el punto).

En el programa siguiente se emplean variables estructuradas que contienen dos números reales para indicar un punto en el plano de forma cartesiana (struct cartesiano_s) y polar (struct polar_s). El programa pide las coordenadas cartesianas de un punto y las transforma en coordenadas polares (ángulo y radio, o distancia respecto del origen). Obsérvese que se declaran dos variables con una inicialización directa: prec para indicar la precisión con que se trabajará y pi para almacenar el valor de la constante π en la misma precisión.

```
#include <stdio.h>
#include <math.h>

main( )
{
    struct cartesiano_s { double x, y; } c;
    struct polar_s { double radio, angulo; } p;
    double prec = 1e-9;
    double pi = 3.141592654;
    printf( "De coordenadas cartesianas a polares.\n" );
    printf( "x = ? "); scanf( "%lf", &(c.x) );
    printf( "y = ? "); scanf( "%lf", &(c.y) );
    p.radio = sqrt( c.x * c.x + c.y * c.y );
    if( p.radio < prec ) { /* si el radio es cero ... */
        p.angulo = 0.0; /* ... el ángulo es cero. */
    } else {
        if( -prec<c.x && c.x<prec ) { /* si c.x es cero ... */
            if( c.y > 0.0 ) p.angulo = 0.5*pi;
            else p.angulo = -0.5*pi;
        } else {
            p.angulo = atan( c.y / c.x );
        } /* if */
    } /* if */
    printf( "radio = %g\n", p.radio );
    printf( "ángulo = %g (%g grados sexagesimales)\n",
           p.angulo,
           p.angulo*180.0/pi
    ); /* printf */
} /* main */
```

El programa anterior hace uso de las funciones matemáticas estándar `sqrt` y `atan` para calcular la raíz cuadrada y el arco tangente, respectivamente. Para ello, es necesario que se incluya el fichero de cabeceras (`#include <math.h>`) correspondiente en el código fuente.

2.8.2. Variables de tipo múltiple

Son variables cuyo contenido puede variar entre datos de distinto tipo. El tipo de datos debe de estar entre alguno de los que se indican

en su declaración y el compilador reserva espacio para contener al que ocupe mayor tamaño de todos ellos. Su declaración es parecida a la de las tuplas.

Ejemplo

```
union numero_s {
    signed entero;
    unsigned natural;
    float real;
} numero;
```

El uso de esta clase de variables puede suponer un cierto ahorro de espacio. No obstante, hay que tener presente que, para gestionar estos campos de tipo variable, es necesario disponer de información (explícita o implícita) del tipo de dato que se almacena en ellos en un momento determinado.

Así pues, suelen ir combinados en tuplas que dispongan de algún campo que permita averiguar el tipo de datos del contenido de estas variables. Por ejemplo, véase la declaración de la siguiente variable (`seguro`), en la que el campo `tipo_bien` permite conocer cuál de las estructuras del tipo múltiple está presente en su contenido:

```
struct seguro_s {
    unsigned    poliza;
    char        tomador[31];
    char        NIF[9];
    char        tipo_bien; /* 'C': vivienda, */
                           /* 'F': vida, */
                           /* 'M': vehículo. */

    union {
        struct {
            char ref_catastro[];
            float superficie;
        } vivienda;
        struct {
            struct fecha_s nacimiento;
            char beneficiario[31];
        } vida;
        struct {
```

```
char matricula[7];
struct fecha_s fabricacion;
unsigned short siniestros;
} vehículo;
} datos;
unsigned valor;
unsigned prima;
} seguro;
```

Con ello, también es posible tener información sobre una serie de seguros en una misma tabla, independientemente del tipo de póliza que tengan asociados:

```
struct seguro_s asegurados [ NUMSEGUROS ];
```

En todo caso, el uso de union resulta bastante infrecuente.

2.9. Tipos de datos abstractos

Los tipos de datos abstractos son tipos de datos a los que se atribuye un significado con relación al problema que se pretende resolver con el programa y, por tanto, a un nivel de abstracción superior al del modelo computacional. Se trata, pues, de tipos de datos transparentes al compilador y, consecuentemente, irrelevantes en el código ejecutable correspondiente.

En la práctica, cualquier estructura de datos que se defina es, de hecho, una agrupación de datos en función del problema y, por lo tanto, un tipo abstracto de datos. De todas maneras, también puede serlo un entero que se emplee con una finalidad distinta; por ejemplo, sólo para almacenar valores lógicos ('cierto' o 'falso').

En cualquier caso, el uso de los tipos abstractos de datos permite aumentar la legibilidad del programa (entre otras cosas que se verán más adelante). Además, hace posible emplear declaraciones de tipos anteriormente descritos sin tener que repetir parte de la declaración, pues basta con indicar el nombre que se le ha asignado.

2.9.1. Definición de tipos de datos abstractos

Para definir un nuevo nombre de tipo de dato es suficiente con hacer una declaración de una variable antecedida por `typedef`. En esta declaración, lo que sería el nombre de la variable será, de hecho, el nombre del nuevo tipo de datos. A continuación, se muestran varios ejemplos.

```
typedef char boolean, logico;
#define MAXSTRLEN 81
typedef char cadena[MAXSTRLEN];
typedef struct persona_s {
    cadena     nombre, direccion, poblacion;
    char       codigo_postal[5];
    unsigned   telefono;
} persona_t;
```

En las definiciones anteriores se observa que la sintaxis no varía respecto de la de la declaración de las variables salvo por la inclusión de la palabra clave `typedef`, cuyo significado es, precisamente, un apócope de “define tipo”. Con estas definiciones, ya es posible declarar variables de los tipos correspondientes:

```
boolean  correcto, ok;
cadena   nombre_profesor;
persona_t alumnos[MAX_GRUPO];
```



Es muy recomendable emplear siempre un nombre de tipo que identifique los contenidos de las variables de forma significativa dentro del problema que ha de resolver el programa que las utiliza.

Por ello, a partir de este punto, todos los ejemplos emplearán tipos abstractos de datos cuando sea necesario.

Por lo que atañe a este texto, se preferirá que los nombres de tipo terminen siempre en “`_t`”.

2.9.2. Tipos enumerados

Los tipos de datos enumerados son un tipo de datos compatible con enteros en el que se realiza una correspondencia entre un entero y un determinado símbolo (la constante de enumeración). En otras palabras, es un tipo de datos entero en el que se da nombre (enumeran) a un conjunto de valores. En cierta manera, su empleo sustituye al comando de definición de constantes simbólicas del preprocesador (#define) cuando éstas son de tipo entero.

El ejemplo siguiente ilustra cómo se declaran y cómo se pueden emplear:

```
/* ... */
enum { ROJO, VERDE, AZUL } rgb;
enum bool_e { FALSE = 0, TRUE = 1 } logico;
enum bool_e encontrado;
int color;
/* ... */
rgb = VERDE;
/* Se puede asignar un enumerado a un entero: */
color = ROJO;
logico = TRUE;
/* Se puede asignar un entero a un enumerado, */
/* aunque no tenga ningún símbolo asociado: */
encontrado = -1;
/* ... */
```

La variable enumerada `rgb` podrá contener cualquier valor entero (de tipo `int`), pero tendrá tres valores enteros identificados con los nombres `ROJO`, `VERDE` y `AZUL`. Si el valor asociado a los símbolos importa, se tiene que asignar a cada símbolo su valor mediante el signo igual, tal como aparece en la declaración de `logico`.

El tipo enumerado puede tener un nombre específico (`bool_e` en el ejemplo) que evite la repetición del enumerado en una declaración posterior de una variable del mismo tipo (en el ejemplo: `encontrado`).

También es posible y recomendable definir un tipo de dato asociado:

```
typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;
```

En este caso particular, se emplea el nombre `bool` en lugar de `bool_t` o `logico` por coincidir con el nombre del tipo de datos primitivo de C++. Dado lo frecuente de su uso, en el resto del texto se considerará definido. (No obstante, habrá que tener presente que una variable de este tipo puede adquirir valores distintos de 1 y ser, conceptualmente, ‘cierta’ o `TRUE`.)

2.9.3. Ejemplo

En los programas vistos con anterioridad se emplean variables de tipos de datos estructurados y que son (o podrían ser) frecuentemente usados en otros programas de la misma índole. Así pues, es conveniente transformar las declaraciones de tipo de los datos estructurados en definición de tipos.

En particular, el programa de evaluación de polinomios por el método de Horner debería haber dispuesto de un tipo de datos estructurado que representara la información de un polinomio (grado máximo y coeficientes).

El programa que se muestra a continuación contiene una definición del tipo de datos `polinomio_t` para identificar a sus componentes como datos de un mismo polinomio. El grado máximo se emplea también para saber qué elementos del vector contienen los coeficientes para cada grado y cuáles no. Este programa realiza la derivación simbólica de un polinomio dado (la derivación simbólica implica obtener otro polinomio que representa la derivada de la función polinómica dada como entrada).

```
#include <stdio.h>

#define MAX_GRADO 16

typedef struct polinomio_s {
    int      grado;
    double   a[MAX_GRADO];
} polinomio_t;
```

```
main( )
{
    polinomio_t p;
    double      x, coef;
    int         i, grado;
    p.grado = 0; /* inicialización de (polinomio_t) p */
    p.a[0] = 0.0;
    printf( "Derivación simbólica de polinomios.\n" );
    printf( "Grado del polinomio = ? " );
    scanf( "%d", &(p.grado) );
    if( ( 0 ≤ p.grado ) && ( p.grado < MAX_GRADO ) ) {
        for( i = 0; i ≤ p.grado; i = i + 1 ) { /* lectura */
            printf( "a[%d]*x^%d = ? ", i, i );
            scanf( "%lf", &coef );
            p.a[i] = coef;
        } /* for */
        for( i = 0; i < p.grado; i = i + 1 ) { /* derivación */
            p.a[i] = p.a[i+1]*(i+1);
        } /* for */
        if( p.grado > 0 ) {
            p.grado = p.grado -1;
        } else {
            p.a[0] = 0.0;
        } /* if */
        printf( "Polinomio derivado:\n" );
        for( i = 0; i < p.grado; i = i + 1 ) { /* impresión */
            printf( "%g*x^%d +", p.a[i], i );
        } /* for */
        printf( "%g\n", p.a[i] );
    } else {
        printf( "El grado del polinomio tiene que estar" );
        printf( " entre 0 y %d!\n", MAX_GRADO-1 );
    } /* if */
} /* main */
```

Ejemplo

Unidades de disquete, de disco duro, de CD, de DVD, de tarjetas de memoria, etc.

2.10. Ficheros

Los ficheros son una estructura de datos homogénea que tiene la particularidad de tener los datos almacenados fuera de la memoria principal. De hecho son estructuras de datos que se encuentran en la llamada memoria externa o secundaria (no obstante, es posible que algunos ficheros temporales se encuentren sólo en la memoria principal).

Para acceder a los datos de un fichero, el ordenador debe contar con los dispositivos adecuados que sean capaces de leer y, opcionalmente, escribir en los soportes adecuados.

Por el hecho de residir en soportes de información permanentes, pueden mantener información entre distintas ejecuciones de un mismo programa o servir de fuente y depósito de información para cualquier programa.

Dada la capacidad de muchos de estos soportes, el tamaño de los ficheros puede ser mucho mayor incluso que el espacio de memoria principal disponible. Por este motivo, en la memoria principal sólo se dispone de una parte del contenido de los ficheros en uso y de la información necesaria para su manejo.

No menos importante es que los ficheros son estructuras de datos con un número indefinido de éstos.

En los próximos apartados se comentarán los aspectos relacionados con los ficheros en C, que se denominan ficheros de flujo de bytes (en inglés, *byte streams*). Estos ficheros son estructuras homogéneas de datos simples en los que cada dato es un único byte. Habitualmente, los hay de dos tipos: los ficheros de texto ASCII (cada byte es un carácter) y los ficheros binarios (cada byte coincide con algún byte que forma parte de algún dato de alguno de los tipos de datos que existen).

2.10.1. Ficheros de flujo de bytes

Los ficheros de tipo *byte stream* de C son secuencias de bytes que se pueden considerar bien como una copia del contenido de la memo-

ria (binarios), bien como una cadena de caracteres (textuales). En este apartado nos ocuparemos especialmente de estos últimos por ser los más habituales.

Dado que están almacenados en un soporte externo, es necesario disponer de información sobre los mismos en la memoria principal. En este sentido, toda la información de control de un fichero de este tipo y una parte de los datos que contiene (o que habrá de contener, en caso de escritura) se recoge en una única variable de tipo FILE.

El tipo de datos FILE es una tupla compuesta, entre otros campos, por el nombre del fichero, la longitud del mismo, la posición del último byte leído o escrito y un buffer (memoria temporal) que contiene BUFSIZ byte del fichero. Este último es necesario para evitar accesos al dispositivo periférico afectado y, dado que las operaciones de lectura y escritura se hacen por bloques de bytes, para que éstas se hagan más rápidamente.

Afortunadamente, hay funciones estándar para hacer todas las operaciones que se acaban de insinuar. Al igual que la estructura FILE y la constante BUFSIZ, están declaradas en el fichero stdio.h. En el próximo apartado se comentarán las más comunes.

2.10.2. Funciones estándar para ficheros

Para acceder a la información de un fichero, primero se tiene que "abrir". Es decir, hay que localizarlo y crear una variable de tipo FILE. Para ello, la función de apertura da como resultado de su ejecución la posición de memoria en la que se encuentra la variable que crea o NULL si no ha conseguido abrir el fichero indicado.

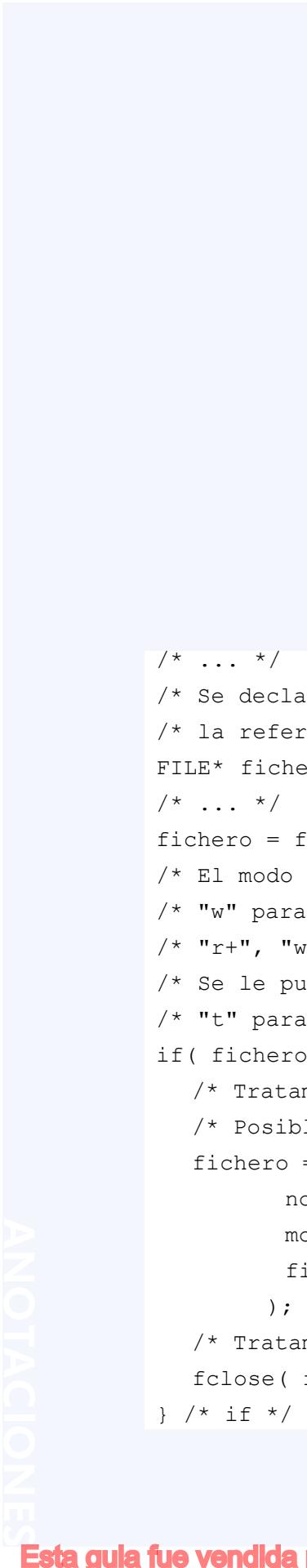
Cuando se abre un fichero, es necesario especificar si se leerá su contenido (modo_apertura = "r"), si se le quieren añadir mas datos (modo_apertura = "a"), o si se quiere crear de nuevo (modo_apertura = "w").

También es conveniente indicar si el fichero es un fichero de texto (los finales de línea pueden transformarse ligeramente) o si es binario.

Esto se consigue añadiendo al modo de apertura una "t" o una

Nota

En el tercer caso es necesario tener cuidado: si el fichero ya existiera, se perdería todo su contenido!



"b", respectivamente. Si se omite esta información, el fichero se abre en modo texto.

Una vez abierto, se puede leer su contenido o bien escribir nuevos datos en él. Después de haber operado con el fichero, hay que cerrarlo. Esto es, hay que indicar al sistema operativo que ya no se trabajará más con él y, sobre todo, hay que acabar de escribir los datos que pudieran haber quedado pendientes en el *buffer* correspondiente. De todo esto se ocupa la función estándar de cierre de fichero.

En el código siguiente se refleja el esquema algorítmico para el trabajo con ficheros y se detalla, además, una función de reapertura de ficheros que aprovecha la misma estructura de datos de control. Hay que tener en cuenta que esto supone cerrar el fichero anteriormente abierto:

```
/* ... */
/* Se declara una variable para que contenga           */
/* la referencia de la estructura FILE:                 */
FILE* fichero;
/* ... */

fichero = fopen( nombre_fichero, modo_apertura );
/* El modo de apertura puede ser "r" para lectura,      */
/* "w" para escritura, "a" para añadidura y            */
/* "r+", "w+" o "a+" para actualización (leer/escribir). */
/* Se le puede añadir el sufijo                         */
/* "t" para texto o "b" para binario.                   */
if( fichero != NULL ) {
    /* Tratamiento de los datos del fichero.             */
    /* Posible reapertura del mismo fichero:            */
    fichero = freopen(
        nombre_fichero,
        modo_apertura,
        fichero
    ); /* freopen */
    /* Tratamiento de los datos del fichero.             */
    fclose( fichero );
} /* if */
```

En los próximos apartados se detallan las funciones estándar para trabajar con ficheros de flujo o *streams*. Las variables que se em-

plean en los ejemplos son del tipo adecuado para lo que se usan y, en particular, `flujo` es de tipo `FILE*`; es decir, referencia a estructura de fichero.

Funciones estándar de entrada de datos (lectura) de ficheros

Estas funciones son muy similares a las ya vistas para la lectura de datos procedentes de la entrada estándar. En éstas, sin embargo, será muy importante saber si ya se ha llegado al final del fichero y, por lo tanto, ya no hay más datos para su lectura.

```
fscanf( flujo, "formato" [,lista_de_variables ] )
```

De funcionamiento similar a `scanf()`, devuelve como resultado el número de argumentos realmente leídos. Por tanto, ofrece una manera indirecta de determinar si se ha llegado al final del fichero. En este caso, de todas maneras, activa la condición de fin de fichero. De hecho, un número menor de asignaciones puede deberse, simplemente, a una entrada inesperada, como por ejemplo, leer un carácter alfabético para una conversión "%d".

Por otra parte, esta función devuelve `EOF` (del inglés *end of file*) si se ha llegado a final de fichero y no se ha podido realizar ninguna asignación. Así pues, resulta mucho más conveniente emplear la función que comprueba esta condición antes que comprobarlo de forma indirecta mediante el número de parámetros correctamente leídos (puede que el fichero contenga más datos) o por el retorno de `EOF` (no se produce si se ha leído, al menos, un dato).

Ejemplo

```
fscanf( flujo, "%u%c", &num_dni, &letra_nif );
fscanf( flujo, "%d%d%d", &codigo, &precio, &cantidad );
```

`feof(flujo)`

Devuelve 0 en caso de que no se haya llegado al final del fichero. En caso contrario devuelve un valor distinto de cero, es decir, que es cierta la condición de final de fichero.

fgetc(flujo)

Lee un carácter del flujo. En caso de no poder efectuar la lectura por haber llegado a su fin, devuelve EOF. Esta constante ya está definida en el fichero de cabeceras stdio.h; por lo tanto, puede emplearse libremente dentro del código.

Nota

Es importante tener presente que puede haber ficheros que tengan un carácter EOF en medio de su flujo, pues el final del fichero viene determinado por su longitud.

fgets(cadena, longitud_maxima, flujo)

Lee una cadena de caracteres del fichero hasta encontrar un final de línea, hasta llegar a longitud_maxima (-1 para la marca de final de cadena) de caracteres, o hasta fin de fichero. Devuelve NULL si encuentra el final de fichero durante la lectura.

Ejemplo

```
if( fgets( cadena, 33, flujo ) != NULL ) puts( cadena );
```

Funciones estándar de salida de datos (escritura) de ficheros

Las funciones que aquí se incluyen también tienen un comportamiento similar al de las funciones para la salida de datos por el dispositivo estándar. Todas ellas escriben caracteres en el flujo de salida indicado:

fprintf(flujo, "formato" [, lista_de_variables])

La función fprintf() escribe caracteres en el flujo de salida indicado con formato. Si ha ocurrido algún problema, esta función devuelve el último carácter escrito o la constante EOF.

fputc(caracter, flujo)

La función fputc() escribe caracteres en el flujo de salida indicado carácter a carácter. Si se ha producido un error de escritura o bien

el soporte está lleno, la función `fputc()` activa un indicador de error del fichero. Este indicador se puede consultar con la función `ferror(flujo)`, que retorna un cero (valor lógico falso) cuando no hay error.

`fputs(cadena, flujo)`

La función `fputs()` escribe caracteres en el flujo de salida indicado permitiendo grabar cadenas completas. Si ha ocurrido algún problema, esta función actúa de forma similar a `fprintf()`.

Funciones estándar de posicionamiento en ficheros de flujo

En los ficheros de flujo es posible determinar la posición de lectura o escritura; es decir, la posición del último byte leído o que se ha escrito. Esto se hace mediante la función `ftell(flujo)`, que devuelve un entero de tipo `long` que indica la posición o `-1` en caso de error.

También hay funciones para cambiar esta posición de lectura (y escritura, si se trata de ficheros que hay que actualizar):

`fseek(flujo, desplazamiento, direccion)`

Desplaza el "cabezal" lector/escritor respecto de la posición actual con el valor del entero largo indicado en `desplazamiento` si `direccion` es igual a `SEEK_CUR`. Si esta dirección es `SEEK_SET`, entonces `desplazamiento` se convierte en un desplazamiento respecto del principio y, por lo tanto, indica la posición final. En cambio, si es `SEEK_END`, indicará el desplazamiento respecto de la última posición del fichero. Si el repositionamiento es correcto, devuelve 0.

`rewind(flujo)`

Sitúa el "cabezal" al principio del fichero. Esta función es equivalente a:

```
seek( flujo, 0L, SEEK_SET );
```

donde la constante de tipo `long int` se indica con el sufijo "L". Esta función permitiría, pues, releer un fichero desde el principio.

Relación con las funciones de entrada/salida por dispositivos estándar

Las entradas y salidas por terminal estándar también se pueden llevar a cabo con las funciones estándar de entrada/salida, o también mediante las funciones de tratamiento de ficheros de flujo. Para esto último es necesario emplear las referencias a los ficheros de los dispositivos estándar, que se abren al iniciar la ejecución de un programa. En C hay, como mínimo, tres ficheros predefinidos: `stdin` para la entrada estándar, `stdout` para la salida estándar y `stderr` para la salida de avisos de error, que suele coincidir con `stdout`.

2.10.3. Ejemplo

Se muestra aquí un pequeño programa que cuenta el número de palabras y de líneas que hay en un fichero de texto. El programa entiende como palabra toda serie de caracteres entre dos espacios en blanco. Un espacio en blanco es cualquier carácter que haga que `isspace()` devuelva cierto. El final de línea se indica con el carácter de retorno de carro (ASCII número 13); es decir, con '`\n`'. Es importante observar el uso de las funciones relacionadas con los ficheros de flujo de bytes.

Como se verá, la estructura de los programas que trabajan con estos ficheros incluye la codificación de alguno de los esquemas algorítmicos para el tratamiento de secuencias de datos (de hecho, los ficheros de flujo son secuencias de bytes) En este caso, como se realiza un conteo de palabras y líneas, hay que recorrer toda la secuencia de entrada. Por lo tanto, se puede observar que el código del programa sigue perfectamente el esquema algorítmico para el recorrido de secuencias.

```
/* Fichero: npalabras.c */  
#include <stdio.h>  
#include <ctype.h> /* Contiene: isspace() */  
  
typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;  
bool esPalabra(char c);
```

```
main()
{
    char nombre_fichero[FILENAME_MAX];
    FILE *flujo;
    bool en_palabra;
    char c;
    unsigned long int npalabras, nlineas;

    printf( "Contador de palabras y líneas.\n" );
    printf( "Nombre del fichero: " );
    gets( nombre_fichero );
    flujo = fopen( nombre_fichero, "rt" );
    if( flujo != NULL ) {
        npalabras = 0;
        nlineas = 0;
        en_palabra = FALSE;
        while( !feof( flujo ) ) {
            c = fgetc( flujo );
            if( c == '\n' ) nlineas = nlineas + 1;
            if( isspace( c ) ) {
                if( en_palabra ) {
                    en_palabra = FALSE;
                    npalabras = npalabras + 1;
                } /* if */
            } else { /* si el carácter no es espacio en blanco */
                en_palabra = TRUE;
            } /* if */
        } /* while */
        printf( "Número de palabras = %lu\n", npalabras );
        printf( "Número de líneas = %lu\n", nlineas );
    } else {
        printf( "No puedo abrir el fichero!\n" );
    } /* if */
} /* main */
```

Nota

La detección de las palabras se hace comprobando los finales de palabra, que tienen que estar formadas por un carácter distinto del espacio en blanco, seguido por uno que lo sea.

2.11. Principios de la programación modular

La lectura del código fuente de un programa implica realizar el seguimiento del flujo de ejecución de sus instrucciones (el flujo de control). Evidentemente, una ejecución en el orden secuencial de las instrucciones no precisa de mucha atención. Pero ya se ha visto que los programas contienen también instrucciones condicionales o alternativas e iterativas. Con todo, el seguimiento del flujo de control puede resultar complejo si el código fuente ocupa más de lo que se puede observar (por ejemplo, más de una veintena de líneas).

Por ello, resulta conveniente agrupar aquellas partes del código que realizan una función muy concreta en un subprograma identificado de forma individual. Es más, esto resulta incluso provechoso cuando se trata de funciones que se realizan en diversos momentos de la ejecución de un programa.

En los apartados siguientes, se verá cómo se distribuye en distintos subprogramas un programa en C. La organización es parecida en otros lenguajes de programación.

2.12. Funciones

En C, a las agrupaciones de código en las que se divide un determinado programa se las llama, precisamente, *funciones*. Más aún, en C, todo el código debe estar distribuido en funciones y, de hecho, el propio programa principal es una función: la función principal (`main`).

Generalmente, una función incluirá en su código, como mucho, la programación de unos pocos esquemas algorítmicos de procesamiento de secuencias de datos y algunas ejecuciones condicionales o alternativas. Es decir, lo necesario para realizar una tarea muy concreta.

2.12.1. Declaración y definición

La declaración de cualquier entidad (variable o función) implica la manifestación de su existencia al compilador, mientras que definirla

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

supone describir su contenido. Tal diferenciación ya se ha visto para las variables, pero sólo se ha insinuado para las funciones.

La declaración consiste exactamente en lo mismo que para las variables: manifestar su existencia. En este caso, de todas maneras hay que describir los argumentos que toma y el resultado que devuelve para que el compilador pueda generar el código, y poderlas emplear.



Los ficheros de cabecera contienen declaraciones de funciones.

En cambio, la definición de una función se corresponde con su programa, que es su contenido. De hecho, de forma similar a las variables, el contenido se puede identificar por la posición del primero de sus bytes en la memoria principal. Este primer byte es el primero de la primera instrucción que se ejecuta para llevar a cabo la tarea que tenga programada.

Declaraciones

La declaración de una función consiste en especificar el tipo de dato que devuelve, el nombre de la función, la lista de parámetros que recibe entre paréntesis y un punto y coma que finaliza la declaración:

```
tipo_de_dato nombre_función( lista_de_parámetros );
```

Hay que tener presente que no se puede hacer referencia a funciones que no estén declaradas previamente. Por este motivo, es necesario incluir los ficheros de cabeceras de las funciones estándar de la biblioteca de C como stdio.h, por ejemplo.



Si una función no ha sido previamente declarada, el compilador supondrá que devuelve un entero. De la misma manera, si se omite el tipo de dato que retorna, supondrá que es un entero.

La lista de parámetros es opcional y consiste en una lista de declaraciones de variables que contendrán los datos tomados como argumentos de la función. Cada declaración se separa de la siguiente por medio de una coma. Por ejemplo:

```
float nota_media( float teo, float prb, float pract );
bool aprueba( float nota, float tolerancia );
```

Si la función no devuelve ningún valor o no necesita ningún argumento, se debe indicar mediante el tipo de datos vacío (`void`):

```
void avisa( char mensaje[] );
bool si_o_no( void );
int lee_codigo( void );
```

Definiciones

La definición de una función está encabezada siempre por su declaración, que ahora debe incluir forzosamente la lista de parámetros si los tiene. Esta cabecera no debe finalizar con punto y coma, sino que irá seguida del cuerpo de la función, delimitada entre llaves de apertura y cierre:

```
tipo_de_dato nombre_función( lista_de_parámetros )
{ /* cuerpo de la función:
   /* 1) declaración de variables locales */
   /* 2) instrucciones de la función */
} /* nombre_función */
```

Tal como se ha comentado anteriormente, la definición de la función ya supone su declaración. Por lo tanto, las funciones que realizan tareas de otros programas y, en particular, del programa principal (la función `main`) se definen con anterioridad.

Llamadas

El mecanismo de uso de una función en el código es el mismo que se ha empleado para las funciones de la biblioteca estándar de C: basta

con referirse a ellas por su nombre, proporcionarles los argumentos necesarios para que puedan llevar a cabo la tarea que les corresponda y, opcionalmente, emplear el dato devuelto dentro de una expresión, que será, habitualmente, de condición o de asignación.

El procedimiento por el cual el flujo de ejecución de instrucciones pasa a la primera instrucción de una función se denomina *procedimiento de llamada*. Así pues, se hablará de llamadas a funciones cada vez que se indique el uso de una función en un programa.

A continuación se presenta la secuencia de un procedimiento de llamada:

1. Preparar el entorno de ejecución de la función; es decir, reservar espacio para el valor de retorno, los parámetros formales (las variables que se identifican con cada uno de los argumentos que tiene), y las variables locales.
2. Realizar el paso de parámetros; es decir, copiar los valores resultantes de evaluar las expresiones en cada uno de los argumentos de la instrucción de llamada a los parámetros formales.
3. Ejecutar el programa correspondiente.
4. Liberar el espacio ocupado por el entorno local y devolver el posible valor de retorno antes de regresar al flujo de ejecución de instrucciones en donde se encontraba la llamada.

El último punto se realiza mediante la instrucción de retorno que, claro está, es la última instrucción que se ejecutará en una función:

```
return expresión;
```

Nota

Esta instrucción debe aparecer vacía o no aparecer si la función es de tipo `void`; es decir, si se la ha declarado explícitamente para no devolver ningún dato.

En el cuerpo de la función se puede realizar una llamada a la misma. Esta llamada se denomina *llamada recursiva*, ya que la defi-

nición de la función se hace en términos de ella misma. Este tipo de llamadas no es incorrecto pero hay que vigilar que no se produzcan indefinidamente; es decir, que haya algún caso donde el flujo de ejecución de las instrucciones no implique realizar ninguna llamada recursiva y, por otra parte, que la transformación que se aplica a los parámetros de éstas conduzca, en algún momento, a las condiciones de ejecución anterior. En particular, **no** se puede hacer lo siguiente:

```
/* ... */
void menu( void )
{
    /* mostrar menú de opciones,      */
    /* ejecutar opción seleccionada */
    menu();
/* ... */
```

La función anterior supone realizar un número indefinido de llamadas a `menu()` y, por tanto, la continua creación de entornos locales sin su posterior liberación. En esta situación, es posible que el programa no pueda ejecutarse correctamente tras un tiempo por falta de memoria para crear nuevos entornos.

2.12.2. Ámbito de las variables

El **ámbito de las variables** hace referencia a las partes del programa que las pueden emplear. Dicho de otra manera, el **ámbito de las variables** abarca todas aquellas instrucciones que pueden acceder a ellas.

En el código de una función se pueden emplear todas las variables globales (las que son “visibles” por cualquier instrucción del programa), todos los parámetros formales (las variables que equivalen a los argumentos de la función), y todas las variables locales (las que se declaran dentro del cuerpo de la función).

En algunos casos puede no ser conveniente utilizar variables globales, pues dificultarían la compresión del código fuente, cosa que di-

ficultaría el posterior depurado y mantenimiento del programa. Para ilustrarlo, veamos el siguiente ejemplo:

```
#include <stdio.h>

unsigned int A, B;

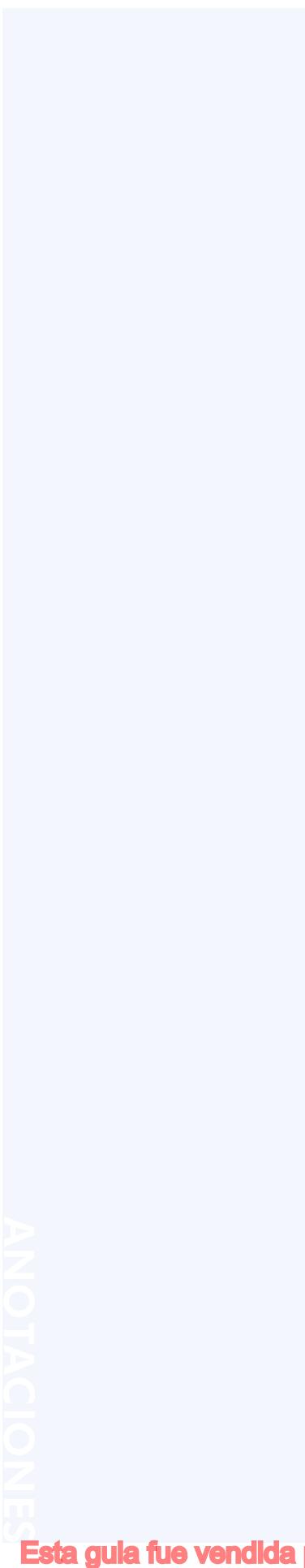
void reduce( void )
{
    if( A < B ) B = B - A;
    else A = A - B;
} /* reduce */

void main( void )
{
    printf( "El MCD de: " );
    scanf( "%u%u", &A, &B );
    while( A!=0 && B!=0 ) reduce();
    printf( "... es %u\n", A + B );
} /* main */
```

Aunque el programa mostrado tiene un funcionamiento correcto, no es posible deducir directamente qué hace la función `reduce()`, ni tampoco determinar de qué variables depende ni a cuáles afecta. Por tanto, hay que adoptar como norma que ninguna función dependa o afecte a variables globales. De ahí el hecho de que, en C, todo el código se distribuye en funciones, se deduce fácilmente que no debe haber ninguna variable global.

Así pues, todas las variables son de ámbito local (parámetros formales y variables locales). En otras palabras, se declaran en el entorno local de una función y sólo pueden ser empleadas por las instrucciones dentro de la misma.

Las variables locales se crean en el momento en que se activa la función correspondiente, es decir, después de ejecutar la instrucción de llamada de la función. Por este motivo, tienen una clase de almacenamiento denominada automática, ya que son creadas y destruidas de forma automática en el procedimiento de llamada a función. Esta



clase de almacenamiento se puede hacer explícita mediante la palabra clave `auto`:

```
int una_funcion_cualquiera( int a, int b )
{
    /* ... */
    auto int variable_local;
    /* resto de la función */
} /* una_funcion_cualquiera */
```

A veces resulta interesante que la variable local se almacene temporalmente en uno de los registros del procesador para evitar tener que actualizarla continuamente en la memoria principal y acelerar, con ello, la ejecución de las instrucciones involucradas (normalmente, las iterativas). En estos casos, se puede aconsejar al compilador que genere código máquina para que se haga así; es decir, para que el almacenamiento de una variable local se lleve a cabo en uno de los registros del procesador. De todas maneras, muchos compiladores son capaces de llevar a cabo tales optimizaciones de forma autónoma.

Esta clase de almacenamiento se indica con la palabra clave `register`:

```
int una_funcion_cualquiera( int a, int b )
{
    /* ... */
    register int contador;
    /* resto de la función */
} /* una_funcion_cualquiera */
```

Para conseguir el efecto contrario, se puede indicar que una variable local resida siempre en memoria mediante la indicación `volatile` como clase de almacenamiento. Esto sólo resulta conveniente cuando la variable puede ser modificada de forma ajena al programa.

```
int una_funcion_cualquiera( int a, int b )
{
```

```
/* ... */  
volatile float temperatura;  
/* resto de la función */  
} /* una_funcion_cualquiera */
```

En los casos anteriores, se trataba de variables automáticas. Sin embargo, a veces resulta interesante que una función pueda mantener la información contenida en alguna variable local entre distintas llamadas. Esto es, permitir que el algoritmo correspondiente “recuerde” algo de su estado pasado. Para conseguirlo, hay que indicar que la variable tiene una clase de almacenamiento **static**; es decir, que se encuentra estática o inamovible en memoria:

```
int una_funcion_cualquiera( int a, int b )  
{  
    /* ... */  
    static unsigned numero_llamadas = 0;  
    numero_llamadas = numero_llamadas + 1;  
    /* resto de la función */  
} /* una_funcion_cualquiera */
```

En el caso anterior es muy importante inicializar las variables en la declaración; de lo contrario, no se podría saber el contenido inicial, previo a cualquier llamada a la función.



Como nota final, indicar que las clases de almacenamiento se utilizan muy raramente en la programación en C. De hecho, a excepción de **static**, las demás prácticamente no tienen efecto en un compilador actual.

2.12.3. Parámetros por valor y por referencia

El paso de parámetros se refiere a la acción de transformar los parámetros formales a parámetros reales; es decir, de asignar un contenido a las variables que representan a los argumentos:

```
tipo funcion_llamada(  
    parámetro_formal_1,  
    parámetro_formal_2,  
    ...  
) ;
```

```
funcion_llamadora( ... )
{
    /* ... */
    funcion_llamada( parámetro_real_1, parámetro_real_2, ... )
    /* ... */
} /* funcion_llamadora */
```

En este sentido, hay dos posibilidades: que los argumentos reciban el resultado de la evaluación de la expresión correspondiente o que se sustituyan por la variable que se indicó en el parámetro real de la misma posición. El primer caso, se trata de un **paso de parámetros por valor**, mientras que el segundo, se trata de un **paso de variable** (cualquier cambio en el argumento es un cambio en la variable que consta como parámetro real).

El paso por valor consiste en asignar a la variable del parámetro formal correspondiente el valor resultante del parámetro real en su misma posición. El paso de variable consiste en sustituir la variable del parámetro real por la del parámetro formal correspondiente y, consecuentemente, poder emplearla dentro de la misma función con el nombre del parámetro formal.



En C, el paso de parámetros sólo se efectúa por valor; es decir, se evalúan todos los parámetros en la llamada y se asigna el resultado al parámetro formal correspondiente en la función.

Para modificar alguna variable que se desee pasar como argumento en la llamada de una función, es necesario pasar la dirección de memoria en la que se encuentra. Para esto se debe de emplear el operador de obtención de dirección (&) que da como resultado la dirección de memoria en la que se encuentra su argumento (variable, campo de tuplo o elemento de matriz, entre otros). Este es el mecanismo que se emplea para que la función `scanf` deposite en las variables que se le pasan como argumento los valores que lee.

Por otra parte, en las funciones llamadas, los parámetros formales que reciben una referencia de una variable en lugar de un valor se deben declarar de manera especial, anteponiendo a su nombre un asterisco. El asterisco, en este contexto, se puede leer como el “contenido cuya posición inicial se encuentra en la variable correspondiente”. Por tanto, en una función como la mostrada a continuación, se leería “el contenido cuya posición inicial se encuentra en el parámetro formal numerador” es de tipo entero. De igual forma se leería para el denominador:

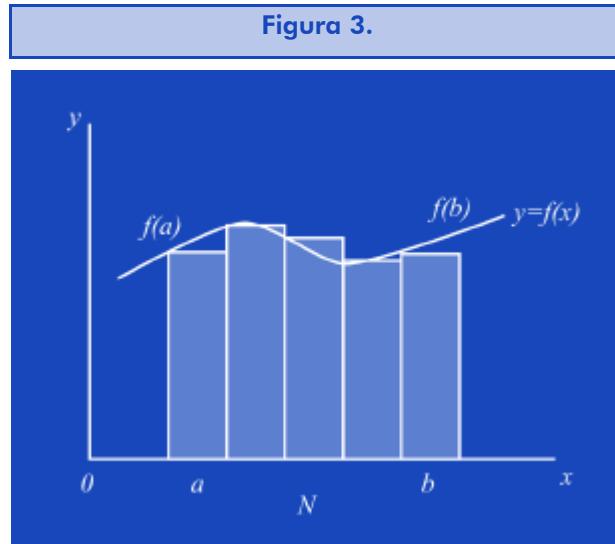
```
void simplifica( int *numerador, int *denominador )  
{  
    int mcd;  
    mcd=maximo_comun_divisor( *numerador, *denominador );  
    *numerador = *numerador / mcd;  
    *denominador = *denominador / mcd;  
} /* simplifica */  
/* ... */  
    simplifica( &a, &b );  
/* ... */
```

Aunque se insistirá en ello más adelante, hay que tener presente que el asterisco, en la parte del código, debería leerse como “el contenido de la variable que está almacenada en la posición de memoria del argumento correspondiente”. Por lo tanto, deberemos emplear `*parámetro_formal` cada vez que se desee utilizar la variable pasada por referencia.

2.12.4. Ejemplo

El programa siguiente calcula numéricamente la integral de una función en un intervalo dado según la regla de Simpson. Básicamente, el método consiste en dividir el intervalo de integración en un número determinado de segmentos de la misma longitud que constituyen la base de unos rectángulos cuya altura viene determinada por el valor de la función a integrar en el punto inicial del segmento. La suma de las áreas de estos rectángulos dará la superficie aproximada defini-

da por la función, el eje de las X y las rectas perpendiculares a él que pasan por los puntos inicial y final del segmento de integración:



```
/* Fichero: simpson.c */
```

```
#include <stdio.h>
#include <math.h>

double f( double x )
{
    return 1.0/(1.0 + x*x);
} /* f */

double integral_f( double a, double b, int n )
{
    double result;
    double x, dx;
    int i;

    result = 0.0;
    if( (a < b) && (n > 0) ) {
        x = a;
        dx = (b-a)/n;
        for( i = 0; i < n; i = i + 1 ) {
            result = result + f(x);
            x = x + dx;
        }
    }
    return result;
}
```

```
x = x + dx;
} /* for */
} /* if */
return result;
} /* integral_f */

void main( void )
{
    double a, b;
    int n;

    printf( "Integración numérica de f(x).\n" );
    printf( "Punto inicial del intervalo, a = ? " );
    scanf( "%lf", &a );
    printf( "Punto final del intervalo, b = ? " );
    scanf( "%lf", &b );
    printf( "Número de divisiones, n = ? " );
    scanf( "%d", &n );
    printf(
        "Resultado, integral(f) [%g,%g] = %g\n",
        a, b, integral_f( a, b, n )
    ); /* printf */
} /* main */
```

2.13. Macros del preprocesador de C

El preprocesador no sólo realiza sustituciones de símbolos simples como las que se han visto. También puede efectuar sustituciones con parámetros. A las definiciones de sustituciones de símbolos con parámetros se las llama “macros”:

```
#define símbolo expresión_constante
#define macro( argumentos ) expresión_const_con_argumentos
```



El uso de las macros puede ayudar a la clarificación de pequeñas partes del código mediante el uso de una sintaxis similar a la de las llamadas a las funciones.

De esta manera, determinadas operaciones simples pueden beneficiarse de un nombre significativo en lugar de emplear unas construcciones en C que pudieran dificultar la comprensión de su intencionalidad.

Ejemplo

```
#define absoluto( x ) ( x < 0 ? -x : x )
#define redondea( x ) ( (int) ( x + 0.5 ) )
#define trunca( x ) ( (int) x )
```

Hay que tener presente que el nombre de la macro y el paréntesis izquierdo no pueden ir separados y que la continuación de la línea, caso de que el comando sea demasiado largo) se hace colocando una barra invertida justo antes del carácter de salto de línea.

Por otra parte, hay que advertir que las macros hacen una sustitución de cada nombre de parámetro aparecido en la definición por la parte del código fuente que se indique como argumento. Así pues:

`absoluto(2*entero + 1)`

se sustituiría por:

`(2*entero + 1 < 0 ? -2*entero + 1 : 2*entero + 1)`

con lo que no sería correcto en el caso de que fuera negativo.

Nota

En este caso, sería posible evitar el error si en la definición se hubieran puesto paréntesis alrededor del argumento.

2.14. Resumen

La organización del código fuente es esencial para confeccionar programas legibles que resulten fáciles de mantener y de actualizar. Esto

es especialmente cierto para los programas de código abierto, es decir, para el software libre.

En esta unidad se han repasado los aspectos fundamentales que intervienen en un código organizado. En esencia, la organización correcta del código fuente de un programa depende tanto de las instrucciones como de los datos. Por este motivo, no sólo se ha tratado de cómo organizar el programa sino que además se ha visto cómo emplear estructuras de datos.

La organización correcta del programa empieza por que éste tenga un flujo de ejecución de sus instrucciones claro. Dado que el flujo de instrucciones más simple es aquél en el que se ejecutan de forma secuencial según aparecen en el código, es fundamental que las instrucciones de control de flujo tengan un único punto de entrada y un único punto de salida. En este principio se basa el método de la programación estructurada. En este método, sólo hay dos tipos de instrucciones de control de flujo: las alternativas y las iterativas.

Las instrucciones iterativas suponen otro reto en la determinación del flujo de control, ya que es necesario determinar que la condición por la que se detiene la iteración se cumple alguna vez. Por este motivo, se han repasado los esquemas algorítmicos para el tratamiento de secuencias de datos y se han visto pequeños programas que, además de servir de ejemplo de programación estructurada, son útiles para realizar operaciones de filtro de datos en tuberías (procesos encadenados).

Para organizar correctamente el código y hacer posible el tratamiento de información compleja es necesario recurrir a la estructuración de los datos. En este aspecto, hay que tener presente que el programa debe reflejar aquellas operaciones que se realizan en la información y no tanto lo que supone llevar a cabo los datos elementales que la componen. Por este motivo, no sólo se ha explicado cómo declarar y emplear datos estructurados, sean éstos de tipo homogéneo o heterogéneo, sino que también se ha detallado cómo definir nuevos tipos de datos a partir de los tipos de datos básicos y estructurados. A estos nuevos tipos de datos se los llama *tipos abstractos de datos* pues son transparentes para el lenguaje de programación.

Al hablar de los datos también se ha tratado de los ficheros de flujo de bytes. Estas estructuras de datos homogéneas se caracterizan por tener un número indefinido de elementos, por residir en memoria secundaria, es decir, en algún soporte de información externo y, finalmente, por requerir de funciones específicas para acceder a sus datos. Así pues, se han comentado las funciones estándar en C para operar con este tipo de ficheros. Fundamentalmente, los programas que los usan implementan esquemas algorítmicos de recorrido o de búsqueda en los que se incluye una inicialización específica para abrir los ficheros, una comprobación de final de fichero para la condición de iteración, operaciones de lectura y escritura para el tratamiento de la secuencia de datos y, para acabar, una finalización que consiste, entre otras cosas, en cerrar los ficheros empleados.

El último apartado se ha dedicado a la programación modular, que consiste en agrupar las secuencias de instrucciones en subprogramas que realicen una función concreta y susceptible de ser empleado más de una vez en el mismo programa o en otros. Así pues, se sustituye en el flujo de ejecución todo el subprograma por una instrucción que se ocupará de ejecutar el subprograma correspondiente. Estos subprogramas se denominan “funciones” en C y a la instrucción que se ocupa de ejecutarlas, “instrucción de llamada”. Se ha visto cómo se lleva a cabo una llamada a una función y que, en este aspecto, lo más importante es el paso de parámetros.

El paso de parámetros consiste en transmitir a una función el conjunto de datos con los que deberá realizar su tarea. Dado que la función puede necesitar devolver resultados que no se puedan almacenar en una variable simple, algunos de estos parámetros se emplean para pasar referencias a variables que también podrán contener valores de retorno. Así pues, se ha analizado también toda la problemática relacionada con el paso de parámetros por valor y por referencia.

2.15. Ejercicios de autoevaluación

- 1) Haced un programa para determinar el número de dígitos necesarios para representar a un número entero dado. El algoritmo

consiste en hacer divisiones enteras por 10 del número hasta que el resultado sea un valor inferior a 10.

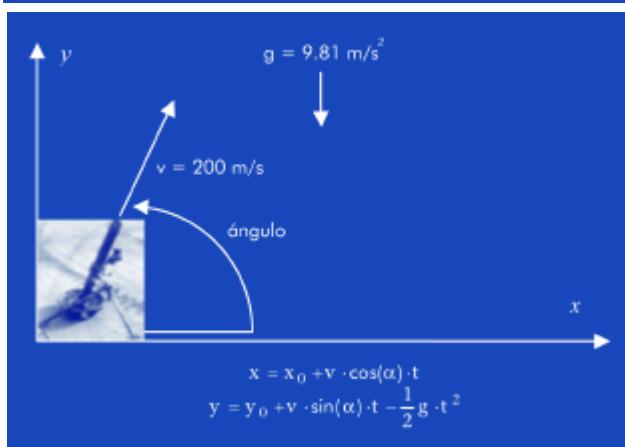
- 2) Haced un programa que determine a cada momento la posición de un proyectil lanzado desde un mortero. Se deberá mostrar la altura y la distancia a intervalos regulares de tiempo hasta que alcance el suelo. Para ello, se supondrá que el suelo es llano y que se le proporcionan, como datos de entrada, el ángulo del cañón y el intervalo de tiempo en el que se mostrarán los datos de salida. Se supone que la velocidad de salida de los proyectiles es de 200 m/s.

Nota

Para más detalles, se puede tener en cuenta que el tubo del mortero es de 1 m de longitud y que los ángulos de tiro varían entre 45 y 85 grados.

En el siguiente esquema se resumen las distintas fórmulas que son necesarias para resolver el problema:

Figura 4.



donde x_0 e y_0 es la posición inicial (puede considerarse 0 para los dos), y α es el ángulo en radianes (Π radianes = 180º).

- 3) Se desea calcular el capital final acumulado de un plan de pensiones sabiendo el capital inicial, la edad del asegurado (se supone que se jubilará a los 65) y las aportaciones y los porcentajes de interés rendidos de cada año. (Se supone que las aportaciones son de carácter anual.)

- 4) Programad un filtro que calcule la media, el máximo y el mínimo de una serie de números reales de entrada.
- 5) Implementad los filtros del último ejemplo del apartado "2.4.2. Filtros y tuberías"; es decir: calculad los importes de una secuencia de datos { código de artículo, precio, cantidad } que genere otra secuencia de datos { código de artículo, importe } y realizad, posteriormente, la suma de los importes de esta segunda secuencia de datos.
- 6) Haced un programa que calcule la desviación típica que tienen los datos de ocupación de un aparcamiento público a lo largo de las 24 horas de un día. Habrá, por tanto, 24 datos de entrada.

Estos datos se refieren al porcentaje de ocupación (número de plazas ocupadas con relación al número total de plazas) calculado al final de cada hora. También se deberá indicar las horas del día que tengan un porcentaje de ocupación inferior a la media menos dos veces la desviación típica, y las que lo tengan superior a la media más dos veces esta desviación.

Nota

La desviación típica se calcula como la raíz cuadrada de la suma de los cuadrados de las diferencias entre los datos y la media, dividida por el número de muestras.

- 7) Averiguad si la letra de un NIF dado es o no correcta. El procedimiento de su cálculo consiste en realizar el módulo 23 del número correspondiente. El resultado da una posición en una secuencia de letras (TRWAGMYFPDXBNJZSQVHLCKE). La letra situada en dicha posición será la letra del NIF.

Nota

Para poder efectuar la comparación entre letras, es conveniente convertir la que proporcione el usuario a mayúscula. Para ello se debe emplear `toupper()`, cuya declaración está en `ctype.h` y que devuelve el carácter correspondiente a la letra mayúscula de la que ha recibido como argumento. Si no es un carácter alfabético o se trata de una letra ya mayúscula, devuelve el mismo carácter.

- 8) Haced un programa que calcule el mínimo número de monedas necesario para devolver el cambio sabiendo el importe total a cobrar y la cantidad recibida como pago. La moneda de importe máximo es la de 2 euros y la más pequeña, de 1 céntimo.

Nota

Es conveniente tener un vector con los valores de las monedas ordenados por valor.

- 9) Resumid la actividad habida en un terminal de venta por artículos. El programa debe mostrar, para cada código de artículo, el número de unidades vendidas. Para ello, contará con un fichero generado por el terminal que consta de pares de números enteros: el primero indica el código del artículo y el segundo, la cantidad vendida. En caso de devolución, la cantidad aparecerá como un valor negativo. Se sabe, además, que no habrá nunca más de 100 códigos de artículos diferentes.

Nota

Es conveniente disponer de un vector de 100 tuplas para almacenar la información de su código y las unidades vendidas correspondientes. Como no se sabe cuántas tuplas serán necesarias, téngase en cuenta que se deberá disponer de una variable que indique los que se hayan almacenado en el vector (de 0 al número de códigos distintos -1).

- 10) Reprogramad el ejercicio anterior de manera que las operaciones que afecten al vector de datos se lleven a cabo en el cuerpo de funciones específicas.

Nota

Definid un tipo de dato nuevo que contenga la información de los productos. Se sugiere, por ejemplo, el que se muestra a continuación.

```
typedef struct productos_s {  
    unsigned int n; /* Número de productos. */  
    venta_t producto[MAX_PRODUCTOS];  
} productos_t;
```

Recuérdese que habrá de pasar la variable de este tipo por referencia.

- 11) Buscad una palabra en un fichero de texto. Para ello, realizad un programa que pida tanto el texto de la palabra como el nombre del fichero. El resultado deberá ser un listado de todas las líneas en que se encuentre dicha palabra.

Se supondrá que una palabra es una secuencia de caracteres alfanuméricos. Es conveniente emplear la macro `isalnum()`, que se encuentra declarada en `ctype.h`, para determinar si un carácter es o no alfanumérico.

Nota

En la solución propuesta, se emplean las funciones que se declaran a continuación.

```
#define LONG_PALABRA 81
typedef char palabra_t[LONG_PALABRA];
bool palabras_iguales( palabra_t p1, palabra_t p2 );
unsigned int lee_palabra( palabra_t p, FILE *entrada );
void primera_palabra(palabra_t palabra, char *frase );
```

2.15.1. Solucionario

```
1)
/* ----- */
/* Fichero: ndigitos.c */
/* ----- */

#include <stdio.h>

main()
{
    unsigned int numero;
    unsigned int digitos;

    printf( "El número de dígitos para representar: " );
    scanf( "%u", &numero );
    digitos = 1;
```

```
while( numero > 10 ) {  
    numero = numero / 10;  
    digitos = digitos + 1;  
} /* while */  
printf( "... es %u.\n", digitos );  
} /* main */
```

2)

```
/* ----- */  
/* Fichero: mortero.c */  
/* ----- */  
  
#include <stdio.h>  
#include <math.h>  
  
#define V_INICIAL 200.00      /* m/s */  
#define L_TUBO     1.0         /* m */  
#define G          9.81        /* m/ (s*s) */  
#define PI         3.14159265  
  
main()  
{  
    double angulo, inc_tiempo, t;  
    double v_x, v_y; /* Velocidades horizontal y vertical. */  
    double x0, x, y0, y;  
  
    printf( "Tiro con mortero.\n" );  
    printf( "Ángulo de tiro [grados sexagesimales] =? " );  
    scanf( "%lf", &angulo );  
    angulo = angulo * PI / 180.0;  
    printf( "Ciclo de muestra [segundos] =? " );  
    scanf( "%lf", &inc_tiempo );  
    x0 = L_TUBO * cos( angulo );  
    y0 = L_TUBO * sin( angulo );  
    t = 0.0;  
    v_x = V_INICIAL * cos( angulo );  
    v_y = V_INICIAL * sin( angulo );  
    do {  
        x = x0 + v_x * t;  
        y = y0 + v_y * t - 0.5 * G * t * t;  
        printf( "%6.2lf s: (%6.2lf, %6.2lf)\n", t, x, y );  
    } while( t < inc_tiempo );
```

```

        t = t + inc_tiempo;
    } while ( y > 0.0 );
} /* main */

```

3)

```

/* ----- */  

/* Fichero: pensiones.c */  

/* ----- */  

#include <stdio.h>

```

```
#define EDAD_JUBILACION 65
```

```
main()
```

{

```
    unsigned int edad;  

    float capital, interes, aportacion;
```

```

    printf( "Plan de pensiones.\n" );  

    printf( "Edad =? " );  

    scanf( "%u", &edad );  

    printf( "Aportación inicial =? " );  

    scanf( "%f", &capital );  

    while( edad < EDAD_JUBILACION ) {  

        printf( "Interés rendido [en %%] =? " );  

        scanf( "%f", &interes );  

        capital = capital*( 1.0 + interes/100.0 );  

        printf( "Nueva aportación =? " );  

        scanf( "%f", &aportacion );  

        capital = capital + aportacion;  

        edad = edad + 1;  

        printf( "Capital acumulado a %u años: %.2f\n",
               edad, capital
        ); /* printf */  

    } /* while */  

} /* main */
```

4)

```

/* ----- */  

/* Fichero: estadistica.c */  

/* ----- */  

#include <stdio.h>

```

```
main()
{
    double      suma, minimo, maximo;
    double      numero;
    unsigned int cantidad, leido_ok;

    printf( "Mínimo, media y máximo.\n" );
    leido_ok = scanf( "%lf", &numero );
    if( leido_ok == 1 ) {
        cantidad = 1;
        suma = numero;
        minimo = numero;
        maximo = numero;
        leido_ok = scanf( "%lf", &numero );
        while( leido_ok == 1 ) {
            suma = suma + numero;
            if( numero > maximo ) maximo = numero;
            if( numero < minimo ) minimo = numero;
            cantidad = cantidad + 1;
            leido_ok = scanf( "%lf", &numero );
        } /* while */
        printf( "Mínimo = %g\n", minimo );
        printf( "Media = %g\n", suma / (double) cantidad );
        printf( "Máximo = %g\n", maximo );
    } else {
        printf( "Entrada vacía.\n" );
    } /* if */
} /* main */
```

5)

```
/* -----
 * Fichero: calc_importes.c
 * -----
 */
#include <stdio.h>
main()
{
    unsigned int leidos_ok, codigo;
    int         cantidad;
    float       precio, importe;

    leidos_ok = scanf( "%u%f%i",

```

```

        &codigo, &precio, &cantidad
    ); /* scanf */
    while( leidos_ok == 3 ) {
        importe = (float) cantidad * precio;
        printf( "%u %.2f\n", codigo, importe );
        leidos_ok = scanf( "%u%f%i",
            &codigo, &precio, &cantidad
        ); /* scanf */
    } /* while */
} /* main */

/* -----
/* Fichero: totaliza.c
/* -----
*/
```

```

#include <stdio.h>
main()
{
    unsigned int leidos_ok, codigo;
    int cantidad;
    float importe;
    double total = 0.0;

    leidos_ok = scanf( "%u%f", &codigo, &importe );
    while( leidos_ok == 2 ) {
        total = total + importe;
        leidos_ok = scanf( "%u%f", &codigo, &importe );
    } /* while */
    printf( "%.2lf\n", total );
} /* main */

6)
/* -----
/* Fichero: ocupa_pk.c
/* -----
```

```

*/
```

```

#include <stdio.h>
#include <math.h>

main()
{
    unsigned int hora;
```

```
float      porcentaje;
float      ratio_acum[24];
double     media, desv, dif;

printf( "Estadística de ocupación diaria:\n" );
/* Lectura de ratios de ocupación por horas:          */
for( hora = 0; hora < 24; hora = hora + 1 ) {
    printf(
        "Porcentaje de ocupación a las %02u horas =? ",
        hora
    ); /* printf */
    scanf( "%f", &porcentaje );
    ratio_acum[hora] = porcentaje;
} /* for */

/* Cálculo de la media:                                */
media = 0.0;
for( hora = 0; hora < 24; hora = hora + 1 ) {
    media = media + ratio_acum[hora];
} /* for */
media = media / 24.0;

/* Cálculo de la desviación típica:                  */
desv = 0.0;
for( hora = 0; hora < 24; hora = hora + 1 ) {
    dif = ratio_acum[ hora ] - media;
    desv = desv + dif * dif;
} /* for */
desv = sqrt( desv ) / 24.0;

/* Impresión de los resultados:                      */
printf( "Media de ocupación en el día: %.2lf\n", media );
printf( "Desviación típica: %.2lf\n", desv );
printf( "Horas con porcentaje muy bajo: ", desv );
for( hora = 0; hora < 24; hora = hora + 1 ) {
    dif = media - 2*desv;
    if( ratio_acum[ hora ] < dif ) printf( "%u ", hora );
} /* for */

printf( "\nHoras con porcentaje muy alto: ", desv );
for( hora = 0; hora < 24; hora = hora + 1 ) {
    dif = media + 2*desv;
    if( ratio_acum[ hora ] > dif ) printf( "%u ", hora );
} /* for */

printf( "\nFin.\n" );
```

```

} /* main */

7)
/* ----- */ /*----- */
/* Fichero: valida_nif.c */ /*----- */
/* ----- */ /*----- */

#include <stdio.h>
#include <ctype.h>
main()
{
    char      NIF[ BUFSIZ ];
    unsigned int DNI;
    char      letra;
    char      codigo[] = "TRWAGMYFPDXBNJZSQVHLCKE" ;

    printf( "Dime el NIF (DNI-letra): " );
    gets( NIF );
    sscanf( NIF, "%u", &DNI );
    DNI = DNI % 23;
    letra = NIF[ strlen(NIF)-1 ];
    letra = toupper( letra );
    if( letra == codigo[ DNI ] ) {
        printf( "NIF válido.\n" );
    } else {
        printf( "¡Letra %c no válida!\n", letra );
    } /* if */
} /* main */

8)
/* ----- */ /*----- */
/* Fichero: cambio.c */ /*----- */
/* ----- */ /*----- */

#include <stdio.h>

#define NUM_MONEDAS 8

main()
{
    double      importe, pagado;
    int         cambio_cents;

```

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

```
int          i, nmonedas;
int          cents[NUM_MONEDAS] = { 1, 2, 5, 10,
                                    20, 50, 100, 200 };

printf( "Importe : " );
scanf( "%lf", &importe );
printf( "Pagado : " );
scanf( "%lf", &pagado );
cambio_cents = (int) 100.00 * ( pagado - importe );
if( cambio_cents < 0 ) {
    printf( "PAGO INSUFICIENTE\n");
} else {
    printf( "A devolver : %.2f\n",
           (float) cambio_cents / 100.0
    ); /* printf */
    i = NUM_MONEDAS - 1;
    while( cambio_cents > 0 ) {
        nmonedas = cambio_cents / cents[i] ;
        if( nmonedas > 0 ) {
            cambio_cents = cambio_cents - cents[i]*nmonedas;
            printf( "%u monedas de %.2f euros.\n",
                    nmonedas,
                    (float) cents[i] / 100.0
            ); /* printf */
        } /* if */
        i = i - 1;
    } /* while */
} /* if */
} /* main */

9)
/* -----
/* Fichero: resumen_tpvc.c
/* ----- */
```



```
#include <stdio.h>

typedef struct venta_s {
    unsigned int codigo;
    int cantidad;
} venta_t;
```

```

#define MAX_PRODUCTOS 100

main()
{
    FILE *entrada;
    char nombre[BUFSIZ];
    unsigned int leidos_ok;
    int num_prod, i;
    venta_t venta, producto[MAX_PRODUCTOS];

    printf( "Resumen del día en TPV.\n" );
    printf( "Fichero: " );
    gets( nombre );
    entrada = fopen( nombre, "rt" );
    if( entrada != NULL ) {
        num_prod = 0;
        leidos_ok = fscanf( entrada, "%u%i",
            &(venta.codigo), &(venta.cantidad)
        ); /* fscanf */
        while( leidos_ok == 2 ) {

            /* Búsqueda del código en la tabla: */ *
            i = 0;
            while( ( i < num_prod ) &&
                ( producto[i].codigo != venta.codigo )
            ) {
                i = i + 1;
            } /* while */
            if( i < num_prod ) { /* Código encontrado: */ *
                producto[i].cantidad = producto[i].cantidad +
                    venta.cantidad;
            } else { /* Código no encontrado ⇒ nuevo producto: */ *
                producto[num_prod].codigo = venta.codigo;
                producto[num_prod].cantidad = venta.cantidad;
                num_prod = num_prod + 1;
            } /* if */
            /* Lectura de siguiente venta: */ *
            leidos_ok = fscanf( entrada, "%u%i",
                &(venta.codigo), &(venta.cantidad)
            ); /* fscanf */
        } /* while */
    }
}

```

```
printf( "Resumen:\n" );
for( i=0; i<num_prod; i = i + 1 ) {
    printf( "%u : %i\n",
        producto[i].codigo, producto[i].cantidad
    ); /* printf */
} /* for */
} else {
    printf( "¡No puedo abrir %s!\n", nombre );
} /* if */
} /* main */

10)
/* ... */

int busca( unsigned int codigo, productos_t *pref )
{
    int i;

    i = 0;
    while( (i < (*pref).n) &&
           ( (*pref).producto[i].codigo != codigo ) )
    {
        i = i + 1;
    } /* while */
    if( i == (*pref).n ) i = -1;
    return i;
} /* busca */

void anyade( venta_t venta, productos_t *pref )
{
    unsigned int n;

    n = (*pref).n;
    if( n < MAX_PRODUCTOS ) {
        (*pref).producto[n].codigo = venta.codigo;
        (*pref).producto[n].cantidad = venta.cantidad;
        (*pref).n = n + 1;
    } /* if */
} /* anyade */

void actualiza( venta_t venta, unsigned int posicion,
```

```
{  
    (*pref).producto[posicion].cantidad =  
        (*pref).producto[posicion].cantidad + venta.cantidad;  
} /* actualiza */  
  
void muestra( productos_t *pref )  
{  
    int i;  
  
    for( i=0; i<(*pref).n; i = i + 1 ) {  
        printf( "%u : %i\n",  
            (*pref).producto[i].codigo,  
            (*pref).producto[i].cantidad  
        ); /* printf */  
    } /* for */  
} /* muestra */  
  
void main( void )  
{  
    FILE      *entrada;  
    char       nombre[BUFSIZ];  
    unsigned int leidos_ok;  
    int       i;  
    venta_t   venta;  
    productos_t productos;  
  
    printf( "Resumen del dia en TPV.\n" );  
    printf( "Fichero: " );  
    gets( nombre );  
    entrada = fopen( nombre, "rt" );  
    if( entrada != NULL ) {  
        productos.n = 0;  
        leidos_ok = fscanf( entrada, "%u%i",  
            &(venta.codigo), &(venta.cantidad)  
        ); /* scanf */  
        while( leidos_ok == 2 ) {  
            i = busca( venta.codigo, &productos );  
            if( i < 0 ) anyade( venta, &productos );  
            else         actualiza( venta, i, &productos );  
            leidos_ok = fscanf( entrada, "%u%i",  
                &(venta.codigo), &(venta.cantidad)  
            ); /* scanf */  
        } /* while */  
    } /* if */  
}
```

```
    ); /* scanf */
} /* while */
printf( "Resumen:\n" );
muestra( &productos );
} else {
    printf( "No puedo abrir %s!\n", nombre );
} /* if */
} /* main */
```

11)

```
/* -----
/* Fichero: busca.c
/* ----- */

#include <stdio.h>
#include <ctype.h>

typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;

#define LONG_PALABRA 81

typedef char palabra_t[LONG_PALABRA];

bool palabras_iguales( palabra_t p1, palabra_t p2 )
{
    int i = 0;

    while( (p1[i]!='\0') && (p2[i]!='\0') && (p1[i]==p2[i])) {
        i = i + 1;
    } /* while */
    return p1[i]==p2[i];
} /* palabras_iguales */

unsigned int lee_palabra( palabra_t p, FILE *entrada )
{
    unsigned int i, nlin;
    bool termino;
    char caracter;
```

```
i = 0;
nlin = 0;
termino = FALSE;
caracter = fgetc( entrada );
while( !feof( entrada ) && !termino ) {
    if( caracter == '\n' ) nlin = nlin + 1;
    if( isalnum( caracter ) ) {
        p[i] = caracter;
        i = i + 1;
        caracter = fgetc( entrada );
    } else {
        if( i > 0 ) {
            termino = TRUE;
        } else {
            caracter = fgetc( entrada );
        } /* if */
    } /* if */
} /* while */
p[i] = '\0';
return nlin;
} /* lee_palabra */

void primera_palabra( palabra_t palabra, char *frase )

{
    int i, j;

    i = 0;
    while( frase[i]!='\0' && isspace( frase[i] ) ) {
        i = i + 1;
    } /* while */
    j = 0;
    while( frase[i]!='\0' && !isspace( frase[i] ) ) {
        palabra[j] = frase[i];
        i = i + 1;
        j = j + 1;
    } /* while */
    palabra[j] = '\0';
} /* primera_palabra */
```

```
{  
FILE *entrada;  
char nombre[BUFSIZ];  
palabra_t palabra, palabra2;  
unsigned int numlin;  
  
printf( "Busca palabras.\n" );  
printf( "Fichero: " );  
gets( nombre );  
entrada = fopen( nombre, "rt" );  
if( entrada != NULL ) {  
    printf( "Palabra: " );  
    gets( nombre );  
    primera_palabra( palabra, nombre );  
    printf( "Buscando %s en fichero...\n", palabra );  
    numlin = 1;  
    while( !feof( entrada ) ) {  
        numlin = numlin + lee_palabra( palabra2, entrada );  
        if( palabras_iguales( palabra, palabra2 ) ) {  
            printf( "... linea %lu\n", numlin );  
        } /* if */  
    } /* while */  
} else {  
    printf( ";No puedo abrir %s!\n", nombre );  
} /* if */  
} /* main */
```

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX • Editorial Guías México

3. Programación avanzada en C. Desarrollo eficiente de aplicaciones

3.1. Introducción

La programación de una aplicación informática suele dar como resultado un código fuente de tamaño considerable. Aun aplicando técnicas de programación modular y apoyándose en funciones estándar de biblioteca, resulta complejo organizar eficazmente el código. Mucho más si se tiene en cuenta que, habitualmente, se trata de un código desarrollado por un equipo de programadores.

Por otra parte, hay que tener presente que mucha de la información con que se trabaja no tiene un tamaño predefinido ni, la mayoría de las veces, se presenta en la forma más adecuada para ser procesada. Esto comporta tener que reestructurar los datos de manera que los algoritmos que los tratan puedan ser más eficientes.

Como último elemento a considerar, pero no por ello menos importante, se debe tener en cuenta que la mayoría de aplicaciones están constituidas por más de un programa. Por lo tanto, resulta conveniente organizarlos aprovechando las facilidades que para ello nos ofrece tanto el conjunto de herramientas de desarrollo de software como el sistema operativo en el que se ejecutarán.

En esta unidad se tratará de diversos aspectos que alivian los problemas antes mencionados. Así pues, desde el punto de vista de un programa en el contexto de una aplicación que lo contenga (o del que sea el único), es importante la adaptación al tamaño real de los datos a procesar y su disposición en estructuras dinámicas, la organización del código para que refleje el algoritmo que implementa y, por último, contar con el soporte del sistema operativo para la coordinación con otros programas dentro y fuera de la misma aplicación y, también, para la interacción con el usuario.

Nota

Sobre este tema se vio un ejemplo al hablar de las tuberías en la unidad anterior.

La representación de información en estructuras dinámicas de datos permite ajustar las necesidades de memoria del programa al mínimo requerido para la resolución del problema y, por otra parte, representar internamente la información de manera que su procesamiento sea más eficiente. Una estructura dinámica de datos no es más que una colección de datos cuya relación no está establecida *a priori* y que puede modificarse durante la ejecución del programa. Esto, por ejemplo, no es factible mediante un vector, pues los datos que contienen están relacionados por su posición dentro de él y, además, su tamaño debe de estar predefinido en el programa.

En el primer apartado se tratará de las variables dinámicas y de su empleo como contenedores de datos que responden a las exigencias de adaptabilidad a la información a representar y acomodamiento respecto del algoritmo que debe tratar con ellas.

El código fuente de una aplicación, sea ésta un conjunto de programas o uno único, debe organizarse de manera que se mantengan las características de un buen código (inteligible, de fácil mantenimiento y coste de ejecución óptimo). Para ello, no sólo hay que emplear una programación estructurada y modular, sino que hay que distribuirlo en diversos ficheros de manera que sea más manejable y, por otra parte, se mantenga su legibilidad.

En el apartado dedicado al diseño descendente de programas se tratará, precisamente, de los aspectos que afectan a la programación más allá de la programación modular. Se tratará de aspectos relacionados con la división del programa en términos algorítmicos y de la agrupación de conjuntos de funciones fuertemente relacionadas. Dado que el código fuente se reparte en diversos ficheros, se tratará también de aquellos aspectos relacionados con su compilación y, en especial, de la herramienta *make*.

Dada la complejidad del código fuente de cualquier aplicación, resulta conveniente emplear todas aquellas funciones estándar que aporta el sistema operativo. Con ello se consigue, además de reducir su complejidad al dejar determinadas tareas como simples instrucciones de llamada, que resulte más independiente de máquina. Así pues, en el último capítulo se trata de la relación entre los programas

y los sistemas operativos, de manera que sea posible comunicar los unos con los otros y, además, los programas entre sí.

Para finalizar, se tratará, aunque brevemente, de cómo distribuir la ejecución de un programa en diversos flujos (o hilos) de ejecución. Esto es, de cómo realizar una programación concurrente de manera que varias tareas puedan resolverse en un mismo intervalo de tiempo.

Esta unidad pretende mostrar aquellos aspectos de la programación más involucrados con el nivel más alto de abstracción de los algoritmos. Así pues, al finalizar su estudio, el lector debería alcanzar los objetivos siguientes:

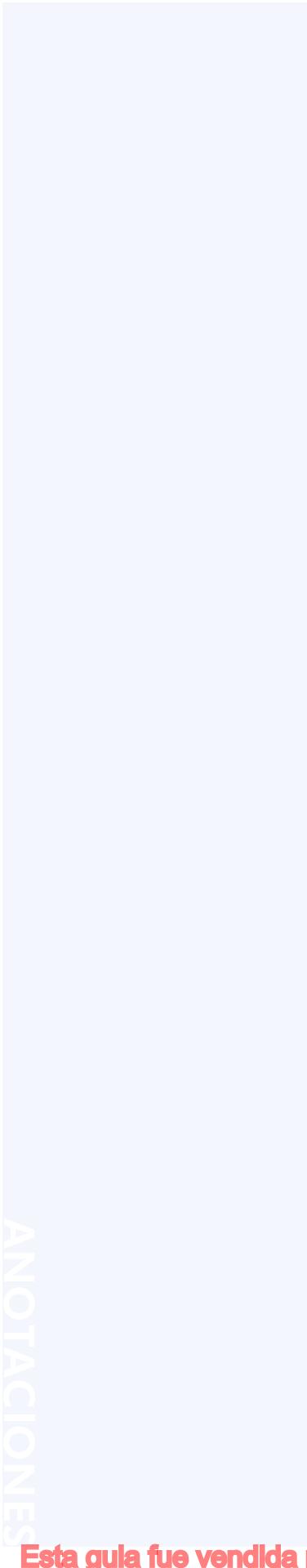
- 1) Emplear adecuadamente variables dinámicas en un programa.
- 2) Conocer las estructuras dinámicas de datos y, en especial, las listas y sus aplicaciones.
- 3) Entender el principio del diseño descendente de programas.
- 4) Ser capaz de desarrollar una biblioteca de funciones.
- 5) Tener conocimientos básicos para la relación del programa con el sistema operativo.
- 6) Asimilar el concepto de hilo de ejecución y los rudimentos de la programación concurrente.

3.2. Las variables dinámicas

La información procesada por un programa está formada por un conjunto de datos que, muy frecuentemente, no tiene ni un tamaño fijo, no se conoce su tamaño máximo, los datos no se relacionan entre ellos de la misma forma, etc.

Ejemplo

Un programa que realice análisis sintácticos (que, por otra parte, podría formar parte de una aplicación de tratamiento de textos) deberá procesar frases de tama-



ños distintos en número y categoría de palabras. Además, las palabras pueden estar relacionadas de muy diferentes maneras. Por ejemplo, los adverbios lo están con los verbos y ambos se diferencian de las que forman el sintagma nominal, entre otras.

Las relaciones existentes entre los elementos que forman una información determinada se pueden representar mediante datos adicionales que las reflejen y nos permitan, una vez calculados, realizar un algoritmo más eficiente. Así pues, en el ejemplo anterior resultaría mucho mejor efectuar los análisis sintácticos necesarios a partir del árbol sintáctico que no de la simple sucesión de palabras de la frase a tratar.

El tamaño de la información, es decir, el número de datos que la forman, afecta significativamente al rendimiento de un programa. Más aún, el uso de variables estáticas para su almacenamiento implica o bien conocer *a priori* su tamaño máximo, o bien limitar la capacidad de tratamiento a sólo una porción de la información. Además, aunque sea posible determinar el tamaño máximo, se puede producir un despilfarro de memoria innecesario cuando la información a tratar ocupe mucho menos.

Las **variables estáticas** son aquellas que se declaran en el programa de manera que disponen de un espacio reservado de memoria durante toda la ejecución del programa. En C, sólo son realmente estáticas las variables globales.

Las **variables locales**, en cambio, son automáticas porque se les reserva espacio sólo durante la ejecución de una parte del programa y luego son destruidas de forma automática. Aun así, son variables de carácter estático respecto del ámbito en el que se emplean, pues tiene un espacio de memoria reservado y limitado.

Las **variables dinámicas**, sin embargo, pueden crearse y destruirse durante la ejecución de un programa y tienen un carácter global; es decir, son "visibles" desde cualquier punto del programa. Dado que es posible crear un número indeterminado de estas variables, permiten ajustarse al tamaño requerido para representar la información.

de un problema particular sin desperdiciar espacio de memoria alguno.

Para poder crear las variables dinámicas durante la ejecución del programa, es necesario contar con operaciones que lo permitan. Por otra parte, las variables dinámicas carecen de nombre y, por tanto, su única identificación se realiza mediante la dirección de la primera posición de memoria en la que residen.

Por ello, es necesario contar con datos que puedan contener referencias a variables dinámicas de manera que permitan utilizarlas. Como sus referencias son direcciones de memoria, el tipo de estos datos será, precisamente, el de dirección de memoria o *apuntador*, pues su valor es el indicador de dónde se encuentra la variable referenciada.

En los siguientes apartados se tratará, en definitiva, de todo aquello que atañe a las variables dinámicas y a las estructuras de datos que con ellas se pueden construir.

3.3. Los apuntadores

Los apuntadores son variables que contienen direcciones de memoria de otras variables; es decir, referencias a otras variables. Evidentemente, el tipo de datos es el de posiciones de memoria y, como tal, es un tipo compatible con enteros. Aun así, tiene sus particularidades, que veremos más adelante.

La declaración de un apuntador se hace declarando el tipo de datos de las variables de las cuales va a contener direcciones. Así pues, se emplea el operador de indirección (un asterisco) o, lo que es lo mismo, el que se puede leer como “contenido de la dirección”. En los ejemplos siguientes se declaran distintos tipos de apuntadores:

```
int      *ref_entero;
char     *cadena;
otro_t   *apuntador;
nodo_t   *ap_nodo;
```

- En `ref_entero` el contenido de la dirección de memoria almacenada es un dato de tipo entero.
- En `cadena` el contenido de la dirección de memoria almacenada es un carácter.
- En `apuntador` el contenido de la dirección de memoria almacenada es de tipo `otro_t`.
- En `ap_nodo`, el contenido de la dirección de memoria almacenada es de tipo `nodo_t`.



La referencia a las variables sin el operador de indirección es, simplemente, la dirección de memoria que contienen.

El tipo de apuntador viene determinado por el tipo de dato del que tiene la dirección. Por este motivo, por ejemplo, se dirá que `ref_entero` es un apuntador de tipo entero, o bien, que se trata de un apuntador “a” un tipo de datos entero. También es posible declarar apuntadores de apuntadores, etc.

En el ejemplo siguiente se puede observar que, para hacer referencia al valor apuntado por la dirección contenida en un apuntador, hay que emplear el operador de “contenido de la dirección de”:

```
int a, b; /* dos variables de tipo entero. */
int *ptr; /* un apuntador a entero. */
int **pptr; /* un apuntador a un apuntador de entero. */
/* ... */
a = b = 5;
ptr = &a;
*ptr = 20; /* a == 20 */
ptr = &b;
pptr = &ptr;
**pptr = 40; /* b == 40 */
/* ... */
```

En la figura siguiente se puede apreciar cómo el programa anterior va modificando las variables a medida que se va ejecutando. Cada columna vertical representa las modificaciones llevadas a cabo en el entorno por una instrucción. Inicialmente (la columna de más a la izquierda), se desconoce el contenido de las variables:

Figura 5.

:						
a	? 5			20		
b	? 5					40
ptr	? &a			&b		
pptr	?				&ptr	
:						

En el caso particular de las tuplas, cabe recordar que el acceso también se hace mediante el operador de indirección aplicado a los apuntadores que contienen sus direcciones iniciales. El acceso a sus campos no cambia:

```
struct alumno_s {
    cadena_t nombre;
    unsigned short dni, nota;
} alumno;
struct alumno_s *ref_alumno;
/* ... */
alumno.nota = *ref_alumno.nota;
/* ... */
```

Para que quede claro el acceso a un campo de una tupla cuya dirección está en una variable, es preferible utilizar lo siguiente:

```
/* ... */
alumno.nota = (*ref_alumno).nota;
/* ... */
```

Si se quiere enfatizar la idea del apuntador, es posible emplear el operador de indirección para tuplos que se asemeja a una flecha que apunta a la tupla correspondiente:

```
/* ... */
alumno.nota = ref_alumno→nota;
/* ... */
```

En los ejemplos anteriores, todas las variables eran de carácter estático o automático, pero su uso primordial es como referencia de las variables dinámicas.

3.3.1. Relación entre apuntadores y vectores

Los vectores, en C, son entelequias del programador, pues se emplea un operador (los corchetes) para calcular la dirección inicial de un elemento dentro de un vector. Para ello, hay que tener presente que los nombres con los que se declaran son, de hecho, apuntadores a las primeras posiciones de los primeros elementos de cada vector. Así pues, las declaraciones siguientes son prácticamente equivalentes:

```
/* ... */
int vector_real[DIMENSION];
int *vector_virtual;
/* ... */
```

En la primera, el vector tiene una dimensión determinada, mientras que en la segunda, el `vector_virtual` es un apuntador a un entero; es decir, una variable que contiene la dirección de datos de tipo entero. Aun así, es posible emplear el identificador de la primera como un apuntador a entero. De hecho, contiene la dirección del primer entero del vector:

```
vector_real == &(vector_real[0])
```



Es muy importante no modificar el contenido del identificador, ipues se podría perder la referencia a todo el vector!

Por otra parte, los apuntadores se manejan con una aritmética especial: se permite la suma y la resta de apuntadores de un mismo tipo

y enteros. En el último caso, las sumas y restas con enteros son, en realidad, sumas y restas con los múltiplos de los enteros, que se multiplican por el tamaño en bytes de aquello a lo que apuntan.

Sumar o restar contenidos de apuntadores resulta algo poco habitual. Lo más frecuente es incrementarlos o decrementarlos para que apunten a algún elemento posterior o anterior, respectivamente. En el ejemplo siguiente se puede intuir el porqué de esta aritmética especial. Con ella, se libera al programador de tener que pensar cuántos bytes ocupa cada tipo de dato:

```
/* ... */  
int vector[DIMENSION], *ref_entero;  
/* ... */  
ref_entero = vector;  
ref_entero = ref_entero + 3;  
*ref_entero = 15; /* Es equivalente a vector[3] = 15 */  
/* ... */
```

En todo caso, existe el operador `sizeof` ("tamaño de") que devuelve como resultado el tamaño en bytes del tipo de datos de su argumento. Así pues, el caso siguiente es, en realidad, un incremento de `ref_entero` de tal manera que se añade `3*sizeof(int)` a su contenido inicial.:

```
/* ... */  
ref_entero = ref_entero + 3;  
/* ... */
```

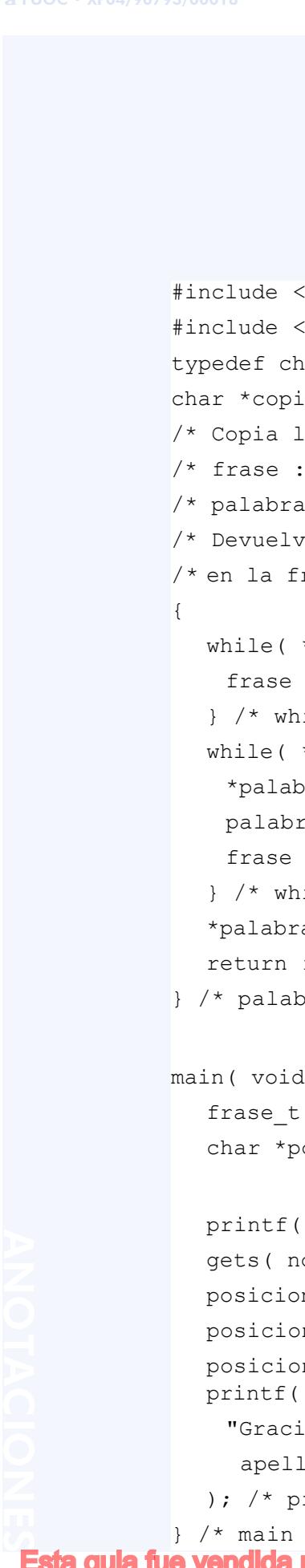
Por tanto, en el caso de los vectores, se cumple que:

```
vector[i] == *(vector+i)
```

Es decir, que el elemento que se encuentra en la posición i -ésima es el que se encuentra en la posición de memoria que resulta de sumar $i*sizeof(*vector)$ a su dirección inicial, indicada en `vector`.

Nota

El operador `sizeof` se aplica al elemento apuntado por `vector`, puesto que, de aplicarse a `vector`, se obtendría el tamaño en bytes que ocupa un apuntador.



En el siguiente ejemplo, se podrá observar con más detalle la relación entre apunadores y vectores. El programa listado a continuación toma como entrada un nombre completo y lo separa en nombre y apellidos:

```
#include <stdio.h>
#include <ctype.h>
typedef char frase_t[256];
char *copia_palabra( char *frase, char *palabra )
/* Copia la primera palabra de la frase en palabra. */
/* frase : apuntador a un vector de caracteres. */
/* palabra : apuntador a un vector de caracteres. */
/* Devuelve la dirección del último carácter leído
/* en la frase. */
{
    while( *frase!='\0' && isspace( *frase ) ) {
        frase = frase + 1;
    } /* while */
    while( *frase]!='\0' && !isspace( *frase ) ) {
        *palabra = *frase;
        palabra = palabra + 1;
        frase = frase + 1;
    } /* while */
    *palabra = '\0';
    return frase;
} /* palabra */

main( void ) {
    frase_t nombre_completo, nombre, apellido1, apellido2;
    char *posicion;

    printf( "Nombre y apellidos? " );
    gets( nombre_completo );
    posicion = copia_palabra( nombre_completo, nombre );
    posicion = copia_palabra( posicion, apellido1 );
    posicion = copia_palabra( posicion, apellido2 );
    printf(
        "Gracias por su amabilidad, Sr/a. %s.\n",
        apellido1
    ); /* printf */
} /* main */
```

Nota

Más adelante, cuando se trate de las cadenas de caracteres, se insistirá de nuevo en la relación entre apuntadores y vectores.

3.3.2. Referencias de funciones

Las referencias de funciones son, en realidad, la dirección a la primera instrucción ejecutable de las mismas. Por lo tanto, se pueden almacenar en apuntadores a funciones.

La declaración de un apuntador a una función se realiza de manera similar a la declaración de los apuntadores a variables: basta con incluir en su nombre un asterisco. Así pues, un apuntador a una función que devuelve el número real, producto de realizar alguna operación con el argumento, se declararía de la manera siguiente:

```
float (*ref_funcion) ( double x );
```

Nota

El paréntesis que encierra al nombre del apuntador y al asterisco que le precede es necesario para que no se confunda con la declaración de una función cuyo valor devuelto sea un apuntador a un número real.

Sirva como ejemplo un programa para la integración numérica de un determinado conjunto de funciones. Este programa es parecido al que ya se vio en la unidad anterior con la modificación de que la función de integración numérica tiene como nuevo argumento la referencia de la función de la que hay que calcular la integral:

```
/* Programa: integrales.c */
#include <stdio.h>
#include <math.h>
double f0( double x ) { return x/2.0; }
double f1( double x ) { return 1+2*log(x); }
double f2( double x ) { return 1.0/(1.0 + x*x); }
```

```

double integral_f( double a, double b, int n,
                   double (*fref)( double x ) )
{
    double result;
    double x, dx;
    int i;
    result = 0.0;
    if( (a < b) && (n > 0) ) {
        x = a;
        dx = (b-a)/n;
        for( i = 0; i < n; i = i + 1 ) {
            result = result + (*fref)(x);
            x = x + dx;
        } /* for */
    } /* if */
    return result;
} /* integral_f */

void main( void )
{
    double a, b;
    int n, fnum;
    double (*fref)( double x );
    printf( "Integración numérica de f(x).\n" );
    printf( "Punto inicial del intervalo, a = ? " );
    scanf( "%lf", &a );
    printf( "Punto final del intervalo, b = ? " );
    scanf( "%lf", &b );
    printf( "Número de divisiones, n = ? " );
    scanf( "%d", &n );
    printf( "Número de función, fnum = ?" );
    scanf( "%d", &fnum );
    switch( fnum ) {
        case 1 : fref = f1; break;
        case 2 : fref = f2; break;
        default: fref = f0;
    } /* switch */
    printf(
        "Resultado, integral(f) [%g,%g] = %g\n",
        a, b, integral_f( a, b, n, fref )
    ); /* printf */
} /* main */

```

Como se puede observar, el programa principal podría perfectamente sustituir las asignaciones de referencias de funciones por llamadas a las mismas. Con esto, el programa sería mucho más claro. Aun así, como se verá más adelante, esto permite que la función que realiza la integración numérica pueda residir en alguna biblioteca y ser empleada por cualquier programa.

3.4. Creación y destrucción de variables dinámicas

Tal como se dijo al principio de esta unidad, las variables dinámicas son aquellas que se crean y destruyen durante la ejecución del programa que las utiliza. Por el contrario, las demás son variables estáticas o automáticas, que no necesitan de acciones especiales por parte del programa para ser empleadas.

Antes de poder emplear una variable dinámica, hay que reservarle espacio mediante la función estándar (declarada en `stdlib.h`) que localiza y reserva en la memoria principal un espacio de tamaño `número_bytes` para que pueda contener los distintos datos de una variable:

```
void * malloc( size_t número_bytes );
```

Como la función desconoce el tipo de datos de la futura variable dinámica, devuelve un apuntador a tipo vacío, que hay que coercer al tipo correcto de datos:

```
/* ... */  
char *apuntador;  
/* ... */  
apuntador = (char *)malloc( 31 );  
/* ... */
```

Si no puede reservar espacio, devuelve `NULL`.

En general, resulta difícil conocer exactamente cuántos bytes ocupa cada tipo de datos y, por otra parte, su tamaño puede depender del compilador y de la máquina que se empleen. Por este motivo, es

Nota

El tipo de datos `size_t` es, simplemente, un tipo de entero sin signo al que se lo ha denominado así porque representa tamaños.

conveniente emplear siempre el operador `sizeof`. Así, el ejemplo anterior tendría que haber sido escrito de la forma siguiente:

```
/* ... */
apuntador = (char *)malloc( 31 * sizeof(char) );
/* ... */
```



`sizeof` devuelve el número de bytes necesario para contener el tipo de datos de la variable o del tipo de datos que tiene como argumento, excepto en el caso de las matrices, en que devuelve el mismo valor que para un apuntador.

A veces, es necesario ajustar el tamaño reservado para una variable dinámica (sobre todo, en el caso de las de tipo vector), bien porque falta espacio para nuevos datos, bien porque se desaprovecha gran parte del área de memoria. Para tal fin, es posible emplear la función de “relocalización” de una variable:

```
void * realloc( void *apuntador, size_t nuevo_tamaño );
```

El comportamiento de la función anterior es similar a la de `malloc`: devuelve `NULL` si no ha podido encontrar un nuevo emplazamiento para la variable con el tamaño indicado.

Cuando una variable dinámica ya no es necesaria, se tiene que destruir; es decir, liberar el espacio que ocupa para que otras variables dinámicas lo puedan emplear. Para ello hay que emplear la función `free`:

```
/* ... */
free( apuntador );
/* ... */
```

Como sea que esta función sólo libera el espacio ocupado pero no modifica en absoluto el contenido del apuntador, resulta que éste aún tiene la referencia a la variable dinámica (su dirección) y, por

tanto, existe la posibilidad de acceder a una variable inexistente. Para evitarlo, es muy conveniente asignar el apuntador a `NULL`:

```
/* ... */  
free( apuntador );  
apuntador = NULL;  
/* ... */
```

De esta manera, cualquier referencia errónea a la variable dinámica destruida causará error; que podrá ser fácilmente corregido.

3.5. Tipos de datos dinámicos

Aquellos datos cuya estructura puede variar a lo largo de la ejecución de un programa se denominan tipos de datos dinámicos.

La variación de la estructura puede ser únicamente en el número de elementos, como en el caso de una cadena de caracteres, o también en la relación entre ellos, como podría ser el caso de un árbol sintáctico.

Los tipos de datos dinámicos pueden ser almacenados en estructuras de datos estáticas, pero al tratarse de un conjunto de datos, tienen que ser vectores, o bien de forma menos habitual, matrices multidimensionales.

Nota

Las estructuras de datos estáticas son, por definición, lo opuesto a las estructuras de datos dinámicas. Es decir, aquéllas en que tanto el número de los datos como su interrelación no varían en toda la ejecución del programa correspondiente. Por ejemplo, un vector siempre tendrá una longitud determinada y todos los elementos a excepción del primero y del último tienen un elemento precedente y otro siguiente.

En caso de almacenar las estructuras de datos dinámicas en estructuras estáticas, es recomendable comprobar si se conoce el nú-

mero máximo y la cantidad media de datos que puedan tener. Si ambos valores son parecidos, se puede emplear una variable de vector estática o automática. Si son muy diferentes, o bien se desconocen, es conveniente ajustar el tamaño del vector al número de elementos que haya en un momento determinado en la estructura de datos y, por lo tanto, almacenar el vector en una variable de carácter dinámico.

Las estructuras de datos dinámicas se almacenan, por lo común, empleando variables dinámicas. Así pues, puede verse una estructura de datos dinámica como una colección de variables dinámicas cuya relación queda establecida mediante apuntadores. De esta manera, es posible modificar fácilmente tanto el número de datos de la estructura (creando o destruyendo las variables que los contienen) como la propia estructura, cambiando las direcciones contenidas en los apuntadores de sus elementos. En este caso, es habitual que los elementos sean tuplos y que se denominen nodos.

En los apartados siguientes se verán los dos casos, es decir, estructuras de datos dinámicas almacenadas en estructuras de datos estáticas y como colecciones de variables dinámicas. En el primer caso, se tratará de las cadenas de caracteres, pues son, de largo, las estructuras de datos dinámicas más empleadas. En el segundo, de las listas y de sus aplicaciones.

3.5.1. Cadenas de caracteres

Las cadenas de caracteres son un caso particular de vectores en que los elementos son caracteres. Además, se emplea una marca de final (el carácter NUL o '`\0`') que delimita la longitud real de la cadena representada en el vector.

La declaración siguiente sería una fuente de problemas derivada de la no inclusión de la marca de final, puesto que es una norma de C que hay que respetar para poder emplear todas las funciones estándar para el proceso de cadenas:

```
char cadena[20] = { 'H', 'o', 'l', 'a' } ;
```

Por lo tanto, habría que inicializarla de la siguiente manera:

```
char cadena[20] = { 'H', 'o', 'l', 'a', '\0' } ;
```

Las declaraciones de cadenas de caracteres inicializadas mediante texto implican que la marca de final se añada siempre. Por ello, la declaración anterior es equivalente a:

```
char cadena[20] = "Hola" ;
```

Aunque el formato de representación de cadenas de caracteres sea estándar en C, no se dispone de instrucciones ni de operadores que trabajen con cadenas: no es posible hacer asignaciones ni comparaciones de cadenas, es necesario recurrir a las funciones estándar (declaradas en `string.h`) para el manejo de cadenas:

```
int      strlen   ( char *cadena );
char *  strcpy    ( char *destino, char *fuente );
char *  strncpy  ( char *destino, char *fuente, int num_car );
char *  strcat    ( char *destino, char *fuente );
char *  strncat  ( char *destino, char *fuente, int num_car );
char *  strdup    ( char *origen );
char *  strcmp    ( char *cadena1, char *cadena2 );
char *  strncmp  ( char *kdna1, char *kdna2, int num_car );
char *  strchr    ( char *cadena, char caracter );
char *  strrchr   ( char *cadena, char caracter );
```

La longitud real de una cadena de caracteres kdna en un momento determinado se puede obtener mediante la función siguiente:

```
strlen ( kdna )
```

El contenido de la cadena de caracteres apuntada por kdna a kdna9, se puede copiar con `strcpy(kdna9, kdna)`. Si la cadena fuente puede ser más larga que la capacidad del vector correspondiente a la de destino, con `strncpy(kdna9, kdna, LONG_KDNA9 - 1)`. En este último caso, hay que prever que la cadena resultante no lleve un '\0' al final. Para solucionarlo, hay que reservar el último carácter de la copia con conteo para poner un

'\0' de guarda. Si la cadena no tuviera espacio reservado, se tendría que hacer lo siguiente:

```
/* ... */
char *kdna9, kdna[LONG_MAX];
/* ... */
kdna9 = (char *) malloc( strlen( kdna ) + 1 );
if( kdna9 != NULL ) strcpy( kdna9, kdna );
/* ... */
```

Nota

De esta manera, kdna9 es una cadena con el espacio ajustado al número de caracteres de la cadena almacenada en kdna, que se deberá liberar mediante un free(kdna9) cuando ya no sea necesaria. El procedimiento anterior se puede sustituir por:

```
/* ... */
kdna9 = strdup( kdna );
/* ... */
```

La comparación entre cadenas se hace carácter a carácter, empezando por el primero de las dos cadenas a comparar y continuando por los siguientes mientras la diferencia entre los códigos ASCII sea 0. La función strcmp() retorna el valor de la última diferencia. Es decir, un valor negativo si la segunda cadena es alfabéticamente mayor que la primera, positivo en caso opuesto, y 0 si son iguales. Para entenderlo mejor, se adjunta un posible código para la función de comparación de cadenas:

```
int strcmp( char *cadena1, char *cadena2 )
{
    while( (*cadena1 != '\0') &&
          (*cadena2 != '\0') &&
          (*cadena1 == *cadena2) )
    {
        cadena1 = cadena1 + 1;
        cadena2 = cadena2 + 1;
    } /* while */
    return *cadena1 - *cadena2;
} /* strcmp */
```

La función `strncmp()` hace lo mismo que `strcmp()` con los primeros `núm_car` caracteres.

Finalmente, aunque hay más, se comentan las funciones de búsqueda de caracteres dentro de cadenas. Estas funciones retornan el apuntador al carácter buscado o `NULL` si no se encuentra en la cadena:

- `strchr()` realiza la búsqueda desde el primer carácter.
- `strrchr()` inspecciona la cadena empezando por la derecha; es decir, por el final.

Ejemplo

```
char *strchr( char *cadena, char caracter )
{
    while( (*cadena != '\0') && (*cadena2 != caracter)
) ) {
    cadena = cadena + 1;
} /* while */
return cadena;
} /* strchr */
```



Todas las funciones anteriores están declaradas en `string.h` y, por tanto, para utilizarlas, hay que incluir este fichero en el código fuente del programa correspondiente.

En `stdio.h` también hay funciones estándar para operar con cadenas, como `gets()` y `puts()`, que sirven para la entrada y la salida de datos que sean cadenas de caracteres y que ya fueron descritas en su momento. También contiene las declaraciones de `sscanf()` y `sprintf()` para la lectura y la escritura de cadenas con formato. Estas dos últimas funciones se comportan exactamente igual que `scanf()` y `printf()` a excepción de que la lectura o escritura se realizan en una cadena de caracteres en lugar de hacerlo en el dispositivo estándar de entrada o salida:

```
sprintf(
    char *destino, /* Cadena en la que "imprime". */
    char *formato
    [, lista_de_variables]
); /* sprintf */
```

```
int sscanf(          /* Devuelve el número de variables      */
                /* cuyo contenido ha sido actualizado.   */
char *origen,     /* Cadena de la que se "lee".           */
char *formato
[,lista_de_variables]
); /* sscanf */
```



Cuando se utiliza `sprintf()` se debe comprobar que la cadena destino tenga espacio suficiente para contener aquello que resulte de la impresión con el formato dado.

Cuando se utiliza `sscanf()`, se debe comprobar siempre que se han leído todos los campos: la inspección de la cadena `origen` se detiene al encontrar la marca de final de cadena independientemente de los especificadores de campo indicados en el formato.

En el siguiente ejemplo, se puede observar el código de una función de conversión de una cadena que represente un valor hexadecimal (por ejemplo: "3D") a un entero positivo (siguiendo el ejemplo anterior: $3D_{(16)} = 61$) mediante el uso de las funciones anteriormente mencionadas. La primera se emplea para prefijar la cadena con "0x", puesto que éste es el formato de los números hexadecimales en C, y la segunda, para efectuar la lectura de la cadena obtenida aprovechando que lee números en cualquier formato estándar de C.

```
unsigned hexaVal( char *hexadecimal )
{
    unsigned numero;
    char *hexaC;

    hexaC = (char *) malloc(
        ( strlen( hexadecimal ) + 3 ) * sizeof( char )
    ); /* malloc */
```

```
if( hexaC != NULL ) {  
    sprintf( hexaC, "0x%s", hexadecimal );  
    sscanf( hexaC, "%x", &numero );  
    free( hexaC );  
} else {  
    numero = 0; /* ¡La conversión no se ha realizado! */  
} /* if */  
return numero;  
} /* hexaVal */
```

Nota

Recordad la importancia de liberar el espacio de las cadenas de caracteres creadas dinámicamente que no se vayan a utilizar. En el caso anterior, de no hacer un `free(hexaC)`, la variable seguiría ocupando espacio, a pesar de que ya no se podría acceder a ella, puesto que la dirección está contenida en el apuntador `hexaC`, que es de tipo automático y, por tanto, se destruye al finalizar la ejecución de la función. Este tipo de errores puede llegar a causar un gran desperdicio de memoria.

3.5.2. Listas y colas

Las listas son uno de los tipos dinámicos de datos más empleados y consisten en secuencias homogéneas de elementos sin tamaño predeterminado. Al igual que las cadenas de caracteres, pueden almacenarse en vectores siempre que se sepa su longitud máxima y la longitud media durante la ejecución. Si esto no es así, se emplearán variables dinámicas “enlazadas” entre sí; es decir, variables que contendrán apuntadores a otras dentro de la misma estructura dinámica de datos.

La ventaja que aporta representar una lista en un vector es que elimina la necesidad de un campo apuntador al siguiente. De todas maneras, hay que insistir en que sólo es posible aprovecharla si no se produce un derroche de memoria excesivo y cuando el programa no deba hacer frecuentes inserciones y eliminaciones de elementos en cualquier posición de la lista.

En este apartado se verá una posible forma de programar las operaciones básicas con una estructura de datos dinámica de tipo lista

mediante variables dinámicas. En este caso, cada elemento de la lista será un nodo del tipo siguiente:

```
typedef struct nodo_s {
    int         dato;
    struct nodo_s *siguiente;
} nodo_t, *lista_t;
```

El tipo `nodo_t` se corresponde con un nodo de la lista y que `lista_t` es un apuntador a un nodo cuya tupla tiene un campo `siguiente` que, a la vez, es un apuntador a otro nodo, y así repetidamente hasta tener enlazados toda una lista de nodos. A este tipo de listas se las denomina **listas simplemente enlazadas**, pues sólo existe un enlace entre un nodo y el siguiente.

Las listas simplemente enlazadas son adecuadas para algoritmos que realicen frecuentes recorridos secuenciales. En cambio, para aquéllos en que se hacen recorridos parciales en ambos sentidos (hacia delante y hacia atrás en la secuencia de nodos) es recomendable emplear **listas doblemente enlazadas**; es decir, listas cuyos nodos contengan apuntadores a los elementos siguientes y anteriores.

En ambos casos, si el algoritmo realiza inserciones de nuevos elementos y destrucciones de elementos innecesarios de forma muy frecuente, puede ser conveniente tener el primer y el último elemento enlazados. Estas estructuras se denominan **listas circulares** y, habitualmente, los primeros elementos están marcados con algún dato o campo especial.

Operaciones elementales con listas

Una lista de elementos debe permitir llevar a cabo las operaciones siguientes:

- Acceder a un nodo determinado.
- Eliminar un nodo existente.
- Insertar un nuevo nodo.

En los apartados siguientes se comentarán estas tres operaciones y se darán los programas de las funciones para llevarlas a cabo sobre una lista simplemente enlazada.

Para acceder a un nodo determinado es imprescindible obtener su dirección. Si se supone que se trata de obtener el enésimo elemento en una lista y devolver su dirección, la función correspondiente necesita como argumentos tanto la posición del elemento buscado, como la dirección del primer elemento de la lista, que puede ser NULL si ésta está vacía.

Evidentemente, la función retornará la dirección del nodo enésimo o NULL si no lo ha encontrado:

```
nodo_t *enesimo_nodo(lista_t lista, unsigned int n)
{
    while( ( lista != NULL ) && ( n != 0 ) ) {
        lista = lista ->siguiente;
        n = n - 1;
    } /* while */
    return lista;
} /* enesimo_nodo */
```

En este caso, se considera que la primera posición es la posición número 0, de forma parecida a los vectores en C. Es imprescindible comprobar que (lista != NULL) se cumple, puesto que, de lo contrario, no podría ejecutarse lista = lista->siguiente;: no se puede acceder al campo siguiente de un nodo que no existe.

Para eliminar un elemento en una lista simplemente enlazada, es necesario disponer de la dirección del elemento anterior, ya que el campo siguiente de este último debe actualizarse convenientemente. Para ello, es necesario extender la función anterior de manera que devuelva tanto la dirección del elemento buscado como la del elemento anterior. Como sea que tiene que devolver dos datos, hay que hacerlo a través de un paso por referencia: hay que pasar las direcciones de los apuntadores a nodo que contendrán las direcciones de los nodos:

```
void enesimo_pq_nodo(
    lista_t    lista,    /* Apuntador al primer nodo.
    unsigned int n,      /* Posición del nodo buscado.
```

*/

*/

```

nodo_t      **pref, /* Ref. apuntador a nodo previo.*/
nodo_t      **qref) /* Ref. apuntador a nodo actual.*/
{
    nodo_t *p, *q;
    p = NULL; /* El anterior al primero no existe. */
    q = lista;
    while( ( q != NULL ) && ( n != 0 ) ) {
        p = q;
        q = q->siguiente;
        n = n - 1;
    } /* while */
    *pref = p;
    *qref = q;
} /* enesimo_pq_nodo */

```



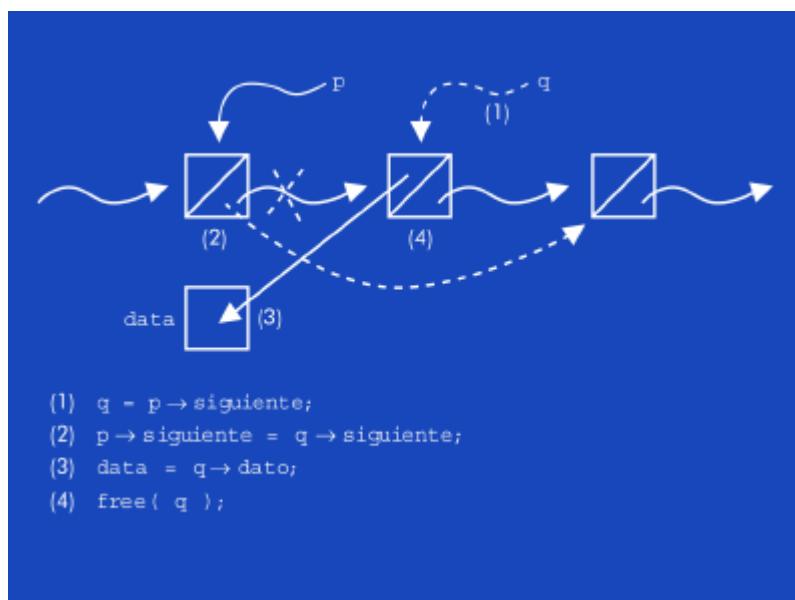
El programa de la función que destruya un nodo debe partir de que se conoce tanto su dirección (almacenada en *q*) como la del posible elemento anterior (guardada en el apuntador *p*). Dicho de otra manera, se pretende eliminar el elemento siguiente al apuntado por *p*.

Cuando se realizan estos programas, es más que conveniente hacer un esquema de la estructura de datos dinámica sobre el que se indiquen los efectos de las distintas modificaciones en la misma.

Así pues, para programar esta función, primero estableceremos el caso general sobre un esquema de la estructura de la cual queremos eliminar un nodo y, luego, se programará atendiendo a las distintas excepciones que puede tener el caso general. Habitualmente, éstas vienen dadas por tratar el primer o el último elemento, pues son aquellos que no tienen precedente o siguiente y, como consecuencia, no siguen la norma de los demás en la lista.

En la siguiente imagen se resume el procedimiento general de eliminación del nodo siguiente al nodo p:

Figura 6.



Nota

Evidentemente, no es posible eliminar un nodo cuando $p == \text{NULL}$. En este caso, tanto (1) como (2) no pueden ejecutarse por implicar referencias a variables inexistentes. Más aún, para $p == \text{NULL}$ se cumple que se desea eliminar el primer elemento, pues es el único que no tiene ningún elemento que le preceda. Así pues, hemos de “proteger” la ejecución de (1) y (2) con una instrucción condicional que decida si se puede llevar a cabo el caso general o, por el contrario, se elimina el primero de los elementos. Algo parecido sucede con (3) y (4), que no pueden ejecutarse si $q == \text{NULL}$.

La función para destruir el nodo podría ser como la siguiente:

```
int destruye_nodo(
    lista_t *listaref, /* Apuntador a referencia 1.er nodo. */
    nodo_t *p,          /* Apuntador a nodo previo. */
    nodo_t *q)          /* Apuntador a nodo a destruir. */
{
    int data = 0         /* Valor por omisión de los datos. */
}
```

```

if( p != NULL ) {
    /* q = p→siguiente; (no es necesario) */
    p→siguiente = q→siguiente;
} else {
    if( q != NULL ) *listaref = q→siguiente;
} /* if */
if( q!= NULL ) {
    data = q→dato;
    free( q );
} /* if */
return data;
} /* destruye_nodo */

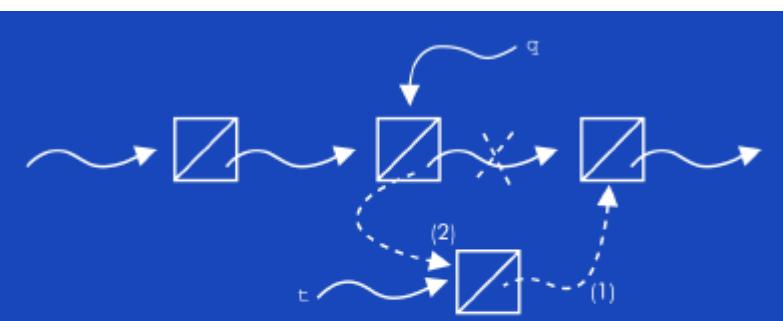
```



Al eliminar el primer elemento, es necesario cambiar la dirección contenida en lista para que apunte al nuevo primer elemento, salvo que q también sea NULL).

Para insertar un nuevo nodo, cuya dirección esté en t, basta con realizar las operaciones indicadas en la siguiente figura:

Figura 7.



- (1) $t \rightarrow \text{siguiente} = q \rightarrow \text{siguiente};$
- (2) $q \rightarrow \text{siguiente} = t$

Nota

En este caso, el nodo t quedará insertado después del nodo q. Como se puede observar, es necesario que $q \neq \text{NULL}$ para poder realizar las operaciones de inserción.

El código de la función correspondiente sería como sigue:

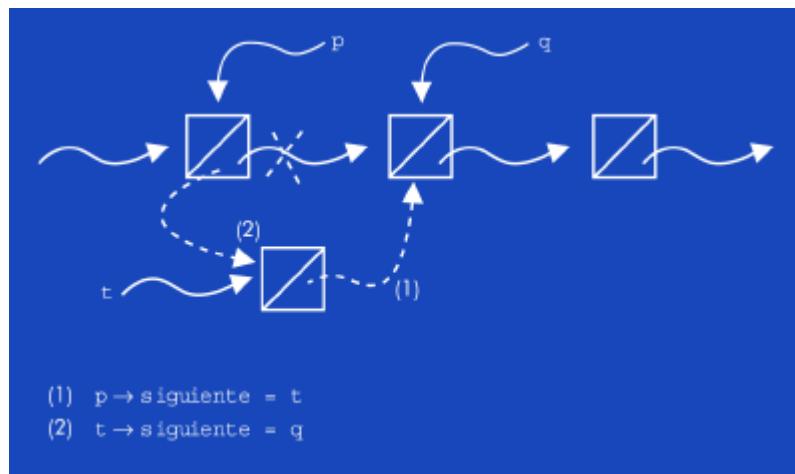
```
void inserta_siguiente_nodo(
    lista_t *listaref, /* Apuntador a referencia 1.er nodo. */
    nodo_t *q,         /* Apuntador a nodo en la posición. */
    nodo_t *t)         /* Apuntador a nodo a insertar. */
{
    if( q != NULL ) {
        t->siguiente = q->siguiente;
        q->siguiente = t;
    } else { /* La lista está vacía. */
        *listaref = t;
    } /* if */
} /* inserta_siguiente_nodo */
```

Para que la función anterior pueda ser útil, es necesario disponer de una función que permita crear nodos de la lista. En este caso:

```
nodo_t *crea_nodo( int data )
{
    nodo_t *nodoref;
    nodoref = (nodo_t *)malloc( sizeof( nodo_t ) );
    if( nodoref != NULL ) {
        nodoref->dato = data;
        nodoref->siguiente = NULL;
    } /* if */
    return nodoref;
} /* crea_nodo */
```

Si se trata de insertar en alguna posición determinada, lo más frecuente suele ser insertar el nuevo elemento como precedente del indicado; es decir, que ocupe la posición del nodo referenciado y desplace al resto de nodos una posición “a la derecha”. Las operaciones que hay que realizar para ello en el caso general se muestran en la siguiente figura:

Figura 8.



Siguiendo las indicaciones de la figura anterior, el código de la función correspondiente sería el siguiente:

```
void inserta_nodo(
    lista_t *listaref, /* Apuntador a referencia 1er nodo. */
    nodo_t *p,          /* Apuntador a nodo precedente. */
    nodo_t *q,          /* Apuntador a nodo en la posición. */
    nodo_t *t)          /* Apuntador a nodo a insertar. */
{
    if( p != NULL ) {
        p->siguiente = t;
    } else { /* Se inserta un nuevo primer elemento. */
        *listaref = t;
    } /* if */
    t->siguiente = q;
} /* inserta_nodo */
```

Con todo, la inserción de un nodo en la posición enésima podría construirse de la siguiente manera:

```
bool inserta_enesimo_lista(
    lista_t *listaref, /* Apuntador a referencia 1.er nodo. */
    unsigned int n,    /* Posición de la inserción. */
    int dato)          /* Dato a insertar. */
{
    /* Devuelve FALSE si no se puede. */
    nodo_t *p, *q, *t;
    bool retval;
```

```
t = crea_nodo( dato );
if( t != NULL ) {
    enesimo_pq_nodo( *listaref, n, &p, &q );
    inserta_nodo( listaref, p, q, t );
    retval = TRUE;
} else {
    retval = FALSE;
} /* if */
return retval;
} /* inserta_enesimo_lista */
```

De igual manera, podría componerse el código para la función de destrucción del enésimo elemento de una lista.

Normalmente, además, las listas no serán de elementos tan simples como enteros y habrá que sustituir la definición del tipo de datos `nodo_t` por uno más adecuado.



Los criterios de búsqueda de nodos suelen ser más sofisticados que la búsqueda de una determinada posición. Por lo tanto, hay que tomar las funciones anteriores como un ejemplo de uso a partir del cual se pueden derivar casos reales de aplicación.

Colas

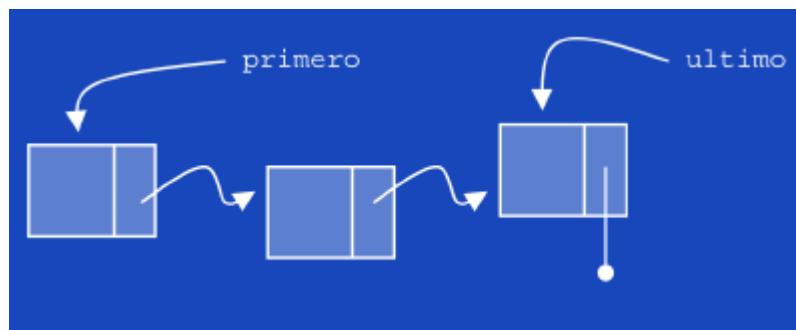
Las colas son, de hecho, listas en las que se inserta por un extremo y se elimina por el otro. Es decir, son listas en las que las operaciones de inserción y de eliminación se restringen a unos casos muy determinados. Esto permite hacer una gestión mucho más eficaz de las mismas. En este sentido, es conveniente disponer de un tuplo que facilite tanto el acceso directo al primer elemento como al último. De esta manera, las operaciones de eliminación e inserción se pueden resolver sin necesidad de búsquedas en la lista.

Así pues, esta clase de colas debería tener una tupla de control como la siguiente:

```
typedef struct cola_s {
    nodo_t *primero;
    nodo_t *ultimo;
} cola_t;
```

Gráficamente:

Figura 9.



Para ejemplificar las dos operaciones, supondremos que los elementos de la cola serán simples enteros, es decir, que la cola será una lista de nodos del tipo de datos `nodo_t` que ya se ha visto.

Con ello, la inserción sería como sigue:

```
bool encola( cola_t *colaref, int dato )
/* Devuelve FALSE si no se puede añadir el dato.*/
{
    nodo_t *q, *t;
    bool    retval;

    t = crea_nodo( dato );
    if( t != NULL ) {
        t->siguiente = NULL;
        q = colaref->ultimo;
        if( q == NULL ) { /* Cola vacía: */
            colaref->primero = t;
            retval = TRUE;
        } else {
            t->siguiente = q->siguiente;
            q->siguiente = t;
            retval = TRUE;
        }
    } else {
        retval = FALSE;
    }
}
```

```
    } else {
        q->siguiente = t;
    } /* if */
    colaref->ultimo = t;
    retval = TRUE;
} else {
    retval = FALSE;
} /* if */
return retval;
} /* encola */
```

Y la eliminación:

```
bool desencola( cola_t *colaref, int *datoref )
/* Devuelve FALSE si no se puede eliminar el dato.      */
{
    nodo_t *q;
    bool retval;
    q = colaref->primero;
    if( q != NULL ) {
        colaref->primero = q->siguiente;
        *datoref = destruye_nodo( &q );
        if( colaref->primero == NULL ) { /* Cola vacía:      */
            colaref->ultimo = NULL;
        } /* if */
        retval = TRUE;
    } else {
        retval = FALSE;
    } /* if */
    return retval;
} /* desencola */
```

La función `destruye_nodo` de la eliminación anterior es como sigue:

```
int destruye_nodo( nodo_t **pref )
{
```

```

int dato = 0;
if( *pref != NULL ) {
    dato = (*pref) →dato;
    free( *pref );
    *pref = NULL;
} /* if */
return dato;
} /* destruye_nodo */

```

Las colas se utilizan frecuentemente cuando hay recursos compartidos entre muchos usuarios.

Ejemplo

- Para gestionar una impresora, el recurso es la propia impresora y los usuarios, los ordenadores conectados a la misma.
- Para controlar una máquina del tipo "su turno": el recurso es el vendedor y los usuarios son los clientes.

En general, cuando se hacen inserciones en cola se tiene en cuenta qué elemento se está insertando; es decir, no se realizan siempre por el final, sino que el elemento se coloca según sus privilegios sobre los demás. En este caso, se habla de **colas con prioridad**. En este tipo de colas, la eliminación siempre se realiza por el principio, pero la inserción implica situar al nuevo elemento en la última posición de los elementos con la misma prioridad.

Ciertamente, hay otros tipos de gestiones especializadas con listas y, más allá de las listas, otros tipos de estructuras dinámicas de datos como los árboles (por ejemplo, un árbol sintáctico) y los grafos (por ejemplo, una red de carreteras). Desafortunadamente, no se dispone de suficiente tiempo para tratarlos, pero hay que tenerlos en cuenta cuando los datos del problema puedan requerirlo.

3.6. Diseño descendente de programas

Recordemos que la programación modular estriba en dividir el código en subprogramas que realicen una función concreta. En esta uni-

dad se tratará especialmente de la manera como pueden agruparse estos subprogramas de acuerdo a las tareas que les son encomendadas y, en definitiva, de cómo organizarlos para una mejor programación del algoritmo correspondiente.

Los algoritmos complejos suelen reflejarse en programas con muchas líneas de código. Por lo tanto se impone una programación muy cuidadosa para que el resultado sea un código legible y de fácil mantenimiento.

3.6.1. Descripción

El fruto de una programación modular es un código constituido por diversos subprogramas de pocas líneas relacionados entre ellos mediante llamadas. Así pues, cada uno de ellos puede ser de fácil comprensión y, consecuentemente, de fácil mantenimiento.

La técnica de diseño descendente es, de hecho, una técnica de diseño de algoritmos en la que se resuelve el algoritmo principal abstrayendo los detalles, que luego se solucionan mediante otros algoritmos de la misma manera. Es decir, se parte del nivel de abstracción más alto y, para todas aquellas acciones que no puedan trasladarse de forma directa a alguna instrucción del lenguaje de programación elegido, se diseñan los algoritmos correspondientes de forma independiente del principal siguiendo el mismo principio.



El diseño descendente consiste en escribir programas de algoritmos de unas pocas instrucciones e implementar las instrucciones no primitivas con funciones cuyos programas sigan las mismas normas anteriores.

Nota

Una instrucción primitiva sería aquella que pueda programarse directamente en un lenguaje de programación.

En la práctica, esto comporta diseñar algoritmos de manera que permite que la programación de los mismos se haga de forma totalmente modular.

3.6.2. Ejemplo

El diseño descendente de programas consiste, pues, en empezar por el programa del algoritmo principal e ir refinando aquellas instrucciones “gruesas” convirtiéndolas en subprogramas con instrucciones más “finas”. De ahí la idea de refinar. Evidentemente, el proceso termina cuando ya no hay más instrucciones “gruesas” para refinrar.

En este apartado se verá un ejemplo simple de diseño descendente para resolver un problema bastante habitual en la programación: el de la ordenación de datos para facilitar, por ejemplo, su consulta.

Este problema es uno de los más estudiados en la ciencia informática y existen diversos métodos para solucionarlo. Uno de los más simples consiste en seleccionar el elemento que debería encabezar la clasificación (por ejemplo, el más pequeño o el mayor, si se trata de números), ponerlo en la lista ordenada y repetir el proceso con el resto de elementos por ordenar. El programa principal de este algoritmo puede ser el siguiente:

```
/* ... */  
lista_t      pendientes, ordenados;  
elemento_t   elemento;  
/* ... */  
inicializa_lista( &ordenados );  
while( ! esta_vacia_lista( pendientes ) ) {  
    elemento = extrae_minimo_de_lista( &pendientes );  
    pon_al_final_de_lista( &ordenados, elemento );  
} /* while */  
/* ... */
```

En el programa anterior hay pocas instrucciones primitivas de C y, por tanto, habrá que refinarlo. Como contrapartida, su funcionamiento resulta fácil de comprender. Es importante tener en cuenta que los operadores de “dirección de” (el signo `&`) en los parámetros de las llamadas de las funciones indican que éstas pueden modificar el contenido de los mismos.

La mayor dificultad que se encuentra en el proceso de refinado es, habitualmente, identificar las partes que deben describirse con ins-

trucciones primitivas; es decir, determinar los distintos niveles de abstracción que deberá tener el algoritmo y, consecuentemente, el programa correspondiente. Generalmente, se trata de conseguir que el programa refleje al máximo el algoritmo del que proviene.

Es un error común pensar que aquellas operaciones que sólo exigen una o dos instrucciones primitivas no pueden ser nunca contempladas como una instrucción no primitiva.

Una norma de fácil adopción es que todas las operaciones que se realicen con un tipo de datos abstracto sean igualmente abstractas; es decir, se materialicen en instrucciones no primitivas (funciones).

3.7. Tipos de datos abstractos y funciones asociadas

La mejor manera de implementar la programación descendente es programar todas las operaciones que puedan hacerse con cada uno de los tipos de datos abstractos que se tengan que emplear. De hecho, se trata de crear una máquina virtual para ejecutar aquellas instrucciones que se ajustan al algoritmo, a la manera que un lenguaje dispone de todas las operaciones necesarias para la máquina que es capaz de procesarlo y, obviamente, luego se traslada a las operaciones del lenguaje de la máquina real que realizará el proceso.

En el ejemplo del algoritmo de ordenación anterior, aparecen dos tipos de datos abstractos (`lista_t` y `elemento_t`) para los que, como mínimo, son necesarias las operaciones siguientes:

```
void inicializa_lista( lista_t *ref_lista );
bool esta_vacia_lista( lista_t lista );
elemento_t extrae_minimo_de_lista( lista_t *ref_lista );
void pon_al_final_de_lista( lista_t *rlst, elemento_t e );
```

Como se puede observar, no hay operaciones que afecten a datos del tipo `elemento_t`. En todo caso, es seguro que el programa hará uso de ellas (lectura de datos, inserción de los mismos en la lista, comparación entre elementos, escritura de resultados, etc.) en al-

guna otra sección. Por lo tanto, también se habrán de programar las operaciones correspondientes. En particular, si se observa el siguiente código se comprobará que aparecen funciones para tratar con datos del tipo `elemento_t`:

```
elemento_t extrae_minimo_de_lista( lista_t *ref_lista )
{
    ref_nodo_t actual, minimo;
    bool es_menor;
    elemento_t pequе;
    principio_de_lista( ref_lista );
    if( esta_vacia_lista( *ref_lista ) ) {
        inicializa_elemento( &peque );
    } else {
        minimo = ref_nodo_de_lista( *ref_lista );
        pequе = elemento_en_ref_nodo( *ref_lista, minimo );
        avanza_posicion_en_lista( ref_lista );
        while( !es_final_de_lista( *ref_lista ) ) {
            actual = ref_nodo_de_lista( *ref_lista );
            es_menor = compara_elementos(
                elemento_en_ref_nodo( *ref_lista, actual ), pequе
            ); /* compara_elementos */
            if( es_menor ) {
                minimo = actual;
                pequе = elemento_en_ref_nodo( *ref_lista, minimo );
            } /* if */
            avanza_posicion_en_lista( ref_lista );
        } /* while */
        muestra_elemento( pequе );
        elimina_de_lista( ref_lista, minimo );
    } /* if */
    return pequе;
} /* extrae_minimo_de_lista */
```

Como se puede deducir del código anterior, al menos son necesarias dos operaciones para datos del tipo `elemento_t`:

```
inicializa_elemento();
compara_elementos();
```

Además, son necesarias cuatro operaciones más para listas:

```
principio_de_lista();
es_final_de_lista();
avanza_posicion_en_lista();
elimina_de_lista();
```

Se ha añadido también el tipo de datos `ref_nodo_t` para tener las referencias de los nodos en las listas y dos operaciones: `ref_nodo_de_lista` para obtener la referencia de un determinado nodo en la lista y `elemento_en_ref_nodo` para obtener el elemento que se guarda en el nodo indicado.

Con esto se demuestra que el refinamiento progresivo sirve para determinar qué operaciones son necesarias para cada tipo de datos y, por otra parte, se refleja que las listas constituyen un nivel de abstracción distinto y mayor que el de los elementos.

Para completar la programación del algoritmo de ordenación, es necesario desarrollar todas las funciones asociadas a las listas, y luego a los elementos. De todas maneras, lo primero que hay que hacer es determinar los tipos de datos abstractos que se emplearán.

Las listas pueden materializarse con vectores o con variables dinámicas, según el tipo de algoritmos que se empleen. En el caso del ejemplo de la ordenación, dependerá en parte de los mismos criterios generales que se aplican para tomar tal decisión y, en parte, de las características del mismo algoritmo. Generalmente, se elegirá una implementación con vectores si el desaprovechamiento medio de los mismos es pequeño, y particularmente se tiene en cuenta que el algoritmo que nos ocupa sólo se puede aplicar para la clasificación de cantidades modestas de datos (por ejemplo, unos centenares como mucho).

Así pues, de escoger la primera opción, el tipo de datos `lista_t` sería:

```
#define LONG_MAXIMA 100
typedef struct lista_e {
```

```

elemento_t     nodo[ LONG_MAXIMA ];
unsigned short posicion; /* Posición actual de acceso. */
unsigned short cantidad; /* Longitud de la lista. */
} lista_t;

```

También haría falta definir el tipo de datos para la referencia de los nodos:

```
typedef unsigned short ref_nodo_t;
```

De esta manera, el resto de operaciones que son necesarias se corresponderían con las funciones siguientes:

```

void inicializa_lista( lista_t *ref_lista )
{
    (*ref_lista).cantidad = 0;
    (*ref_lista).posicion = 0;
} /* inicializa_lista */

bool esta_vacia_lista( lista_t lista )
{
    return ( lista.cantidad == 0 );
} /* esta_vacia_lista */

bool es_final_de_lista( lista_t lista )
{
    return ( lista_posicion == lista.cantidad );
} /* es_final_de_lista */

void principio_de_lista( lista_t *lista_ref )
{
    lista_ref->posicion = 0;
} /* principio_de_lista */

ref_nodo_t ref_nodo_de_lista( lista_t lista )
{
    return lista.posicion;
} /* ref_nodo_de_lista */

elemento_t elemento_en_ref_nodo(
    lista_t lista,

```

```
ref_nodo_t refnodo)
{
    return lista.nodo[ refnodo ];
} /* elemento_en_ref_nodo */

void avanza_posicion_en_lista( lista_t *lista_ref )
{
    if( !es_final_de_lista( *lista_ref ) ) {
        (*lista_ref).posicion= (*lista_ref).posicion + 1;
    } /* if */
} /* avanza_posicion_en_lista */

elemento_t elimina_de_lista(
    lista_t *ref_lista,
    ref_nodo_t refnodo)
{
    elemento_t eliminado;
    ref_nodo_t pos, ultimo;

    if( esta_vacia_lista( *ref_lista ) ) {
        inicializa_elemento( &eliminado );
    } else {
        eliminado = (*ref_lista).nodo[ refnodo ];
        ultimo = (*ref_lista).cantidad - 1;
        for( pos= refnodo; pos < ultimo; pos= pos + 1 ) {
            (*ref_lista).nodo[pos] = (*ref_lista).nodo[pos+1];
        } /* for */
        (*ref_lista).cantidad= (*ref_lista).cantidad - 1;
    } /* if */
    return eliminado;
} /* elimina_de_lista */

elemento_t extrae_minimo_de_lista( lista_t *ref_lista );

void pon_al_final_de_lista(
    lista_t *ref_lista,
    elemento_t elemento )
{
    if( (*ref_lista).cantidad < LONG_MAXIMA ) {
        (*ref_lista).nodo[(*ref_lista).cantidad] = elemento;
        (*ref_lista).cantidad = (*ref_lista).cantidad + 1;
    }
}
```

```
    } /* if */
} /* pon_al_final_de_lista */
```

Si se examinan las funciones anteriores, todas asociadas al tipo de datos `lista_t`, se verá que sólo requieren de una operación con el tipo de datos `elemento_t`: `compara_elementos`. Por lo tanto, para completar el programa de ordenación, ya sólo falta definir el tipo de datos y la operación de comparación.

Si bien las operaciones con las listas sobre vectores eran genéricas, todo aquello que afecta a los elementos dependerá de la información cuyos datos se quieran ordenar.

Por ejemplo, suponiendo que se deseen ordenar por número de DNI las notas de un examen, el tipo de datos de los elementos podría ser como sigue:

```
typedef struct elemento_s {
    unsigned int      DNI;
    float            nota;
} dato_t, *elemento_t;
```

La función de comparación sería como sigue:

```
bool compara_elementos(
    elemento_t menor,
    elemento_t mayor )
{
    return ( menor→DNI < mayor→DNI );
} /* compara_elementos */
```

La función de inicialización sería como sigue:

```
void inicializa_elemento( elemento_t *ref_elem )
{
    *ref_elem = NULL;
} /* inicializa_elemento */
```

Nótese que los elementos son, en realidad, apuntadores de variables dinámicas. Esto, a pesar de requerir que el programa las construya

y destruya, simplifica, y mucho, la codificación de las operaciones en niveles de abstracción superiores. No obstante, es importante recalcar que siempre habrá que preparar funciones para la creación, destrucción, copia y duplicado de cada uno de los tipos de datos para los que existen variables dinámicas.



Las funciones de copia y de duplicado son necesarias puesto que la simple asignación constituye una copia de la dirección de una variable a otro apuntador, es decir, se tendrán dos referencias a la misma variable en lugar de dos variables distintas con el mismo contenido.

Ejemplo

Comprobad la diferencia que existe entre estas dos funciones:

```
/* ... */
elemento_t original, otro, copia;
/* ... */
otro = original; /* Copia de apuntadores. */
/* la dirección guardada en 'otro'
   es la misma que la contenida en 'original'
 */

/* ... */
copia_elemento( original, copia );
otro = duplica_elemento( original );
/* las direcciones almacenadas en 'copia' y en 'otro'
   son distintas de la contenida en 'original'
 */
```

La copia de contenidos debe, pues, realizarse mediante una función específica y, claro está, si la variable que debe contener la copia no está creada, deberá procederse primero a crearla, es decir, se hará un duplicado.

En resumen, cuando hayamos de programar un algoritmo en diseño descendente, habremos de programar las funciones de creación, destrucción y copia para cada uno de los tipos de datos abstractos que contenga. Además, se programará como función toda operación que se realice con cada tipo de datos para que queden reflejados los distintos niveles de abstracción presentes en el algoritmo. Con todo, aun a costa de alguna línea de código más, se consigue un programa inteligible y de fácil manutención.

3.8. Ficheros de cabecera

Es común que varios equipos de programadores colaboren en la realización de un mismo programa, si bien es cierto que, muchas veces, esta afirmación es más bien la manifestación de un deseo que el reflejo de una realidad. En empresas pequeñas y medianas se suele traducir en que los equipos sean de una persona e, incluso, que los diferentes equipos se reduzcan a uno solo. De todas maneras, la afirmación, en el fondo, es cierta: la elaboración de un programa debe aprovechar, en una buena práctica de la programación, partes de otros programas. El reaprovechamiento permite no sólo reducir el tiempo de desarrollo, sino también tener la garantía de emplear componentes de probada funcionalidad.

Todo esto es, además, especialmente cierto para el software libre, en el que los programas son, por norma, producto de un conjunto de programadores diverso y no forzosamente coordinado: un programador puede haber aprovechado código elaborado previamente por otros para una aplicación distinta de la que motivó su desarrollo original.

Para permitir el aprovechamiento de un determinado código, es conveniente eliminar los detalles de implementación e indicar sólo el tipo de datos para el cual está destinado y qué operaciones se pueden realizar con las variables correspondientes. Así pues, basta con disponer de un fichero donde se definan el tipo de datos abstracto y se declaren las funciones que se proveen para las variables del mismo. A estos ficheros se los llama ficheros de cabecera por constituirse como la parte inicial del código fuente de las funciones cuyas declaraciones o cabeceras se han incluido en los mismos.



En los ficheros de cabecera se da a conocer todo aquello relativo a un tipo de datos abstracto y a las funciones para su manejo.

3.8.1. Estructura

Los ficheros de cabecera llevan la extensión ".h" y su contenido debe organizarse de tal manera que sea de fácil lectura. Para ello, primero

se deben colocar unos comentarios indicando la naturaleza de su contenido y, sobre todo, de la funcionalidad del código en el fichero ".c" correspondiente. Después, es suficiente con seguir la estructura de un programa típico de C: inclusión de ficheros de cabecera, definición de constantes simbólicas y, por último, declaración de las funciones.



La definición debe estar siempre en el fichero ".c".

Sirva como ejemplo el siguiente fichero de cabecera para operar con números complejos:

```
/* Fichero:    complejos.h
 * Contenido:  Funciones para operar con números
 *              complejos del tipo (X + iY) en los que
 *              X es la parte real e Y, la imaginaria.
 * Revisión: 0.0 (original) */

#ifndef _NUMEROS_COMPLEJOS_H_
#define _NUMEROS_COMPLEJOS_H_

#include <stdio.h>
#define PRECISION 1E-10
typedef struct complex_s {
    double real, imaginario;
} *complex_t;
complex_t nuevo_complejo( double real, double imaginario );
void borra_complejo( complex_t complejo );
void imprime_complejo( FILE *fichero, complex_t complejo );
double modulo_complejo( complex_t complejo );
complex_t opuesto_complejo( complex_t complejo );
complex_t suma_complejos( complex_t c1, complex_t c2 );
/* etcétera */

#endif /* _NUMEROS_COMPLEJOS_H_ */
```

Las definiciones de tipos y constantes y las declaraciones de funciones se han colocado como el cuerpo del comando del preprocesador

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todo¹⁶⁷os Derechos Reservados.**

`#ifndef ... #endif`. Este comando pregunta si una determinada constante está definida y, de no estarlo, traslada al compilador lo que contenga el fichero hasta la marca de final. El primer comando del cuerpo de este comando condicional consiste, precisamente, en definir la constante `_NUMEROS_COMPLEJOS_H_` para evitar que una nueva inclusión del mismo fichero genere el mismo código fuente para el compilador (es innecesario si ya lo ha procesado una vez).

Los llamados **comandos de compilación condicional** del preprocesador permiten decidir si un determinado trozo de código fuente se suministra al compilador o no. Los resumimos en la tabla siguiente:

Tabla 8.

Comando	Significado
<code>#if expresión</code>	Las líneas siguientes son compiladas si expresión ? 0.
<code>#ifdef SÍMBOLO</code>	Se compilan las líneas siguientes si SÍMBOLO está definido.
<code>#ifndef SÍMBOLO</code>	Se compilan las líneas siguientes si SÍMBOLO no está definido.
<code>#else</code>	Finaliza el bloque compilado si se cumple la condición e inicia el bloque a compilar en caso contrario.
<code>#elif expresión</code>	Encadena un else con un if.
<code>#endif</code>	Indica el final del bloque de compilación condicional.

Nota

Un símbolo definido (por ejemplo: SÍMBOLO) puede anularse mediante:

`#undef SÍMBOLO`

y, a partir de ese momento, se considera como no definido.

Las formas:

```
#ifdef SÍMBOLO  
#ifndef SÍMBOLO
```

son abreviaciones de:

```
#if defined( SÍMBOLO )  
#if !defined( SÍMBOLO )
```

respectivamente. La función `defined` puede emplearse en expresiones lógicas más complejas.

Para acabar esta sección, cabe indicar que el fichero de código fuente asociado debe incluir, evidentemente, su fichero de cabecera:

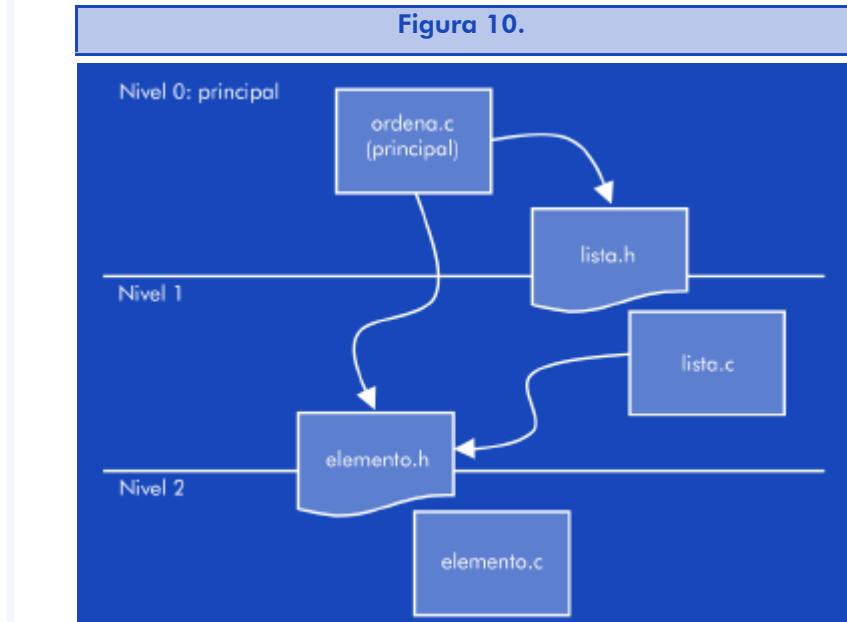
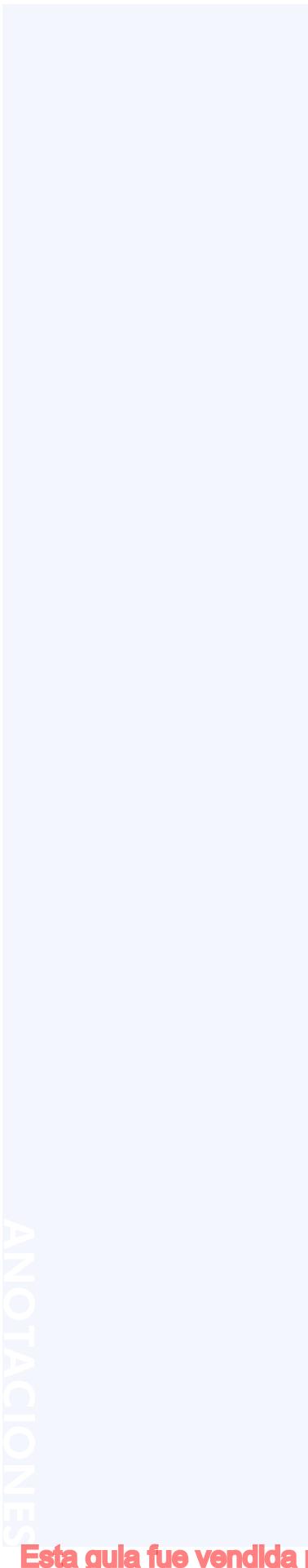
```
/* Fichero:  complejos.c          */  
/* ... */  
#include "complejos.h"  
/* ... */
```

Nota

La inclusión se hace indicando el fichero entre comillas dobles en lugar de utilizar “paréntesis angulares” (símbolos de mayor y menor que) porque se supone que el fichero que se incluye se encuentra en el mismo directorio que el fichero que contiene la directiva de inclusión. Los paréntesis angulares se deben usar si se requiere que el preprocesador examine el conjunto de caminos de acceso a directorios estándar de ficheros de inclusión, como es el caso de `stdio.h`, por ejemplo.

3.8.2. Ejemplo

En el ejemplo de la ordenación por selección, tendríamos un fichero de cabecera para cada tipo de datos abstracto y, evidentemente, los correspondientes ficheros con el código en C. Además, dispondríamos de un tercer fichero que contendría el código del programa principal. La figura siguiente refleja el esquema de relaciones entre estos ficheros:



Cada fichero de código fuente puede compilarse independientemente. Es decir, en este ejemplo habría tres unidades de compilación. Cada una de ellas puede, pues, ser desarrollada independientemente de las demás. Lo único que deben respetar aquellas unidades que tienen un fichero de cabecera es no modificar ni los tipos de datos ni las declaraciones de las funciones. De esta manera, las demás unidades que hagan uso de ellas no deberán modificar sus llamadas y, por tanto, no requerirán ningún cambio ni compilación.

Es importante tener en cuenta que sólo puede existir una unidad con una función `main` (que es la que se corresponde con `ordena.c`, en el ejemplo dado). Las demás deberán ser compendios de funciones asociadas a un determinado tipo de datos abstracto.

El uso de ficheros de cabecera permite, además, cambiar el código de alguna función, sin tener que cambiar por ello nada de los programas que la emplean. Claro está, siempre que el cambio no afecte al “contrato” establecido en el fichero de cabecera correspondiente.



En el fichero de cabecera se describe toda la funcionalidad que se provee para un determinado tipo de datos abstracto y, con esta descripción, se adquiere el compromiso de que se mantendrá independientemente de cómo se materialice en el código asociado.

Para ilustrar esta idea se puede pensar en que, en el ejemplo, es posible cambiar el código de las funciones que trabajan con las listas para que éstas sean de tipo dinámico sin cambiar el contrato adquirido en el fichero de cabecera.

A continuación se lista el fichero de cabecera para las listas en el caso de la ordenación:

```
/* Fichero: lista.h
#ifndef _LISTA_VEC_H_
#define _LISTA_VEC_H_

#include <stdio.h>
#include "bool.h"
#include "elemento.h"

#define LONG_MAXIMA 100
typedef struct lista_e {
    elemento_t     nodo[ LONG_MAXIMA ];
    unsigned short posicion; /* Posición actual de acceso. */
    unsigned short cantidad; /* Longitud de la lista. */
} lista_t;

void inicializa_lista( lista_t *ref_lista );
bool esta_vacia_lista( lista_t lista );
bool es_final_de_lista( lista_t lista );
void principio_de_lista( lista_t *lista_ref );
ref_nodo_t ref_nodo_de_lista( lista_t lista );
elemento_t elemento_en_ref_nodo(
    lista_t     lista,
    ref_nodo_t refnodo
);
void avanza_posicion_en_lista( lista_t *lista_ref );
elemento_t extrae_minimo_de_lista( lista_t *ref_lista );
void pon_al_final_de_lista(
    lista_t *ref_lista,
    elemento_t elemento
);

#endif /* _LISTA_VEC_H_ */
```

Como se puede observar, se podría cambiar la forma de extracción del elemento mínimo sin necesidad de modificar el fichero de cabecera, y menos aún la llamada en el programa principal. Sin embargo, un cambio en el tipo de datos para, por ejemplo, implementar las listas mediante variables dinámicas, implicaría volver a compilar todas las unidades que hagan uso de las mismas, aunque no se modifiquen las cabeceras de las funciones. De hecho, en estos casos, resulta muy conveniente mantener el contrato al respecto de las mismas, pues evitará modificaciones en los códigos fuente de las unidades que hagan uso de ellas.

3.9. Bibliotecas

Las bibliotecas de funciones son, de hecho, unidades de compilación. Como tales, cada una dispone de un fichero de código fuente y uno de cabecera. Para evitar la compilación repetitiva de las bibliotecas, el código fuente ya ha sido compilado (ficheros de extensión ".o") y sólo quedan pendientes de su enlace con el programa principal.

Las bibliotecas de funciones, de todas maneras, se distinguen de las unidades de compilación por cuanto sólo se incorporan dentro del programa final aquellas funciones que son necesarias: las que no se utilizan, no se incluyen. Los ficheros compilados que permiten esta opción tienen la extensión ".l".

3.9.1. Creación

Para obtener una unidad de compilación de manera que sólo se incluyan en el programa ejecutable las funciones utilizadas, hay que compilarla para obtener un fichero de tipo objeto; es decir, con el código ejecutable sin enlazar:

```
$ gcc -c -o biblioteca.o biblioteca.c
```

Se supone, que en el fichero de biblioteca.c tal como se ha indicado, hay una inclusión del fichero de cabecera apropiado.

Una vez generado el fichero objeto, es necesario incluirlo en un archivo (con extensión ".a") de ficheros del mismo tipo (puede ser el único si la biblioteca consta de una sola unidad de compilación). En este contexto, los "archivos" son colecciones de ficheros objeto reunidos en un único fichero con un índice para localizarlos y, sobre todo, para determinar qué partes del código objeto corresponden a qué funciones. Para crear un archivo hay que ejecutar el siguiente comando:

```
$ ar biblioteca.a biblioteca.o
```

Para construir el índice (la tabla de símbolos y localizaciones) debe de ejecutarse el comando:

```
$ ar -s biblioteca.a
```

o bien:

```
$ ranlib biblioteca.a
```

El comando de gestión de archivos `ar` permite, además, listar los ficheros objeto que contiene, añadir o reemplazar otros nuevos o modificados, actualizarlos (se lleva a cabo el reemplazo si la fecha de modificación es posterior a la fecha de inclusión en el archivo) y eliminarlos si ya no son necesarios. Esto se hace, respectivamente, con los comandos siguientes:

```
$ ar -t biblioteca.a  
$ ar -r nuevo.o biblioteca.a  
$ ar -u actualizable.o biblioteca.a  
$ ar -d obsoleto.o biblioteca.a
```

Con la información de la tabla de símbolos, el enlazador monta un programa ejecutable empleando sólo aquellas funciones a las que se refiere. Para lo demás, los archivos son similares a los ficheros de código objeto simples.

3.9.2. Uso

El empleo de las funciones de una biblioteca es exactamente igual que el de las funciones de cualquier otra unidad de compilación.

Basta con incluir el fichero de cabecera apropiado e incorporar las llamadas a las funciones que se requieran dentro del código fuente.

3.9.3. Ejemplo

En el ejemplo de la ordenación se ha preparado una unidad de compilación para las listas. Como las listas son un tipo de datos dinámico que se utiliza mucho, resulta conveniente disponer de una biblioteca de funciones para operar con ellas. De esta manera, no hay que programarlas de nuevo en ocasiones posteriores.

Para transformar la unidad de listas en una biblioteca de funciones de listas hay que hacer que el tipo de datos no dependa en absoluto de la aplicación. En caso contrario, habría que compilar la unidad de listas para cada nuevo programa.

En el ejemplo, las listas contenían elementos del tipo `elemento_t`, que era un apuntador a `dato_t`. En general, las listas podrán tener elementos que sean apuntadores a cualquier tipo de datos. Sea como sea, todo son direcciones de memoria. Por este motivo, en lo que atañe a las listas, los elementos serán de un tipo vacío, que el usuario de la biblioteca de funciones habrá de definir. Así pues, en la unidad de compilación de las listas se incluye:

```
typedef void *elemento_t;
```

Por otra parte, para realizar la función de extracción del elemento más pequeño, ha de saber a qué función llamar para realizar la comparación. Por tanto, se le añade un parámetro más que consiste en la dirección de la función de comparación:

```
elemento_t extrae_minimo_de_lista(
    lista_t *ref_lista,
    bool (*compara_elementos)( elemento_t, elemento_t )
); /* extrae_minimo_de_lista */
```

El segundo argumento es un apuntador a una función que toma como parámetros dos elementos (no es necesario dar un nombre a

los parámetros formales) y devuelve un valor lógico de tipo `bool`. En el código fuente de la definición de la función, la llamada se efectuaría de la siguiente manera:

```
/* ... */  
es_menor = (*compara_elementos) (  
    elemento_en_ref_nodo( lista, actual ),  
    peque  
) ; /* compara_elementos */  
/* ... */
```

Por lo demás, no habría de hacerse cambio alguno.

Así pues, la decisión de transformar alguna unidad de compilación de un programa en biblioteca dependerá fundamentalmente de dos factores:

- El tipo de datos y las operaciones han de poder ser empleados en otros programas.
- Raramente se deben emplear todas las funciones de la unidad.

3.10. Herramienta make

La compilación de un programa supone, normalmente, compilar algunas de sus unidades y luego enlazarlas todas junto con las funciones de biblioteca que se utilicen para montar el programa ejecutable final. Por lo tanto, para obtenerlo, no sólo hay que llevar a cabo una serie de comandos, sino que también hay que tener en cuenta qué ficheros han sido modificados.

Las herramientas de tipo *make* permiten establecer las relaciones entre ficheros de manera que sea posible determinar cuáles dependen de otros. Así, cuando detecta que alguno de los ficheros tiene una fecha y hora de modificación anterior a alguno de los que depende, se ejecuta el comando indicado para generarlos de nuevo.

De esta manera, no es necesario preocuparse de qué ficheros hay que generar y cuáles no hace falta actualizar. Por otra parte, evita tener que ejecutar individualmente una serie de comandos que, para programas grandes, puede ser considerable.



El propósito de las herramientas de tipo *make* es determinar automáticamente qué piezas de un programa deben ser recompiladas y ejecutar los comandos pertinentes.

La herramienta *gmake* (o, simplemente, *make*) es una utilidad *make* de GNU (www.gnu.org/software/make) que se ocupa de lo anteriormente mencionado. Para poder hacerlo, necesita un fichero que, por omisión, se denomina *makefile*. Es posible indicarle un fichero con otro nombre si se la invoca con la opción *-f*:

```
$ make -f fichero_objetivos
```

3.10.1. Fichero *makefile*

En el fichero *makefile* hay que especificar los objetivos (habitualmente, ficheros a construir) y los ficheros de los que dependen (los requisitos para cumplir los objetivos). Para cada objetivo es necesario construir una regla, cuya estructura es la siguiente:

```
# Sintaxis de una regla:  
objetivo : fichero1 fichero2 ... ficheroN  
comando1  
comando2  
...  
comandoK
```

Nota

La primera de estas líneas debe empezar forzosamente por un carácter de tabulación.

El símbolo # se emplea para introducir una línea de comentarios. Toda regla requiere que se indique cuál es el objetivo y, después de los dos puntos, se indiquen los ficheros de los cuales depende. En las líneas siguientes se indicarán los comandos que deben de ejecutarse. Cualquier línea que quiera continuarse deberá finalizar con un carácter de tabulación.

zar con un salto de línea precedido del carácter de “escape” o barra inversa (\).

Tomando como ejemplo el programa de ordenación de notas de un examen por DNI, podría construirse un *makefile* como el mostrado a continuación para simplificar la actualización del ejecutable final:

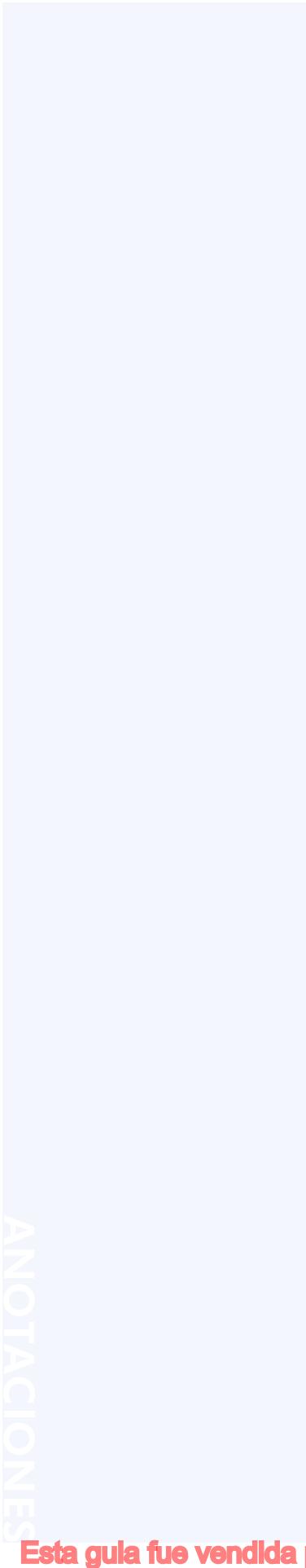
```
# Compilación del programa de ordenación:  
clasifica : ordena.o nota.o lista.a  
    gcc -g -o clasifica ordena.o nota.o lista.a  
ordena.o : ordena.c  
    gcc -g -c -o ordena.o ordena.c  
nota.o : nota.c nota.h  
    gcc -g -c -o nota.o nota.c  
lista.a : lista.o  
    ar -r lista.a lista.o ;  
    ranlib lista.a  
lista.o : lista.c lista.h  
    gcc -g -c -o lista.o lista.c
```

La herramienta *make* procesaría el fichero anterior revisando los prerequisitos del primer objetivo, y si éstos son a su vez objetivos de otras reglas, se procederá de la misma manera para estos últimos. Cualquier prerequisito terminal (que no sea un objetivo de alguna otra regla) modificado con posterioridad al objetivo al que afecta provoca la ejecución en serie de los comandos especificados en las siguientes líneas de la regla.

Es posible especificar más de un objetivo, pero *make* sólo examinará las reglas del primer objetivo final que encuentre. Si se desea que procese otros objetivos, será necesario indicarlo así en su invocación.

Es posible tener objetivos sin prerequisitos, en cuyo caso, siempre se ejecutarán los comandos asociados a la regla en la que aparecen.

Es posible tener un objetivo para borrar todos los ficheros objeto que ya no son necesarios.

**Ejemplo**

Retomando el ejemplo anterior, sería posible limpiar el directorio de ficheros innecesarios añadiendo el siguiente texto al final del *makefile*:

```
# Limpieza del directorio de trabajo:  
limpia :  
    rm -f ordena.o nota.o lista.o
```

Para conseguir este objetivo, bastaría con introducir el comando:

```
$ make limpia
```

Es posible indicar varios objetivos con unos mismos prerequisitos:

```
# Compilación del programa de ordenación:  
todo depura óptimo : clasifica  
depura : CFLAGS := -g  
óptimo : CFLAGS := -O  
clasifica : ordena.o nota.o lista.a  
    gcc $(CFLAGS) -o clasifica ordena.o nota.o lista.a  
# resto del fichero ...
```

Nota

A la luz del fragmento anterior, podemos ver lo siguiente:

- Si se invoca `make` sin argumento, se actualizará el objetivo `todo` (el primero que encuentra), que depende de la actualización del objetivo secundario `clasifica`.
- Si se especifica el objetivo `depura` o `óptimo`, habrá de comprobar la consistencia de dos reglas: en la primera se indica que para conseguirlos hay que actualizar `clasifica` y, en la segunda, que hay que asignar a la variable `CFLAGS` un valor.

Dado que *make* primero analiza las dependencias y luego ejecuta los comandos oportunos, el resultado es que el posible montaje de *compila* se hará con el contenido asignado a la variable `CCFLAGS` en la regla precedente: el acceso al contenido de una variable se realiza mediante el operador correspondiente, que se representa por el símbolo del dólar.

En el ejemplo anterior ya puede apreciarse que la utilidad de *make* va mucho más allá de lo que aquí se ha contado y, de igual forma, los *makefile* tienen muchas maneras de expresar reglas de forma más potente. Aun así, se han repasado sus principales características desde el punto de vista de la programación y visto algunos ejemplos significativos.

3.11. Relación con el sistema operativo. Paso de parámetros a programas

Los programas se traducen a instrucciones del lenguaje máquina para ser ejecutados. Sin embargo, muchas operaciones relacionadas con los dispositivos de entrada, de salida y de almacenamiento (discos y memoria, entre otros) son traducidas a llamadas a funciones del sistema operativo.

Una de las funciones del sistema operativo es, precisamente, permitir la ejecución de otros programas en la máquina y, en definitiva, del software. Para ello, provee al usuario de ciertos mecanismos para que elija qué aplicaciones desea ejecutar. Actualmente, la mayoría son, de hecho, interfaces de usuario gráficos. Aun así, siguen existiendo los entornos textuales de los intérpretes de comandos, denominados comúnmente *shells*.

En estos *shells*, para ejecutar programas (algunos de ellos, utilidades del propio sistema y otros, de aplicaciones) basta con suministrarles el nombre del fichero de código ejecutable correspondiente. Ciertamente, también existe la posibilidad de que un programa sea invocado por otro.

De hecho, la función `main` de los programas en C puede tener distintos argumentos. En particular, existe la convención de que el pri-

mer parámetro sea el número de argumentos que se le pasan a través del segundo, que consiste en un vector de cadenas de caracteres. Para aclarar cómo funciona este procedimiento de paso de parámetros, sirva como ejemplo el siguiente programa, que nos descubrirá qué sucede cuando lo invocamos desde un *shell* con un cierto número de argumentos:

```
/* Fichero: args.c */
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int i;
    printf( "Núm. argumentos, argc = %i\n", argc );
    for( i = 0; i < argc; i = i + 1 ) {
        printf( "Argumento argv[%i] = \"%s\"\n", i, argv[i] );
    } /* for */
    return argc;
```

Si se hace la invocación con el comando siguiente, el resultado será el que se indica a continuación:

```
$ args -prueba número 1
Núm. argumentos, argc = 4
Argumento argv[0] = "args"
Argumento argv[1] = "-prueba"
Argumento argv[2] = "número"
Argumento argv[3] = "1"
$
```

Es importante tener presente que la propia invocación de programa se toma como argumento 0 del comando. Así pues, en el ejemplo anterior, la invocación en la que se dan al programa tres parámetros se convierte, finalmente, en una llamada a la función `main` en la que se adjuntará también el texto con el que ha sido invocado el propio programa como primera cadena de caracteres del vector de argumentos.

Puede consultarse el valor returned por `main` para determinar si ha habido algún error durante la ejecución o no. Generalmente, se

toma por convenio que el valor devuelto debe ser el código del error correspondiente o 0 en ausencia de errores.

Nota

En el código fuente de la función es posible emplear las constantes `EXIT_FAILURE` y `EXIT_SUCCESS`, que están definidas en `stdlib.h`, para indicar el retorno con o sin error, respectivamente.

En el siguiente ejemplo, se muestra un programa que efectúa la suma de todos los parámetros presentes en la invocación. Para ello, emplea la función `atof`, declarada en `stdlib.h`, que convierte cadenas de texto a números reales. En caso de que la cadena no representara un número, devuelve un cero:

```
/* Fichero: suma.c */
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
    float    suma = 0.0;
    int      pos = 1;

    while( pos < argc ) {
        suma = suma + atof( argv[ pos ] );
        pos = pos + 1;
    } /* while */
    printf( " = %g\n", suma );
    return 0;
} /* main */
```

Nota

En este caso, el programa devuelve siempre el valor 0, puesto que no hay errores de ejecución que indicar.

3.12. Ejecución de funciones del sistema operativo

Un sistema operativo es un software que permite a las aplicaciones que se ejecutan en un ordenador abstraerse de los detalles de la máquina. Es decir, las aplicaciones pueden trabajar con una máquina virtual que es capaz de realizar operaciones que la máquina real no puede entender.

Además, el hecho de que los programas se definan en términos de las rutinas (o funciones) proveídas por el SO aumenta su portabilidad, su capacidad de ser ejecutados en máquinas distintas. En definitiva, los hace independientes de la máquina, pero no del SO, obviamente.

En C, muchas de las funciones de la biblioteca estándar emplean las rutinas del SO para llevar a cabo sus tareas. Entre estas funciones se encuentran las de entrada y salida de datos, de ficheros y de gestión de memoria (variables dinámicas, sobre todo).

Generalmente, todas las funciones de la biblioteca estándar de C tienen la misma cabecera y el mismo comportamiento, incluso con independencia del sistema operativo; no obstante, hay algunas que dependen del sistema operativo: puede haber algunas diferencias entre Linux y Microsoft. Afortunadamente, son fácilmente detectables, pues las funciones relacionadas a un determinado sistema operativo están declaradas en ficheros de cabecera específicos.

En todo caso, a veces resulta conveniente ejecutar los comandos del *shell* del sistema operativo en lugar de ejecutar directamente las funciones para llevarlos a término. Esto permite, entre otras cosas, que el programa correspondiente pueda describirse a un nivel de abstracción más alto y, con ello, aprovechar que sea el mismo intérprete de comandos el que complete los detalles necesarios para llevar a término la tarea encomendada. Generalmente, se trata de ejecutar órdenes internas del propio *shell* o aprovechar sus recursos (caminos de búsqueda y variables de entorno, entre otros) para ejecutar otros programas.

Para poder ejecutar un comando del *shell*, basta con suministrar a la función `system` la cadena de caracteres que lo describa. El valor que devuelve es el código de retorno del comando ejecutado o -1 en caso de error.

En Linux, `system(comando)` ejecuta `/bin/sh -c comando`; es decir, emplea `sh` como intérprete de comandos. Por lo tanto, éstos tienen que ajustarse a la sintaxis del mismo.

En el programa siguiente muestra el código devuelto por la ejecución de un comando, que hay que introducir entre comillas como argumento del programa:

```
/* Fichero: ejecuta.c */
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
    int codigo;

    if( argc == 2 ) {
        codigo = system( argv[1] );
        printf( "%i = %s\n", codigo, argv[1] );
    } else {
        printf( "Uso: ejecuta \"comando\"\n" );
    } /* if */
    return 0;
} /* main */
```

Aunque simple, el programa anterior nos da una cierta idea de cómo emplear la función `system`.

El conjunto de rutinas y servicios que ofrece el sistema operativo va más allá de dar soporte a las funciones de entrada y salida de datos, de manejo de ficheros y de gestión de memoria, también permite lanzar la ejecución de programas dentro de otros. En el apartado siguiente, se tratará con más profundidad el tema de la ejecución de programas.

3.13. Gestión de procesos

Los sistemas operativos actuales son capaces, además de todo lo visto, de ejecutar una diversidad de programas en la misma máquina en un mismo período de tiempo. Evidentemente, esto sólo es posible si se ejecutan en procesadores distintos o si su ejecución se realiza de forma secuencial o intercalada en un mismo procesador.

Normalmente, a pesar de que el sistema operativo sea capaz de gestionar una máquina con varios procesadores, habrá más programas a ejecutar que recursos para ello. Por este motivo, siempre tendrá que poder planificar la ejecución de un determinado conjunto de programas en un único procesador. La planificación puede ser:

- Secuencial. Una vez finalizada la ejecución de un programa, se inicia la del programa siguiente (también se conoce como ejecución por *lotes*).
- Intercalada. Cada programa dispone de un cierto tiempo en el que se lleva a cabo una parte de su flujo de ejecución de instrucciones y, al término del periodo de tiempo asignado, se ejecuta una parte de otro programa.

De esta manera, se pueden ejecutar diversos programas en un mismo intervalo de tiempo dando la sensación de que su ejecución progresiva paralelamente.

Apoyándose en los servicios (funciones) que ofrece el SO al respecto de la ejecución del software, es posible, entre otras cosas, ejecutarlo disociado de la entrada/salida estándar y/o partir su flujo de ejecución de instrucciones en varios paralelos.

3.13.1. Definición de proceso

Respecto del sistema operativo, cada flujo de instrucciones que debe gestionar es un proceso. Por lo tanto, repartirá su ejecución entre los diversos procesadores de la máquina y en el tiempo para que lleven a cabo sus tareas progresivamente. Habrá, eso sí, algunos procesos que se dividirán en dos paralelos, es decir, el programa consistirá, a partir de ese momento, en dos procesos distintos.

Cada proceso tiene asociado, al iniciarse, una entrada y salida estándar de datos de la que puede disociarse para continuar la ejecución en segundo plano, algo habitual en los procesos permanentes, que tratamos en el apartado siguiente.

Los procesos que comparten un mismo entorno o estado, con excepción evidente de la referencia a la instrucción siguiente, son denominados *hilos* o *hebras* (en inglés, *threads*), mientras que los que tienen entornos distintos son denominados simplemente *procesos*.

Con todo, un programa puede organizar su código de manera que lleve a término su tarea mediante varios flujos de instrucciones paralelos, sean éstos simples hilos o procesos completos.

3.13.2. Procesos permanentes

Un proceso permanente es aquel que se ejecuta indefinidamente en una máquina. Suelen ser procesos que se ocupan de la gestión automatizada de entradas y salidas de datos y, por lo tanto, con escasa interacción con los usuarios.

Así pues, muchas de las aplicaciones que funcionan con el modelo cliente-servidor se construyen con procesos permanentes para el servidor y con procesos interactivos para los clientes. Un ejemplo claro de tales aplicaciones son las relacionadas con Internet: los clientes son programas, como el gestor de correo electrónico o el navegador, y los servidores son programas que atienden a las peticiones de los clientes correspondientes.

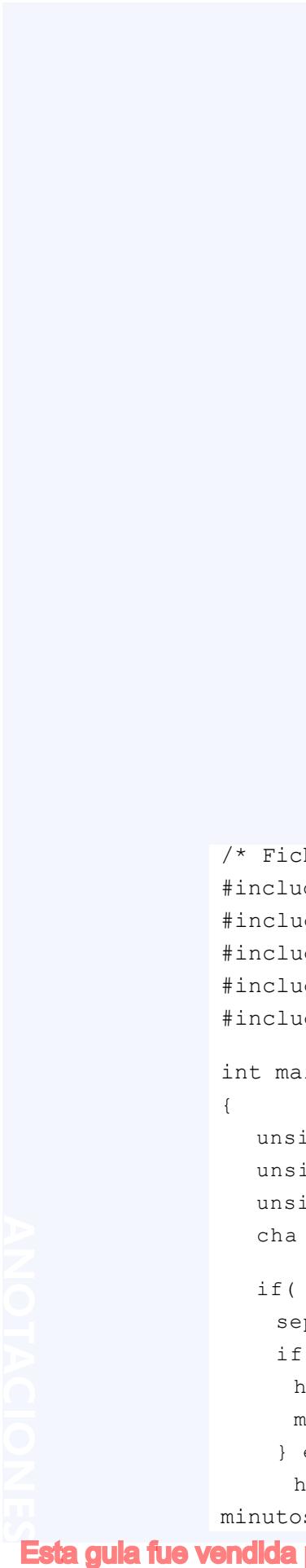
En Linux, a los procesos permanentes se les llama, gráficamente, demonios (*daemons*, en inglés) porque aunque los usuarios no pueden observarlos puesto que no interactúan con ellos (en especial, no lo hacen a través del terminal estándar), existen: los demonios son “espíritus” de la máquina que el usuario no ve pero cuyos efectos percibe.

Para crear un demonio, basta con llamar a la función `daemon`, declarada en `unistd.h`, con los parámetros adecuados. El primer argumento indica si no cambia de directorio de trabajo y el segundo, si no se disocia del terminal estándar de entrada/salida; es decir, una llamada común habría de ser como sigue:

```
/* ... */
if( daemon( FALSE, FALSE ) ) == 0 ) {
    /* cuerpo */
} /* if */
/* resto del programa, tanto si se ha creado como si no. */
```

Nota

Literalmente, un demonio es un espíritu maligno, aunque se supone que los procesos denominados como tales no deberían serlo.

**Nota**

Esta llamada consigue que el cuerpo del programa sea un demonio que trabaja en el directorio raíz (como si hubiera hecho un `cd /`) y que está disociado de las entradas y salidas estándar (en realidad, redirigidas al dispositivo vacío: `/dev/null`). La función devuelve un código de error, que es cero si todo ha ido bien.

Ejemplo

Para ilustrar el funcionamiento de los demonios, se muestra un programa que avisa al usuario de que ha transcurrido un cierto tiempo. Para ello, habrá que invocar al programa con dos parámetros: uno para indicar las horas y minutos que deben transcurrir antes de advertir al usuario y otro que contenga el texto del aviso. En este caso, el programa se convertirá en un demonio no disociado del terminal de entrada/salida estándar, puesto que el aviso aparecerá en el mismo.

```
/* Fichero: alarma.c */  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include "bool.h"  
  
int main( int argc, char *argv[] )  
{  
    unsigned int horas;  
    unsigned int minutos;  
    unsigned int segundos;  
    char         *aviso, *separador;  
  
    if( argc == 3 ) {  
        separador = strchr( argv[1], ':' );  
        if( separador != NULL ) {  
            horas = atoi( argv[1] );  
            minutos = atoi( separador+1 );  
        } else {  
            horas = 0;  
            minutos = atoi( argv[1] );  
        }  
    } else {  
        horas = 0;  
        minutos = 0;  
    }  
    segundos = (horas*3600) + (minutos*60);  
    sleep( segundos );  
    write( 1, aviso, strlen(aviso) );  
}
```

```
    } /* if */
    segundos = (horas*60 + minutos) * 60;
    aviso = argv[2];
    if( daemon( FALSE, TRUE ) ) {
        printf( "No puede instalarse el avisador :-(\n" );
    } else {
        printf( "Alarma dentro de %i horas y %i minutos.\n",
            horas, minutos
        ); /* printf */
        printf( "Haz $ kill %li para apagarla.\n",
            getpid()
        ); /* printf */
    } /* if */
    sleep( segundos );
    printf( "%s\007\n", aviso );
    printf( "Alarma apagada.\n" );
} else {
    printf( "Uso: %s horas:minutos \"aviso\"\n", argv[0] );
} /* if */
return 0;
} /* main */
```

La lectura de los parámetros de entrada ocupa buena parte del código. En particular, lo que necesita más atención es la extracción de las horas y de los minutos; para ello, se buscan los dos puntos (con `strchr`, declarada en `string.h`) y luego se toma la cadena entera para determinar el valor de las horas y la cadena a partir de los dos puntos para los minutos.

La espera se realiza mediante una llamada a la función `sleep`, que toma como argumento el número de segundos en los que el programa debe “dormir”, es decir, suspender su ejecución.

Finalmente, para dar al usuario la posibilidad de parar la alarma, se le informa del comando que debe introducir en el *shell* para “matar” el proceso (es decir, para finalizar su ejecución). A tal efecto, se le muestra el número de proceso que se corresponde con el demonio instalado. Este identificador se consigue llamando a la función `getpid()`, en el que PID significa, precisamente, identificador de proceso



Uno de los usos fundamentales de los demonios es el de la implementación de procesos proveedores de servicios.

3.13.3. Procesos concurrentes

Los procesos concurrentes son aquellos que se ejecutan simultáneamente en un mismo sistema. Al decir *simultáneamente*, en este contexto, entendemos que se llevan a cabo en un mismo período de tiempo bien en procesadores distintos, bien repartidos temporalmente en un mismo procesador, o bien en los dos casos anteriores.

El hecho de repartir la ejecución de un programa en diversos flujos de instrucciones concurrentes puede perseguir alguno de los objetivos siguientes:

- Aprovechar los recursos en un sistema multiprocesador. Al ejecutarse cada flujo de instrucciones en un procesador distinto, se consigue una mayor rapidez de ejecución. De hecho, sólo en este caso se trata de procesos de ejecución verdaderamente simultánea.

Nota

Cuando dos o más procesos comparten un mismo procesador, no hay más remedio que ejecutarlos por tramos en un determinado período de tiempo dentro del que, efectivamente, se puede observar una evolución progresiva de los mismos.

- Aumentar el rendimiento respecto de la entrada/salida de datos. Para aumentar el rendimiento respecto de la E/S del programa puede resultar conveniente que uno de los procesos se ocupe de la relación con la entrada de datos, otro del cálculo que haya que hacer con ellos y, finalmente, otro de la salida de resultados. De esta manera, es posible realizar el cálculo sin detenerse a dar salida a los resultados o esperar datos de entrada. Ciertamente, no siempre cabe hacer tal partición y el número de procesos puede variar mucho en función de las necesidades del

programa. De hecho, en este caso se trata, fundamentalmente, de separar procesos de naturaleza lenta (por ejemplo, los que deben comunicarse con otros bien para recibir bien para transmitir datos) de otros procesos más rápidos, es decir, con mayor atención al cálculo.

En los siguientes apartados se comentan varios casos de programación concurrente tanto con “procesos ligeros” (los hilos) como con procesos completos o “pesados” (por contraposición a los ligeros).

3.14. Hilos

Un hilo, hebra o *thread* es un proceso que comparte el entorno con otros del mismo programa, lo que comporta que el espacio de memoria sea el mismo. Por tanto, la creación de un nuevo hilo sólo implica disponer de información sobre el estado del procesador y la instrucción siguiente para el mismo. Precisamente por este motivo son denominados “procesos ligeros”.



Los hilos son flujos de ejecución de instrucciones independientes que tienen mucha relación entre sí.

Para emplearlos en C sobre Linux, es necesario hacer llamadas a funciones de hilos del estándar de POSIX. Este estándar define una interfaz portable de sistemas operativos (originalmente, Unix) para entornos de computación, de cuya expresión en inglés toma el acrónimo.

Las funciones de POSIX para hilos están declaradas en el fichero `pthread.h` y se debe de enlazar el archivo de la biblioteca correspondiente con el programa. Para ello, hay que compilar con el comando siguiente:

```
$ gcc -o ejecutable codigo.c -lpthread
```

Nota

La opción `-lpthread` indica al enlazador que debe incluir también la biblioteca de funciones POSIX para hilos.

3.14.1. Ejemplo

Mostraremos un programa para determinar si un número es primo o no como ejemplo de un programa desenhebrado en dos hilos. El hilo principal se ocupará de buscar posibles divisores mientras que el secundario actuará de "observador" para el usuario: leerá los datos que maneja el hilo principal para mostrarlos por el terminal de salida estándar. Evidentemente, esto es posible porque comparten el mismo espacio de memoria.

La creación de los hilos requiere que su código esté dentro de una función que sólo admite un parámetro del tipo `(void *)`. De hecho las funciones creadas para ser hilos POSIX deben obedecer a la cabecera siguiente:

```
(void *)hilo( void *referencia_parámetros );
```

Así pues, será necesario colocar en una tupla toda la información que se quiera hacer visible al usuario y pasar su dirección como parámetro de la misma. Pasamos a definir la tupla de elementos que se mostrará:

```
/* ... */
typedef struct s_visible {
    unsigned long numero;
    unsigned long divisor;
    bool fin;
} t_visible;
/* ... */
```

Nota

El campo `fin` servirá para indicar al hilo hijo que el hilo principal (el padre) ha acabado su tarea. En este caso, ha determinado si el número es o no, primo.

La función del hilo hijo será la siguiente:

Notg

El carácter '\b' se corresponde con un retroceso y que, dado que los números se imprimen con 12 dígitos (los ceros a la izquierda se muestran como espacios), la impresión de 12 retrocesos implica borrar el número que se haya escrito anteriormente.

Para crear el hilo del observador, basta con llamar a `pthread_create()` con los argumentos adecuados. A partir de ese momento, un nuevo hilo se ejecuta concurrentemente con el código del programa:

```
/* ... */

int main( int argc, char *argv[] )
{
    int         codigo_error; /* Código de error a devolver. */
    pthread_t   id_hilo;     /* Identificador del hilo. */
    t_visible  vista;        /* Datos observables. */
    bool        resultado;   /* Indicador de si es primo. */
```

```

{
    int         codigo_error; /* Código de error a devolver. */
    pthread_t   id_hilo;     /* Identificador del hilo. */
    t_visible   vista;       /* Datos observables. */
    bool        resultado;   /* Indicador de si es primo. */

    if( argc == 2 ) {
        vista.numero = atol( argv[1] );
        vista.fin = FALSE;
        codigo_error = pthread_create(
            &id_hilo,           /* Referencia en la que poner el ID. */
            NULL,               /* Referencia a posibles atributos. */
            observador,         /* Función que ejecutará el hilo. */
            (void *)&vista /* Argumento de la función. */
        ); /* pthread_create */
        if( codigo_error == 0 ) {
            resultado = es_primo( &vista );
            vista.fin = TRUE;
            pthread_join( id_hilo, NULL );
            if( resultado )printf( "Es primo.\n" );
            else             printf( "No es primo.\n" );
            codigo_error = 0;
        } else {
            printf( "No he podido crear un hilo observador!\n" );
            codigo_error = 1;
        } /* if */
    } else {
        printf( "Uso: %s número\n", argv[0] );
        codigo_error = -1;
    } /* if */
    return codigo_error;
} /* main */

```

Nota

Después de crear el hilo del observador, se comprueba que el número sea primo y, al regresar de la función `es_primo()`, se pone el campo `fin` a `TRUE` para que el observador finalice su ejecución. Para esperar a que efectivamente haya acabado, se llama a `pthread_join()`. Esta función espera a que el hilo cuyo identificador se haya dado como primer argumento llegue al final de su ejecución y, por tanto, se produzca una unión de los hilos (de ahí el apelativo en inglés *join*). El segundo argumento se emplea para recoger posibles datos devueltos por el hilo.

Para terminar el ejemplo, sería necesario codificar la función `es_primo()`, que tendría como cabecera la siguiente:

```
/* ... */  
bool es_primo( t_visible *ref_datos );  
/* ... */
```

La programación se deja como ejercicio. Para resolverlo adecuadamente, cabe tener presente que hay que emplear `ref_datos->divisor` como tal, puesto que la función `observador()` la lee para mostrarla al usuario.

En este caso, no existe ningún problema en que los hilos de un programa tengan el mismo espacio de memoria; es decir, el mismo entorno o contexto. De todas maneras, suele ser habitual que el acceso a datos compartidos por más de un hilo sea sincronizado. En otras palabras, que se habilite algún mecanismo para impedir que dos hilos accedan simultáneamente al mismo dato, especialmente para modificarlo, aunque también para que los hilos lectores lean los datos debidamente actualizados. Estos mecanismos de exclusión mutua son, de hecho, convenios de llamadas a funciones previas y posteriores al acceso a los datos.

Nota

Se puede imaginar como funciones de control de un semáforo de acceso a una plaza de aparcamiento: si está libre, el semáforo estará en verde y, si está ocupada, estará en rojo hasta que se libere.

3.15. Procesos

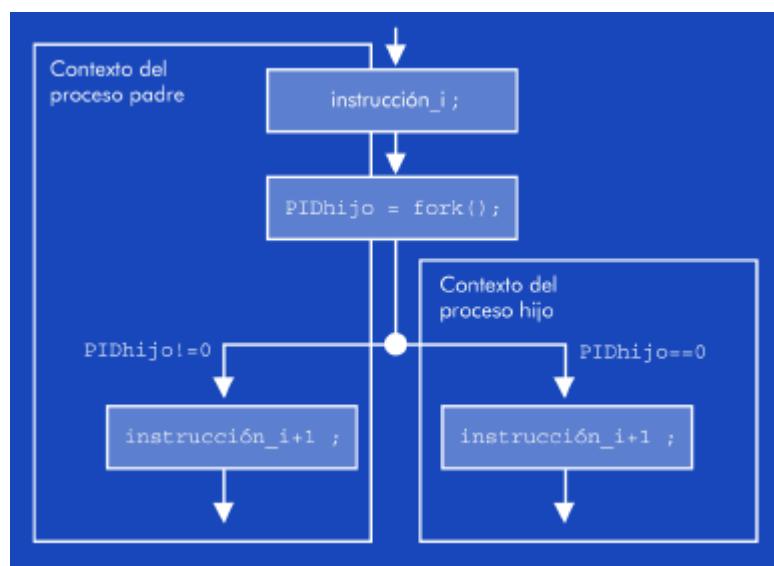
Un proceso es un flujo de ejecución de instrucciones con un entorno propio y, por tanto, con todas las atribuciones de un programa. Puede dividir, pues, el flujo de ejecución de instrucciones en otros procesos (ligeros o no) si así se considera conveniente por razones de eficiencia.

En este caso, la generación de un nuevo proceso a partir del proceso principal implica realizar la copia de todo el entorno de este último. Con ello, el proceso hijo tiene una copia exacta del entorno del padre en el momento de la división. A partir de ahí, tanto los contenidos de las variables como la indicación de la instrucción siguiente puede divergir. De hecho, actúan como dos procesos distintos con entornos evidentemente diferentes del mismo código ejecutable.

Dada la separación estricta de los entornos de los procesos, generalmente éstos se dividen cuando hay que realizar una misma tarea sobre datos distintos, que lleva a cabo cada uno de los procesos hijos de forma autónoma. Por otra parte, existen también mecanismos para comunicar procesos entre sí: las tuberías, las colas de mensajes, las variables compartidas (en este caso, se cuenta con funciones para implementar la exclusión mutua) y cualquier otro tipo de comunicación que pueda establecerse entre procesos distintos.

Para crear un nuevo proceso, basta con llamar a `fork()`, cuya declaración se encuentra en `unistd.h`, y que devuelve el identificador del proceso hijo en el padre y cero en el nuevo proceso:

Figura 11.



Nota

El hecho de que `fork()` devuelva valores distintos en el proceso padre y en el hijo permite a los flujos de instrucciones siguientes determinar si pertenecen a uno u otro.

El programa siguiente es un ejemplo simple de división de un proceso en dos: el original o padre y la copia o hijo. Para simplificar, se supone que tanto el hijo como el padre realizan una misma tarea. En este caso, el padre espera a que el hijo finalice la ejecución con `wait()`, que requiere de la inclusión de los ficheros `sys/types.h` y `sys/wait.h`:

```
/* Fichero: ej_fork.c */  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
  
/* ... */  
  
int main( void )  
  
{  
    pid_t proceso;  
    int estado;  
    printf( "Proceso padre (%li) iniciado.\n", getpid() );  
    proceso = fork();  
    if( proceso == 0 ) {  
        printf( "Proceso hijo (%li) iniciado.\n", getpid() );  
        tarea( "hijo" );  
        printf( "Fin del proceso hijo.\n" );  
    } else {  
        tarea( "padre" );  
        wait( &estado ); /* Espera la finalización del hijo. */  
        printf( "Fin del proceso padre.\n" );  
    } /* if */  
    return 0;  
} /* main */
```

Para poner de manifiesto que se trata de dos procesos que se ejecutan paralelamente, resulta conveniente que las tareas del padre y del hijo sean distintas o que se hagan con datos de entrada diferentes.

Para ilustrar tal caso, se muestra una posible programación de la función, tarea en la que se hace una repetición de esperas. Para poder observar que la ejecución de los dos procesos puede no estar intercalada siempre de la misma manera, tanto el número de repeticiones como el tiempo de las esperas se pone en función de números aleatorios provistos por la función `random()`, que arranca con un valor “semilla” calculado mediante `srandom()` con un argumento que varíe con las distintas ejecuciones y con el tipo de proceso (padre o hijo):

```
/* ... */
void tarea( char *nombre )
{
    unsigned int contador;

    srandom( getpid() % ( nombre[0] * nombre[2] ) );
    contador = random() % 11 + 1;
    while( contador > 0 ) {
        printf( "... paso %i del %s\n", contador, nombre );
        sleep( random() % 7 + 1 );
        contador = contador - 1;
    } /* while */
} /* tarea */
/* ... */
```

En el ejemplo anterior, el padre espera a un único hijo y realiza una misma tarea. Esto, evidentemente, no es lo habitual. Es mucho más común que el proceso padre se ocupe de generar un proceso hijo para cada conjunto de datos a procesar. En este caso, el programa principal se complica ligeramente y hay que seleccionar en función del valor devuelto por `fork()` si las instrucciones pertenecen al padre o a uno de los hijos.

Para ilustrar la codificación de tales programas, se muestra un programa que toma como argumentos una cantidad indeterminada de números naturales para los que averigua si se trata o no, de números primos. En este caso, el programa principal creará un proceso hijo

```
/* ... */

int main( int argc, char *argv[] )
{
    int                 contador;
    unsigned long int   numero, divisor;
    pid_t               proceso;
    int                 estado;

    if( argc > 1 ) {
        proceso = getpid();
        printf( "Proceso %li iniciado.\n", proceso );
        contador = 1;
        while( proceso != 0 && contador < argc ) {
            /* Creación de procesos hijo: */
            numero = atol( argv[ contador ] );
            contador = contador + 1;
            proceso = fork();
            if( proceso == 0 ) {
                printf( "Proceso %li para %lu\n",
                    getpid(),
                    numero
                ); /* printf */
                divisor = es_primo( numero );
                if( divisor > 1 ) {
                    printf( "%lu no es primo.\n", numero );
                    printf( "Su primer divisor es %lu\n", divisor );
                } else {
                    printf( "%lu es primo.\n", numero );
                } /* if */
            } /* if */
        } /* while */
        while( proceso != 0 && contador > 0 ) {
            /* Espera de finalización de procesos hijo:*/
            wait( &estado );
            contador = contador - 1;
        } /* while */
        if( proceso !=0 ) printf( "Fin.\n");
    } else {
        printf( "Uso: %s natural_1 ... natural_N\n", argv[0] );
    } /* if */
    return 0;
} /* main */
```

Nota

El bucle de creación se interrumpe si `proceso==0` para evitar que los procesos hijo puedan crear “nietos” con los mismos datos que algunos de sus “hermanos”.

Hay que tener presente que el código del programa es el mismo tanto para el proceso padre, como para el hijo.

Por otra parte, el bucle final de espera sólo debe aplicarse al padre, es decir, al proceso en el que se cumpla que la variable `proceso` sea distinta de cero. En este caso, basta con descontar del contador de procesos generados una unidad para cada espera cumplida.

Para comprobar su funcionamiento, falta diseñar la función `es_primo()`, que queda como ejercicio. Para ver el funcionamiento de tal programa de manera ejemplar, es conveniente introducir algún número primo grande junto con otros menores o no primos.

3.15.1. Comunicación entre procesos

Tal como se ha comentado, los procesos (tanto si son de un mismo programa como de programas distintos) se pueden comunicar entre sí mediante mecanismos de tuberías, colas de mensajes y variables compartidas, entre otros. Por lo tanto, estos mecanismos también se pueden aplicar en la comunicación entre programas distintos de una misma aplicación o, incluso, de aplicaciones distintas. En todo caso, siempre se trata de una comunicación poco intensa y que requiere de exclusión mutua en el acceso a los datos para evitar conflictos (aun así, no siempre se evitan todos).

Como ejemplo, se mostrará un programa que descompone en suma de potencias de divisores primos cualquier número natural dado. Para ello, dispone de un proceso de cálculo de divisores primos y otro, el padre, que los muestra a medida que se van calculando. Cada factor de la suma es un dato del tipo:

```
typedef struct factor_s {
    unsigned long int divisor;
    unsigned long int potencia;
} factor_t;
```

La comunicación entre ambos procesos se realiza mediante una tubería.

Nota

Recordad que en la unidad anterior ya se definió *tubería*. Una tubería consiste, de hecho, en dos ficheros de flujo de bytes, uno de entrada y otro de salida, por el que se comunican dos procesos distintos.

Como se puede apreciar en el código siguiente, la función para abrir una tubería se llama `pipe()` y toma como argumento la dirección de un vector de dos enteros en donde depositará los descriptores de los ficheros de tipo `stream` que haya abierto: en la posición 0 el de salida, y en la 1, el de entrada. Después del `fork()`, ambos procesos tienen una copia de los descriptores y, por tanto, pueden acceder a los mismos ficheros tanto para entrada como para salida de datos. En este caso, el proceso hijo cerrará el fichero de entrada y el padre, el de salida; puesto que la tubería sólo comunicará los procesos en un único sentido: de hijo a padre. (Si la comunicación se realizara en ambos sentidos, sería necesario establecer un protocolo de acceso a los datos para evitar conflictos.):

```
/* ... */
int main( int argc, char *argv[] )
{
    unsigned long int    numero;
    pid_t                proceso;
    int                  estado;
    int                  desc_tuberia[2];

    if( argc == 2 ) {
        printf( "Divisores primos.\n" );
        numero = atol( argv[ 1 ] );
        if( pipe( desc_tuberia ) != -1 ) {
            proceso = fork();
            if( proceso == 0 ) { /* Proceso hijo: */                      */
                close( desc_tuberia[0] );
                divisores_de( numero, desc_tuberia[1] );
                close( desc_tuberia[1] );
            } else { /* Proceso principal o padre: */                     */
                /* Cierre el descriptor de escritura (desc_tuberia[0]) */
                /* y lee los resultados (desc_tuberia[1]) */
                /* ... */
            }
        }
    }
}
```

```

        close( desc_tuberia[1] );
        muestra_divisores( desc_tuberia[0] );
        wait( &estado );
        close( desc_tuberia[0] );
        printf( "Fin.\n" );
    } /* if */
} else {
    printf( "No puedo crear la tuberia!\n" );
} /* if */
} else {
    printf( "Uso: %s numero_natural\n", argv[0] );
} /* if */
return 0;
} /* main */

```

Con todo, el código de la función `muestra_divisores()` en el proceso padre podría ser como el que se muestra a continuación. En él, se emplea la función de lectura `read()`, que intenta leer un determinado número de bytes del fichero cuyo descriptor se le pasa como primer argumento. Devuelve el número de bytes efectivamente leídos y su contenido lo deposita a partir de la dirección de memoria indicada:

```

/* ... */
void muestra_divisores( int desc_entrada )
{
    size_t      nbytes;
    factor_t   factor;

    do {
        nbytes = read( desc_entrada,
                      (void *)&factor,
                      sizeof( factor_t ) );
    } /* read */
    if( nbytes > 0 ) {
        printf( "%lu ^ %lu\n",
                factor.divisor,
                factor.potencia );
    } /* printf */
    } while( nbytes > 0 );
} /* muestra_divisores */
/* ... */

```

Para completar el ejemplo, se muestra una posible programación de la función `divisores_de()` en el proceso hijo. Esta función emplea `write()` para depositar los factores recién calculados en el fichero de salida de la tubería:

```
/* ... */  
void divisores_de(  
    unsigned long    int numero,  
    int              desc_salida )  
{  
    factor_t f;  
  
    f.divisor = 2;  
    while( numero > 1 ) {  
        f.potencia = 0;  
        while( numero % f.divisor == 0 ) {  
            f.potencia = f.potencia + 1;  
            numero = numero / f.divisor;  
        } /* while */  
        if( f.potencia > 0 ) {  
            write( desc_salida, (void *)&f, sizeof( factor_t ) );  
        } /* if */  
        f.divisor = f.divisor + 1;  
    } /* while */  
} /* divisores_de */  
/* ... */
```

Con este ejemplo se ha mostrado una de las posibles formas de comunicación entre procesos. En general, cada mecanismo de comunicación tiene unos usos preferentes

Nota

Las tuberías son adecuadas para el paso de una cantidad relativamente alta de datos entre procesos, mientras que las colas de mensajes se adaptan mejor a procesos que se comunican poco frecuentemente o de forma irregular.

En todo caso, hay que tener presente que repartir las tareas de un programa en varios procesos supondrá un cierto incremento de la complejidad por la necesaria introducción de mecanismos de comunicación entre ellos. Así pues, es importante valorar los beneficios que tal división pueda aportar al desarrollo del programa correspondiente.

3.16. Resumen

Los algoritmos que se emplean para procesar la información pueden ser más o menos complejos según la representación que se escoja para la misma. Como consecuencia, la eficiencia de la programación está directamente relacionada con las estructuras de datos que se empleen en ésta.

Por este motivo se han introducido las estructuras dinámicas de datos, que permiten, entre otras cosas, aprovechar mejor la memoria y cambiar la relación entre ellos como parte del procesado de la información.

Las estructuras de datos dinámicas son, pues, aquéllas en las que el número de datos puede variar durante la ejecución del programa y cuyas relaciones, evidentemente, pueden cambiar. Para ello, se apoyan en la creación y destrucción de variables dinámicas y en los mecanismos para acceder a ellas. Fundamentalmente, el acceso a tales variables se debe hacer mediante apuntadores, puesto que las variables dinámicas no disponen de nombres con los que identificarlas.

Se ha visto también un ejemplo común de estructuras de datos dinámicas como las cadenas de caracteres y las listas de nodos. En particular, para este último caso se ha revisado no sólo la posible programación de las funciones de gestión de los nodos en una lista, sino también una forma especial de tratamiento de las mismas en la que se emplean como representaciones de colas.

Dado lo habitual del empleo de muchas de estas funciones para estructuras de datos dinámicas comunes, resulta conveniente agruparlas en archivos de ficheros objeto: las bibliotecas de funciones. De

esta manera, es posible emplear las mismas funciones en programas diversos sin preocuparse de su programación. Aun así, es necesario incluir los ficheros de cabecera para indicar al compilador la forma de invocar a tales funciones. Con todo, se repasa el mecanismo de creación de bibliotecas de funciones y, además, se introduce el uso de la utilidad *make* para la generación de ejecutables que resultan de la compilación de diversas unidades del mismo programa y de los archivos de biblioteca requeridos.

Por otra parte, también se ha visto cómo la relación entre los distintos tipos de datos abstractos de un programa facilitan la programación modular. De hecho, tales tipos se clasifican según niveles de abstracción o, según se mire, de dependencia de otros tipos de datos. Así pues, el nivel más bajo de abstracción lo ostentan los tipos de datos abstractos que se definen en términos de tipos de datos primitivos.

De esta manera, el programa principal será aquel que opere con los tipos de datos de mayor nivel de abstracción. El resto de módulos del programa serán los que provean al programa principal de las funciones necesarias para realizar tales operaciones.

Por lo tanto, el diseño descendente de algoritmos, basado en la jerarquía que se establece entre los distintos tipos de datos que emplean, es una técnica con la que se obtiene una programación modular eficiente.

En la práctica, cada tipo de datos abstracto deberá acompañarse de las funciones para operaciones elementales como creación, acceso a datos, copia, duplicado y destrucción de las variables dinámicas correspondientes. Más aun, deberá estar contenida en una unidad de compilación independiente, junto con el fichero de cabecera adecuado.

Finalmente, en el último capítulo, se ha insistido en la organización del código, no tanto con relación a la información que debe procesar, sino más en relación con la forma de hacerlo. En este sentido, resulta conveniente aprovechar al máximo las facilidades que nos ofrece el lenguaje de programación C para utilizar las rutinas de servicio del sistema operativo.

Cuando la información a tratar deba ser procesada por otro programa, es posible ejecutarlos desde el flujo de ejecución de instrucciones del que se está ejecutando. En este caso, sin embargo, la comunicación entre el programa llamado y el llamador es mínima. Como consecuencia, debe ser el mismo programa llamado el que obtenga la mayor parte de la información a tratar y el que genere el resultado.

Se ha tratado también de la posibilidad de dividir el flujo de ejecución de instrucciones en varios flujos diferentes que se ejecutan concurrentemente. De esta manera, es posible especializar cada flujo en un determinado aspecto del tratamiento de la información o, en otros casos, realizar el mismo tratamiento sobre partes distintas de la información.

Los flujos de ejecución de instrucciones se pueden dividir en hilos (*threads*) o procesos. A los primeros también se les denomina procesos ligeros, pues son procesos que comparten el mismo contexto (entorno) de ejecución. El tipo de tratamiento de la información será el que determine qué forma de división es la mejor. Como norma se puede tomar la del grado de compartimiento de la información: si es alto, entonces es mejor un hilo, y si es bajo, un proceso (entre ellos, no obstante, hay diversos mecanismos de comunicación según el grado particular de relación que tengan).

En todo caso, parte del contenido de esta unidad se verá de nuevo en las próximas pues, tanto C++ como Java, facilitan la programación con tipos de datos abstractos, el diseño modular y la distribución de la ejecución en diversos flujos de instrucciones.

3.17. Ejercicios de autoevaluación

- 1) Haced un buscador de palabras en ficheros, de forma similar al último ejercicio de la unidad anterior. El programa deberá pedir el nombre del fichero y la palabra a buscar. En este caso, la función principal deberá ser la siguiente:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;
typedef char *palabra_t;
```

```
palabra_t siguiente_palabra(
    char *frase,
    unsigned int inicio
) { /* ... */ }
int main( void )
{
    FILE             *entrada;
    char             nombre[BUFSIZ];
    palabra_t       palabra, palabra2;
    unsigned int    numlin, pos;

    printf( "Busca palabras.\n" );
    printf( "Fichero: " );
    gets( nombre );
    entrada = fopen( nombre, "rt" );
    if( entrada != NULL ) {
        printf( "Palabra: " );
        gets( nombre );
        palabra = siguiente_palabra( nombre, 0 );
        printf( "Buscando %s en fichero...%s\n", palabra );
        numlin = 1;
        while( fgets( nombre, BUFSIZ-1, entrada ) != NULL ) {
            numlin = numlin + 1;
            pos = 0;
            palabra2 = siguiente_palabra( nombre, pos );
            while( palabra2 != NULL ) {
                if( !strcmp( palabra, palabra2 ) ) {
                    printf( "... línea %lu\n", numlin );
                } /* if */
                pos = pos + strlen( palabra2 );
                free( palabra2 );
                palabra2 = siguiente_palabra( nombre, pos );
            } /* while */
        } /* while */
        free( palabra );
        fclose( entrada );
        printf( "Fin.\n" );
    } else {
        printf( ";No puedo abrir %s!\n", nombre );
    } /* if */
    return 0;
} /* main */
```

Se debe de programar, pues, la función siguiente `palabra()`.

- 2) Componed, a partir de las funciones provistas en el apartado 3.5.2, la función para eliminar el elemento enésimo de una lista de enteros. El programa principal deberá ser el siguiente:

```

int main( void )
{
    lista_t      lista;
    char         opcion;
    int          dato;
    unsigned int n;

    printf( "Gestor de listas de enteros.\n" );
    lista = NULL;
    do {
        printf(
            "[I]nsertar, [E]liminar, [M]ostrar o [S]alir? "
        ); /* printf */
        do opcion = getchar(); while( isspace(opcion) );
        opcion = toupper( opcion );
        switch( opcion ) {
            case 'I':
                printf( "Dato =? " );
                scanf( "%i", &dato );
                printf( "Posicion =? " );
                scanf( "%u", &n );
                if( !inserta_enesimo_lista( &lista, n, dato ) ) {
                    printf( "No se insertó.\n" );
                } /* if */
                break;
            case 'E':
                printf( "Posicion =? " );
                scanf( "%u", &n );
                if( elimina_enesimo_lista( &lista, n, &dato ) ) {
                    printf( "Dato = %i\n", dato );
                } else {
                    printf( "No se eliminó.\n" );
                } /* if */
                break;
            case 'M':
                muestra_lista( lista );
                break;
        } /* switch */
    } while( opcion != 'S' );
    while( lista != NULL ) {
        elimina_enesimo_lista( &lista, 0, &dato );
    } /* while */
    printf( "Fin.\n" );
    return 0;
} /* main */

```

También hay que programar la función `muestra_lista()` para poder ver su contenido.

- 3) Haced un programa que permita insertar y eliminar elementos de una cola de enteros. Las funciones que deben emplearse se encuentran en el apartado referente a colas del apartado 3.5.2. Por lo tanto, sólo cabe desarrollar la función principal de dicho programa, que puede inspirarse en la mostrada en el ejercicio anterior.
- 4) Programad el algoritmo de ordenación por selección visto en el apartado 3.6 para clasificar un fichero de texto en el que cada línea tenga el formato siguiente:

```
DNI nota '\n'
```

Por tanto, los elementos serán del mismo tipo de datos que el visto en el ejemplo. El programa principal será:

```
int main( void )
{
    FILE        *entrada;
    char        nombre[ BUFSIZ ];
    lista_t     pendientes, ordenados;
    ref_nodo_t  refnodo;
    elemento_t  elemento;

    printf( "Ordena listado de nombres.\n" );
    printf( "Fichero =? " ); gets( nombre );
    entrada = fopen( nombre, "rt" );
    if( entrada != NULL ) {
        inicializa_lista( &pendientes );
        while( fgets( nombre, BUFSIZ-1, entrada ) != NULL ) {
            elemento = lee_elemento( nombre );
            pon_al_final_de_lista( &pendientes, elemento );
        } /* if */
        inicializa_lista( &ordenados );
        while( ! esta_vacia_lista( pendientes ) ) {
            elemento = extrae_minimo_de_lista( &pendientes );
            pon_al_final_de_lista( &ordenados, elemento );
        } /* while */
        printf( "Lista ordenada por DNI:\n" );
        principio_de_lista( &ordenados );
    }
}
```

```

while( !es_final_de_lista( ordenados ) ) {
    refnodo = ref_nodo_de_lista( ordenados );
    elemento = elemento_en_ref_nodo(ordenados, refnodo);
    muestra_elemento( elemento );
    avanza_posicion_en_lista( &ordenados );
} /* while */
printf( "Fin.\n" );
} else {
    printf( ";No puedo abrir %s!\n", nombre );
} /* if */
return 0;
} /* main */

```

Por lo tanto, también será necesario programar las funciones siguientes:

- `elemento_t crea_elemento(unsigned int DNI, float nota);`
- `elemento_t lee_elemento(char *frase);`
- `void muestra_elemento(elemento_t elemento);`

Nota

En este caso, los elementos de la lista no son destruidos antes de finalizar la ejecución del programa porque resulta más simple y, además, se sabe que el espacio de memoria que ocupa se liberará en su totalidad. Aun así, no deja de ser una mala práctica de la programación y, por lo tanto, se propone como ejercicio libre, la incorporación de una función para eliminar las variables dinámicas correspondientes a cada elemento antes de acabar la ejecución del programa.

- 5) Implementad el programa anterior en tres unidades de compilación distintas: una para el programa principal, que también puede dividirse en funciones más manejables, una para los elementos y otra para las listas, que puede transformarse en biblioteca.
- 6) Haced un programa que acepte como argumento un NIF y valide la letra. Para ello, tómese como referencia el ejercicio de autoevaluación número 7 de la unidad anterior.

- 7) Transformad la utilidad de búsqueda de palabras en ficheros de texto del primer ejercicio para que tome como argumentos en la línea de comandos tanto la palabra a buscar como el nombre del fichero de texto en el que tenga que realizar la búsqueda.
- 8) Cread un comando que muestre el contenido del directorio como si de un `ls -als | more` se tratara. Para ello, hay que hacer un programa que ejecute tal mandato y devuelva el código de error correspondiente.
- 9) Programad un “despertador” para que muestre un aviso cada cierto tiempo o en una hora determinada. Para ello, tomar como referencia el programa ejemplo visto en la sección de procesos permanentes.

El programa tendrá como argumento la hora y minutos en que se debe mostrar el aviso indicado, que será el segundo argumento. Si la hora y minutos se precede con el signo '+', entonces se tratará como en el ejemplo, es decir, como el lapso de tiempo que debe pasar antes de mostrar el aviso.

Hay que tener presente que la lectura del primer valor del primer argumento puede hacerse de igual forma que en el programa “avisador” del tema, puesto que el signo '+' se interpreta como indicador de signo del mismo número. Eso sí, hay que leer específicamente `argv[1][0]` para saber si el usuario ha introducido el signo o no.

Para saber la hora actual, es necesario emplear las funciones de biblioteca estándar de tiempo, que se encuentran declaradas en `time.h`, y cuyo uso se muestra en el programa siguiente:

```
/* Fichero: horamin.c */  
#include <stdio.h>  
#include <time.h>  
int main( void )  
{  
    time_t tiempo;  
    struct tm *tiempo_desc;  
    time( &tiempo );  
    tiempo_desc = localtime( &tiempo );  
    printf( "Son las %2d y %2d minutos.\n",
```

```

        tiempo_desc->tm_hour,
        tiempo_desc->tm_min
    ); /* printf */
    return 0;
} /* main */

```

- 10) Probad los programas de detección de números primos mediante hilos y procesos. Para ello, es necesario definir la función `es_primo()` de manera adecuada. El siguiente programa es una muestra de tal función, que aprovecha el hecho de que ningún divisor entero será mayor que la raíz cuadrada del número (se aproxima por la potencia de 2 más parecida):

```

/* Fichero: es_primo.c                                         */
#include <stdio.h>
#include "bool.h"

int main( int argc, char *argv[] )
{
    unsigned long int numero, maximo, divisor;
    bool         primo;

    if( argc == 2 ) {
        numero = atol( argv[1] );
        primo = numero < 4; /* 0 ... 3, considerados primos.      */
        if( !primo ) {
            divisor = 2;
            primo = numero % divisor != 0;
            if( primo ) {
                maximo = numero / 2;
                while( maximo*maximo > numero ) maximo = maximo/2;
                maximo = maximo * 2;
                divisor = 1;
                while( primo && (divisor < maximo) ) {
                    divisor = divisor + 2;
                    primo = numero % divisor != 0;
                } /* while */
            } /* if */
        } /* if */
        printf( "... %s primo.\n", primo? "es" : "no es" );
    } else {
        printf( "Uso: %s número_natural\n", argv[0] );
    } /* if */
    return 0;
} /* main */

```

3.17.1. Solucionario

- 1) Como ya tenemos el programa principal, es suficiente con mostrar la función siguiente_palabra:

```
palabra_t siguiente_palabra(
    char         *frase,
    unsigned int inicio)
{
    unsigned int fin, longitud;
    palabra_t    palabra;

    while( frase[inicio]!='\0' && !isalnum(frase[inicio]) ) {
        inicio = inicio + 1;
    } /* while */
    fin = inicio;
    while( frase[fin]!='\0' && isalnum( frase[fin] ) ) {
        fin = fin + 1;
    } /* while */
    longitud = fin - inicio;
    if( longitud > 0 ) {
        palabra = (palabra_t)malloc((longitud+1)*sizeof(char));
        if( palabra != NULL ) {
            strncpy( palabra, &(frase[inicio]), longitud );
            palabra[longitud] = '\0';
        } /* if */
    } else {
        palabra = NULL;
    } /* if */
    return palabra;
} /* siguiente_palabra */
```

2)

```
bool elimina_enesimo_lista(
    lista_t *listaref, /* Apuntador a referencia 1er nodo.*/
    unsigned int n,      /* Posición de la eliminación.          */
    int       *datoref) /* Referencia del dato eliminado.      */
{
    /* Devuelve FALSE si no se puede.      */
    nodo_t *p, *q, *t;
    bool retval;
```

```

enesimo_pq_nodo( *listaref, n, &p, &q );
if( q != NULL ) {
    *datoref = destruye_nodo( listaref, p, q );
    retval = TRUE;
} else {
    retval = FALSE;
} /* if */
return retval;
} /* elimina_enesimo_lista */

void muestra_lista( lista_t lista )
{
nodo_t *q;
if( lista != NULL ) {
    q = lista;
    printf( "Lista = " );
    while( q != NULL ) {
        printf( "%i ", q->dato );
        q = q->siguiente;
    } /* while */
    printf( "\n" );
} else {
    printf( "Lista vacía.\n" );
} /* if */
} /* muestra_lista */

```

3)

```

int main( void )
{
    cola_t    cola;
    char      opcion;
    int       dato;

    printf( "Gestor de colas de enteros.\n" );
    cola.primero = NULL; cola.ultimo = NULL;
    do {
        printf(
            "[E]ncolar, [D]esencolar, [M]ostrar o [S]alir? "
        ); /* printf */
        do opcion = getchar(); while( isspace(opcion) );
        opcion = toupper( opcion );

```

```

switch( opcion ) {
    case 'E':
        printf( "Dato =? " );
        scanf( "%i", &dato );
        if( !encola( &cola, dato ) ) {
            printf( "No se insertó.\n" );
        } /* if */
        break;
    case 'D':
        if( desencola( &cola, &dato ) ) {
            printf( "Dato = %i\n", dato );
        } else {
            printf( "No se eliminó.\n" );
        } /* if */
        break;
    case 'M':
        muestra_lista( cola.primero );
        break;
} /* switch */
} while( opcion != 'S' );
while( desencola( &cola, &dato ) ) { ; }
printf( "Fin.\n" );
return 0;
} /* main */

```

4)

```

elemento_t crea_elemento( unsigned int DNI, float nota )
{
    elemento_t elemento;
    elemento = (elemento_t)malloc( sizeof( dato_t ) );
    if( elemento != NULL ) {
        elemento->DNI = DNI;
        elemento->nota = nota;
    } /* if */
    return elemento;
} /* crea_elemento */

elemento_t lee_elemento( char *frase )
{
    unsigned int DNI;
    double       nota;
    int          leido_ok;

```

```

elemento_t    elemento;
leido_ok = sscanf( frase, "%u%lf", &DNI, &nota );
if( leido_ok == 2 ) {
    elemento = crea_elemento( DNI, nota );
} else {
    elemento = NULL;
} /* if */
return elemento;
} /* lee_elemento */
void muestra_elemento( elemento_t elemento )
{
    printf( "%10u %.2f\n", elemento->DNI, elemento->nota );
} /* muestra_elemento */

```

5) Véase el apartado 3.6.2.

6)

```

char letra_de( unsigned int DNI )
{
    char codigo[] = "TRWAGMYFPDXBNJZSQVHLCKE" ;
    return codigo[ DNI % 23 ];
} /* letra_de */

int main( int argc, char *argv[] )
{
    unsigned int DNI;
    char         letra;
    int          codigo_error;

    if( argc == 2 ) {
        sscanf( argv[1], "%u", &DNI );
        letra = argv[1][ strlen(argv[1])-1 ];
        letra = toupper( letra );
        if( letra == letra_de( DNI ) ) {
            printf( "NIF válido.\n" );
            codigo_error = 0;
        } else {
            printf( "¡Letra %c no válida!\n", letra );
            codigo_error = -1;
        } /* if */
    } else {
        printf( "Uso: %s DNI-letra\n", argv[0] );
        codigo_error = 1;
    } /* if */
    return codigo_error;
} /* main */

```

7)

```
int main( int argc, char *argv[] )  
{  
    FILE          *entrada;  
    char           nombre[BUFSIZ];  
    palabra_t      palabra, palabra2;  
    unsigned int   numlin, pos;  
    int            codigo_error;  
    if( argc == 3 ) {  
        palabra = siguiente_palabra( argv[1], 0 );  
        if( palabra != NULL ) {  
            entrada = fopen( argv[2], "rt" );  
            if( entrada != NULL ) {  
                /* (Véase: Enunciado del ejercicio 1.) */  
                } else {  
                    printf( "¡No puedo abrir %s!\n", argv[2] );  
                    codigo_error = -1;  
                } /* if */  
            } else {  
                printf( "¡Palabra %s inválida!\n", argv[1] );  
                codigo_error = -1;  
            } /* if */  
        } else {  
            printf( "Uso: %s palabra fichero\n", argv[0] );  
            codigo_error = 0;  
        } /* if */  
        return codigo_error;  
    } /* main */
```

8)

```
int main( int argc, char *argv[] )  
{  
    int codigo_error;  
  
    codigo_error = system( "ls -als | more" );  
    return codigo_error;  
} /* main */
```

9)

```

int main( int argc, char *argv[] )
{
    unsigned int horas;
    unsigned int minutos;
    unsigned int segundos;
    char         *aviso, *separador;
    time_t       tiempo;
    struct tm    *tiempo_desc;

    if( argc == 3 ) {
        separador = strchr( argv[1], ':' );
        if( separador != NULL ) {
            horas = atoi( argv[1] ) % 24;
            minutos = atoi( separador+1 ) % 60;
        } else {
            horas = 0;
            minutos = atoi( argv[1] ) % 60;
        } /* if */
        if( argv[1][0]!='+' ) {
            time( &tiempo );
            tiempo_desc = localtime( &tiempo );
            if( minutos < tiempo_desc->tm_min ) {
                minutos = minutos + 60;
                horas = horas - 1;
            } /* if */
            if( horas < tiempo_desc->tm_hour ) {
                horas = horas + 24;
            } /* if */
            minutos = minutos - tiempo_desc->tm_min;
            horas = horas - tiempo_desc->tm_hour;
        } /* if */
        segundos = (horas*60 + minutos) * 60;
        aviso = argv[2];
        if( daemon( FALSE, TRUE ) ) {

            printf( "No puede instalarse el avisador :-(\n" );

        } else {
            printf( "Alarma dentro de %i horas y %i minutos.\n",

```

```
horas, minutos
); /* printf */
printf( "Haz $ kill %li para apagarla.\n",
getpid()
); /* printf */
} /* if */
sleep( segundos );
printf( "%s\007\n", aviso );
printf( "Alarma apagada.\n" );
} else {
printf( "Uso: %s [+]HH:MM \"aviso\"\n", argv[0] );
printf( "(con + es respecto de la hora actual)\n" );
printf( "(sin + es la hora del dia)\n" );
} /* if */
return 0;
} /* main */
```

- 10) Se trata de repetir el código dado dentro de una función que tenga la cabecera adecuada para cada caso.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX • Editorial Guías México

4. Programación orientada a objetos en C++

4.1. Introducción

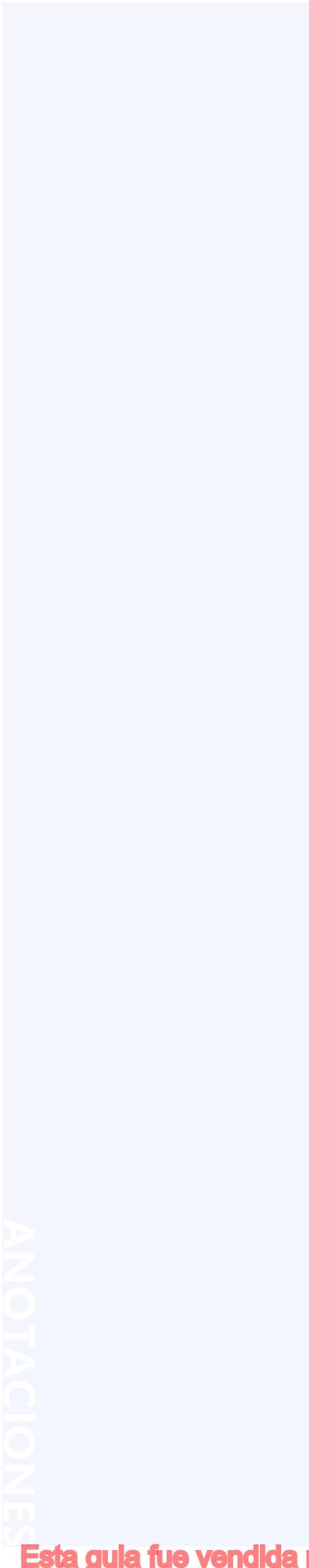
Hasta el momento se ha estudiado cómo abordar un problema utilizando los paradigmas de programación modular y el diseño descendente de algoritmos. Con ellos se consigue afrontar un problema complejo mediante la descomposición en problemas más simples, reduciendo progresivamente su nivel de abstracción, hasta obtener un nivel de detalle manejable. Al final, el problema se reduce a estructuras de datos y funciones o procedimientos.

Para trabajar de forma eficiente, las buenas prácticas de programación nos aconsejan agrupar los conjuntos de rutinas y estructuras relacionados entre sí en unidades de compilación, que luego serían enlazados con el archivo principal. Con ello, se consigue lo siguiente:

- Localizar con rapidez el código fuente que realiza una tarea determinada y limitar el impacto de las modificaciones a unos archivos determinados.
- Mejorar la legibilidad y comprensión del código fuente en conjunto al no mezclarse entre sí cada una de las partes.

No obstante, esta recomendable organización de los documentos del proyecto sólo proporciona una separación de los diferentes archivos y, por otro lado, no refleja la estrecha relación que suele haber entre los datos y las funciones.

En la realidad muchas veces se desea implementar entidades de forma que se cumplan unas propiedades generales: conocer las entradas que precisan, una idea general de su funcionamiento y las salidas que generan. Generalmente, los detalles concretos de la implemen-



tación no son importantes: seguramente habrá decenas de formas posibles de hacerlo.

Se puede poner como ejemplo un televisor. Sus propiedades pueden ser la marca, modelo, medidas, número de canales; y las acciones a implementar serían encender o apagar el televisor, cambiar de canal, sintonizar un nuevo canal, etc. Cuando utilizamos un aparato de televisión, lo vemos como una caja cerrada, con sus propiedades y sus conexiones. No nos interesa en absoluto sus mecanismos internos, sólo deseamos que actúe cuando apretamos el botón adecuado. Además, éste se puede utilizar en varias localizaciones y siempre tendrá la misma función. Por otro lado, si se estropea puede ser sustituido por otro y sus características básicas (tener una marca, encender, apagar, cambiar de canal, etc.) continúan siendo las mismas independientemente de que el nuevo aparato sea más moderno. El televisor es tratado como un objeto por sí mismo y no como un conjunto de componentes.

Este mismo principio aplicado a la programación se denomina **encapsulamiento**. El encapsulamiento consiste en implementar un elemento (cuyos detalles se verán más adelante) que actuará como una “caja negra”, donde se especificarán unas entradas, una idea general de su funcionamiento y unas salidas. De esta forma se facilita lo siguiente:

- La reutilización de código. Si ya se dispone de una “caja negra” que tenga unas características coincidentes con las necesidades definidas, se podrá incorporar sin interferir con el resto del proyecto.
- El mantenimiento del código. Se pueden realizar modificaciones sin que afecten al proyecto en conjunto, mientras se continúen cumpliendo las especificaciones de dicha “caja negra”.

A cada uno de estos elementos los llamaremos **objetos** (en referencia a los objetos de la vida real a los que representa). Al trabajar con objetos, lo que supone un nivel de abstracción mayor, se afronta el diseño de una aplicación no pensando en la secuencia de instrucciones a realizar, sino en la definición de los objetos que intervienen y las relaciones que se establecen entre ellos.

En esta unidad estudiaremos un nuevo lenguaje que nos permite implementar esta nueva visión que supone el paradigma de la programación orientada a objetos: C++.

Este nuevo lenguaje se basa en el lenguaje C al que se le dota de nuevas características. Por este motivo, en primer lugar, se establece una comparación entre ambos lenguajes en los ámbitos comunes que nos permite un aprendizaje rápido de sus bases. A continuación, se nos presenta el nuevo paradigma y las herramientas que el nuevo lenguaje proporciona para la implementación de los objetos y sus relaciones. Finalmente, se muestra cómo este cambio de filosofía afecta al diseño de aplicaciones.

En esta unidad se pretende que los lectores, partiendo de sus conocimientos del lenguaje C, puedan conocer los principios básicos de la programación orientada a objetos utilizando el lenguaje C++ y del diseño de aplicaciones siguiendo este paradigma. En concreto, al finalizar el estudio de esta unidad, el lector habrá alcanzado los objetivos siguientes:

- 1) Conocer las diferencias principales entre C y C++, inicialmente sin explorar aún la tecnología de objetos.
- 2) Comprender el paradigma de la programación orientada a objetos.
- 3) Saber implementar clases y objetos en C++.
- 4) Conocer las propiedades principales de los objetos: la herencia, la homonimia y el polimorfismo.
- 5) Poder diseñar una aplicación simple en C++ aplicando los principios del diseño orientado a objetos.

4.2. De C a C++

4.2.1. El primer programa en C++

Elegir el entorno de programación C++ para la implementación del nuevo paradigma de la programación orientada a objetos supone una gran ventaja por las numerosas similitudes existentes con el len-

guaje C. No obstante, puede convertirse en una limitación si el programador no explora las características adicionales que nos proporciona el nuevo lenguaje y que aportan una serie de mejoras bastante interesantes.

Tradicionalmente, en el mundo de la programación, la primera toma de contacto con un lenguaje de programación se hace a partir del clásico mensaje de “hola mundo” y, en este caso, no haremos una excepción.

Nota

La extensión “.cpp” indica al compilador que el tipo de código fuente es C++.

Por tanto, en primer lugar, escribid en vuestro editor el siguiente texto y guardadlo con el nombre `ejemplo01.cpp`:

```
#include <iostream>
int main()
{
    cout << "hola mundo \n" ;
    return 0;
}
```

Comparando este programa con el primer programa en C, observamos que la estructura es similar. De hecho, como se ha comentado, el C++ se puede ver como una evolución del C para implementar la programación orientada a objetos y, como tal, mantiene la compatibilidad en un porcentaje muy alto del lenguaje.

La única diferencia observable la encontramos en la forma de gestionar la salida que se hace a través de un objeto llamado `cout`. La naturaleza de los objetos y de las clases se estudiará en profundidad más adelante, pero, de momento, podremos hacernos una idea considerando la clase como un tipo de datos nuevo que incluye atributos y funciones asociadas, y el objeto como una variable de dicho tipo de datos.

La definición del objeto `cout` se halla dentro de la librería `<iostream>`, que se incluye en la primera línea del código fuente. También llama la atención la forma de uso, mediante el direccionamiento (con el símbolo `<<`) del texto de “hola mundo” sobre el objeto `cout`, que genera la salida de este mensaje en la pantalla.

Como el tema del tratamiento de las funciones de entrada/salida es una de las principales novedades del C++, comenzaremos por él para desarrollar las diferencias entre un lenguaje y el otro.

4.2.2. Entrada y salida de datos

Aunque ni el lenguaje C ni el C++ definen las operaciones de entrada/salida dentro del lenguaje en sí, es evidente que es indispensable su tratamiento para el funcionamiento de los programas. Las operaciones que permiten la comunicación entre los usuarios y los programas se encuentran en bibliotecas que provee el compilador. De este modo, podremos trasladar un código fuente escrito para un entorno Sun a nuestro PC en casa y así obtendremos una independencia de la plataforma. Al menos, en teoría.

Tal como se ha comentado en unidades anteriores, el funcionamiento de la entrada/salida en C se produce a través de librerías de funciones, la más importante de las cuales es la `<stdio.h>` o `<cstdio>` (entrada/salida estándar). Dichas funciones (`printf`, `scanf`, `fprint`, `fscanf`, etc.) continúan siendo operativas en C++, aunque no se recomienda su uso al no aprovechar los beneficios que proporciona el nuevo entorno de programación.

Nota

Ambas formas de expresar el nombre de la librería `<xxxx.h>` o `<xxxx>` son correctas, aunque la segunda se considera actualmente la forma estándar de incorporar librerías C dentro del lenguaje C++ y la única recomendada para su uso en nuevas aplicaciones.

C++, al igual que C, entiende la comunicación de datos entre el programa y la pantalla como un flujo de datos: el programa va enviando datos y la pantalla los va recibiendo y mostrando. De la misma manera se entiende la comunicación entre el teclado (u otros dispositivos de entrada) y el programa.

Para gestionar estos flujos de datos, C++ incluye la clase iostream, que crea e inicializa cuatro objetos:

- `cin`. Maneja flujos de entrada de datos.
- `cout`. Maneja flujos de salida de datos.
- `cerr`. Maneja la salida hacia el dispositivo de error estándar, la pantalla.
- `clog`. Maneja los mensajes de error.

A continuación, se presentan algunos ejemplos simples de su uso.

```
#include <iostream>
int main()
{
    int numero;
    cout << "Escribe un número: ";
    cin >> numero;
}
```

En este bloque de código se observa lo siguiente:

- La declaración de una variable entera con la que se desea trabajar.
- El texto “Escribe un número” (que podemos considerar como un flujo de datos literal) que deseamos enviar a nuestro dispositivo de salida.

Para conseguir nuestro objetivo, se direcciona el texto hacia el objeto `cout` mediante el operador `>>`. El resultado será que el mensaje saldrá por pantalla.

- Una variable donde se desea guardar la entrada de teclado. Otra vez el funcionamiento deseado consistirá en dirigir el flujo de entrada recibido en el teclado (representado/gestionado por el objeto `cin`) sobre dicha variable.

La primera sorpresa para los programadores en C, acostumbrados al `printf` y al `scanf`, es que no se le indica en la instrucción el

formato de los datos que se desea imprimir o recibir. De hecho, ésta es una de las principales ventajas de C++: el compilador reconoce el tipo de datos de las variables y trata el flujo de datos de forma consecuente. Por tanto, simplificando un poco, se podría considerar que los objetos `cin` y `cout` se adaptan al tipo de datos. Esta característica nos permitirá adaptar los objetos `cin` y `cout` para el tratamiento de nuevos tipos de datos (por ejemplo, `structs`), cosa impensable con el sistema anterior.

Si se desea mostrar o recoger diversas variables, simplemente, se encadenan flujos de datos:

```
#include <iostream>
int main()
{
    int i, j, k;
    cout << "Introducir tres números";
    cin >> i >> j >> k;
    cout << "Los números son: "
    cout << i << ", " << j << " y " << k;
}
```

En la última línea se ve cómo en primer lugar se envía al `cout` el flujo de datos correspondiente al texto “Los números son:”; después, el flujo de datos correspondiente a la variable `i`; posteriormente, el texto literal “ , ”, y así hasta el final.

En el caso de la entrada de datos por teclado, `cin` leerá caracteres hasta la introducción de un carácter de salto de línea (return o “\n”). Después, irá extrayendo del flujo de datos introducido caracteres hasta encontrar el primer espacio y dejará el resultado en la variable `i`. El resultado de esta operación será también un flujo de datos (sin el primer número que ya ha sido extraído) que recibirá el mismo tratamiento: ir extrayendo caracteres del flujo de datos hasta el siguiente separador para enviarlo a la siguiente variable. El proceso se repetirá hasta leer las tres variables.

Por tanto, la línea de lectura se podría haber escrito de la siguiente manera y hubiera sido equivalente, pero menos clara:

```
( ( ( cin >> i ) >> j ) >> k )
```

Si se desea mostrar la variable en un formato determinado, se debe enviar un manipulador del objeto que le indique el formato deseado. En el siguiente ejemplo, se podrá observar su mecánica de funcionamiento:

```
#include <iostream>
#include <iomanip>
// Se debe incluir para la definición de los
// manipuladores de objeto cout con parámetros

int main()
{
    int i = 5;
    float j = 4.1234;

    cout << setw(4) << i << endl;
    //muestra i con anchura de 4 car.
    cout << setprecision(3) << j << endl;
    // muestra j con 3 decimales
}
```



Hay muchas otras posibilidades de formato, pero no es el objetivo de este curso. Esta información adicional está disponible en la ayuda del compilador.

4.2.3. Utilizando C++ como C

Como se ha comentado, el lenguaje C++ nació como una evolución del C, por lo que, para los programadores en C, es bastante simple adaptarse al nuevo entorno. No obstante, además de introducir todo el tratamiento para la programación orientada a objetos, C++ aporta algunas mejoras respecto a la programación clásica que es interesante conocer y que posteriormente, en la programación orientada a objetos, adquieran toda su dimensión.

A continuación, procederemos a analizar diferentes aspectos del lenguaje.

4.2.4. Las instrucciones básicas

En este aspecto, C++ se mantiene fiel al lenguaje C: las instrucciones mantienen su aspecto general (finalizadas en punto y coma, los bloques de código entre llaves, etc.) y las instrucciones básicas de control de flujo, tanto las de selección como las iterativas, conservan su sintaxis (`if`, `switch`, `for`, `while`, `do ... while`). Estas características garantizan una aproximación rápida al nuevo lenguaje.

Dentro de las instrucciones básicas, podríamos incluir las de entrada/salida. En este caso, sí que se presentan diferencias significativas entre C y C++, diferencias que ya hemos comentado en el apartado anterior.

Además, es importante resaltar que se ha incluido una nueva forma de añadir comentarios dentro del código fuente para contribuir a mejorar la lectura y el mantenimiento del mismo. Se conserva la forma clásica de los comentarios en C como el texto incluido entre las secuencias de caracteres `/*` (inicio de comentario) y `*/` (fin de comentario), pero se añade una nueva forma: la secuencia `//` que nos permite un comentario hasta final de línea.

Ejemplo

```
/*
Este texto está comentado utilizando la forma clásica de C.
Puede contener la cantidad de líneas que se desee.
*/

//
// Este texto utiliza el nuevo formato de comentarios
// hasta final de línea que incorpora C++
//
```

4.2.5. Los tipos de datos

Los tipos de datos fundamentales de C (`char`, `int`, `long int`, `float` y `double`) se conservan en C++, y se incorpora el nuevo tipo `bool` (tipo booleano o lógico), que puede adquirir dos valores posibles: falso (`false`) o verdadero (`true`), ahora definidos dentro del lenguaje.

```
// ...
{
    int i = 0, num;
    bool continuar;

    continuar = true;
    do
    {
        i++;
        cout << "Para finalizar este bucle que ha pasado";
        cout << i << "veces, teclea un 0 ";
        cin >> num;
        if (num == 0) continuar = false;
    } while (continuar);
}
```

Aunque el uso de variables de tipo lógico o booleano ya era común en C, utilizando como soporte los números enteros (el 0 como valor falso y cualquier otro valor como verdadero), la nueva implementación simplifica su uso y ayuda a reducir errores. Además, el nuevo tipo de datos sólo ocupa un byte de memoria, en lugar de los dos bytes que utilizaba cuando se simulaba con el tipo int.

Por otro lado, hay que destacar las novedades respecto de los tipos estructurados (`struct`, `enum` o `union`). En C++ pasan a ser considerados descriptores de tipos de datos completos, evitando la necesidad del uso de la instrucción `typedef` para definir nuevos tipos de datos.

En el ejemplo que sigue, se puede comprobar que la parte de la definición del nuevo tipo no varía:

```
struct fecha {
    int dia;
    int mes;
    int año;
};

enum diasSemana {LUNES, MARTES, MIERCOLES, JUEVES,
VIERNES, SABADO, DOMINGO};
```

Lo que se simplifica es la declaración de una variable de dicho tipo, puesto que no se tienen que volver a utilizar los términos `struct`, `enum` o `union`, o la definición de nuevos tipos mediante la instrucción `typedef`:

```
fecha aniversario;  
diasSemana festivo;
```

Por otro lado, la referencia a los datos tampoco varía.

```
// ...  
aniversario.dia = 2;  
aniversario.mes = 6;  
aniversario.anyo = 2001;  
festivo = LUNES;
```

En el caso de la declaración de variables tipo `enum` se cumplen dos funciones:

- Declarar `diasSemana` como un tipo nuevo.
- Hacer que `LUNES` corresponda a la constante 0, `MARTES` a la constante 1 y así sucesivamente.

Por tanto, cada constante enumerada corresponde a un valor entero. Si no se especifica nada, el primer valor tomará el valor de 0 y las siguientes constantes irán incrementando su valor en una unidad. No obstante, C++ permite cambiar este criterio y asignar un valor determinado a cada constante:

```
enum comportamiento {HORRIBLE = 0, MALO, REGULAR = 100,  
BUENO = 200, MUY_BUENO, EXCELENTE};
```

De esta forma, `HORRIBLE` tomaría el valor de 0, `MALO` el valor de 1, `REGULAR` el valor de 100, `BUENO` el de 200, `MUY_BUENO`, el de 201 y `EXCELENTE`, el de 202.

Otro aspecto a tener en cuenta es la recomendación en esta nueva versión que se refiere a realizar las coerciones de tipos de forma explícita. La versión C++ se recomienda por su legibilidad:

```
int i = 0;
long v = (long) i; // coerción de tipos en C
long v = long (i); // coerción de tipos en C++
```

4.2.6. La declaración de variables y constantes

La declaración de variables en C++ continua teniendo el mismo formato que en C, pero se introduce un nuevo elemento que aportará más seguridad en nuestra forma de trabajar. Se trata del especificador `const` para la definición de constantes.

En programación se utiliza una constante cuando se conoce con certeza que dicho valor no debe variar durante el proceso de ejecución de la aplicación:

```
const float PI = 3.14159;
```

Una vez definida dicha constante, no se le puede asignar ningún valor y, por tanto, siempre estará en la parte derecha de las expresiones. Por otro lado, una constante siempre tiene que estar inicializada:

```
const float radio;// ERROR!!!!!!
```

El uso de constantes no es nuevo en C. La forma clásica para definirlas es mediante la instrucción de preprocesador `#define`.

```
#define PI 3.14159
```

En este caso, el comportamiento real es sustituir cada aparición del texto `PI` por su valor en la fase de preprocesamiento del texto. Por tanto, cuando analiza el texto, el compilador sólo ve el número `3.14159` en lugar de `PI`.

No obstante, mientras el segundo caso corresponde a un tratamiento especial durante el proceso previo a la compilación, el uso de `const` permite un uso normalizado y similar al de una variable pero con capacidades limitadas. A cambio, recibe el mismo tratamiento que las variables respecto al ámbito de actuación (sólo en el fichero de trabajo, a menos que se le indique lo contrario mediante la palabra reservada `extern`) y tiene un tipo asignado, con lo cual se podrán realizar todas las comprobaciones de tipos en fase de compilación haciendo el código fuente resultante más robusto.

4.2.7. La gestión de variables dinámicas

La gestión directa de la memoria es una de las armas más poderosas de las que dispone el C, y una de las más peligrosas: cualquier acceso inadecuado a zonas de memoria no correspondiente a los datos deseados puede provocar resultados imprevisibles en el mejor de los casos, cuando no desastrosos.

En los capítulos anteriores, se ha visto que las operaciones con direcciones de memoria en C se basan en los apuntadores (`*apunt`), usados para acceder a una variable a partir de su dirección de memoria. Utilizan el operador de indirección o desreferencia (`*`), y sus características son:

- apuntador contiene una dirección de memoria.
- `*apuntador` indica al contenido existente en dicha dirección de memoria.
- Para acceder a la dirección de memoria de una variable se utiliza el operador dirección (`&`) precediendo el nombre de la variable.

```
// Ejemplo de uso de apuntadores
int i = 10;
int *apunt_i = &i;
    // apunt_i toma la dirección
    // de la variable i de tipo entero
    // Si no lo asignáramos aquí, sería
    // recomendable inicializarlo a NULL
```

```
*apunt_i = 3;
    // Se asigna el valor 3 a la posición
```

```

// de memoria apunt_i
//Por tanto, se modifica el valor de i.

cout << "Valor Original      : " << i << endl ;
cout << "A través del apuntador : "
     << *apunt_i << endl ;
// La salida mostrará:
// Valor original: 3
// A través del apuntador: 3

```

Operadores new y delete

El principal uso de los apuntadores está relacionado con las variables dinámicas. Las dos principales funciones definidas en C para realizar estas operaciones son `malloc()` y `free()` para reservar memoria y liberarla, respectivamente. Ambas funciones continúan siendo válidas en C++. No obstante, C++ proporciona dos nuevos operadores que permiten un control más robusto para dicha gestión. Son `new` y `delete`. Al estar incluidos en el lenguaje, no se precisa la inclusión de ninguna librería específica.

El operador `new` realiza la reserva de memoria. Su formato es `new` y un tipo de datos. A diferencia del `malloc`, en este caso no es preciso indicarle el tamaño de la memoria a reservar, pues el compilador lo calcula a partir del tipo de datos utilizado.

Para liberar la memoria se dispone del operador `delete` que también ofrece más robustez que la función `free()`, pues protege internamente el hecho de intentar liberar memoria apuntada por un apuntador nulo.

```

fecha * aniversario = new fecha;
...
delete aniversario;

```

Si se desea crear varios elementos, basta especificarlo en forma de vector o matriz. El resultado es declarar la variable como apuntador al primer elemento del vector.

De forma similar, se puede liberar toda la memoria reservada por el vector (cada uno de los objetos creados y el vector mismo) utilizando la forma `delete []`.

```
fecha * lunasLlenas = new fecha[12];  
...  
delete [] lunasLlenas;
```

Si se omiten los corchetes, el resultado será eliminar únicamente el primer objeto del vector, pero no el resto creando una fuga de memoria; es decir, memoria reservada a la cual no se puede acceder en el futuro.

Apuntadores const

En la declaración de apuntadores en C++ permite el uso de la palabra reservada `const`. Y además existen diferentes posibilidades.

```
const int * ap_i;           // *ap_i permanece constante  
int * const ap_j;  
                           // Dirección ap_j es constante, pero no su valor  
const int * const ap_k;  
                           // Tanto la dirección ap_k como su valor *ap_k serán constantes
```

Es decir, en el caso de apuntadores se puede hacer constante su valor (`*ap_i`) o su dirección de memoria (`ap_i`), o las dos cosas. Para no confundirse, basta con fijarse en el texto posterior a la palabra reservada `const`.

Con la declaración de apuntadores constantes el programador le indica al compilador cuándo se desea que el valor o la dirección que contiene un apuntador no sufran modificaciones. Por tanto, cualquier intento de asignación no prevista se detecta en tiempo de compilación. De esta forma se reduce el riesgo de errores de programación.

Referencias

Para la gestión de variables dinámicas C++ añade un nuevo elemento para facilitar su uso: las referencias. Una referencia es un

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

alias o un sinónimo. Cuando se crea una referencia, se inicializa con el nombre de otra variable y actúa como un nombre alternativo de ésta.

Para crearla se escribe el tipo de la variable destino, seguido del operador de referencia (&) y del nombre de la referencia. Por ejemplo,

```
int i;
int & ref_i = i;
```

En la expresión anterior se lee: la variable `ref_i` es una referencia a la variable `i`. Las referencias siempre se tienen que inicializar en el momento de la declaración (como si fuera una `const`).

Hay que destacar que aunque el operador de referencia y el de dirección se representan de la misma forma (&), corresponden a operaciones diferentes, aunque están relacionados entre sí. De hecho, la característica principal de las referencias es que si se pide su dirección, devuelven la de la variable destino.

```
#include <iostream>

int main()
{
    int i = 10;
    int & ref_i = i;

    ref_i = 3; //Se asigna el valor 3 a la posición

    cout << "valor de i " << i << endl;
    cout << "dirección de i " << &i << endl;
    cout << "dirección de ref_i " << &ref_i << endl;
}
```

Con este ejemplo se puede comprobar que las dos direcciones son idénticas, y como la asignación sobre `ref_i` tiene el mismo efecto que si hubiera sido sobre `i`.

Otras características de las referencias son las siguientes:

- No se pueden reasignar. El intento de reasignación se convierte en una asignación en la variable sinónima.
- No se le pueden asignar un valor nulo.

El uso principal de las referencias es el de la llamada a funciones, que se verá a continuación.

4.2.8. Las funciones y sus parámetros

El uso de las funciones, elemento básico de la programación modular, continúa teniendo el mismo formato: un tipo del valor de retorno, el nombre de la función y un número de parámetros precedidos por su tipo. A esta lista de parámetros de una función también se la conoce como la **firma de una función**.

Uso de parámetros por valor o por referencia

Como se ha comentado en unidades anteriores, en C existen dos formas de pasar parámetros a una función: por valor o por variable. En el primer caso, la función recibe una copia del valor original del parámetro mientras que en el segundo se recibe la dirección de dicha variable. De esta forma, se puede acceder directamente a la variable original, que puede ser modificada. Para hacerlo, la forma tradicional en C es pasar a la función como parámetro el apuntador a una variable.

A continuación, veremos una función que permite intercambiar el contenido de dos variables:

```
#include <iostream>

void intercambiar(int *i, int *j);

int main()
```

Nota

Aquí no utilizaremos el término por referencia para no inducir a confusión con las referencias de C++.

```

int x = 2, y = 3;
cout << " Antes. x = " << x << " y = " << y << endl;
intercambiar(&x , &y);
cout << " Después. x = " << x << " y = " << y << endl;
}
void intercambiar(int *i, int *j)
{
    int k;
    k = *i;
    *i = *j;
    *j = k;
}

```

Se puede comprobar que el uso de las desreferencias (*) dificulta su comprensión. Pero en C++ se dispone de un nuevo concepto comentado anteriormente: las referencias. La nueva propuesta consiste en recibir el parámetro como referencia en lugar de apuntador:

```

#include <iostream>

void intercambiar(int &i, int &j);

int main()
{
    int x = 2, y = 3;
    cout << " Antes. x = " << x << " y = " << y << endl;
    intercambiar(x , y); //NO intercambiar(&x , &y);
    cout << " Despues. x= " << x << " y = " << y << endl;
}
void intercambiar(int & i, int & j)
{
    int k;
    k = i;
    i = j;
    j = k;
}

```

El funcionamiento de esta nueva propuesta es idéntico al de la anterior, pero la lectura del código fuente es mucho más simple al utilizar el operador de referencia (&) para recoger las direcciones de memoria de los parámetros.

No obstante, hay que recordar que las referencias tienen una limitaciones (no pueden tomar nunca un valor nulo y no pueden reasignarse). Por tanto, no se podrán utilizar las referencias en el paso de parámetros cuando se desee pasar un apuntador como parámetro y que éste pueda ser modificado (por ejemplo, obtener el último elemento en una estructura de datos de cola). Tampoco se podrán utilizar referencias para aquellos parámetros que deseamos considerar como opcionales, al existir la posibilidad de que no pudieran ser asignados a ningún parámetro de la función que los llama, por lo que tendrían que tomar el valor null, (lo que no es posible).

En estos casos, el paso de parámetros por variable se debe continuar realizando mediante el uso de apuntadores.

Uso de parámetros const

En la práctica, al programar en C, a veces se utiliza el paso por variable como una forma de eficiencia al evitar tener que realizar una copia de los datos dentro de la función. Con estructuras de datos grandes (estructuras, etc.) esta operación interna de salvaguarda de los valores originales puede ocupar un tiempo considerable y, además, se corre el riesgo de una modificación de los datos por error.

Para reducir estos riesgos, C++ permite colocar el especificador `const` justo antes del parámetro (tal como se ha comentado en el apartado de apuntadores `const`).

Si en el ejemplo anterior de la función intercambiar se hubiera definido los parámetros `i` y `j` como `const` (lo cual no tiene ningún sentido práctico y sólo se considera a efectos explicativos), nos daría errores de compilación:

```
void intercambiar(const int & i, const int & j);  
{  
    int k;  
    k = i;  
    i = j; // Error de compilación. Valor i constante.  
    j = k; // Error de compilación. Valor j constante.  
}
```

De este modo se pueden conseguir las ventajas de eficiencia al evitar los procesos de copia no deseados sin tener el inconveniente de estar desprotegido ante modificaciones no deseadas.

Sobrecarga de funciones

C admite un poco de flexibilidad en las llamadas a funciones al permitir el uso de un número de parámetros variables en la llamada a una función, siempre y cuando sean los parámetros finales y en la definición de dicha función se les haya asignado un valor para el caso que no se llegue a utilizar dicho parámetro.

C++ ha incorporado una opción mucho más flexible, y que es una de las novedades respecto al C más destacadas: permite el uso de diferentes funciones con el mismo nombre (*homonímia de funciones*). A esta propiedad también se la denomina *sobrecarga de funciones*.



Las funciones pueden tener el mismo nombre pero deben tener diferencias en su lista de parámetros, sea en el número de parámetros o sea en variaciones en el tipo de éstos.

Hay que destacar que el tipo del valor de retorno de la función no se considera un elemento diferencial de la función, por tanto, el compilador muestra error si se intenta definir dos funciones con el mismo nombre e idéntico número y tipo de parámetros pero que retornen valores de distintos tipos. El motivo es que el compilador no puede distinguir a cuál de las funciones definidas se desea llamar.

A continuación se propone un programa que eleva números de diferente tipo al cuadrado:

```
#include <iostream>

int elevarAlCuadrado (int);
float elevarAlCuadrado (float);

int main()
```

```
{  
    int numEntero = 123;  
    float numReal = 12.3;  
    int numEnteroAlCuadrado;  
    float numRealAlCuadrado;  
  
    cout << "Ejemplo para elevar números al cuadrado\n";  
    cout << "Números originales \n";  
    cout << "Número entero: " << numEntero << "\n";  
    cout << "Número real: " << numReal << "\n";  
  
    numEnteroAlCuadrado = elevarAlCuadrado (numEntero);  
    numRealAlCuadrado = elevarAlCuadrado (numReal);  
  
    cout << "Números elevados al cuadrado \n";  
    cout << "Número entero:" << numEnteroAlCuadrado << "\n";  
    cout << "Número real: " << numRealAlCuadrado << "\n";  
  
    return 0;  
}  
  
int elevarAlCuadrado (int num)  
{  
    cout << "Elevando un número entero al cuadrado \n";  
    return ( num * num);  
}  
float elevarAlCuadrado (float num)  
{  
    cout << "Elevando un número real al cuadrado \n";  
    return ( num * num);  
}
```

El hecho de sobregregar la función ElevarAlCuadrado ha permitido que con el mismo nombre de función se pueda realizar lo que es intrínsecamente la misma operación. Con ello, hemos evitado tener que definir dos nombres de función diferentes:

- ElevarAlCuadradoNumerosEnteros
- ElevarAlCuadradoNumerosReales

De esta forma, el mismo compilador identifica la función que se desea ejecutar por el tipo de sus parámetros y realiza la llamada correcta.

4.3. El paradigma de la programación orientada a objetos

En las unidades anteriores, se han analizado una serie de paradigmas de programación (modular y descendente) que se basan en la progresiva organización de los datos y la resolución de los problemas a partir de su división en un conjunto de instrucciones secuenciales. La ejecución de dichas instrucciones sólo se apoya en los datos definidos previamente.

Este enfoque, que nos permite afrontar múltiples problemas, también muestra sus limitaciones:

- El uso compartido de los datos provoca que sea difícil modificar y ampliar los programas por sus interrelaciones.
- El mantenimiento de los grandes programas se vuelve realmente complicado al no poder asegurar el control de todas las implicaciones que suponen cambios en el código.
- La reutilización de código también puede provocar sorpresas, otra vez por no poder conocer todas las implicaciones que suponen.

Nota

¿Cómo es posible que se tengan tantas dificultades si las personas son capaces de realizar acciones complejas en su vida cotidiana? La razón es muy sencilla: en nuestra vida cotidiana no se procede con los mismos criterios. La descripción de nuestro entorno se hace a partir de objetos: puertas, ordenadores, automóviles, ascensores, personas, edificios, etc., y estos objetos cumplen unas relaciones más o menos simples: si una puerta está abierta se puede pasar y si está cerrada no se puede. Si un automóvil tiene una rueda pinchada, se sustituye y se puede volver a circular. ¡Y no necesitamos conocer toda la mecánica del automóvil para poder realizar esta operación! Ahora bien, ¿nos podríamos imaginar nuestro mundo si al cambiar el neumático nos dejara de funcionar el limpiapar-

rabrías? Sería un caos. Eso es casi lo que sucede, o al menos no podemos estar completamente seguros de que no suceda, con los paradigmas anteriores.

El paradigma de la orientación a objetos nos propone una forma diferente de enfocar la programación sobre la base de la definición de objetos y de las relaciones entre ellos.

Cada objeto se representa mediante una **abstracción** que contiene su información esencial, sin preocuparse de sus demás características.

Esta información está compuesta de datos (variables) y acciones (funciones) y, a menos que se indique específicamente lo contrario, su ámbito de actuación está limitado a dicho objeto (**ocultamiento de la información**). De esta forma, se limita el alcance de su código de programación, y por tanto su repercusión, sobre el entorno que le rodea. A esta característica se le llama **encapsulamiento**.

Las relaciones entre los diferentes objetos pueden ser diversas, y normalmente suponen acciones de un objeto sobre otro que se implementan mediante mensajes entre los objetos.

Una de las relaciones más importante entre los objetos es la pertenencia a un tipo más general. En este caso, el objeto más específico comparte una serie de rasgos (información y acciones) con el más genérico que le vienen dados por esta relación de inclusión. El nuevo paradigma proporciona una herramienta para poder reutilizar todos estos rasgos de forma simple: **la herencia**.

Finalmente, una característica adicional es el hecho de poder comportarse de forma diferente según el contexto que lo envuelve. Es conocida como **polimorfismo** (un objeto, muchas formas). Además, esta propiedad adquiere toda su potencia al ser capaz de adaptar este comportamiento en el momento de la ejecución y no en tiempo de compilación.



El paradigma de programación orientada a objetos se basa en estos cuatro pilares: abstracción, encapsulación, herencia y polimorfismo.

Ejemplo

Ejemplo de acciones sobre objetos en la vida cotidiana: llamar por teléfono, descolgar el teléfono, hablar, contestar, etc.

Ejemplo

Un perro es un animal, un camión es un vehículo, etc.

4.3.1. Clases y objetos

A nivel de implementación, una clase corresponde a un nuevo tipo de datos que contiene una colección de datos y de funciones que nos permiten su manipulación.

Ejemplo

Deseamos describir un aparato de vídeo.

La descripción se puede hacer a partir de sus características como marca, modelo, número de cabezales, etc. o a través de sus funciones como reproducción de cintas de vídeo, grabación, rebobinado, etc. Es decir, tenemos dos visiones diferentes para tratar el mismo aparato.

El primer enfoque correspondería a una colección de variables, mientras que el segundo correspondería a una colección de funciones.



El uso de las clases nos permite integrar datos y funciones dentro de la misma entidad.

El hecho de reunir el conjunto de características y funciones en el mismo contenedor facilita su interrelación, así como su aislamiento del resto del código fuente. En el ejemplo del aparato de vídeo, la reproducción de una cinta implica, una vez puesta la cinta, la acción de unos motores que hacen que la cinta se vaya desplazando por delante de unos cabezales que leen la información.

En realidad, a los usuarios el detalle del funcionamiento nos es intrascendente, sencillamente, vemos el aparato de vídeo como una caja que tiene una ranura y unos botones, y sabemos que en su interior contiene unos mecanismos que nos imaginamos bastante complejos. Pero también sabemos que a nosotros nos basta con pulsar el botón "Play".

A este concepto se le denomina **encapsulación de datos y funciones en una clase**. Las variables que están dentro de la clase reciben el

nombre de variables *miembro* o *datos miembro*, y a las funciones se las denomina *funciones miembro* o *métodos de la clase*.

Pero las clases corresponden a elementos abstractos; es decir, a ideas genéricas y en nuestra casa no disponemos de un aparato de vídeo en forma de idea, sino de un elemento real con unas características y funciones determinadas. De la misma forma, en C++ necesitaremos trabajar con los elementos concretos. A estos elementos los llamaremos *objetos*.

Declaración de una clase

La sintaxis para la declaración de una clase es utilizar la palabra reservada `class` seguida del nombre de la clase *y*, entre llaves, la lista de las variables miembro y de las funciones miembro.

```
class Perro
{
    // lista de variables miembro
    int edad;
    int altura;

    // lista de funciones miembro
    void ladrar();
};
```

La declaración de la clase no implica reserva de memoria alguna, aunque informa de la cantidad de memoria que precisará cada uno de los objetos de dicha clase.

Ejemplo

En la clase `Perro` presentada, cada uno de los objetos ocupará 8 bytes de memoria: 4 bytes para la variable miembro `edad` de tipo entero y 4 bytes para la variable miembro `altura`. Las definiciones de las funciones miembro, en nuestro caso `ladrar`, no implican reserva de espacio.

Implementación de las funciones miembro de una clase

Hasta el momento, en la clase Perro hemos declarado como miembros dos variables (edad y altura) y una función (ladrar). Pero no se ha especificado la implementación de la función.

La definición de una función miembro se hace mediante el nombre de la clase seguido por el operador de ámbito (::), el nombre de la función miembro y sus parámetros.

```
class Perro
{
    // lista de variables miembro
    int edad;
    int altura;

    // lista de funciones miembro
    void ladrar();
};

Perro:: ladrar()
{
    cout << "Guau";
}
```

Nota

Aunque ésta es la forma habitual, también es posible implementar las funciones miembro de forma *inline*. Para ello, después de la declaración del método y antes del punto y coma (;) se introduce el código fuente de la función:

```
class Perro
{
    // lista de variables miembro
    int edad;
    int altura;

    // lista de funciones miembro
    void ladrar()
    {
        cout << "Guau";
    }
};
```

Nota

```
void ladRAR()  
{ cout << "Guau"; };  
};
```

Este tipo de llamadas sólo es útil cuando el cuerpo de la función es muy reducido (una o dos instrucciones).

Funciones miembros const

En el capítulo anterior se comentó la utilidad de considerar las variables que no deberían sufrir modificaciones en el transcurso de la ejecución del programa, y su declaración mediante el especificador `const`. También se comentó la seguridad que aportaban los apuntadores `const`. Pues de forma similar se podrá definir a una función miembro como `const`.

```
void ladRAR() const;
```

Para indicar que una función miembro es `const`, sólo hay que poner la palabra reservada `const` entre el símbolo de cerrar paréntesis después del paso de parámetros y el punto y coma (`;`) final.

Al hacerlo, se le indica al compilador que esta función miembro no puede modificar al objeto. Cualquier intento en su interior de asignar una variable miembro o llamar a alguna función no constante generará un error por parte del compilador. Por tanto, es una medida más a disposición del programador para asegurar la coherencia de las líneas de código fuente.

Declaración de un objeto

Así como una clase se puede asimilar como un nuevo tipo de datos, un objeto sólo corresponde a la definición de un elemento de dicho tipo. Por tanto, la declaración de un objeto sigue el mismo modelo:

```
unsigned int numeroPulgas; // variable tipo unsigned int  
Perro sultan; // objeto de la clase Perro.
```



Un objeto es una instancia individual de una clase.

4.3.2. Acceso a objetos

Para acceder a las variables y funciones miembro de un objeto se utiliza el operador punto (.), es decir, poner el nombre del objeto seguido de punto y del nombre de la variable o función que se desea.

En el apartado anterior donde se ha definido un objeto sultan de la clase Perro, si se desea inicializar la edad de sultán a 4 o llamar a su función ladurar(), sólo hay que referirse a lo siguiente:

```
Sultan.edad = 4;
Sultan.ladurar();
```

No obstante, una de las principales ventajas proporcionadas por las clases es que sólo son visibles aquellos miembros (tanto datos como funciones) que nos interesa mostrar. Por este motivo, si no se indica lo contrario, los miembros de una clase sólo son visibles desde las funciones de dicha clase. En este caso, diremos que dichos miembros son **privados**.

Nota

`protected` corresponde a un caso más específico y se estudiará en el apartado "Herencia".

Para poder controlar el ámbito de los miembros de una clase, se dispone de las siguientes palabras reservadas: `public`, `private` y `protected`.

Cuando se declara un miembro (tanto variables como funciones) como `private` se le está indicando al compilador que el uso de esta variable es privado y está restringido al interior de dicha clase. En cambio, si se declara como `public`, es accesible desde cualquier lugar donde se utilicen objetos de dicha clase.

En el código fuente, estas palabras reservadas se aplican con forma de etiqueta delante de los miembros del mismo ámbito:

```
class Perro
{
    public:
        void ladurar();
```

```
private:  
    int edad;  
    int altura;  
};
```

Nota

Se ha declarado la función ladrar() como pública permitiendo el acceso desde fuera de la clase, pero se han mantenido ocultos los valores de edad y altura al declararlos como privados.

Veámoslo en la implementación de un programa en forma completa:

```
#include <iostream>  
class Perro  
{  
public:  
    void ladrar() const  
    { cout << "Guau"; }  
  
private:  
    int edad;  
    int altura;  
};  
  
int main()  
{  
    Perro sultan;  
  
    //Error compilación. Uso variable privada  
    sultan.edad = 4;  
    cout << sultan.edad;  
}
```

En el bloque main se ha declarado un objeto sultan de la clase Perro. Posteriormente, se intenta asignar a la variable miembro edad el valor de 4. Como dicha variable no es pública, el compilador da error indicando que no se tiene acceso a ella. Igualmente, nos mostraría un error de compilación similar para la siguiente fila. Una solución en este caso sería declarar la variable miembro edad como public.

Privacidad de los datos miembro

El hecho de declarar una variable miembro como pública limita la flexibilidad de las clases, pues una modificación del tipo de la variable afectará a los diferentes lugares del código donde se utilicen dichos valores.



Una regla general de diseño recomienda mantener los datos miembro como privados, y manipularlos a través de funciones públicas de acceso donde se obtiene o se le asigna un valor.

En nuestro ejemplo, se podría utilizar las funciones `obtenerEdad()` y `asignarEdad()` como métodos de acceso a los datos. Declarar la variable `edad` como privada nos permitiría cambiar el tipo en entero a byte o incluso sustituir el dato por la fecha de nacimiento. La modificación se limitaría a cambiar el código fuente en los métodos de acceso, pero continuaría de forma transparente fuera de la clase, pues se puede calcular la edad a partir de la fecha actual y la fecha de nacimiento o asignar una fecha de nacimiento aproximada a partir de un valor para la edad.

```
class Perro
{
    public:

        Perro(int, int); // método constructor
        Perro(int);      //
        Perro();          //
        ~Perro();         // método destructor

        void asignarEdad(int); // métodos de acceso
        int obtenerEdad();    //
        void asignarAltura(int); //
        int obtenerAltura();   //

        void ladrar();           // métodos de la clase
```

```
private:  
    int edad;  
    int altura;  
};  
  
Perro:: ladrar()  
{ cout << "Guau"; }  
  
void Perro:: asignarAltura (int nAltura)  
{ altura = nAltura; }  
  
int Perro:: obtenerAltura (int nAltura)  
{ return (altura); }  
  
void Perro:: asignarEdad (int nEdad)  
{ edad = nEdad; }  
  
int Perro:: obtenerEdad ()  
{ return (edad); }
```

El apuntador this

Otro aspecto a destacar de las funciones miembro es que siempre tienen acceso al propio objeto a través del apuntador `this`.

De hecho, la función miembro `obtenerEdad` también se podría expresar de la siguiente forma:

```
int Perro:: obtenerEdad ()  
{ return (this->edad); }
```

Visto de este modo, parece que su importancia sea escasa. No obstante, poder referirse al objeto sea como apuntador `this` o en su forma desreferenciada (`*this`) le da mucha potencia. Veremos ejemplos más avanzados al respecto cuando se estudie la sobrecarga de operadores.

4.3.3. Constructores y destructores de objetos

Generalmente, cada vez que se define una variable, después se inicializa. Ésta es una práctica correcta en la que se intenta prevenir resultados imprevisibles al utilizar una variable sin haberle asignado ningún valor previo.

Las clases también se pueden inicializar. Cada vez que se crea un nuevo objeto, el compilador llama a un método específico de la clase para inicializar sus valores que recibe el nombre de **constructor**.



El constructor siempre recibe el nombre de la clase, sin valor de retorno (ni siquiera `void`) y puede tener parámetros de inicialización.

```
Perro::Perro()
{
    edad = 0;
}
```

o

```
Perro::Perro(int nEdad) // Nueva edad del perro
{
    edad = nEdad;
}
```

En caso de que no se defina específicamente el constructor en la clase, el compilador utiliza el **constructor predeterminado**, que consiste en el nombre de la clase, sin ningún parámetro y tiene un bloque de instrucciones vacío. Por tanto, no hace nada.

```
Perro::Perro()
{ }
```

Esta característica suena desconcertante, pero permite mantener el mismo criterio para la creación de todos los objetos.

En nuestro caso, si deseamos inicializar un objeto de la clase Perro con una edad inicial de 1 año, utilizamos la siguiente definición del constructor:

```
Perro(int nuevaEdad);
```

De esta manera, la llamada al constructor sería de la siguiente forma:

```
Perro sultan(1);
```

Si no hubiera parámetros, la declaración del constructor a utilizar dentro de la clase sería la siguiente:

```
Perro();
```

La declaración del constructor en main o cualquier otra función del cuerpo del programa quedaría del modo siguiente:

```
Perro sultan();
```

Pero en este caso especial se puede aplicar una excepción a la regla que indica que todas las llamadas a funciones van seguidas de paréntesis aunque no tengan parámetros. El resultado final sería el siguiente:

```
Perro sultan;
```

El fragmento anterior es una llamada al constructor Perro() y coincide con la forma de declaración presentada inicialmente y ya conocida.

De la misma forma, siempre que se declara un método constructor, se debería declarar un método **destructor** que se encarga de limpiar cuando ya no se va a usar más el objeto y liberar la memoria utilizada.



El destructor siempre va antecedido por una tilde (~), tiene el nombre de la clase, y no tiene parámetros ni valor de retorno.

```
~Perro();
```

En caso de que no definamos ningún destructor, el compilador define un **destructor predeterminado**. La definición es exactamente la misma, pero siempre tendrá un cuerpo de instrucciones vacío:

```
Perro::~Perro()
{ }
```

Incorporando las nuevas definiciones al programa, el resultado final es el siguiente:

```
#include <iostream>

class Perro
{
public:
    Perro(int edad); // constructor con un parámetro
    Perro();          // constructor predeterminado
    ~Perro();         // destructor
    void ladrar();

private:
    int edad;
    int altura;
};

Perro:: ladrar()
{ cout << "Guau"; }

int main()
{
    Perro sultan(4); // Inicializando el objeto
                      // con una edad de 4.
    sultan.ladrar();
}
```

El constructor de copia

El compilador, además de proporcionar de forma predeterminada a las clases un método constructor y un método destructor, también proporciona un método constructor de copia.

Cada vez que se crea una copia de un objeto se llama al constructor de copia. Esto incluye los casos en que se pasa un objeto como parámetro por valor a una función o se devuelve dicho objeto como retorno de la función. El propósito del constructor de copia es realizar una copia de los datos miembros del objeto a uno nuevo. A este proceso también se le llama *copia superficial*.

Este proceso, que generalmente es correcto, puede provocar fuertes conflictos si entre las variables miembros a copiar hay apuntadores. El resultado de la copia superficial haría que dos apuntadores (el del objeto original y el del objeto copia) apunten a la misma dirección de memoria: si alguno de ellos liberara dicha memoria, provocaría que el otro apuntador, al no poder percatarse de la operación, se quedara apuntando a una posición de memoria perdida generando una situación de resultados impredecibles.

La solución en estos casos pasa por sustituir la copia superficial por una **copia profunda** en la que se reservan nuevas posiciones de memoria para los elementos tipo apuntador y se les asigna el contenido de las variables apuntadas por los apuntadores originales.

La forma de declarar dicho constructor es la siguiente:

```
Perro :: Perro (const Perro & unperro);
```

En esta declaración se observa la conveniencia de pasar el parámetro como referencia constante pues el constructor no debe alterar el objeto.

La utilidad del constructor de copia se observa mejor cuando alguno de los atributos es un apuntador. Por este motivo y para esta prueba cambiaremos el tipo de edad a apuntador a entero. El resultado final sería el siguiente:

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

```
class Perro
{
public:
    Perro();                                // constructor predeterminado
    ~Perro();                                 // destructor
    Perro(const Perro & rhs);                // constructor de copia
    int obtenerEdad();                      // método de acceso

private:
    int *edad;
};

Perro :: obtenerEdad()
{ return (*edad) }

Perro :: Perro ()                         // Constructor
{
    edad = new int;
    * edad = 3;
}

Perro :: ~Perro ()                        // Destructor
{
    delete edad;
    edad = NULL;
}

Perro :: Perro (const Perro & rhs) // Constructor de copia
{
    edad = new int;                  // Se reserva nueva memoria
    *edad = rhs.obtenerEdad();      // Copia el valor edad
                                    // en la nueva posición

}

int main()
{
    Perro sultan(4);                 // Inicializando con edad 4.
}
```

Inicializar valores en los métodos constructores

Hay otra forma de inicializar los valores en un método constructor de una forma más limpia y eficiente que consiste en interponer dicha inicialización entre la definición de los parámetros del método y la llave que indica el inicio del bloque de código.

```
Perro:: Perro () :  
    edad (0),  
    altura (0)  
{ }
```

```
Perro:: Perro (int nEdad, int nAltura):  
    edad (nEdad),  
    altura(nAltura)  
{ }
```

Tal como se ve en el fragmento anterior, la inicialización consiste en un símbolo de dos puntos (:) seguido de la variable a inicializar y, entre paréntesis, el valor que se le desea asignar. Dicho valor puede corresponder tanto a una constante como a un parámetro de dicho constructor. Si hay más de una variable a inicializar, se separan por comas (,).

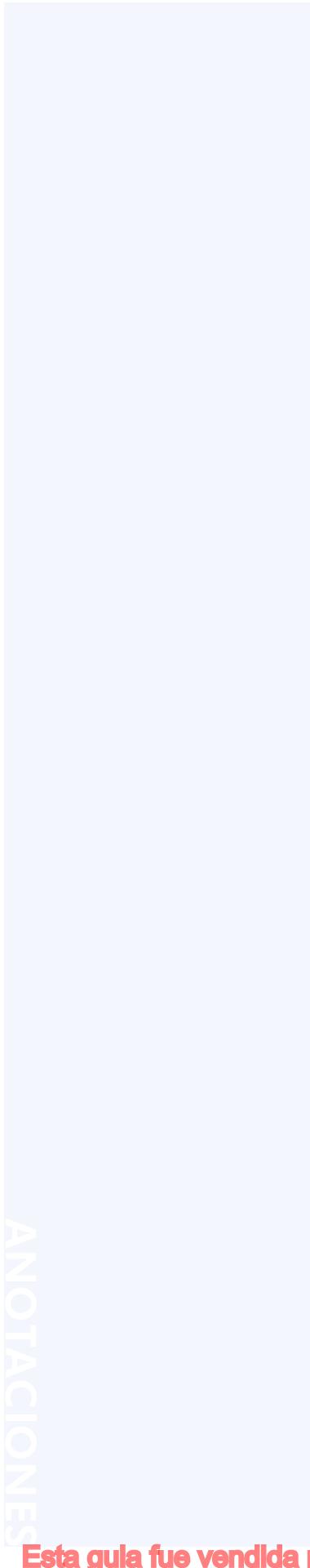
Variables miembro y funciones miembro estáticas

Hasta el momento, cuando nos hemos referido a las clases y a los objetos los hemos situado en planos diferentes: las clases describían entes abstractos, y los objetos describían elementos creados y con valores concretos.

No obstante, hay momentos en que los objetos necesitan referirse a un atributo o a un método común con los demás objetos de la misma clase.

Ejemplo

Si estamos creando una clase Animales, nos puede interesar conservar en alguna variable el número total de



perros que se han creado hasta el momento, o hacer una función que nos permita contar los perros aunque todavía no se haya creado ninguno.

La solución es preceder la declaración de las variables miembro o de las funciones miembro con la palabra reservada `static`. Con ello le estamos indicando al compilador que dicha variable, o dicha función, se refiere a la clase en general y no a ningún objeto en concreto. También se puede considerar que se está compartiendo este dato o función con todas las instancias de dicho objeto.

En el siguiente ejemplo se define una variable miembro y una función miembro estáticas:

```
class Perro {  
    // ...  
    static int numeroDePerros; //normalmente será privada  
    static int cuantosPerros() { return numeroDePerros; }  
};
```

Para acceder a ellos se puede hacer de dos formas:

- Desde un objeto de la clase Perro.

```
Perro sultan = new Perro();  
sultan.numeroDePerros; //normalmente será privada  
sultan.cuantosPerros();
```

- Utilizando el identificador de la clase sin definir ningún objeto.

```
Perro::numeroDePerros; //normalmente será privada  
Perro::cuantosPerros();
```

Pero hay que tener presente un aspecto importante: las variables y las funciones miembro `static` se refieren siempre a la clase y no a ningún objeto determinado, con lo cual el objeto `this` no existe.

Como consecuencia, en las funciones miembro estáticas no se podrá hacer referencia ni directa ni indirectamente al objeto `this` y:

- Sólo podrán hacer llamadas a funciones estáticas, pues las funciones no estáticas siempre esperan implícitamente a dicho objeto como parámetro.
- Sólo podrán tener acceso a variables estáticas, porque a las variables no estáticas siempre se accede a través del mencionado objeto.
- No se podrán declarar dichas funciones como `const` pues ya no tiene sentido.

4.3.4. Organización y uso de bibliotecas en C++

Hasta el momento se ha incluido en el mismo archivo la definición de la clase y el código que la utiliza, pero ésta no es la organización aconsejada si se desea reutilizar la información.

Se recomienda dividir el código fuente de la clase en dos ficheros separando la definición de la clase y su implementación:

- El fichero de cabecera incorpora la definición de la clase. La extensión de dichos ficheros puede variar entre diversas posibilidades, y la decisión final es arbitraria.
- El fichero de implementación de los métodos de la clase y que contiene un `include` del fichero de cabecera. La extensión utilizada para dichos ficheros también puede variar.

Posteriormente, cuando se desee reutilizar esta clase, bastará con incluir en el código fuente una llamada al fichero de cabecera de la clase.

En nuestro ejemplo, la implementación se encuentra en el fichero `perro.cpp`, que hace un `include` del fichero de cabecera `perro.hpp`.

Nota

Las extensiones utilizadas de forma más estándar son `.hpp` (más utilizadas en entornos Windows), `.H` y `.hxx` (más utilizadas en entornos Unix) o incluso `.h` (al igual que en C).

Nota

Las extensiones utilizadas de forma más estándar son `.cpp` (más frecuentes en entornos Windows) `.C` y `.cxx` (más utilizadas en entornos Unix).

Fichero perro.hpp (fichero de cabecera de la clase)

```

class Perro
{
    public:

        Perro(int edad);      //métodos constructores
        Perro();
        ~Perro();             //método destructor
        int obtenerEdad();   // métodos de acceso
        int asignarEdad(int);
        int asignarAltura(int);
        int obtenerAltura();
        void ladrar();        // métodos propios

    private:
        int edad;
        int altura;
};

```

Fichero perro.cpp (fichero de implementación de la clase)

```

#include <iostream>      //necesaria para el cout
#include <perro.hpp>

Perro:: ladrar()
{ cout << "Guau"; }

void Perro:: asignarAltura (int nAltura)
{ altura = nAltura; }

int Perro:: obtenerAltura (int nAltura)
{ return (altura); }

void Perro:: asignarEdad (int nEdad)
{ edad = nEdad; }

int Perro:: obtenerEdad ()
{ return (edad); }

```

Fichero ejemplo.cpp

```
#include <perro.hpp>
int main()
{
    //Inicializando el objeto con edad 4.
    Perro sultan (4);
    sultan.ladrar();
}
```

Las bibliotecas estándar

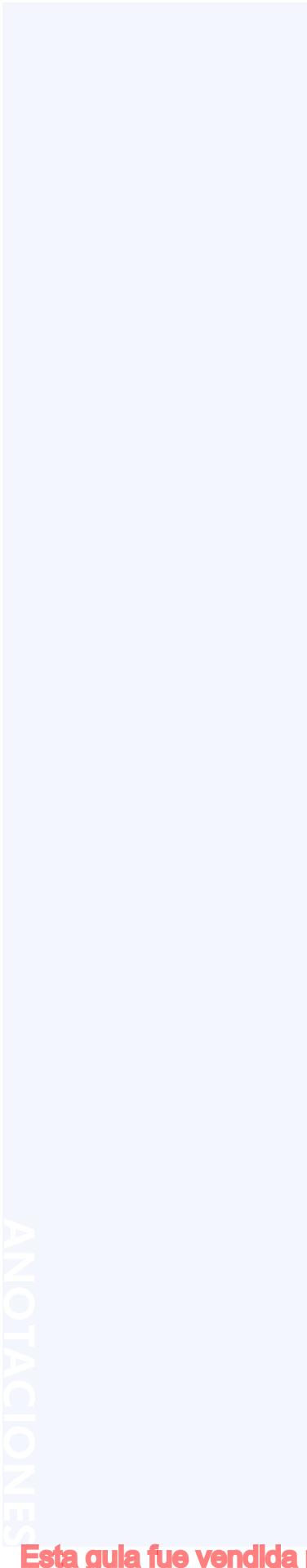
Los compiladores suelen incluir una serie de funciones adicionales para que el programador las utilice. En el caso de GNU, proporciona una biblioteca estándar de funciones y objetos para los programadores de C++ denominada `libstdc++`.

Esta librería proporciona las operaciones de entrada/salida con *streams*, *strings*, vectores, listas, algoritmos de comparación, operaciones matemáticas y algoritmos de ordenación entre muchas otras operaciones.

Uso de la biblioteca STL

C++ ha incorporado un nivel más de abstracción al introducir las **plantillas**, que también se conocen como *tipos parametrizados*. No es objeto de este curso el desarrollar este tema, pero la potencia que proporciona al C++ la inclusión de STL (Biblioteca Estándar de Plantillas) nos obliga a hacer una breve reseña de cómo utilizarla.

La idea básica de las plantillas es simple: cuando se implementa una operación general con un objeto (por ejemplo, una lista de perros) definiremos las diferentes operaciones de manipulación de una lista sobre la base de la clase Perro. Si posteriormente se desea realizar una operación similar con otros objetos (una lista de gatos), el código resultante para el mantenimiento de la lista es similar pero definiendo los elementos en base a la clase Gato. Por tanto, seguramente, nuestra forma de actuar sería hacer un copiar y pegar y modificar el bloque copiado para trabajar con la nueva clase deseada. No obstante, este proceso se repite



cada vez que se desea implementar una nueva lista de otro tipo de objetos (por ejemplo, una lista de caballos).

Además, cada vez que se deseara modificar una operación de las listas, se debería cambiar cada una de sus personalizaciones. Por tanto, rápidamente se intuye que ésta implementación no sería eficiente.

La forma eficiente de hacerlo es generar un código genérico que realice las operaciones de las listas para un tipo que se le puede indicar con posterioridad. Este código genérico es el de las plantillas o tipos parametrizados.

Después de este breve comentario, se intuye la eficiencia y la potencia de esta nueva característica y también su complejidad, que, como ya hemos comentado, sobrepasa el objetivo de este curso. No obstante, para un dominio avanzado del C++ este tema es imprescindible y se recomienda la consulta de otras fuentes bibliográficas para completar estos conocimientos.

No obstante, mientras que la definición e implementación de una librería de plantillas puede llegar a adquirir una gran complejidad, el uso de la Librería Estándar de Plantillas (STL) es accesible.

En el siguiente ejemplo, se trabaja con la clase `set` que define un conjunto de elementos. Para ello, se ha incluido la librería `set` incluida en la STL.

```
#include <iostream>
#include <set>

int main()
{
    // define un conjunto de enteros <int>
    set<int> setNumeros;

    // añade tres números al conjunto de números
    setNumeros.insert(123);
    setNumeros.insert(789);
    setNumeros.insert(456);

    // muestra cuántos números tiene
    // el conjunto de números
```

```
cout << "Conjunto números: "
    << setNumeros.size() << endl;

// se repite el proceso con un conjunto de letras
//define conjunto de caracteres <char>
set<char> setLetras;

setLetras.insert('a');
setLetras.insert('z');
cout << "Conjunto letras: "
    << setLetras.size() << endl;
return 0;
}
```

En este ejemplo, se han definido un conjunto de números y un conjunto de letras. Para el conjunto de números, se ha definido la variable `setNumeros` utilizando la plantilla `set` indicándole que se utilizarán elementos de tipo `<int>`. Este conjunto define varios métodos, entre los cuales el de insertar un elemento al conjunto (`.insert`) o contar el número de elementos (`.size`). Para el segundo se ha definido la variable `setLetras` también utilizando la misma plantilla `set` pero ahora con elementos tipo `<char>`.

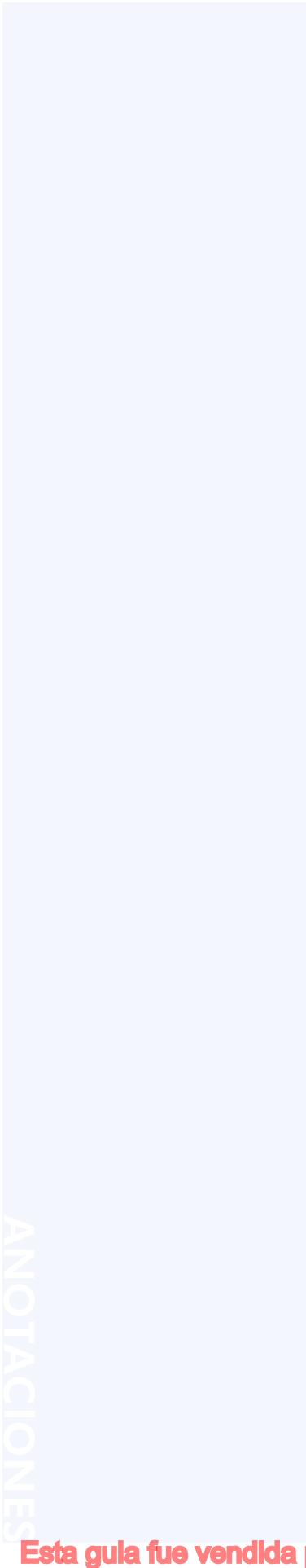
La salida del programa nos muestra el número de elementos introducidos en el conjunto de números y, posteriormente, el número de elementos introducidos en el conjunto de letras.

4.4. Diseño de programas orientados a objetos

La potencia del paradigma de la programación orientada a objetos no radica sólo en la definición de clases y objetos, sino en todas las consecuencias que implican y que se pueden implementar en el lenguaje de programación.

En esta unidad se estudiarán las principales propiedades de este paradigma:

- La homonimia
- La herencia
- El polimorfismo



Una vez asimilado el alcance del cambio de paradigma, se proporcionarán nuevas reglas para el diseño de aplicaciones.

4.4.1. La homonimia

Tal como su definición indica, homonimia se refiere al uso de dos o más sentidos (en nuestro caso, léase operaciones) con el mismo nombre.

En programación orientada a objetos, hablaremos de homonimia al utilizar el mismo nombre para definir la misma operación diversas veces en diferentes situaciones aunque, normalmente, con la misma idea de fondo. El ejemplo más claro es definir con el mismo nombre las operaciones que básicamente cumplen el mismo objetivo pero para diferentes objetos.

En nuestro caso, diferenciaremos entre sus dos formas principales: la **homonimia (o sobrecarga) de funciones** y la **homonimia de operadores**.

Sobrecarga de funciones y métodos

La sobrecarga de funciones se estudió anteriormente como una de las mejoras que proporciona más flexibilidad a C++ respecto de C, y es una de las características más utilizadas dentro de la definición de las clases.

Los constructores son un caso práctico de sobrecarga de métodos. Para cada clase, se dispone de un constructor por defecto que no tiene parámetros y que inicializa los objetos de dicha clase.

En nuestro ejemplo,

```
Perro:::Perro()
{ }
```

O, tal como se vio, se puede dar el caso de que siempre se desee inicializar dicha clase a partir de una edad determinada, o de una edad y una altura determinadas:

```
Perro::Perro(int nEdad) // Nueva edad del perro
{ edad = nEdad; }
```

```
Perro::Perro(int nEdad, int nAltura) // Nueva defin.
{
    edad = nEdad;
    altura = nAltura;
}
```

En los tres casos, se crea una instancia del objeto `Perro`. Por tanto, básicamente están realizando la misma operación aunque el resultado final sea ligeramente diferente.

Del mismo modo, cualquier otro método o función de una clase puede ser sobrecargado.

Sobrecarga de operadores

En el fondo, un operador no es más que una forma simple de expresar una operación entre uno o más operandos, mientras que una función nos permite realizar operaciones más complejas.

Por tanto, la sobrecarga de operadores es una forma de simplificar las expresiones en las operaciones entre objetos.

En nuestro ejemplo, se podría definir una función para incrementar la edad de un objeto `Perro`.

```
Perro Perro::incrementarEdad()
{
    ++edad;
    return (*this);
}
// la llamada resultante sería Sultan.IncrementarEdad()
```

Aunque la función es muy simple, podría resultar un poco incómoda de utilizar. En este caso, podríamos considerar sobrecargar el operador ++ para que, al aplicarlo sobre un objeto Perro, se incremente automáticamente su edad.

La sobrecarga del operador se declara de la misma forma que una función. Se utiliza la palabra reservada operator, seguida del operador que se va a sobrecargar. En el caso de las funciones de operadores unitarios, no llevan parámetros (a excepción del incremento o decremento postfijo que utilizan un entero como diferenciador).

```
#include <iostream>
class Perro
{
public:
    Perro();
    Perro(int nEdad);
    ~Perro();
    int obtenerEdad();
    const Perro & operator++(); // Operador ++
    const Perro & operator++(int); // Operador i++

private:
    int edad;
};

Perro::Perro():
edad(0)
{ }

Perro::Perro(int nEdad):
edad(nEdad)
{ }

int Perro::obtenerEdad()
{ return (edad); }

const Perro & Perro::operator++()
{ }
```

```
    ++edad;
    return (*this);
}

const Perro & Perro::operator++(int x)
{
    Perro temp = *this;
    ++edad;
    return (temp);
}

int main()
{
    Perro sultan(3);
    cout << "Edad de Sultan al comenzar el programa \n " ;
    cout << sultan.obtenerEdad() << endl;

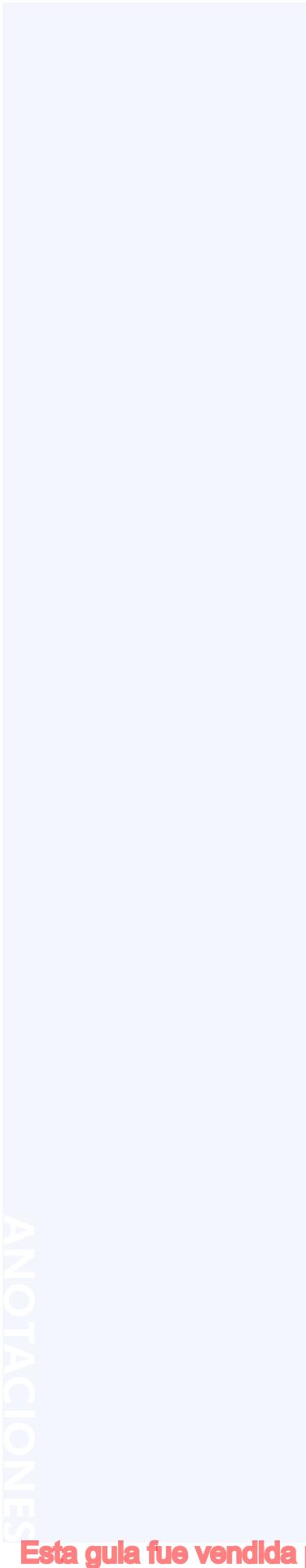
    ++sultan;
    cout << "Edad de Sultan después de un aniversario \n ";
    cout << sultan.obtenerEdad() << endl;

    sultan++;
    cout << "Edad de Sultan después de otro aniversario\n";
    cout << sultan.obtenerEdad();
}
```

En la declaración de sobrecarga de los operadores, se observa cómo se devuelve una referencia `const` a un objeto tipo `Perro`. De esta forma, se protege la dirección del objeto original de cualquier cambio no deseado.

También es posible observar que las declaraciones para el operador postfijo y prefijo son prácticamente iguales, y sólo cambia el tipo de argumento. Para diferenciar ambos casos, se estableció la convención de que el operador postfijo tuviera en la declaración un parámetro tipo `int` (aunque este parámetro no se usa).

```
const Perro & operator++();      // Operador ++i
const Perro & operator++(int);   // Operador i++
```



En la implementación de ambas funciones, también hay diferencias significativas:

- En el caso del operador prefijo, se procede a incrementar el valor de la edad del objeto y se retorna el objeto modificado a través del apuntador `this`.

```
const Perro & Perro::operator++()
{
    ++edad;
    return (*this);
}
```

- En el caso del operador postfijo, se requiere devolver el valor del objeto anterior a su modificación. Por este motivo, se establece una variable temporal que recoge el objeto original, se procede a su modificación y se retorna la variable temporal.

```
const Perro & Perro::operator++(int x)
{
    Perro temp = *this;
    ++edad;
    return (temp);
}
```

La definición de la sobrecarga del operador suma, que es un operador binario, sería como sigue:

```
// En este caso, la suma de dos objetos tipo Perro
// no tiene NINGÚN SENTIDO LÓGICO.
// SÓLO a efectos de mostrar como sería
// la declaración del operador, se ha considerado
// como resultado "possible"
// retornar el objeto Perro de la izquierda
// del operador suma con la edad correspondiente
// a la suma de las edades de los dos perros.
```

```
const Perro & Perro::operator+(const Perro & rhs)
{
    Perro temp = *this;
    temp.edad = temp.edad + rhs.edad;
    return (temp);
}
```

Nota

Dado lo desconcertante de la lógica empleada en el ejemplo anterior, queda claro también que no se debe abusar de la sobrecarga de operadores. Sólo debe utilizarse en aquellos casos en que su uso sea intuitivo y ayude a una mayor legibilidad del programa.

4.4.2. La herencia simple

Los objetos no son elementos aislados. Cuando se estudian los objetos, se establecen relaciones entre ellos que nos ayudan a comprenderlos mejor.

Ejemplo

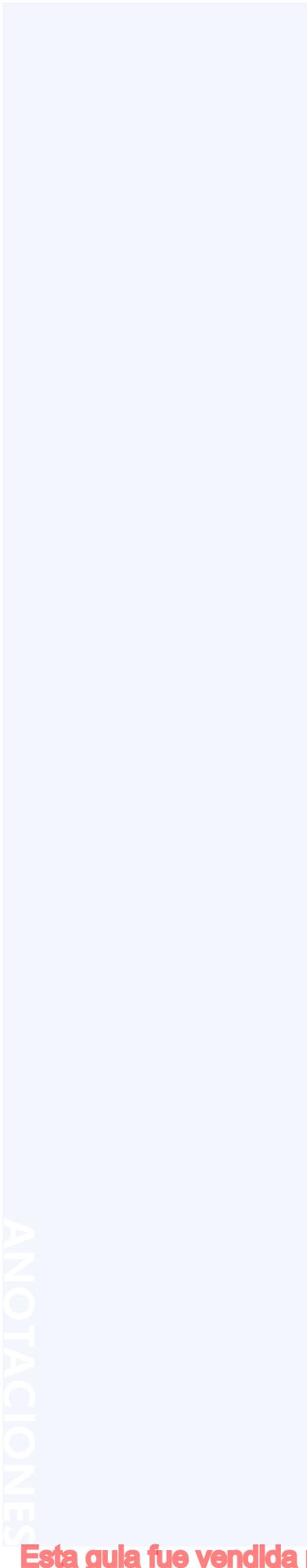
Un perro y un gato son objetos diferentes, pero tienen una cosa en común: los dos son mamíferos. También lo son los delfines y las ballenas, aunque se muevan en un entorno muy diferente, aunque no los tiburones, que entrarían dentro de la categoría de peces. ¿Qué tienen todos estos objetos en común? Todos son animales.

Se puede establecer una jerarquía de objetos, en la cual un perro es un mamífero, un mamífero es un animal, un animal es un ser vivo, etc. Entre ellos se establece la relación es *un*. Este tipo de relación es muy habitual: un guisante es una verdura, que es un tipo de vegetal; un disco duro es una unidad de almacenamiento, que a su vez es un tipo de componente de un ordenador.

Al decir que un elemento es un tipo de otro, establecemos una especialización: decimos que el elemento tiene unas características generales compartidas y otras propias.

La herencia es una manera de representar las características que se reciben del nivel más general.

La idea de perro hereda todas las características de mamífero; es decir, mama, respira con pulmones, se mueve, etc., pero también



presenta unas características concretas propias como ladrar o mover la cola. A su vez, los perros también se pueden dividir según su raza: pastor alemán, caniche, doberman, etc. Cada uno con sus particularidades, pero heredando todas las características de los perros.

Para representar estas relaciones, C++ permite que una clase derive de otra. En nuestro caso, la clase Perro deriva de la clase Mamífero. Por tanto, en la clase Perro no será necesario indicarle que mama, ni que respira con pulmones, ni que se mueve. Al ser un mamífero, hereda dichas propiedades además de aportar datos o funciones nuevas.

De la misma forma, un Mamífero se puede implementar como una clase derivada de la clase Animal, y a su vez hereda información de dicha clase como la de pertenecer a los seres vivos y moverse.

Dada la relación entre la clase Perro y la clase Mamífero, y entre la clase Mamífero y la clase Animal, la clase Perro también heredará la información de la clase Animal: ¡Un perro es un ser vivo que se mueve!

Implementación de la herencia

Para expresar esta relación en C++, en la declaración de una clase después de su nombre se pone dos puntos (:), el tipo de derivación (pública o privada) y el nombre de la clase de la cual deriva:

```
class Perro : public Mamifero
```

El tipo de derivación, que de momento consideraremos pública, se estudiará con posterioridad. Ahora, enfocaremos nuestra atención sobre cómo queda la nueva implementación:

```
#include <iostream>

enum RAZAS { PASTOR_ALEMAN, CANICHE,
              DOBERMAN, YORKSHIRE };

class Mamifero
```

```
{  
public:  
    Mamifero();           // método constructor  
  
    ~Mamifero();          // método destructor  
void asignarEdad(int nEdad)  
{ edad = nEdad } ;        // métodos de acceso  
int obtenerEdad() const  
{ return (edad) } ;  
protected:  
    int edad;  
};  
  
class Perro : public Mamifero  
{  
public:  
    Perro();           // método constructor  
  
    ~Perro();          // método destructor  
void asignarRaza(RAZAS); // métodos de acceso  
int obtenerRaza() const;  
void ladrar() const  
{ cout << "Guau " ; } ; // métodos propios  
private:  
    RAZAS raza;  
};  
  
class Gato : public Mamifero  
{  
public:  
    Gato();   // método constructor  
    ~Gato(); // método destructor  
void maullar() const  
{ cout << "Miauuu"; } // métodos propios  
};
```

En la implementación de la clase Mamífero, en primer lugar se han definido su constructor y destructor por defecto. Dado que el dato miembro edad que disponíamos en la clase Perro no es una característica exclusiva de esta clase, sino que todos los mamíferos tienen una edad, se ha trasladado el dato miembro edad y sus métodos de acceso (obtenerEdad y asignarEdad) a la nueva clase.

Nota

Cabe destacar que la declaración de tipo `protected` para el dato miembro `edad` permite que sea accesible desde las clases derivadas. Si se hubiera mantenido la declaración de `private`, no hubieran podido verlo ni utilizarlo otras clases, ni siquiera las derivadas. Si se hubiese declarado `public`, habría sido visible desde cualquier objeto, pero se recomienda evitar esta situación.

Dentro de la clase `Perro`, hemos añadido como dato nuevo su raza, y hemos definido sus métodos de acceso (`obtenerRaza` y `asignarRaza`), así como su constructor y destructor predefinido. Se continúa manteniendo el método `ladrar` como una función de la clase `Perro`: los demás mamíferos no ladran.

Constructores y destructores de clases derivadas

Al haber hecho la clase `Perro` derivada de la clase `Mamífero`, en esencia, los objetos `Perro` son objetos `Mamífero`. Por tanto, en primer lugar llama a su constructor base, con lo que se crea un `Mamífero`, y posteriormente se completa la información llamando al constructor de `Perro`.

Datos Mamífero
Datos Perro

A la hora de destruir un objeto de la clase `Perro`, el proceso es el inverso: en primer lugar se llama al destructor de `Perro`, liberando así su información específica, y, posteriormente, se llama al destructor de `Mamífero`.

Ejemplo

Siguiendo con el ejemplo, con la clase `Perro`, además de inicializar su raza, también podemos inicializar su edad (como es un mamífero, tiene una edad).

Hasta el momento, sabemos inicializar los datos de un objeto de la clase que estamos definiendo, pero también es habitual que en el constructor de una clase se desee inicializar datos pertenecientes a su clase base.

El constructor de la clase `Mamífero` ya realiza esta tarea, pero es posible que sólo nos interese hacer esta operación para los perros y no

para todos los animales. En este caso, podemos realizar la siguiente inicialización en el constructor de la clase Perro:

```
Perro :: Perro()
{
    asignarRaza(CANICHE); // Acceso a raza
    asignarEdad(3);      // Acceso a edad
};
```

Al ser `asignarEdad` un método público perteneciente a su clase base, ya lo reconoce automáticamente.

En el ejemplo anterior, hemos definido dos métodos `ladrar` y `maullar` para las clases `Perro` y `Gato` respectivamente. La gran mayoría de los animales tienen la capacidad de emitir sonidos para comunicarse; por tanto, se podría crear un método común que podríamos llamar `emitirSonido`, al cual podríamos dar un valor general para todos los animales, excepto para los perros y los gatos, caso en que se podría personalizar:

```
#include <iostream>

enum RAZAS { PASTOR_ALEMAN, CANICHE,
              DOBERMAN, YORKSHIRE };

class Mamifero
{
public:
    Mamifero();           // método constructor
    ~Mamifero();          // método destructor
    void asignarEdad(int nEdad)
    { edad = nEdad; } ;   // métodos de acceso
    int obtenerEdad() const
    { return (edad); };
    void emitirSonido() const
    { cout << "Sonido"; };
protected:
    int edad;
};

class Perro : public Mamifero
```

```

{

    public:
        Perro();                      // método constructor
        ~Perro();                     // método destructor
        void asignarRaza(RAZAS); // métodos de acceso
        int obtenerRaza() const;
        void emitirSonido() const
        { cout << "Guau"; }          // métodos propios

    private:
        RAZAS raza;
};

class Gato : public Mamifero
{
    public:
        Gato(); // método constructor
        ~Gato(); // método destructor
        void emitirSonido () const
        { cout << "Miauuu"; } // métodos propios
};

int main()
{
    Perro unperro;
    Gato ungato;
    Mamifero unmamifero;

    unmamifero.emitirSonido; // Resultado: "Sonido"
    unperro.emitirSonido;    // Resultado: Guau
    ungato.emitirSonido;    // Resultado: Miau
}

```

El método `emitirSonido` tendrá un resultado final según lo llame un Mamífero, un Perro o un Gato. En el caso de las clases derivadas (Perro y Gato) se dice que se ha **redefinido la función** miembro de la clase base. Para ello, la clase derivada debe definir la función base con la misma firma (tipo de retorno, parámetros y sus tipos, y el especificador `const`).



Hay que diferenciar la redefinición de funciones de la sobrecarga de funciones. En el primer caso, se trata de funciones con el mismo nombre y la misma signatura en clases diferentes (la clase base y la clase derivada). En el segundo caso, son funciones con el mismo nombre y diferente signatura, que están dentro de la misma clase.

El resultado de la redefinición de funciones es la **ocultación** de la función base para las clases derivadas. En este aspecto, hay que tener en cuenta que si se redefine una función en una clase derivada, quedarán ocultas también todas las sobrecargas de dicha función de la clase base. Un intento de usar alguna función ocultada generará un error de compilación. La solución consistirá en realizar en la clase derivada las mismas sobrecargas de la función existentes en la clase base.

No obstante, si se desea, todavía se puede acceder al método ocultado anteponiendo al nombre de la función el nombre de la clase base seguido del operador de ámbito (::).

```
unperro.Mamifero::emitirSonido();
```

4.4.3. El polimorfismo

En el ejemplo que se ha utilizado hasta el momento, sólo se ha considerado que la clase Perro (clase derivada) hereda los datos y métodos de la clase Mamifero (clase base). De hecho, la relación existente es más fuerte.

C++ permite el siguiente tipo de expresiones:

```
Mamifero *ap_unmamifero = new Perro;
```

En estas expresiones a un apuntador a una clase Mamifero no le asignamos directamente ningún objeto de la clase Mamifero, sino

que le asignamos un objeto de una clase diferente, la clase Perro, aunque se cumple que Perro es una clase derivada de Mamifero.

De hecho, ésta es la naturaleza del polimorfismo: un mismo objeto puede adquirir diferentes formas. A un apuntador a un objeto mamífero se le puede asignar un objeto mamífero o un objeto de cualquiera de sus clases derivadas.

Además, como se podrá comprobar más adelante, esta asignación se podrá hacer en tiempo de ejecución.

Funciones virtuales

Con el apuntador que se presenta a continuación, se podrá llamar a cualquier método de la clase Mamifero. Pero lo interesante sería que, en este caso concreto, llamaría a los métodos correspondientes de la clase Perro. Esto nos lo permiten las **funciones o métodos virtuales**:

```
#include <iostream>

enum RAZAS { PASTOR_ALEMAN, CANICHE,
              DOBERMAN, YORKSHIRE };

class Mamifero
{
public:
    Mamifero();                      // método constructor
    virtual ~Mamifero();             // método destructor
    virtual void emitirSonido() const
        { cout << "emitir un sonido" << endl; };
protected:
    int edad;
};

class Perro : public Mamifero
{
public:
    Perro();                         // método constructor
    virtual ~Perro();                // método destructor
    int obtenerRaza() const;
```

```
virtual void emitirSonido() const
    { cout << "Guau" << endl; }; // métodos propios
private:
    RAZAS raza;
};

class Gato : public Mamifero
{
public:
    Gato(); // método constructor
    virtual ~Gato(); // método destructor
    virtual void emitirSonido() const
        { cout << "Miau" << endl; }; // métodos propios
};

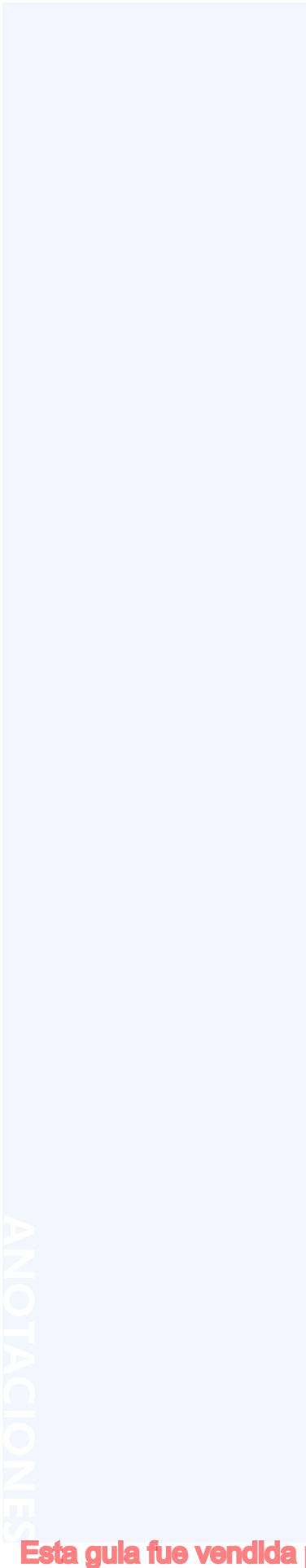
class Vaca : public Mamifero
{
public:
    Vaca(); // método constructor
    virtual ~Vaca(); // método destructor
    virtual void emitirSonido() const
        { cout << "Muuu" << endl; }; //métodos propios
};

int main()
{
    Mamifero * ap_mamiferos[3];
    int i;

    ap_mamiferos [0] = new Gato;
    ap_mamiferos [1] = new Vaca;
    ap_mamiferos [2] = new Perro;

    for (i=0; i<3 ; i++)
        ap_mamiferos[i] → emitirSonido();

    return 0;
}
```



Al ejecutar el programa, en primer lugar se declara un vector de 3 elementos tipo apuntador a Mamifero, y se inicializa a diferentes tipos de Mamifero (Gato, Vaca y Perro).

Posteriormente, se recorre este vector y se procede a llamar al método `emitirSonido` para cada uno de los elementos. La salida obtenida será:

- Miau
- Muuu
- Guau

El programa ha detectado en cada momento el tipo de objeto que se ha creado a través del `new` y ha llamado a la función `emitirSonido` correspondiente.

Esto hubiera funcionado igualmente aunque se le hubiera pedido al usuario que le indicara al programa el orden de los animales. El funcionamiento interno se basa en detectar en tiempo de ejecución el tipo del objeto al que se apunta y éste, en cierta forma, sustituye las funciones virtuales del objeto de la clase base por las que correspondan al objeto derivado.

Por todo ello, se ha definido la función miembro `emitirSonido` de la clase `Mamifero` como función virtual.

Declaración de las funciones virtuales

Al declarar una función de la clase base como virtual, implícitamente se están declarando como virtuales las funciones de las clases derivadas, por lo que no es necesaria su declaración explícita como tales. No obstante, para una mayor claridad en el código, se recomienda hacerlo.

Si la función no estuviera declarada como virtual, el programa entendería que debe llamar a la función de la clase base, independientemente del tipo de apuntador que fuera.

También es importante destacar que, en todo momento, se trata de apuntadores a la clase base (aunque se haya inicializado con un

objeto de la clase derivada), con lo cual sólo pueden acceder a funciones de la clase base. Si uno de estos apuntadores intentara acceder a una función específica de la clase derivada, como por ejemplo obtenerRaza(), provocaría un error de función desconocida. A este tipo de funciones sólo se puede acceder directamente desde apuntadores a objetos de la clase derivada.

Constructores y destructores virtuales

Por definición, los constructores no pueden ser virtuales. En nuestro caso, al inicializar new Perro se llama al constructor de la clase Perro y al de la clase Mamífero, por lo que ya crea un apuntador a la clase derivada.

Al trabajar con estos apuntadores, una operación posible es su eliminación. Por tanto, nos interesaría que, para su destrucción, primero se llame al destructor de la clase derivada y después al de la clase base. Para ello, nos basta con declarar el destructor de la clase base como virtual.

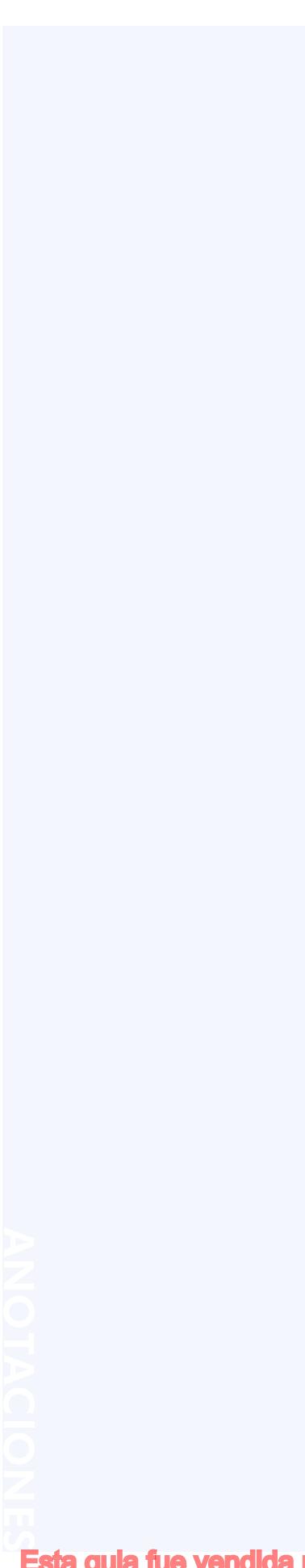
La regla práctica a seguir es declarar un destructor como virtual en el caso de que haya alguna función virtual dentro de la clase.

Tipos abstractos de datos y funciones virtuales puras

Ya hemos comentado que las clases corresponden al nivel de las ideas mientras que los objetos corresponden a elementos concretos.

De hecho, nos podemos encontrar clases en las que no tenga sentido instanciar ningún objeto, aunque sí que lo tuviera instanciarlos de clases derivadas. Es decir, clases que deseáramos mantener exclusivamente en el mundo de las ideas mientras que sus clases derivadas generaran nuestros objetos.

Un ejemplo podría ser una clase ObraDeArte, de la cual derivarán las subclases Pintura, Escultura, Literatura, Arquitectura, etc. Se podría considerar a la clase ObraDeArte como un concepto abstracto y cuando se tratara de obras concretas, referirnos a través de una de



las variedades de arte (sus clases derivadas). El criterio para declarar una clase como tipo de datos abstracto siempre es subjetivo y dependerá del uso que se desea de las clases en la aplicación.

```

class ObraDeArte
{
public:
    ObraDeArte();
    virtual ~ObraDeArte ();
    virtual void mostrarObraDeArte() = 0; //virtual pura
    void asignarAutor(String autor);
    String obtenerAutor();
    String autor;
};

class Pintura : public ObraDeArte
{
public:
    Pintura();
    Pintura ( const Pintura & );
    virtual ~Pintura ();
    virtual void mostrarObraDeArte();
    void asignarTitulo(String titulo);
    String obtenerTitulo();
private:
    String titulo;
};

Pintura :: mostrarObraDeArte()
{ cout << "Fotografía Pintura \n" }

class Escultura : public ObraDeArte
{
public:
    Escultura();
    Escultura ( const Escultura & );
    virtual ~Escultura ();
    virtual void mostrarObraDeArte();
    void asignarTitulo(String titulo);
    String obtenerTitulo();
private:
    String titulo;
};

Escultura :: mostrarObraDeArte()
{ cout << "Fotografía Escultura \n" }

```

Dentro de esta clase abstracta, se ha definido una función virtual que nos muestra una reproducción de la obra de arte. Esta reproducción varía según el tipo de obra. Podría ser en forma de fotografía, de video, de una lectura de un texto literario o teatral, etc.

Para declarar la clase ObraDeArte como un tipo de datos abstracto, y por tanto, no instanciable por ningún objeto sólo es necesario declarar una **función virtual pura**.

Para asignar una función virtual pura, basta con tomar una función virtual y asignarla a 0:

```
virtual void mostrarObraDeArte() = 0;
```

Ahora, al intentar instanciar un objeto ObraDeArte (mediante new ObraDeArte) daría error de compilación.

Al declarar funciones virtuales puras se debe tener en cuenta que esta función miembro también se hereda. Por tanto, en las clases derivadas se debe redefinir esta función. Si no se redefine, la clase derivada se convierte automáticamente en otra clase abstracta.

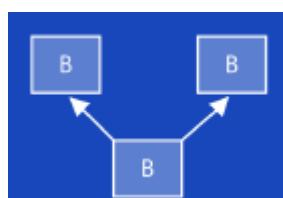
4.4.4. Operaciones avanzadas con herencia

A pesar de la potencia que se vislumbra en la herencia simple, hay situaciones en las que no es suficiente. A continuación, se presenta una breve introducción a los conceptos más avanzados relativos a la herencia.

Herencia múltiple

La herencia múltiple permite que una clase se derive de más de una clase base.

Figura 12.



```
class A : public B, public C
```

En este ejemplo, la clase A se deriva de la clase B y de la clase C. Ante esta situación, surgen algunas preguntas:

- ¿Qué sucede cuando las dos clases derivadas tienen una función con el mismo nombre? Se podría producir un conflicto de ambigüedad para el compilador que se puede resolver añadiendo a la clase A una función virtual que redefina esta función, con lo que se resuelve explícitamente la ambigüedad.
- ¿Qué sucede si las clases derivan de una clase base común? Como la clase A deriva de la clase D por parte de B y por parte de C, se producen dos copias de la clase D (ved la ilustración), lo cual puede provocar ambigüedades. La solución en este caso la proporciona la **herencia virtual**.

Figura 13.

Mediante la herencia virtual se indica al compilador que sólo se desea una clase base D compartida; para ello, las clases B y C se definen como virtuales.

Figura 14.

```
class B: virtual D  
class C: virtual D  
class A : public B, public C
```

Generalmente, una clase inicializa sólo sus variables y su clase base. Al declarar una clase como virtual, el constructor que inicializa las variables corresponde al de la clase más derivada.

Herencia privada

A veces no es necesario, o incluso no se desea, que las clases derivadas tengan acceso a los datos o funciones de la clase base. En este caso se utiliza la herencia privada.

Con la herencia privada las variables y funciones miembro de la clase base se consideran como privadas, independientemente de la accesibilidad declarada en la clase base. Por tanto, para cualquier función que no sea miembro de la clase derivada son inaccesibles las funciones heredadas de la clase base.

4.4.5. Orientaciones para el análisis y diseño de programas

La complejidad de los proyectos de software actuales hace que se utilice la estrategia “divide y vencerás” para afrontar el análisis de un problema, descomponiendo el problema original en tareas más reducidas y más fácilmente abordables.

La forma tradicional para realizar este proceso se basa en descomponer el problema en funciones o procesos más simples (diseño descendente), de manera que se obtiene una estructura jerárquica de procesos y subprocessos.

Con la programación orientada a objetos, la descomposición se reabre de forma alternativa enfocando el problema hacia los objetos que lo componen y sus relaciones, y no hacia las funciones.

El proceso a seguir es el siguiente:

- **Conceptualizar.** Los proyectos normalmente surgen de una idea que guía el desarrollo completo de éste. Es muy útil identificar el objetivo general que se persigue y velar porque se mantenga en las diferentes fases del proyecto.
- **Analizar.** Es decir, determinar las necesidades (requerimientos) que debe cubrir el proyecto. En esta fase, el esfuerzo se centra en comprender el dominio (el entorno) del problema en el mundo real (qué elementos intervienen y cómo se relacionan) y capturar los requerimientos.

Ejemplo

Si se diseña una aplicación para cajeros automáticos, un caso de uso podría ser retirar dinero de la cuenta.

El primer paso para conseguir el análisis de requerimientos es identificar los **casos de uso** que son descripciones en lenguaje natural de los diferentes procesos del dominio. Cada caso de uso describe la interacción entre un actor (sea persona o elemento) y el sistema. El actor envía un mensaje al sistema y éste actúa consecuentemente (respondiendo, cancelando, actuando sobre otro elemento, etc.).

Ejemplo

En el caso del proyecto del cajero automático, serían objetos el cliente, el cajero automático, el banco, el recibo, el dinero, la tarjeta de crédito, etc.

A partir de un conjunto completo de casos de uso, se puede comenzar a desarrollar el **modelo del dominio**, el documento donde se refleja todo lo que se conoce sobre el dominio. Como parte de esta modelización, se describen todos los objetos que intervienen (que al final podrán llegar a corresponder a las clases del diseño).

Nota

UML sólo es una convención comúnmente establecida para representar la información de un modelo.

El modelo se suele expresar en **UML** (lenguaje de modelado unificado), cuya explicación no es objetivo de esta unidad.

Ejemplo

En el caso del proyecto del cajero automático, un posible escenario sería que el cliente deseara retirar el dinero de la cuenta y no hubiera fondos.

A partir de los casos de uso, podemos describir diferentes **escenarios**, circunstancias concretas en las cuales se desarrolla el caso de uso. De esta forma, se puede ir completando el conjunto de interacciones posibles que debe cumplir nuestro modelo. Cada escenario se caracteriza también en un entorno, con unas condiciones previas y elementos que lo activan.

Todos estos elementos se pueden representar gráficamente mediante diagramas que muestren estas interacciones.

Además, se deberán tener en cuenta las restricciones que suponen el entorno en que funcionará u otros requerimientos proporcionados por el cliente.

- **Diseñar.** A partir de la información del análisis, se enfoca el problema en crear la solución. Podemos considerar el diseño como la conversión de los requerimientos obtenidos en un modelo implementable en software. El resultado es un documento que contiene el plan del diseño.

En primer lugar, se debe identificar las clases que intervienen. Una primera (y simple) aproximación a la solución del problema consiste en escribir los diferentes escenarios, y crear una clase para cada sustantivo. Posteriormente, se puede reducir este número mediante la agrupación de los sinónimos.

Una vez definidas las clases del modelo, podemos añadir las clases que nos serán útiles para la implementación del proyecto (las vistas, los informes, clases para conversiones o manipulaciones de datos, uso de dispositivos, etc.).

Establecido el conjunto inicial de clases, que posteriormente se puede ir modificando, se puede proceder a modelar las relaciones e interacciones entre ellas. Uno de los puntos más importantes en la definición de una clase es determinar sus **responsabilidades**: un principio básico es que cada clase debe ser responsable de algo. Si se identifica claramente esta responsabilidad única, el código resultante será más fácil de mantener. Las responsabilidades que no corresponden a una clase, las delega a las clases relacionadas.

En esta fase también se establecen las **relaciones** entre los objetos del diseño que pueden coincidir, o no, con los objetos del análisis. Pueden ser de diferentes tipos. El tipo de relación que más se ha comentado en esta unidad son las relaciones de generalización que posteriormente se han implementado a partir de la herencia pública, pero hay otras cada una con sus formas de implementación.

La información del diseño se completa con la inclusión de la **dinámica** entre las clases: la modelación de la interacción de las clases entre ellas a través de diversos tipos de diagramas gráficos.



El documento que recoge toda la información sobre el diseño de un programa se denomina *plan de diseño*.

- **Implementar.** Para que el proyecto se pueda aplicar, se debe convertir el plan de diseño a un código fuente, en nuestro caso, en C++. El lenguaje elegido proporciona las herramientas y mecánicas de trabajo para poder trasladar todas las definiciones de las clases, sus requerimientos, sus atributos y sus relaciones desde el mundo del diseño a nuestro entorno real. En esta fase nos centraremos en codificar de forma eficiente cada uno de los elementos del diseño.
- **Probar.** En esta fase se comprueba que el sistema realiza lo que se espera de él. Si no es así, se deben revisar las diferentes especificaciones a nivel de análisis, diseño o implementación. El diseño de un buen conjunto de pruebas basado en los casos de uso nos puede evitar muchos disgustos en el producto final. Siempre es preferible disponer de un buen conjunto de pruebas que provoque muchos fallos en las fases de análisis, diseño o implementación (y por tanto se pueden corregir), a encontrarse dichos errores en la fase de distribución.
- **Distribuir.** Se entrega al cliente una implementación del proyecto para su evaluación (del prototipo) o para su instalación definitiva.

Formas de desarrollo de un proyecto

Normalmente, el proceso descrito se lleva a cabo mediante un **proceso de cascada**: se va completando sucesivamente cada una de las etapas y cuando se ha finalizado y revisado, se pasa a la siguiente sin posibilidad de retroceder a la etapa anterior.

Este método, que en teoría parece perfecto, en la práctica es fatal. Por eso, en el análisis y diseño orientado a objetos se suele utilizar un **proceso iterativo**. En este proceso, a medida que se va desarrollando el software, se va repitiendo las etapas procediendo

cada vez a un refinamiento superior, lo que permite adaptarlo a los cambios producidos por el mayor conocimiento del proyecto por parte de los diseñadores, de los desarrolladores y del mismo cliente.

Este método también proporciona otra ventaja en la vida real: se facilita la entrega en la fecha prevista de versiones completas aunque en un estado más o menos refinado. De alguna manera, permite introducir la idea de versiones “lo suficientemente buenas”, y que posteriormente se pueden ir refinando según las necesidades del cliente.

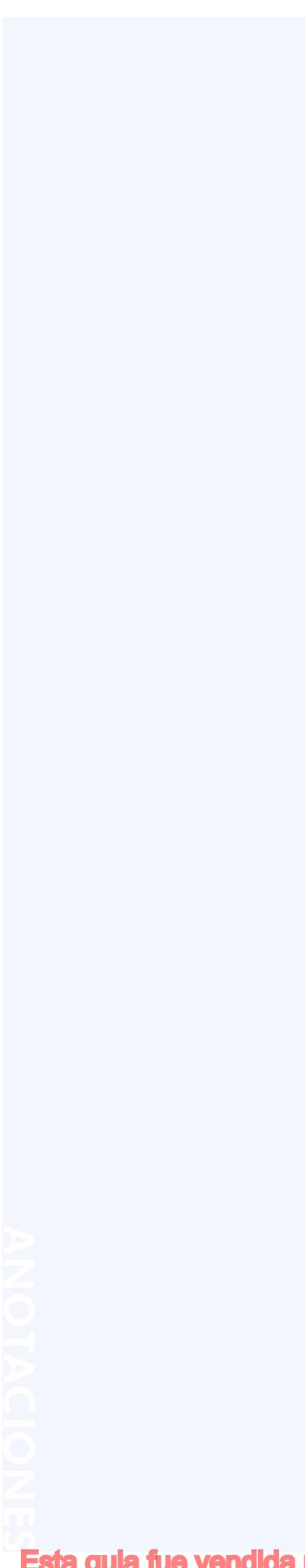
4.5. Resumen

En esta unidad, hemos evolucionado desde un entorno de programación C que sigue el modelo imperativo, donde se toma como base de actuación las instrucciones y su secuencia, a un modelo orientado a objetos donde la unidad base son los objetos y su interrelación.

Para ello, hemos tenido que comprender las ventajas que supone utilizar un modelo de trabajo más abstracto, pero más cercano a la descripción de las entidades que se manejan en el mundo real y sus relaciones, y que nos permite enfocar nuestra atención en los conceptos que se desea implementar más que en el detalle final que suponen las líneas de código.

Posteriormente, hemos estudiado las herramientas que proporciona C++ para su implementación: las clases y los objetos. Además de su definición, también se han revisado las propiedades principales relativas al modelo de orientación a objetos que proporciona C++. En concreto, se han estudiado la herencia entre clases, la homonimia y el polimorfismo.

Finalmente, hemos visto que, debido al cambio de filosofía en el nuevo paradigma de programación, no se pueden aplicar los mismos principios para el diseño de los programas y por ello se han introducido nuevas reglas de diseño coherentes con él.



4.6. Ejercicios de autoevaluación

1) Diseñad una aplicación que simule el funcionamiento de un ascensor. Inicialmente, la aplicación debe definir tres operaciones:

ASCENSOR: [1] Entrar [2] Salir [0] Finalizar

Después de cada operación, debe mostrar la ocupación del ascensor.

[1] Entrar corresponde a que una persona entra en el ascensor.

[2] Salir corresponde a que una persona sale del ascensor.

2) Ampliad el ejercicio anterior para incorporar los requerimientos siguientes:

- Una operación que sea dar el estado del ascensor

ASCENSOR: [1] Entrar [2] Salir [3] Estado [0] Finalizar

- Limitación de capacidad del ascensor a 6 plazas.

- Limitación de carga del ascensor a 500 kgs.

- Petición de código y peso de los usuarios para permitirles acceder al ascensor según los límites establecidos.

Si no se permite el acceso del usuario al ascensor, se le presenta un mensaje con los motivos:

- < El ascensor está completo >
- < El ascensor superaría la carga máxima autorizada >

Si se permite el acceso del usuario al ascensor, se le presenta el mensaje siguiente: < #Código# entra en el ascensor>.

Al entrar, como muestra de educación, saluda en general a las personas del ascensor (<#código# dice> Hola) si las hubiera, y éstas le corresponden el saludo de forma individual (<#código# responde> Hola).

Al salir un usuario del ascensor, se debe solicitar su código y actualizar la carga del ascensor al tiempo que se presenta el siguiente mensaje: <#codigo# sale del ascensor>.

Al salir, el usuario se despide de las personas del ascensor(<#codigo# dice> Adiós) si las hubiera, y éstas le corresponden el saludo de forma individual (<#codigo# responde> Adiós).

Para simplificar, consideraremos que no puede haber nunca dos pasajeros con el mismo código.

Después de cada operación, se debe poder mostrar el estado del ascensor (ocupación y carga).

3) Ampliad el ejercicio anterior incorporando tres posibles idiomas en los que los usuarios puedan saludar.

Al entrar, se debe solicitar también cuál es el idioma de la persona:

IDIOMA: [1] Catalán [2] Castellano [3] Inglés

- En catalán, el saludo es “Bon dia” y la despedida, “Adéu”.
- En castellano, el saludo es “Buenos días” y la despedida, “Adiós”.
- En inglés, el saludo es “Hello” y la despedida, “Bye”.

4.6.1. Solucionario

1)

```
ascensor01.hpp
class Ascensor {
    private:
        int ocupacion;
        int ocupacionmaxima;
    public:
        // Constructores y destructores
        Ascensor();
        ~Ascensor();

        // Funciones de acceso
```

```

        void mostrarOcupacion();
        int obtenerOcupacion();
        void modificarOcupacion(int difOcupacion);

        // Funciones del método
        bool persona_puedeEntrar();
        bool persona_puedeSalir();

        void persona_entrar();
        void persona_salir();
    };

ascensor01.cpp
#include <iostream>
#include "ascensor01.hpp"

// Constructores y destructores
Ascensor::Ascensor():
    ocupacion(0), ocupacionmaxima(6)
{ }

Ascensor::~Ascensor()
{ }

// Funciones de acceso
int Ascensor::obtenerOcupacion()
{ return (ocupacion); }

void Ascensor::modificarOcupacion(int difOcupacion)
{ ocupacion += difOcupacion; }

void Ascensor::mostrarOcupacion()
{ cout << "Ocupacion actual: " << ocupacion << endl; }

bool Ascensor::persona_puedeEntrar()
{ return (true); }

bool Ascensor::persona_puedeSalir()
{
    bool hayOcupacion;
    if (obtenerOcupacion() > 0) hayOcupacion = true;
}

```

```
else hayOcupacion = false;

return (hayOcupacion);
}

void Ascensor::persona_entrar()
{ modificarOcupacion(1); }

void Ascensor::persona_salir()
{
    int ocupacionActual = obtenerOcupacion();
    if (ocupacionActual>0)
        modificarOcupacion(-1);
}
```

ejerc01.cpp

```
#include <iostream>
#include "ascensor01.hpp"

int main(int argc, char *argv[])
{
    char opc;
    bool salir = false;
    Ascensor unAscensor;
    do
    {
        cout << endl
        cout << "ASCENSOR: [1]Entrar [2]Salir [0]Finalizar ";
        cin >> opc;
        switch (opc)
        {
            case '1':
                cout << "opc Entrar" << endl;
                unAscensor.persona_entrar();
                break;
            case '2':
                cout << "opc Salir" << endl;
                if (unAscensor.persona_puedeSalir())
                    unAscensor.persona_salir();
                else cout << "Ascensor vacio " << endl;
                break;
        }
    } while (!salir);
}
```

```
        case '0':
            salir = true;
            break;
    }
    unAscensor.mostrarOcupacion();
} while (! salir);
return 0;
}
```

2)

ascensor02.hpp

```
#ifndef _ASCENSOR02
#define _ASCENSOR02
#include "persona02.hpp"
```

```
class Ascensor {
private:
    int ocupacion;
    int carga;
    int ocupacionMaxima;
    int cargaMaxima;
    Persona *pasajeros[6];
public:
    // Constructores y destructores
    Ascensor();
    ~Ascensor();

    // Funciones de acceso
    void mostrarOcupacion();
    int obtenerOcupacion();
    void modificarOcupacion(int difOcupacion);
    void mostrarCarga();
    int obtenerCarga();
    void modificarCarga(int difCarga);

    void mostrarListaPasajeros();

    // Funciones del método
    bool persona_puedeEntrar(Persona *);
```

```
bool persona_seleccionar(Persona *localizarPersona,
                           Persona **unaPersona);

void persona_entrar(Persona *);
void persona_salir(Persona *);

void persona_saludarRestoAscensor(Persona *);
void persona_despedirseRestoAscensor(Persona *);

};

#endif

ascensor 02.cpp
#include <iostream>
#include "ascensor02.hpp"

// 
// Constructores y destructores
//

// En el constructor, inicializamos los valores máximos
// de ocupación y carga máxima de ascensor
// y el vector de pasajeros a apuntadores NULL

Ascensor::Ascensor():
    ocupacion(0), carga(0),
    ocupacionMaxima(6), cargaMaxima(500)
{ for (int i=0;i<=5;++i) {pasajeros[i]=NULL;} }

Ascensor::~Ascensor()
{ // Liberar codigos de los pasajeros
    for (int i=0;i<=5;++i)
        { if (! (pasajeros[i]==NULL)) {delete(pasajeros[i]);} }
}

// Funciones de acceso
int Ascensor::obtenerOcupacion()
{ return (ocupacion); }

void Ascensor::modificarOcupacion(int difOcupacion)
{ ocupacion += difOcupacion; }
```

```
void Ascensor::mostrarOcupacion()
{ cout << "Ocupacion actual: " << ocupacion ; }

int Ascensor::obtenerCarga()
{ return (carga); }

void Ascensor::modificarCarga(int difCarga)
{ carga += difCarga; }

void Ascensor::mostrarCarga()
{ cout << "Carga actual: " << carga ; }

bool Ascensor::persona_puedeEntrar(Persona *unaPersona)
{
    bool tmpPuedeEntrar;

    // si la ocupación no sobrepasa el límite de ocupación y
    // si la carga no sobrepasa el límite de carga
    // → puede entrar

    if (ocupacion + 1 > ocupacionMaxima)
    {
        cout << " Aviso: Ascensor completo. No puede entrar. "
        cout << endl;
        return (false);
    }

    if (unaPersona→obtenerPeso() + carga > cargaMaxima)
    {
        cout << "Aviso: El ascensor supera su carga máxima.";
        cout << " No puede entrar. " << endl;
        return (false);
    }

    return (true);
}

bool Ascensor::persona_seleccionar(Persona *localizarPersona,
                                    Persona **unaPersona)
{
    int contador;
```

```
// Se debe seleccionar un pasajero del ascensor.  
bool personaEncontrada = false;  
if (obtenerOcupacion() > 0)  
{  
    contador=0;  
    do  
    {  
        if (pasajeros[contador] !=NULL)  
        {  
            if ((pasajeros[contador]→obtenerCodigo()==  
                localizarPersona→obtenerCodigo() ))  
            {  
                *unaPersona=pasajeros[contador];  
                personaEncontrada=true;  
                break;  
            }  
        }  
        contador++;  
    } while (contador<ocupacionMaxima);  
    if (contador>=ocupacionMaxima) {*unaPersona=NULL;}  
}  
return (personaEncontrada);  
  
}  
  
void Ascensor::persona_entrar(Persona *unaPersona)  
{  
    int contador;  
    modificarOcupacion(1);  
    modificarCarga(unaPersona->obtenerPeso());  
    cout << unaPersona→obtenerCodigo();  
    cout << " entra en el ascensor " << endl;  
    contador=0;  
    // hemos verificado anteriormente que hay plazas libres  
    do  
    {  
        if (pasajeros[contador]==NULL )  
        {  
            pasajeros[contador]=unaPersona;  
            break;  
        }  
    }
```

```
    contador++;
} while (contador<ocupacionMaxima);
}

void Ascensor::persona_salir(Persona *unaPersona)
{
int contador;
contador=0;
do
{
if ((pasajeros[contador]==unaPersona ))
{
cout << unaPersona→obtenerCodigo();
cout << " sale del ascensor " << endl;
pasajeros[contador]=NULL;
// Modificamos la ocupación y la carga
modificarOcupacion(-1);
modificarCarga(-1 * (unaPersona->obtenerPeso()));
break;
}
contador++;
} while (contador<ocupacionMaxima);
if (contador == ocupacionMaxima)
{ cout << "Ninguna persona con este código. ";
cout << "Nadie sale del ascensor" << endl;}
}

void Ascensor::mostrarListaPasajeros()
{
int contador;
Persona *unaPersona;

if (obtenerOcupacion() > 0)
{
cout << "Lista de pasajeros del ascensor: " << endl;
contador=0;
do
{
if (!(pasajeros[contador]==NULL ))
{
unaPersona=pasajeros[contador];
}
```

```
        cout << unaPersona->obtenerCodigo() << "; ";
    }
    contador++;
} while (contador<ocupacionMaxima);
cout << endl;
}
else
{ cout << "El ascensor esta vacío" << endl; }
}

void Ascensor::persona_saludarRestoAscensor( Persona *unaPersona)
{
    int contador;
    Persona *otraPersona;
    if (obtenerOcupacion() > 0)
    {
        contador=0;
        do
        {
            if (! (pasajeros[contador]==NULL ))
            {
                otraPersona=pasajeros[contador];
                if (! (unaPersona->obtenerCodigo()==
                    otraPersona->obtenerCodigo()))
                {
                    cout << otraPersona->obtenerCodigo();
                    cout << " responde: " ;
                    otraPersona->saludar();
                    cout << endl;
                }
            }
            contador++;
        } while (contador<ocupacionMaxima);
    }
}

void Ascensor::persona_despedirseRestoAscensor( Persona *unaPersona)
{
    int contador;
    Persona *otraPersona;

    if (obtenerOcupacion() > 0)
```

```

    {
        contador=0;
        do
        {
            if (! (pasajeros[contador]==NULL ))
            {
                otraPersona=pasajeros[contador];
                if (! (unaPersona->obtenerCodigo()==
                    otraPersona->obtenerCodigo()))
                {
                    cout << otraPersona->obtenerCodigo();
                    cout << " responde: " ;
                    otraPersona->despedirse();
                    cout << endl;
                }
            }
            contador++;
        } while (contador<ocupacionMaxima);
    }
}

persona02.hpp
#ifndef _PERSONA02
#define _PERSONA02

class Persona
{
private:
    int codigo;
    int peso;
public:
    // Constructores
    Persona();
    Persona(int codigo, int peso);
    Persona(const Persona &);
    ~Persona();

    // Funciones de acceso
    int obtenerCodigo();
    void asignarCodigo(int);
    int obtenerPeso() const;
}

```

```
void asignarPeso(int nPeso);
void asignarPersona(int);
void asignarPersona(int,int);
void solicitarDatos();
void solicitarCodigo();

void saludar();
void despedirse();

};

#endif

persona02.cpp
#include <iostream>
#include "persona02.hpp"

Persona::Persona()
{
}

Persona::Persona(int nCodigo, int nPeso)
{
    codigo = nCodigo;
    peso = nPeso;
}

Persona::~Persona()
{
}

int Persona::obtenerPeso() const
{
    return (peso);
}

void Persona::asignarPeso(int nPeso)
{
    peso = nPeso;
}

int Persona::obtenerCodigo()
{
    return (codigo);
}

void Persona::asignarCodigo(int nCodigo)
{
    codigo= nCodigo;
}

void Persona::asignarPersona(int nCodigo)
{
    this->codigo = nCodigo;
}
```

```
void Persona::asignarPersona(int nCodigo, int nPeso)
{
    asignarCodigo(nCodigo);
    asignarPeso(nPeso);
}

void Persona:: saludar()
{ cout << "Hola \n" ; };

void Persona:: despedirse ()
{ cout << "Adios \n" ; };

void Persona::solicitarCodigo()
{
    int nCodigo;
    cout << "Codigo: ";
    cin >> nCodigo;
    cout << endl;
    codigo = nCodigo;
}

ejerc02.cpp
#include <iostream>
#include "ascensor02.hpp"
#include "persona02.hpp"

void solicitarDatos(int *nCodigo, int *nPeso)
{
    cout << endl;
    cout << "Codigo: ";
    cin >> *nCodigo;
    cout << endl;
    cout << "Peso: ";
    cin >> *nPeso;
    cout << endl;
}

int main(int argc, char *argv[])
{
    char opc;
```

```
bool salir = false;
Ascensor unAscensor;
Persona * unaPersona;
Persona * localizarPersona;

do
{
    cout << endl << "ASCENSOR: ";
    cout << "[1]Entrar [2]Salir [3]Estado [0]Finalizar ";
    cin >> opc;
    switch (opc)
    {
        case '1': // opción Entrar
        {
            int nPeso;
            int nCodigo;

            solicitarDatos(&nCodigo, &nPeso);
            unaPersona = new Persona(nCodigo, nPeso);
            if (unAscensor.persona_puedeEntrar(unaPersona))
            {
                unAscensor.persona_entrar(unaPersona);
                if (unAscensor.obtenerOcupacion()>1)
                {
                    cout << unaPersona->obtenerCodigo();
                    cout << " dice: " ;
                    unaPersona->saludar();
                    cout << endl; // Ahora responden las demás
                    unAscensor.persona_saludarRestoAscensor(unaPersona);
                }
            }
            break;
        }
        case '2': // opción Salir
        {
            unaPersona = NULL;
            localizarPersona = new Persona;
            localizarPersona->solicitarCodigo();
            if (unAscensor.persona_seleccionar ( localizarPersona,
                                                &unaPersona))
            {

```

```
unAscensor.persona_salir(unaPersona);
if (unAscensor.obtenerOcupacion()>0)
{
    cout << unaPersona->obtenerCodigo()
    cout << " dice: " ;
    unaPersona->despedirse();
    cout << endl; // Ahora responden las demás
    unAscensor.persona_despedirseRestoAscensor(unaPersona);
    delete (unaPersona);
}
else
{
    cout<<"No hay ninguna persona con este código";
    cout << endl;
}
delete localizarPersona;
break;
}
case '3': //Estado
{
    unAscensor.mostrarOcupacion();
    cout << " - "; // Para separar ocupación de carga
    unAscensor.mostrarCarga();
    cout << endl;
    unAscensor.mostrarListaPasajeros();
    break;
}
case '0':
{
    salir = true;
    break;
}
}
} while (! salir);
return 0;
}
```

3) ascensor03.hpp y ascensor03.cpp coinciden con ascensor02.hpp y ascensor02.cpp del ejercicio 2.

```
persona03.hpp
#ifndef _PERSONA03
#define _PERSONA03

class Persona
{
private:
    int codigo;
    int peso;
public:
    // Constructores
    Persona();
    Persona(int codigo, int peso);
    Persona(const Persona &);

    ~Persona();

    // Funciones de acceso
    int obtenerCodigo();
    void asignarCodigo(int);
    int obtenerPeso() const;
    void asignarPeso(int nPeso);
    void asignarPersona(int,int);
    void solicitarCodigo();

    virtual void saludar();
    virtual void despedirse();
};

class Catalan: public Persona
{
public:
    Catalan()
    {
        asignarCodigo (0);
        asignarPeso (0);
    }

    Catalan(int nCodigo, int nPeso)
    {
        asignarCodigo (nCodigo);
        asignarPeso (nPeso);
    };
}
```

```
virtual void saludar()
{ cout << "Bon dia"; };

virtual void despedirse()
{ cout << "Adeu"; };
};

class Castellano: public Persona
{
public:
    Castellano()
    {
        asignarCodigo (0);
        asignarPeso (0);
    };

    Castellano(int nCodigo, int nPeso)
    {
        asignarCodigo (nCodigo);
        asignarPeso (nPeso);
    };

    virtual void saludar()
    { cout << "Buenos días"; };

    virtual void despedirse()
    { cout << "Adiós"; };
};

class Ingles : public Persona
{
public:
    Ingles()
    {
        asignarCodigo (0);
        asignarPeso (0);
    };

    Ingles(int nCodigo, int nPeso)
    {
        asignarCodigo (nCodigo);
        asignarPeso (nPeso);
    };
}
```

```
virtual void saludar()
{ cout << "Hello"; };

virtual void despedirse()
{ cout << "Bye"; };
};

#endif

persona03.cpp
#include <iostream>
#include "persona03.hpp"

Persona::Persona()
{
}

Persona::Persona(int nCodigo, int nPeso)
{
    codigo = nCodigo;
    peso = nPeso;
}

Persona::~Persona()
{
}

int Persona::obtenerPeso() const
{
    return (peso);
}

void Persona::asignarPeso(int nPeso)
{
    peso = nPeso;
}

int Persona::obtenerCodigo()
{
    return (codigo);
}

void Persona::asignarCodigo(int nCodigo)
{
    this->codigo = nCodigo;
}

void Persona::asignarPersona(int nCodigo, int nPeso)
{
    asignarCodigo(nCodigo);
    asignarPeso(nPeso);
}
```

```
void Persona:: saludar()
{ cout << "Hola \n" ; };

void Persona:: despedirse ()
{ cout << "Adiós \n" ; };

void Persona::solicitarCodigo{
    int nCodigo;

    cout << "Codigo: ";
    cin >> nCodigo;
    cout << endl;

    asignarCodigo (nCodigo);
}

ejerc03.cpp
#include <iostream>
#include "ascensor03.hpp"
#include "persona03.hpp"

void solicitarDatos(int *nCodigo, int *nPeso, int *nIdioma)
{
    cout << endl;
    cout << "Codigo: ";
    cin >> *nCodigo;
    cout << endl;
    cout << "Peso: ";
    cin >> *nPeso;
    cout << endl;
    cout << "Idioma: [1] Catalán [2] Castellano [3] Inglés ";
    cin >> *nIdioma;
    cout << endl;
}

int main(int argc, char *argv[])
{
    char opc;
    bool salir = false;
    Ascensor unAscensor;
    Persona * unaPersona;
    Persona * localizarPersona;
```

```
do
{
    cout << endl << "ASCENSOR: ";
    cout << "[1]Entrar [2]Salir [3]Estado [0]Finalizar";
    cin >> opc;
    switch (opc)
    {
        case '1': // Opción Entrar
        {
            int nPeso;
            int nCodigo;
            int nIdioma;
            solicitarDatos(&nCodigo, &nPeso, &nIdioma);
            switch (nIdioma)
            {
                case 1:
                {
                    unaPersona = new Catalan(nCodigo, nPeso);
                    break;
                }
                case 2:
                {
                    unaPersona=new Castellano(nCodigo, nPeso);
                    break;
                }
                case 3:
                {
                    unaPersona = new Ingles(nCodigo, nPeso);
                    break;
                }
            }
            if (unAscensor.persona_puedeEntrar(unaPersona))
            {
                unAscensor.persona_entrar(unaPersona);
                if (unAscensor.obtenerOcupacion()>1)
                {
                    cout << unaPersona->obtenerCodigo();
                    cout << " dice: " ;
                    unaPersona->saludar();
                    cout << endl; // Ahora responden las demás
                    unAscensor.persona_saludarRestoAscensor (unaPersona);
                }
            }
        }
    }
}
```

```
        break;
    }
    case '2': //Opción Salir
    {
        localizarPersona = new Persona;
        unaPersona = NULL;

        localizarPersona->solicitarCodigo();
        if (unAscensor.persona_seleccionar(localizarPersona,
            & unaPersona))
        {
            unAscensor.persona_salir(unaPersona);
            if (unAscensor.obtenerOcupacion()>0)
            {
                cout << unaPersona->obtenerCodigo();
                cout << " dice: " ;
                unaPersona->despedirse();
                cout << endl; // Ahora responden las demás
                unAscensor.persona_despedirseRestoAscensor (unaPersona);
                delete (unaPersona) ;
            }
        }
        else
        {
            cout<<"No hay ninguna persona con este código";
            cout << endl;
        }
        delete localizarPersona;
        break;
    }
    case '3': //Estado
    {
        unAscensor.mostrarOcupacion();
        cout << " - "; // Para separar Ocupacion de Carga
        unAscensor.mostrarCarga();
        cout << endl;
        unAscensor.mostrarListaPasajeros();
        break;
    }
    case '0':
```

```
{  
    salir = true;  
    break;  
}  
}  
} while (! salir);  
return 0;  
}
```

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX • Editorial Guías México

Unidad 5. Programación en Java

5.1. Introducción

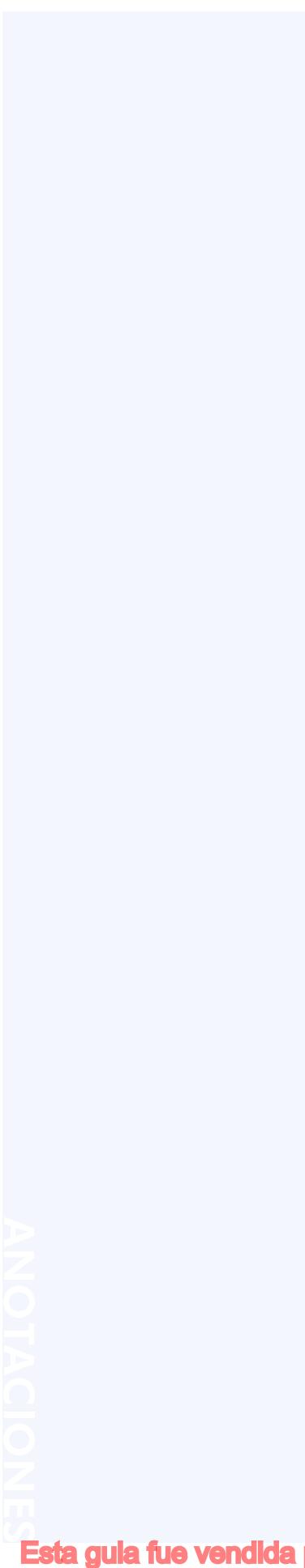
En las unidades anteriores, se ha mostrado la evolución que han experimentado los lenguajes de programación en la historia y que han ido desembocando en los diferentes paradigmas de programación.

Inicialmente, el coste de un sistema informático estaba marcado principalmente por el hardware: los componentes internos de los ordenadores eran voluminosos, lentos y caros. En comparación, el coste que generaban las personas que intervenían en su mantenimiento y en el tratamiento de la información era casi despreciable. Además, por limitaciones físicas, el tipo de aplicaciones que se podían manejar eran más bien simples. El énfasis en la investigación en informática se centraba básicamente en conseguir sistemas más pequeños, más rápidos y más baratos.

Con el tiempo, esta situación ha cambiado radicalmente. La revolución producida en el mundo del hardware ha permitido la fabricación de ordenadores en los que no se podía ni soñar hace 25 años, pero esta revolución no ha tenido su correspondencia en el mundo del software. En este aspecto, los costes materiales se han reducido considerablemente mientras que los relativos a personal han aumentado progresivamente. También se ha incrementado la complejidad en el uso del software, entre otras cosas debido al aumento de interactividad con el usuario.

En la actualidad muchas de las líneas de investigación buscan mejorar el rendimiento en la fase de desarrollo de software donde, de momento, la intervención humana es fundamental. Mucho de este esfuerzo se centra en la generación de código correcto y en la reutilización del trabajo realizado.

En este camino, el paradigma de la programación orientada a objetos ha supuesto una gran aproximación entre el proceso de desarrollo de



aplicaciones y la realidad que intentan representar. Por otro lado, la incorporación de la informática en muchos componentes que nos rodean también ha aumentado en gran medida el número de plataformas diversas sobre las cuales es posible desarrollar programas.

Java es un lenguaje moderno que ha nacido para dar solución a este nuevo entorno. Básicamente, es un lenguaje orientado a objetos pensado para trabajar en múltiples plataformas. Su planteamiento consiste en crear una plataforma común intermedia para la cual se desarrollan las aplicaciones y, después, trasladar el resultado generado para dicha plataforma común a cada máquina final.

Este paso intermedio permite:

- Escribir la aplicación **sólo una vez**. Una vez compilada hacia esta plataforma común, la aplicación podrá ser ejecutada por todos los sistemas que dispongan de dicha plataforma intermedia.
- Escribir la plataforma común **sólo una vez**. Al conseguir que una máquina real sea capaz de ejecutar las instrucciones de dicha plataforma común, es decir, que sea capaz de trasladarlas al sistema subyacente, se podrán ejecutar en ella todas las aplicaciones desarrolladas para dicha plataforma.

Por tanto, se consigue el máximo nivel de reutilización. El precio es el sacrificio de parte de la velocidad.

En el orden de la generación de código correcto, Java dispone de varias características que se irán viendo a lo largo de esta unidad. En todo caso, de momento se desea destacar que Java se basa en C++, con lo cual se consigue mayor facilidad de aprendizaje para gran número de desarrolladores (reutilización del conocimiento), pero se le libera de muchas de las cadenas que C++ arrastraba por su compatibilidad con el C.

Esta "limpieza" tiene consecuencias positivas:

- El lenguaje es más simple, pues se eliminan conceptos complejos raras veces utilizados.

- El lenguaje es más directo. Se ha estimado que Java permite reducir el número de líneas de código a la cuarta parte.
- El lenguaje es más puro, pues sólo permite trabajar en el paradigma de la orientación a objetos.

Además, la juventud del lenguaje le ha permitido incorporar dentro de su núcleo algunas características que sencillamente no existían cuando se crearon otros lenguajes, como las siguientes:

- La programación de hilos de ejecución (*threads*), que permite aprovechar las arquitecturas con multiprocesadores.
- La programación de comunicaciones (TCP/IP, etc.) que facilita el trabajo en red, sea local o Internet.
- La programación de *applets*, miniaplicaciones pensadas para ser ejecutadas por un navegador web.
- El soporte para crear interfaces gráficas de usuario y un sistema de gestión de eventos, que facilitan la creación de aplicaciones siguiendo el paradigma de la programación dirigida por eventos.

En esta unidad se desea introducir al lector en este nuevo entorno de programación y presentar sus principales características, y se pretende de que, partiendo de sus conocimientos del lenguaje C++, alcance los objetivos siguientes:

- 1) Conocer el entorno de desarrollo de Java.
- 2) Ser capaz de programar en Java.
- 3) Entender los conceptos del uso de los hilos de ejecución y su aplicación en el entorno Java.
- 4) Comprender las bases de la programación dirigida por eventos y ser capaz de desarrollar ejemplos simples.
- 5) Poder crear *applets* simples.

5.2. Origen de Java

En 1991, ingenieros de Sun Microsystems intentaban introducirse en el desarrollo de programas para electrodomésticos y pequeños equipos electrónicos donde la potencia de cálculo y memoria era reducida. Ello requería un lenguaje de programación que, principalmente, aportara fiabilidad del código y facilidad de desarrollo, y pudiera adaptarse a múltiples dispositivos electrónicos.

Nota

Por la variedad de dispositivos y procesadores existentes en el mercado y sus continuos cambios buscaban un entorno de trabajo que no dependiera de la máquina en la que se ejecutara.

Para ello diseñaron un esquema basado en una plataforma intermedia sobre la cual funcionaría un nuevo código máquina ejecutable, y esta plataforma se encargaría de la traslación al sistema subyacente. Este código máquina genérico estaría muy orientado al modo de funcionar de la mayoría de dichos dispositivos y procesadores, por lo cual la traslación final habría de ser rápida.

El proceso completo consistiría, pues, en escribir el programa en un lenguaje de alto nivel y compilarlo para generar código genérico (los bytecodes) preparado para ser ejecutado por dicha plataforma (la "máquina virtual"). De este modo se conseguiría el objetivo de poder escribir el código una sola vez y poder ejecutarlo en todas partes donde estuviera disponible dicha plataforma (*Write Once, Run Everywhere*).

Teniendo estas referencias, su primer intento fue utilizar C++, pero por su complejidad surgieron numerosas dificultades, por lo que decidieron diseñar un nuevo lenguaje basándose en C++ para facilitar su aprendizaje. Este nuevo lenguaje debía recoger, además, las propiedades de los lenguajes modernos y reducir su complejidad eliminando aquellas funciones no absolutamente imprescindibles.

El proyecto de creación de este nuevo lenguaje recibió el nombre inicial de Oak, pero como el nombre estaba registrado, se rebautizó

con el nombre final de Java. Consecuentemente, la máquina virtual capaz de ejecutar dicho código en cualquier plataforma recibió el nombre de máquina virtual de Java (JVM - Java virtual machine).

Los primeros intentos de aplicación comercial no fructificaron, pero el desarrollo de Internet fomentó tecnologías multiplataforma, por lo que Java se reveló como una posibilidad interesante para la compañía. Tras una serie de modificaciones de diseño para adaptarlo, Java se presentó por primera vez como lenguaje para ordenadores en el año 1995, y en enero de 1996, Sun formó la empresa Java Soft para desarrollar nuevos productos en este nuevo entorno y facilitar la colaboración con terceras partes. El mismo mes se dio a conocer una primera versión, bastante rudimentaria, del kit de desarrollo de Java, el JDK 1.0.

A principios de 1997 apareció la primera revisión Java, la versión 1.1, mejorando considerablemente las prestaciones del lenguaje, y a finales de 1998 apareció la revisión Java 1.2, que introdujo cambios significativos. Por este motivo, a esta versión y posteriores se las conoce como plataformas Java 2. En diciembre del 2003, la última versión de la plataforma Java2 disponible para su descarga en la página de Sun es Java 1.4.2.

La verdadera revolución que impulsó definitivamente la expansión del lenguaje la causó la incorporación en 1997 de un intérprete de Java en el navegador Netscape.

Nota

Podéis encontrar esta versión en la dirección siguiente:
<http://java.sun.com>

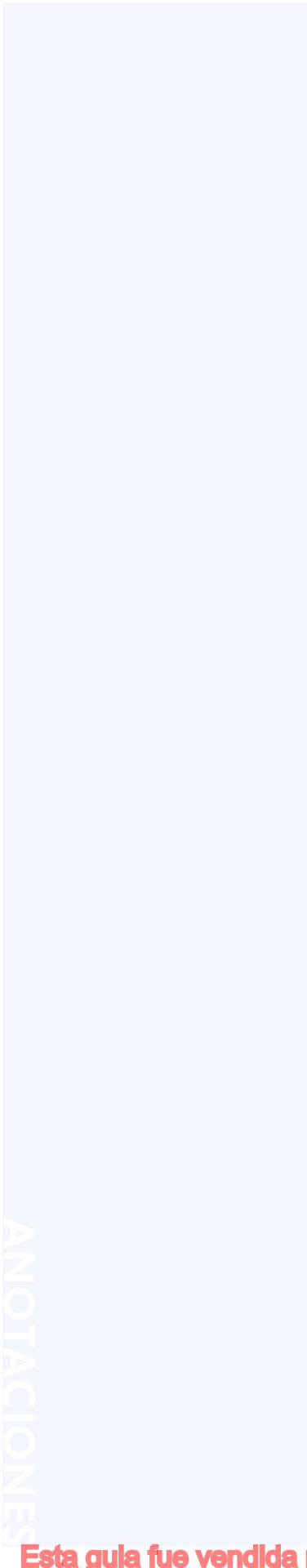
5.3. Características generales de Java



Sun Microsystems describe Java como un lenguaje simple, orientado a objetos, distribuido, robusto, seguro, de arquitectura neutra, portable, interpretado, de alto rendimiento, multitarea y dinámico.

Analicemos esta descripción:

- Simple . Para facilitar el aprendizaje, se consideró que los lenguajes más utilizados por los programadores eran el C y el C++.



Descartado el C++, se diseñó un nuevo lenguaje que fuera muy cercano a él para facilitar su comprensión.

Con este objetivo, Java elimina una serie de características poco utilizadas y de difícil comprensión del C++, como, por ejemplo, la herencia múltiple, las coerciones automáticas y la sobrecarga de operadores.

- **Orientado a objetos.** En pocas palabras, el diseño orientado a objetos enfoca el diseño hacia los datos (objetos), sus funciones e interrelaciones (métodos). En este punto, se siguen esencialmente los mismos criterios que C++.
- **Distribuido.** Java incluye una amplia librería de rutinas que permiten trabajar fácilmente con los protocolos de TCP/IP como HTTP o FTP. Se pueden crear conexiones a través de la red a partir de direcciones URL con la misma facilidad que trabajando en forma local.
- **Robusto.** Uno de los propósitos de Java es buscar la fiabilidad de los programas. Para ello, se puso énfasis en tres frentes:
 - Estricto control en tiempo de compilación con el objetivo de detectar los problemas lo antes posible. Para ello, utiliza una estrategia de fuerte control de tipos, como en C++, aunque evitando algunos de sus agujeros normalmente debidos a su compatibilidad con C. También permite el control de tipos en tiempo de enlace.
 - Chequeo en tiempo de ejecución de los posibles errores dinámicos que se pueden producir.
 - Eliminación de situaciones propensas a generar errores. El caso más significativo es el control de los apuntadores. Para ello, los trata como vectores verdaderos, controlando los valores posibles de índices. Al evitar la aritmética de apuntadores (sumar desplazamiento a una posición de memoria sin controlar sus límites) se evita la posibilidad de sobreescritura de memoria y corrupción de datos.

- **Seguro.** Java está orientado a entornos distribuidos en red y, por este motivo, se ha puesto mucho énfasis en la seguridad contra virus e intrusiones, y en la autenticación.
- **Arquitectura neutra.** Para poder funcionar sobre variedad de procesadores y arquitecturas de sistemas operativos, el compilador de Java proporciona un código común ejecutable desde cualquier sistema que tenga la presencia de un sistema en tiempo de ejecución de Java.

Esto evita que los autores de aplicaciones deban producir versiones para sistemas diferentes (como PC, Apple Macintosh, etc.). Con Java, el mismo código compilado funciona para todos ellos.

Para ello, Java genera instrucciones *bytecodes* diseñadas para ser fácilmente interpretadas por una plataforma intermedia (la máquina virtual de Java) y traducidas a cualquier código máquina nativo al vuelo.

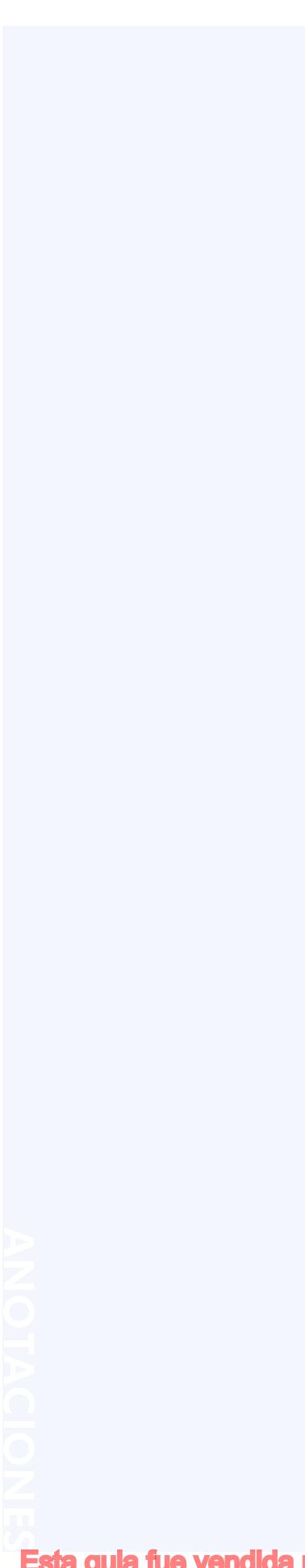
- **Portable.** La arquitectura neutra ya proporciona un gran avance respecto a la portabilidad, pero no es el único aspecto que se ha cuidado al respecto.

Ejemplo

En Java no hay detalles que dependan de la implementación, como podría ser el tamaño de los tipos primitivos. En Java, a diferencia de C o C++, el tipo `int` siempre se refiere a un número entero de 32 bits con complemento a 2 y el tipo `float` un número de 32 bits siguiendo la norma IEEE 754.

La portabilidad también viene dada por las librerías. Por ejemplo, hay una clase Windows abstracta y sus implementaciones para Windows, Unix o Macintosh.

- **Interpretado.** Los *bytecodes* en Java se traducen en tiempo de ejecución a instrucciones de la máquina nativa (son interpretadas) y no se almacenan en ningún lugar.
- **Alto rendimiento.** A veces se requiere mejorar el rendimiento producido por la interpretación de los *bytecodes*, que ya es bas-



tante bueno de por sí. En estos casos, es posible traducirlos en tiempo de ejecución al código nativo de la máquina donde la aplicación se está ejecutando. Esto es, compilar el lenguaje de la JVM al lenguaje de la máquina en la que se haya de ejecutar el programa.

Por otro lado, los bytecodes se han diseñado pensando en el código máquina por lo que el proceso final de la generación de código máquina es muy simple. Además, la generación de los bytecodes es eficiente y se le aplican diversos procesos de optimización.

- **Multitarea.** Java proporciona dentro del mismo lenguaje herramientas para construir aplicaciones con múltiples hilos de ejecución, lo que simplifica su uso y lo hace más robusto.
- **Dinámico.** Java se diseñó para adaptarse a un entorno cambiante. Por ejemplo, un efecto lateral del C++ se produce debido a la forma en la que el código se ha implementado. Si un programa utiliza una librería de clases y ésta cambia, hay que recompilar todo el proyecto y volverlo a redistribuir. Java evita estos problemas al hacer las interconexiones entre los módulos más tarde, permitiendo añadir nuevos métodos e instancias sin tener ningún efecto sobre sus clientes.

Mediante las interfaces se especifican un conjunto de métodos que un objeto puede realizar, pero deja abierta la manera como los objetos pueden implementar estos métodos. Una clase Java puede implementar múltiples interfaces, aunque sólo puede heredar de una única clase. Las interfaces proporcionan flexibilidad y reusabilidad conectando objetos según lo que queremos que hagan y no por lo que hacen.

Las clases en Java se representan en tiempo de ejecución por una clase llamada Class, que contiene las definiciones de las clases en tiempo de ejecución. De este modo, se pueden hacer comprobaciones de tipo en tiempo de ejecución y se puede confiar en los tipos en Java, mientras que en C++ el compilador solo confía en que el programador hace lo correcto.

5.4. El entorno de desarrollo de Java

Para desarrollar un programa en Java, existen diversas opciones comerciales en el mercado. No obstante, la compañía Sun distribuye de forma gratuita el Java Development Kit (JDK) que es un conjunto de programas y librerías que permiten el desarrollo, compilación y ejecución de aplicaciones en Java además de proporcionar un *debugger* para el control de errores.

También existen herramientas que permiten la integración de todos los componentes anteriores (IDE – *integrated development environment*), de utilización más agradable, aunque pueden presentar fallos de compatibilidad entre plataformas o ficheros resultantes no tan optimizados. Por este motivo, y para familiarizarse mejor con todos los procesos de la creación de software, se ha optado en este material por desarrollar las aplicaciones directamente con las herramientas proporcionadas por Sun.

Nota

Entre los IDEs disponibles actualmente se puede destacar el proyecto Eclipse, que, siguiendo la filosofía de código abierto, ha conseguido un paquete de desarrollo muy completo (SDK . - *standard development kit*) para diversos sistemas operativos (Linux, Windows, Sun, Apple, etc.).

Este paquete está disponible para su descarga en <http://www.eclipse.org>.

Otro IDE interesante es JCreator, que además de desarrollar una versión comercial, dispone de una versión limitada, de fácil manejo.

Este paquete está disponible para su descarga en <http://www.jcreator.com>.

Otra característica particular de Java es que se pueden generar varios tipos de aplicaciones:

- **Aplicaciones independientes.** Un fichero que se ejecuta directamente sobre la máquina virtual de la plataforma.

- **Applets.** Miniaplicaciones que no se pueden ejecutar directamente sobre la máquina virtual, sino que están pensadas para ser cargadas y ejecutadas desde un navegador web. Por este motivo, incorpora unas limitaciones de seguridad extremas.
- **Servlets.** Aplicaciones sin interfaz de usuario para ejecutarse desde un servidor y cuya función es dar respuesta a las acciones de navegadores remotos (petición de páginas HTML, envío de datos de un formulario, etc.). Su salida generalmente es a través de ficheros, como por ejemplo, ficheros HTML.

Para generar cualquiera de los tipos de aplicaciones anteriores, sólo se precisa lo siguiente:

- Un editor de textos donde escribir el código fuente en lenguaje Java.
- La plataforma Java, que permite la compilación, depurado, ejecución y documentación de dichos programas.

5.4.1. La plataforma Java

Entendemos como plataforma el entorno hardware o software que necesita un programa para ejecutarse. Aunque la mayoría de plataformas se pueden describir como una combinación de sistema operativo y hardware, la plataforma Java se diferencia de otras en que se compone de una plataforma software que funciona sobre otras plataformas basadas en el hardware (GNU/Linux, Solaris, Windows, Macintosh, etc.).

La plataforma Java tiene dos componentes:

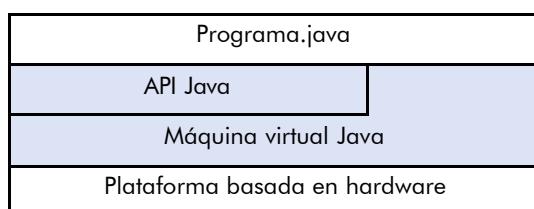
- Máquina virtual (MV). Como ya hemos comentado, una de las principales características que proporciona Java es la independencia de la plataforma hardware: una vez compilados, los programas se deben poder ejecutar en cualquier plataforma.

La estrategia utilizada para conseguirlo es generar un código ejecutable “neutro” (bytecode) como resultado de la compilación. Este código neutro, que está muy orientado al código máquina, se ejecuta

desde una “máquina hipotética” o “máquina virtual”. Para ejecutar un programa en una plataforma determinada basta con disponer de una “máquina virtual” para dicha plataforma.

- *Application programming interface (API)*. El API de Java es una gran colección de software ya desarrollado que proporciona múltiples capacidades como entornos gráficos, comunicaciones, multiproceso, etc. Está organizado en librerías de clases relacionadas e interfaces. Las librerías reciben el nombre de *packages*.

En el siguiente esquema, se puede observar la estructura de la plataforma Java y como la máquina virtual aísla el código fuente (.java) del hardware de la máquina:



5.4.2. Mi primer programa en Java

Otra vez nuestro primer contacto con el lenguaje será mostrando un saludo al mundo. El desarrollo del programa se dividirá en tres fases:

- 1) **Crear un fichero fuente.** Mediante el editor de textos escogido, escribiremos el texto y lo salvaremos con el nombre *HolaMundo.java*.

HolaMundo.java

```
/**  
 * La clase HolaMundo muestra el mensaje  
 * "Hola Mundo" en la salida estándar.  
 */  
public class HolaMundo {  
    public static void main(String[] args) {  
        // Muestra "Hola Mundo!"  
        System.out.println("¡Hola Mundo!");  
    }  
}
```

2) **Compilar el programa** generando un fichero *bytecode*. Para ello, utilizaremos el compilador **javac**, que nos proporciona el entorno de desarrollo, y que traduce el código fuente a instrucciones que la JVM pueda interpretar.

Si después de teclear “javac HolaMundo.java” en el intérprete de comandos, no se produce ningún error, obtenemos nuestro primer programa en Java: un fichero *HolaMundo.class*.

3) **Ejecutar el programa** en la máquina virtual de Java. Una vez generado el fichero de *bytecodes*, para ejecutarlo en la JVM sólo deberemos escribir la siguiente instrucción, para que nuestro ordenador lo pueda interpretar, y nos aparecerá en pantalla el mensaje de bienvenida ¡Hola mundo!:

```
java HolaMundo
```

5.4.3. Las instrucciones básicas y los comentarios

En este punto, Java continua manteniéndose fiel a C++ y C y conserva su sintaxis.

La única consideración a tener en cuenta es que, en Java, las expresiones condicionales (por ejemplo, la condición `if`) deben retornar un valor de tipo `boolean`, mientras que C++, por compatibilidad con C, permitía el retorno de valores numéricos y asimilaba 0 a `false` y los valores distintos de 0 a `true`.

Respecto a los comentarios, Java admite las formas provenientes de C++ (`/* ... */` y `// ...`) y añade una nueva: incluir el texto entre las secuencias `/**` (inicio de comentario) y `*/` (fin de comentario).

De hecho, la utilidad de esta nueva forma no es tanto la de comentar, sino la de documentar. Java proporciona herramientas (por ejemplo, `javadoc`) para generar documentación a partir de los códigos fuentes que extraen el contenido de los comentarios realizados siguiendo este modelo.

Ejemplo

```
/**  
 * Texto comentado con la nueva forma de Java para su  
 * inclusión en documentación generada automáticamente  
 */
```

5.5. Diferencias entre C++ y Java

Como se ha comentado, el lenguaje Java se basó en C++ para proporcionar un entorno de programación orientado a objetos que resultará muy familiar a un gran número de programadores. Sin embargo, Java intenta mejorar C++ en muchos aspectos y, sobre todo, elimina aquellos que permitían a C++ trabajar de forma “no orientada a objetos” y que fueron incorporados por compatibilidad con el lenguaje C.

5.5.1. Entrada/salida

Como Java está pensado principalmente para trabajar de forma gráfica, las clases que gestionan la entrada / salida en modo texto se han desarrollado de manera muy básica. Están reguladas por la clase `System` que se encuentra en la librería `java.lang`, y de esta clase se destacan tres objetos estáticos que son los siguientes:

- `System.in`. Recibe los datos desde la entrada estándar (normalmente el teclado) en un objeto de la clase `InputStream` (flujo de entrada).
- `System.out`. Imprime los datos en la salida estándar (normalmente la pantalla) un objeto de la clase `OutputStream` (flujo de salida).
- `System.err`. Imprime los mensajes de error en pantalla.

Los métodos básicos de que disponen estos objetos son los siguientes:

- `System.in.read()`. Lee un carácter y lo devuelve en forma de entero.
- `System.out.print(var)`. Imprime una variable de cualquier tipo primitivo.
- `System.out.println(var)`. Igual que el anterior pero añadiendo un salto de línea final.

Por tanto, para escribir un mensaje nos basta utilizar básicamente las instrucciones `System.out.print()` y `System.out.println()`:

```
int unEntero = 35;
double unDouble = 3.1415;

System.out.println("Mostrando un texto");
System.out.print("Mostrando un entero ");
System.out.println (unEntero);
System.out.print("Mostrando un double ");
System.out.println (unDouble);
```

Mientras que la salida de datos es bastante natural, la entrada de datos es mucho menos accesible pues el elemento básico de lectura es el carácter. A continuación se presenta un ejemplo en el que se puede observar el proceso necesario para la lectura de una cadena de caracteres:

```
String miVar;
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
// La entrada finaliza al pulsar la tecla Entrar
miVar = br.readLine();
```

Si se desea leer líneas completas, se puede hacer a través del objeto `BufferedReader`, cuyo método `readLine()` llama a un lector de caracteres (un objeto `Reader`) hasta encontrar un símbolo de final de línea ("`\n`" o "`\r`"). Pero en este caso, el flujo de entrada es un objeto `InputStream`, y no tipo `Reader`. Entonces, necesitamos una clase que actúe como lectora para un flujo de datos `InputStream`. Será la clase `InputStreamReader`.

No obstante, el ejemplo anterior es válido para `Strings`. Cuando se desea leer un número entero u otros tipos de datos, una vez realizada la lectura, se debe hacer la conversión. Sin embargo, esta conversión puede llegar a generar un error fatal en el sistema si el texto introducido no coincide con el tipo esperado. En este caso, Java nos obliga a considerar siempre dicho control de errores. La gestión de errores (que provocan las llamadas excepciones) se hace, igual que en C++, a través de la sentencia `try {...} catch {...} finally {...}`.

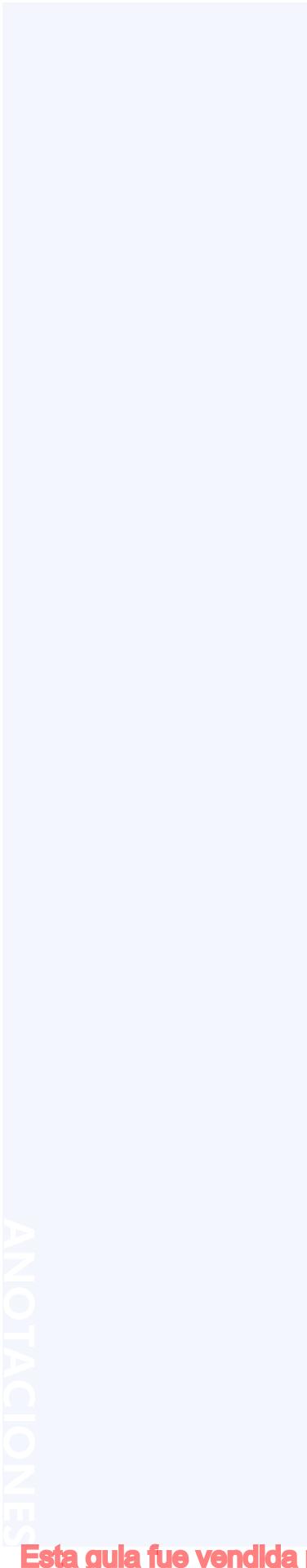
A continuación, veremos cómo se puede diseñar una clase para que devuelva un número entero leído desde teclado:

```
Leer.java
import java.io.*;
public class Leer
{
    public static String getString()
    {
        String str = "";
        try
        {
            InputStreamReader isr = new
                InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            str = br.readLine();
        }
        catch(IOException e)
        {
            System.err.println("Error: " + e.getMessage());
        }
        return str; // devolver el dato tecleado
    }

    public static int getInt()
    {
        try
        {
            return Integer.parseInt(getString());
        }
        catch(NumberFormatException e)
        {
            return Integer.MIN_VALUE; // valor más pequeño
        }
    }

    // getInt
    // se puede definir una función para cada tipo...
    public static double getDouble() {} // getDouble

}
// Leer
```



En el bloque `try { ... }` se incluye el trozo de código susceptible de sufrir un error. En caso de producirse, se lanza una excepción que es recogida por el bloque `catch { ... }`.

En el caso de la conversión de tipos `String` a números, la excepción que se puede producir es del tipo `NumberFormatException`. Podría haber más bloques `catch` para tratar diferentes tipos de excepción. En el ejemplo, si se produce error el valor numérico devuelto corresponde al mínimo valor posible que puede tomar un número entero.

El bloque `finally { ... }` corresponde a un trozo de código a ejecutar tanto si ha habido error, como si no (por ejemplo, cerrar ficheros), aunque su uso es opcional.

De forma similar, se pueden desarrollar funciones para cada uno de los tipos primitivos de Java. Finalmente, la lectura de un número entero sería como sigue:

```
int i;  
...  
i = Leer.getInt();
```

5.5.2. El preprocesador

Java no dispone de preprocesador, por lo que diferentes órdenes (generalmente, originarias de C) se eliminan. Entre éstas, las más conocidas son las siguientes:

- `defines`. Estas órdenes para la definición de constantes, ya en C++ habían perdido gran parte de su sentido al poder declarar variables `const`, y ahora se implementan a partir de las variables `final`.
- `include`. Esta orden, que se utilizaba para incluir el contenido de un fichero, era muy útil en C++, principalmente para la reutilización de los ficheros de cabeceras. En Java, no hay ficheros de cabecera y las librerías (o paquetes) se incluyen mediante la instrucción `import`.

5.5.3. La declaración de variables y constantes

La declaración de variables se mantiene igual, pero la definición de constantes cambia de forma: en Java, se antecede la variable con la palabra reservada `final`; no es necesario asignarle un valor en el momento de la declaración. No obstante, en el momento en que se le asigne un valor por primera vez, ya no puede ser modificado.

```
final int i;  
int j = 2;  
...  
i = j + 2; // asignado el valor, no se podrá modificar
```

5.5.4. Los tipos de datos

Java clasifica los tipos de datos en dos categorías: primitivos y referencias. Mientras el primero contiene el valor, el segundo sólo contiene la dirección de memoria donde está almacenada la información.

Los tipos primitivos de datos de Java (`byte`, `short`, `int`, `long`, `float`, `double`, `char` y `boolean`) básicamente coinciden con los de C++, aunque con algunas modificaciones, que presentamos a continuación:

- Los tipos numéricos tienen el mismo tamaño independientemente de la plataforma en que se ejecute.
- Para los tipos numéricos no existe el especificador `unsigned`.
- El tipo `char` utiliza el conjunto de caracteres Unicode, que tiene 16 bits. Los caracteres del 0 al 127 coinciden con los códigos ASCII.
- Si no se inicializa las variables explícitamente, Java inicializa los datos a cero (o a su equivalente) automáticamente eliminando así los valores basura que pudieran contener.

Los tipos referencia en Java son los vectores, clases e interfaces. Las variables de estos tipos guardan su dirección de memoria, lo que po-

dría asimilarse a los apuntadores en otros lenguajes. No obstante, al no permitir las operaciones explícitas con las direcciones de memoria, para acceder a ellas bastará con utilizar el nombre de la variable.

Por otro lado, Java elimina los tipos `struct` y `union` que se pueden implementar con `class` y que se mantenían en C++ por compatibilidad con C. También elimina el tipo `enum`, aunque se puede emular utilizando constantes numéricas con la palabra clave `final`.

También se eliminan definitivamente los `typedefs` para la definición de tipos, que en C++ ya habían perdido gran parte de su sentido al hacer que las clases, Structs, Union y Enum fueran tipos propios.

Finalmente, sólo admite las coerciones de tipos automáticas (`type casting`) en el caso de conversiones seguras; es decir, donde no haya riesgo de perder ninguna información. Por ejemplo, admite las conversiones automáticas de tipo `int` a `float`, pero no en sentido inverso donde se perderían los decimales. En caso de posible pérdida de información, hay que indicarle explícitamente que se desea realizar la conversión de tipos.

Otra característica muy destacable de Java es la implementación que realiza de los vectores. Los trata como a objetos reales y genera una excepción (error) cuando se superan sus límites. También dispone de un miembro llamado `length` para indicar su longitud, lo que proporciona un incremento de seguridad del lenguaje al evitar accesos indeseados a la memoria.

Para trabajar con cadenas de caracteres, Java dispone de los tipos `String` y `StringBuffer`. Las cadenas definidas entre comillas dobles se convierten automáticamente a objetos `String`, y no pueden modificarse. El tipo `StringBuffer` es similar, pero permite la modificación de su valor y proporciona métodos para su manipulación.

5.5.5. La gestión de variables dinámicas

Tal como se comentó al explicar C++, la gestión directa de la memoria es un arma muy potente pero también muy peligrosa: cual-

quier error en su gestión puede acarrear problemas muy graves en la aplicación y, quizás, en el sistema.

De hecho, la presencia de los apuntadores en C y C++ se debía al uso de cadenas y de vectores. Java proporciona objetos tanto para las cadenas, como los vectores, por lo que, para estos casos, ya no son necesarios los apuntadores. La otra gran necesidad, los pasos de parámetros por variable, queda cubierta por el uso de referencias.

Como en Java el tema de la seguridad es primordial, se optó por no permitir el uso de apuntadores, al menos en el sentido en que se entendían en C y C++.

En C++, se preveían dos formas de trabajar con apuntadores:

- Con su dirección, permitiendo incluso operaciones aritméticas sobre ella (apuntador).
- Con su contenido (* apuntador).

En Java se eliminan todas las operaciones sobre las direcciones de memoria. Cuando se habla de referencias se hace con un sentido diferente de C++. Una variable dinámica corresponde a la referencia al objeto (apuntador):

- Para ver el contenido de la variable dinámica, basta utilizar la forma (apuntador).
- Para crear un nuevo elemento, se mantiene el operador `new`.
- Si se asigna una variable tipo referencia (por ejemplo, un objeto) a otra variable del mismo tipo (otro objeto de la misma clase) el contenido no se duplica, sino que la primera variable apunta a la misma posición de la segunda variable. El resultado final es que el contenido de ambas es el mismo.



Java no permite operar directamente con las direcciones de memoria, lo que simplifica el acceso a su contenido: se hace a través del nombre de la variable (en lugar de utilizar la forma desreferenciada `*nombre_variable`).

Otro de los principales riesgos que entraña la gestión directa de la memoria es la de liberar correctamente el espacio ocupado por las variables dinámicas cuando se dejan de utilizar. Java resuelve esta problemática proporcionando una herramienta que libera automáticamente dicho espacio cuando detecta que ya no se va a volver a utilizar más. Esta herramienta conocida como *recolector de basura* (*garbage collector*) forma parte del Java durante la ejecución de sus programas. Por tanto, no es necesaria ninguna instrucción `delete`, basta con asignar el apuntador a `null`, y el recolector de memoria detecta que la zona de memoria ya no se utiliza y la libera.

Si lo deseamos, en lugar de esperar a que se produzca la recolección de basura automáticamente, podemos invocar el proceso a través de la función `gc()`. No obstante, para la JVM dicha llamada sólo se considera como una sugerencia.

5.5.6. Las funciones y el paso de parámetros

Como ya sabemos, Java sólo se permite programación orientada a objetos. Por tanto, no se admiten las funciones independientes (siempre deben incluirse en clases) ni las funciones globales. Además, la implementación de los métodos se debe realizar dentro de la definición de la clase. De este modo, también se elimina la necesidad de los ficheros de cabeceras. El mismo compilador detecta si una clase ya ha sido cargada para evitar su duplicación. A pesar de su similitud con las funciones `inline`, ésta sólo es formal porque internamente tienen comportamientos diferentes: en Java no se implementan las funciones `inline`.

Por otro lado, Java continua soportando la sobrecarga de funciones, aunque no permite al programador la sobrecarga de operadores, a pesar de que el compilador utiliza esta característica internamente.



En Java todos los parámetros se pasan por valor.

En el caso de los tipos de datos primitivos, los métodos siempre reciben una copia del valor original, que no se puede modificar.

En el caso de tipo de datos de referencia, también se copia el valor de dicha referencia. No obstante, por la naturaleza de las referencias, los cambios realizados en la variable recibida por parámetro también afectan a la variable original.

Para modificar las variables pasadas por parámetro a la función, debemos incluirlas como variables miembro de la clase y pasar como argumento la referencia a un objeto de dicha clase.

5.6. Las clases en Java

Como ya hemos comentado, uno de los objetivos que motivaron la creación de Java fue disponer de un lenguaje orientado a objetos “puro”, en el sentido que siempre se debería cumplir dicho paradigma de programación. Esto, por su compatibilidad con C, no ocurría en C++. Por tanto, las clases son el componente fundamental de Java: todo está incluido en ellas. La manera de definir las clases en Java es similar a la utilizada en C++, aunque se presentan algunas diferencias:

```
Punto2D.java
class Punto2D
{
    int x, y;

    // inicializando al origen de coordenadas
    Punto2D()
    {
        x = 0;
        y = 0;
    }

    // inicializando a una coordenada x,y determinada
    Punto2D(int coordx, int coordy)
    {
```

```

        x = coordx;
        y = coordy;
    }

    // calcula la distancia a otro punto
    float distancia(Punto2D npunto)
    {
        int dx = x - npunto.x;
        int dy = y - npunto.y;
        return ( Math.sqrt(dx * dx + dy * dy));
    }
}

```

- La primera diferencia es la inclusión de la definición de los métodos en el interior de la clase y no separada como en C++. Al seguir este criterio, ya no es necesario el operador de ámbito (::).
- La segunda diferencia es que en Java no es preciso el punto y coma (;) final.
- Las clases se guardan en un fichero con el mismo nombre y con la extensión .java (Punto2.java).

Una característica común a C y C++ es que Java también es sensible a las mayúsculas, por lo cual la clase Punto2D es diferente a punto2d o pUnto2d.

Java permite guardar más de una clase en un fichero pero sólo permite que una de ellas sea pública. Esta clase será la que dará el nombre al archivo. Por tanto, salvo raras excepciones, se suele utilizar un archivo independiente para cada clase.

En la definición de la clase, de forma similar a C++, se declaran los atributos (o variables miembro) y los métodos (o funciones miembro) tal como se puede observar en el ejemplo anterior.

5.6.1. Declaración de objetos

Una vez definida una clase, para declarar un objeto de dicha clase basa ta con anteponer el nombre de la clase (como un tipo más) al del objeto.

```
Punto2D puntoUno;
```

El resultado es que `puntoUno` es una referencia a un objeto de la clase `Punto2D`. Inicialmente, esta referencia tiene valor `null` y no ha hecho ninguna reserva de memoria. Para poder utilizar esta variable para guardar información, es necesario crear una instancia mediante el operador `new`. Al utilizarlo, se llama al **constructor** del objeto `Punto2D` definido.

```
puntoUno = new Punto2D(2,2); // inicializando a (2,2)
```

Una diferencia importante en Java respecto a C++, es el uso de referencias para manipular los objetos. Como se ha comentado anteriormente, la asignación de dos variables declaradas como objetos sólo implica la asignación de su referencia:

```
Punto2D puntoDos;
puntoDos = puntoUno;
```

Si se añade la instrucción anterior, no se ha hecho ninguna reserva específica de memoria para la referencia a objeto `puntoDos`. Al realizar la asignación, `puntoDos` hará referencia al mismo objeto apuntado por `puntoUno`, y no a una copia. Por tanto, cualquier cambio sobre los atributos de `puntoUno` se verán reflejados en `puntoDos`.

5.6.2. Acceso a los objetos

Una vez creado un objeto, se accede a cualquiera de sus atributos y métodos a través del operador punto `(.)` tal como hacíamos en C++.

```
int i;
float dist;

i = puntoUno.x;
dist = puntoUno.distancia(5,1);
```

En C++ se podía acceder al objeto a través de la desreferencia de un apuntador a dicho objeto (`*apuntador`), en cuyo caso, el acceso a sus atributos o métodos podía hacerse a través del operador

punto (*apuntador.atributo) o a través de su forma de acceso abreviada mediante el operador → (apuntador→atributo). En Java, al no existir la forma desreferenciada *apuntador, tampoco existe el operador →.

Finalmente Java, igual que C++, permite el acceso al objeto dentro de los métodos de la clase a través del objeto this.

5.6.3. Destrucción de objetos

Cada vez que se crea un objeto, cuando se deja de utilizar debe ser destruido. La forma de operar de la gestión de memoria en Java permite evitar muchos de los conflictos que aparecen en otros lenguajes y es posible delegar esta responsabilidad a un proceso automático: el recolector de basura (*garbage collector*), que detecta cuando una zona de memoria no está referenciada y, cuando el sistema dispone de un momento de menor intensidad de procesador, la libera.

Algunas veces, al trabajar con una clase se utilizan otros recursos adicionales, como los ficheros. Frecuentemente, al finalizar la actividad de la clase, también se debe poder cerrar la actividad de dichos recursos adicionales. En estos casos, es preciso realizar un proceso manual semejante a los destructores en C++. Para ello, Java permite la implementación de un método llamado `finalize()` que, en caso de existir, es llamado por el mismo recolector. En el interior de este método, se escribe el código que libera explícitamente los recursos adicionales utilizados. El método `finalize` siempre es del tipo `static void`.

```
class MiClase
{
    MiClase()    //constructor
    {
        ...
        //instrucciones de inicialización
    }

    static void finalize()    //destructor
    {
        ...
        //instrucciones de liberación de recursos
    }
}
```

5.6.4. Constructores de copia

C++ dispone de los constructores de copia para asegurar que se realiza una copia completa de los datos en el momento de hacer una asignación, o de asignar un parámetro o un valor de retorno de una función.

Tal como se ha comprobado, Java tiene una filosofía diferente. Las asignaciones entre objetos no implican una copia de su contenido, sino que la segunda referencia pasa a referenciar al primer objeto. Por tanto, siempre se accede al mismo contenido y no es necesaria ninguna operación de reserva de memoria adicional. Como consecuencia de este cambio de filosofía, Java no precisa de constructores de copia.

5.6.5. Herencia simple y herencia múltiple

En Java, para indicar que una clase deriva de otra (es decir, hereda total o parcialmente sus atributos y métodos) se hace a través del término extends. Retomaremos el ejemplo de los perros y los mamíferos.

```
class Mamifero
{
    int edad;

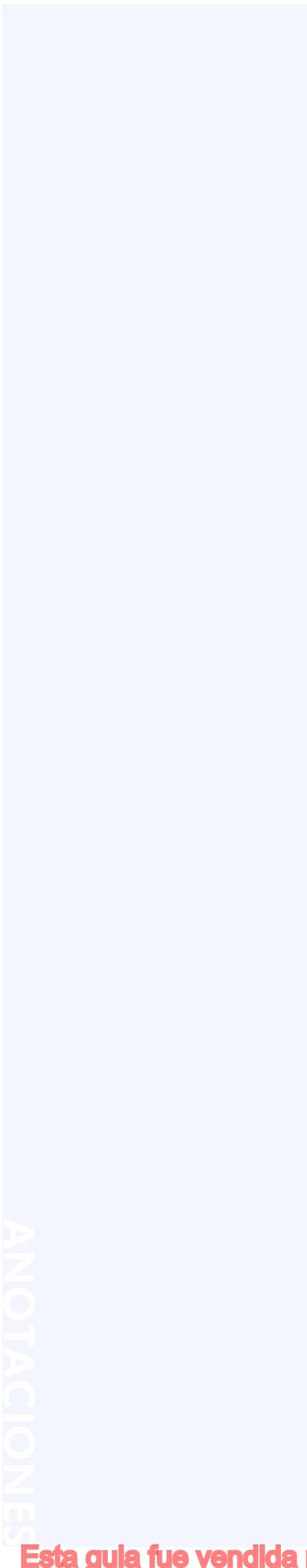
    Mamifero()
    { edad = 0; }

    void asignarEdad(int nEdad)
    { edad = nEdad; }

    int obtenerEdad()
    { return (edad); }

    void emitirSonido()
    { System.out.println("Sonido "); }
}

class Perro extends Mamifero
{
    void emitirSonido()
    { System.out.println("Guau "); }
}
```



En el ejemplo anterior, se dice que la clase Perro es una clase derivada de la clase Mamífero. También es posible leer la relación en el sentido contrario indicando que la clase Mamífero es una superclase de la clase Perro.



En C++ era posible la herencia múltiple, es decir, recibir los atributos y métodos de varias clases. Java no admite esta posibilidad, aunque en cierta manera permite una funcionalidad parecida a través de las interfaces.

5.7. Herencia y polimorfismo

La herencia y el polimorfismo son propiedades esenciales dentro del paradigma del diseño orientado a objetos. Estos conceptos ya han sido comentados en la unidad dedicada a C++ y continúan siendo vigentes en Java. No obstante, hay variaciones en su implementación que comentamos a continuación.

5.7.1. Las referencias this y super

En algunas ocasiones, es necesario acceder a los atributos o métodos del objeto que sirve de base al objeto en el cual se está. Tal como se ha visto, tanto Java como C++ proporciona este acceso a través de la referencia `this`.

La novedad que proporciona Java es poder acceder también a los atributos o métodos del objeto de la superclase a través de la referencia `super`.

5.7.2. La clase Object

Otra de las diferencias de Java respecto a C++ es que todos los objetos pertenecen al mismo árbol de jerarquías, cuya raíz es la clase `Object` de la cual heredan todas las demás: si una clase, en su definición, no tiene el término `Extends`, se considera que hereda directamente de `Object`.



Podemos decir que la clase Object es la superclase de la cual derivan directa o indirectamente todas las demás clases en Java.

La clase Object proporciona una serie de métodos comunes, entre los cuales los siguientes:

- `public boolean equals (Object obj)`. Se utiliza para comparar el contenido de dos objetos y devuelve `true` si el objeto recibido coincide con el objeto que lo llama. Si sólo se desean comparar dos referencias a objeto, se pueden utilizar los operadores de comparación `==` y `!=`.
- `protected Object Clone()`. Retorna una copia del objeto.

5.7.3. Polimorfismo

C++ implementaba la capacidad de una variable de poder tomar varias formas a través de apuntadores a objetos. Como se ha comentado, Java no dispone de apuntadores y cubre esta función a través de referencias, pero el funcionamiento es similar.

```
Mamifero mamiferoUno = new Perro;  
Mamifero mamiferoDos = new Mamifero;
```



Recordemos que, en Java, la declaración de un objeto siempre corresponde a una referencia a éste.

5.7.4. Clases y métodos abstractos

En C++ se comentó que, en algunos casos, las clases corresponden a elementos teóricos de los cuales no tiene ningún sentido instanciar objetos, sino que siempre se tenía que crear objetos de sus clases derivadas.

La implementación en C++ se hacía a través de las funciones virtuales puras, cuya forma de representarla es, como menos, un poco peculiar: se declaraban asignando la función virtual a 0.

La implementación de Java para estos casos es mucho más sencilla: anteponer la palabra reservada `abstract` al nombre de la función. Al declarar una función como `abstract`, ya se indica que la clase también lo es. No obstante, es recomendable explicitarlo en la declaración anteponiendo la palabra `abstract` a la palabra reservada `class`.

El hecho de definir una función como `abstract` obliga a que las clases derivadas que puedan recibir este método la redefinan. Si no lo hacen, heredan la función como abstracta y, como consecuencia, ellas también lo serán, lo que impedirá instanciar objetos de dichas clases.

```
abstract class ObraDeArte
{
    String autor;

    ObraDeArte() {} //constructor

    abstract void mostrarObraDeArte(); //abstract
    void asignarAutor(String nAutor)
    {
        autor = nAutor;
    }
    String obtenerAutor();
    {
        return (autor);
    }
}
```

En el ejemplo anterior, se ha declarado como abstracta la función `mostrarObraDeArte()`, lo que obliga a redefinirla en las clases derivadas. Por tanto, no incluye ninguna definición. Por otro lado, destacamos que, al ser una clase abstracta, no será posible hacer un `new ObraDeArte`.

5.7.5. Clases y métodos finales

En la definición de variables, ya se ha tratado el concepto de variables finales. Hemos dicho que las variables finales, una vez inicializadas,

zadas, no pueden ser modificadas. El mismo concepto se puede aplicar a clases y métodos:

- Las clases finales no tienen ni pueden tener clases derivadas.
- Los métodos finales no pueden ser redefinidos en las clases derivadas.



El uso de la palabra reservada `final` se convierte en una medida de seguridad para evitar usos incorrectos o maliciosos de las propiedades de la herencia que pudiesen suplantar funciones establecidas.

5.7.6. Interfaces

Una interfaz es una colección de definiciones de métodos (sin sus implementaciones), cuya función es definir un protocolo de comportamiento que puede ser implementado por cualquier clase independientemente de su lugar en la jerarquía de clases.

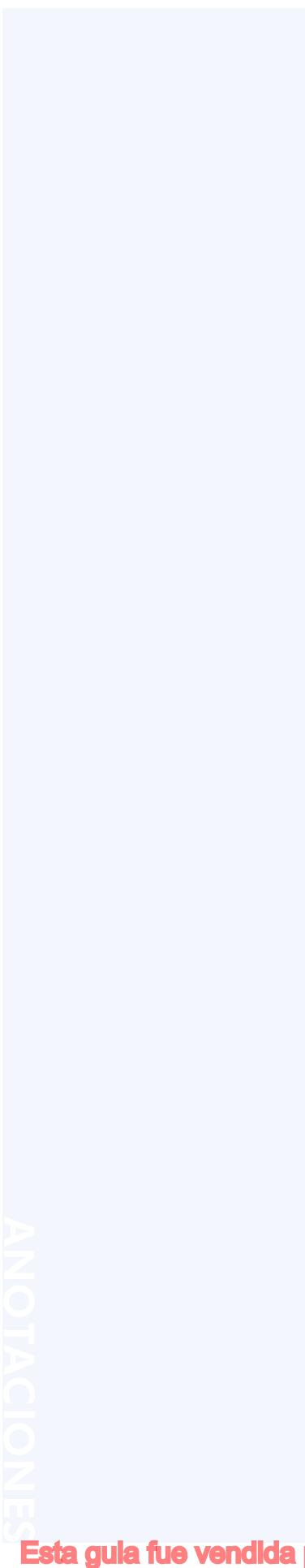
Al indicar que una clase implementa una interfaz, se le obliga a redefinir todos los métodos definidos. En este aspecto, las interfaces se asemejan a las clases abstractas. No obstante, mientras una clase sólo puede heredar de una superclase (sólo permite herencia simple), puede implementar varias interfaces. Ello sólo indica que cumple con cada uno de los protocolos definidos en cada interfaz.

A continuación presentamos un ejemplo de declaración de interfaz:

```
public interface NombreInterfaz  
Extends SuperInterfaz1, SuperInterfaz2  
{ cuerpo interfaz }
```



Si una interfaz no se especifica como pública, sólo será accesible para las clases definidas en su mismo paquete.



El cuerpo de la interfaz contiene las declaraciones de todos los métodos incluidos en ella. Cada declaración se finaliza en punto y coma (;) pues no tienen implementaciones e implícitamente se consideran `public` y `abstract`.

El cuerpo también puede incluir constantes en cuyo caso se consideran `public`, `static` y `final`.

Para indicar que una clase implementa una interface, basta con añadir la palabra clave `implements` en su declaración. Java permite la herencia múltiple de interfaces:

```
class MiClase extends SuperClase  
implements Interfaz1, interfaz2  
{ ... }
```

Cuando una clase declara una interfaz, es como si firmara un contrato por el cual se compromete a implementar los métodos de la interfaz y de sus superinterfaces. La única forma de no hacerlo es declarar la clase como `abstract`, con lo cual no se podrá instanciar objetos y se transmitirá esa obligación a sus clases derivadas.

De hecho, a primera vista parece que hay muchas similitudes entre las clases abstractas y las interfaces pero las diferencias son significativas:

- Una interfaz no puede implementar métodos, mientras que las clases abstractas sí que lo hacen.
- Una clase puede tener varias interfaces, pero sólo una superclase.
- Las interfaces no forman parte de la jerarquía de clases y, por tanto, clases no relacionadas pueden implementar la misma interfaz.

Otra característica relevante de las interfaces es que al definirlas se está declarando un nuevo tipo de datos referencia. Una variable de dicho tipo de datos se podrá instanciar por cualquier clase que implemente esa interfaz. Esto proporciona otra forma de aplicar el polimorfismo.

5.7.7. Paquetes

Para organizar las clases, Java proporciona los paquetes. Un paquete (package) es una colección de clases e interfaces relacionadas que proporcionan protección de acceso y gestión del espacio de nombres. Las clases e interfaces siempre pertenecen a un paquete.

Nota

De hecho, las clases e interfaces que forman parte de la plataforma de Java pertenecen a varios paquetes organizados por su función: `java.lang` incluye las clases fundamentales, `java.io` las clases para entrada/salida, etc.



El hecho de organizar las clases en paquetes evita en gran medida que pueda haber una colisión en la elección del nombre.

Para definir una clase o una interfaz en un paquete, basta con incluir en la primera línea del archivo la expresión siguiente:

```
package miPaquete;
```

Si no se define ningún paquete, se incluye dentro del paquete por defecto (`default package`), lo que es una buena solución para pequeñas aplicaciones o cuando se comienza a trabajar en Java.

Para acceder al nombre de la clase, se puede hacer a través del nombre largo:

```
miPaquete.MiClase
```

Otra posibilidad es la importación de las clases públicas del paquete mediante la palabra clave `import`. Después, es posible utilizar el nombre de la clase o de la interfaz en el programa sin el prefijo de éste:

```
import miPaquete.MiClase; //importa sólo la clase  
import miPaquete.*          // importa todo el paquete
```

Ejemplo

La importación de `java.awt` no incluye las clases del subpaquete `java.awt.event`.

Hay que tener en cuenta que importar un paquete no implica importar los diferentes subpaquetes que pueda contener.



Por convención, Java siempre importa por defecto del paquete `java.lang`.

Para organizar todas las clases y paquetes posibles, se crea un subdirectorio para cada paquete donde se incluyen las diferentes clases de dicho paquete. A su vez, cada paquete puede tener sus subpaquetes, que se encontrarán en un subdirectorio. Con esta organización de directorios y archivos, tanto el compilador como el intérprete tienen un mecanismo automático para localizar las clases que necesitan otras aplicaciones.

Ejemplo

La clase `graficos.figuras.rectangulo` se encontraría dentro del paquete `graficos.figuras` y el archivo estaría localizado en `graficos\figuras\rectangulo.java`.

5.7.8. El API (*applications programming interface*) de Java

La multitud de bibliotecas de funciones que proporciona el mismo lenguaje es una de las bazas primordiales de Java; bibliotecas, que están bien documentadas, son estándar y funcionan para las diferentes plataformas.

Este conjunto de bibliotecas está organizado en paquetes e incluido en la API de Java. Las principales clases son las siguientes:

Tabla 9.

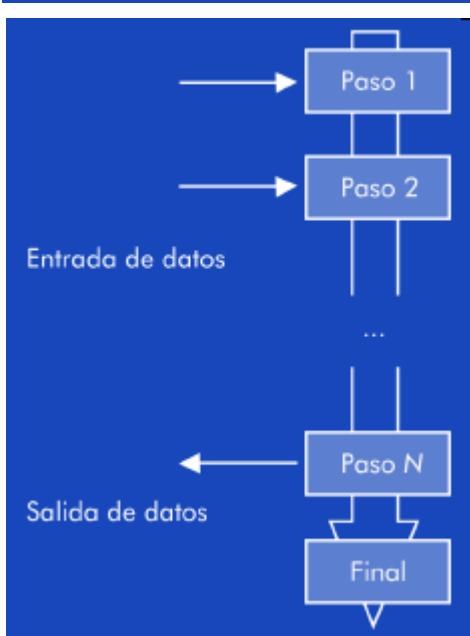
Paquete	Clases incorporadas
<code>java.lang</code> .	Clases fundamentales para el lenguaje como la clase <code>String</code> y otras.
<code>java.io</code>	Clases para la entrada y salida a través de flujos de datos, y ficheros del sistema.
<code>java.util</code>	Clases de utilidad como colecciones de datos y clases, el modelo de eventos, facilidades horarias, generación aleatoria de números, y otras.

Paquete	Clases incorporadas
java.math	Clase que agrupa todas las funciones matemáticas.
java.applet	Clase con utilidades para crear <i>applets</i> y clases que las <i>applets</i> utilizan para comunicarse con su contexto.
java.awt	Clases que permiten la creación de interfaces gráficas con el usuario, y dibujar imágenes y gráficos.
javax.swing	Clases con componentes gráficos que funcionan igual en todas las plataformas Java.
java.security	Clases responsables de la seguridad en Java (encriptación, etc.).
java.net	Clases con funciones para aplicaciones en red.
java.sql	Clase que incorpora el JDBC para la conexión de Java con bases de datos.

5.8. El paradigma de la programación orientada a eventos

Los diversos paradigmas de programación que se han revisado hasta el momento se caracterizan por tener un flujo de instrucciones secuencial y considerar los datos como el complemento necesario para el desarrollo de la aplicación. Su funcionamiento implica normalmente un inicio, una secuencia de acciones y un final de programa:

Figura 15.



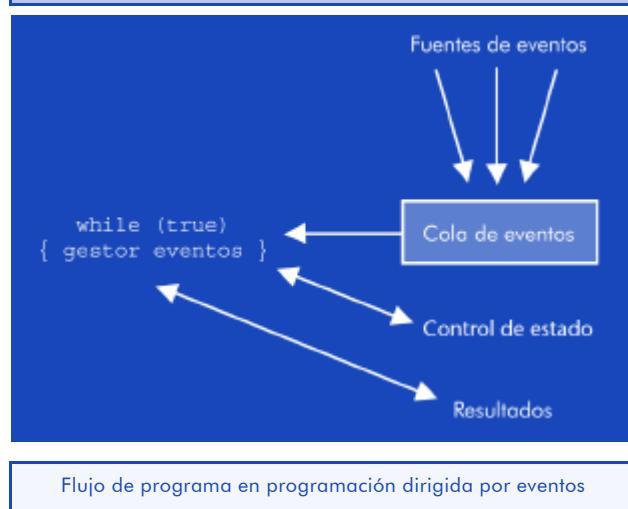
Flujo de programa en programación imperativa

Dentro de este funcionamiento secuencial, el proceso recibe sucesos externos que pueden ser esperados (entradas de datos del usuario por teclado, ratón u otras formas, lecturas de información del sistema, etc.) o inesperados (errores de sistema, etc.). A cada uno de estos sucesos externos lo denominaremos **evento**.

En los paradigmas anteriores, los eventos no alteran el orden del flujo de instrucciones previsto: se les atiende para resolverlos o, si no es posible, se produce una finalización del programa.

En el paradigma de programación dirigida por eventos no se fija una secuencia única de acciones, sino que prepara reacciones a los eventos que puedan ir sucediendo una vez iniciada la ejecución del programa. Por tanto, en este modelo son los datos introducidos los que regulan la secuencia de control de la aplicación. También se puede observar que las aplicaciones difieren en su diseño respecto de los paradigmas anteriores: están preparadas para permanecer en funcionamiento un tiempo indefinido, recibiendo y gestionando eventos.

Figura 16.



5.8.1. Los eventos en Java

Para la gestión de los eventos, Java propone utilizar **el modelo de delegación de eventos**. En este modelo, un componente recibe un evento y se lo transmite al gestor de eventos que tiene asignado para que lo gestione (*event listener*). Por tanto, tendremos una separación

del código entre la generación del evento y su manipulación que nos facilitará su programación.

Diferenciaremos los cuatro tipos de elementos que intervienen:

- El evento (qué se recibe). En la gran mayoría de los casos, es el sistema operativo quien proporciona el evento y gestiona finalmente todas las operaciones de comunicaciones con el usuario y el entorno. Se almacena en un objeto derivado de la clase Event y que depende del tipo de evento sucedido. Los principales tienen relación con el entorno gráfico y son: ActionEvent, KeyEvent, MouseEvent, AdjustmentEvent, WindowEvent, TextEvent, ItemEvent, FocusEvent, ComponentEvent, ContainerEvent.

Cada una de estas clases tiene sus atributos y sus métodos de acceso.

- La fuente del evento (dónde se recibe). Corresponde al elemento donde se ha generado el evento y, por tanto, recoge la información para tratarla o, en nuestro caso, para traspasarla a su gestor de eventos. En entornos gráficos, suele corresponder al elemento con el cual el usuario ha interactuado (un botón, un cuadro de texto, etc.).
- El gestor de eventos (quién lo gestiona). Es la clase especializada que indica, para cada evento, cuál es la respuesta deseada. Cada gestor puede actuar ante diferentes tipos de eventos con sólo asignarle los perfiles adecuados.
- El perfil del gestor (qué operaciones debe implementar el gestor). Para facilitar esta tarea existen interfaces que indican los métodos a implementar para cada tipo de evento. Normalmente, el nombre de esta interfaz es de la forma <nombreEvento>Listener (literalmente, "el que escucha el evento").

Ejemplo

KeyListener es la interfaz para los eventos de teclado y considera los tres métodos siguientes: keyPressed, keyReleased y keyTyped. En algunos casos, la obligación de implementar todos los métodos supone una carga innecesaria. Para estas situaciones, Java proporciona adaptadores <nombreEvento>Adapter que implementan los diferentes métodos vacíos permitiendo así redefinir sólo aquellos métodos que nos interesan.

Ejemplo

Si a un objeto botón de la clase Button deseamos añadirle un Listener de los eventos de ratón haremos:

```
boton.addMouseListener(gestorEventos);
```

Los principales perfiles (o interfaces) definidos por Java son los siguientes: ActionListener, KeyListener, MouseListener, WindowListener, TextListener, ItemListener, FocusListener, AdjustmentListener, ComponentListener y ContainerListener. Todos ellos derivados de la interfaz EventListener.

Finalmente, basta con establecer la relación entre la fuente del evento y su gestor. Para ello, en la clase fuente añadiremos un método del tipo add<nombreEvento>Listener.

De hecho, se podría considerar que los eventos no son realmente enviados al gestor de eventos, sino que es el propio gestor de eventos el que es asignado al evento.

Nota

Comprenderemos más fácilmente el funcionamiento de los eventos a través de un ejemplo práctico, como el que muestra la creación de un applet mediante la librería gráfica Swing que se verá más adelante en esta unidad.

5.9. Hilos de ejecución (threads)

Los sistemas operativos actuales permiten la multitarea, al menos en apariencia, pues si el ordenador dispone de un único procesador, solo podrá realizar una actividad a la vez. No obstante, se puede organizar el funcionamiento de dicho procesador para que reparta su tiempo entre varias actividades o para que aproveche el tiempo que le deja libre una actividad para continuar la ejecución de otra.

A cada una de estas actividades se le llama proceso. Un proceso es un programa que se ejecuta de forma independiente y con un espacio propio de memoria. Por tanto, los sistemas operativos multitarea permiten la ejecución de varios procesos a la vez.

Cada uno de estos procesos puede tener uno o varios hilos de ejecución, cada uno de los cuales corresponde a un flujo secuencial de

instrucciones. En este caso, todos los hilos de ejecución comparten el mismo espacio de memoria y se utiliza el mismo contexto y los mismos recursos asignados al proceso.

Java incorpora la posibilidad de que un proceso tenga múltiples hilos de ejecución simultáneos. El conocimiento completo de su implementación en Java supera los objetivos del curso y, a continuación, nos limitaremos a conocer las bases para la creación de los hilos y su ciclo de vida.

5.9.1. Creación de hilos de ejecución

En Java, hay dos formas de crear hilos de ejecución:

- Crear una nueva clase que herede de `java.lang.Thread` y sobrecargar el método `run()` de dicha clase.
- Crear una nueva clase con la interfaz `java.lang.Runnable` donde se implementará el método `run()`, y después crear un objeto de tipo `Thread` al que se le pasa como argumento un objeto de la nueva clase.

Siempre que sea posible se utilizará la primera forma, por su simplicidad. No obstante, si la clase ya hereda de alguna otra superclase, no será posible derivar también de la clase `Thread` (Java no permite la herencia múltiple), con lo cual se deberá escoger la segunda forma.

Veamos un ejemplo de cada una de las formas de crear hilos de ejecución:

Creación de hilos de ejecución derivando de la clase `Thread`

`ProbarThread.java`

```
class ProbarThread
{
    public static void main(String args[])
    {
        AThread a = new AThread();
        BThread b = new BThread();
    }
}
```

```
a.start();
b.start();
}

}

class AThread extends Thread
{
    public void run()
    {
        int i;
        for (i=1;i<=10; i++)
            System.out.print(" A"+i);
    }
}

class BThread extends Thread
{
    public void run()
    {
        int i;
        for (i=1;i<=10; i++)
            System.out.print(" B"+i);
    }
}
```

En el ejemplo anterior, se crean dos nuevas clases que derivan de la clase *Thread*: las clases *AThread* y *BThread*. Cada una de ellas muestra en pantalla un contador precedido por la inicial del proceso.

En la clase *ProbarThreads*, donde tenemos el método *main()*, se procede a la instanciación de un objeto para cada una de las clases *Thread* y se inicia su ejecución. El resultado final será del tipo (aunque no por fuerza en este orden):

A1 B1 A2 B2 A3 B3 A4 B4 A5 B5 A6 B6 A7 B7 A8 B8 A9 B9 A10 B10

Finalmente, solo hacer notar que en la ejecución *ProbarThreads* se ejecutan 3 hilos: el principal y los dos creados.

Creación de hilos de ejecución implementando la interfaz Runnable**Probar2Thread.java**

```
class Probar2Thread
{
    public static void main(String args[])
    {
        AThread a = new AThread();
        BThread b = new BThread();
        a.start();
        b.start();
    }
}

class AThread implements Runnable
{
    Thread t;
    public void start()
    {
        t = new Thread(this);
        t.start();
    }

    public void run()
    {
        int i;
        for (i=1;i<=50; i++)
            System.out.print(" A"+i);
    }
}

class BThread implements Runnable
{
    Thread t;

    public void start()
    {
        t = new Thread(this);
        t.start();
    }

    public void run()
    {
        int i;
        for (i=1;i<=50; i++)
            System.out.print(" B"+i);
    }
}
```

En este ejemplo, se puede observar que la clase principal main() no ha cambiado, pero sí lo ha hecho la implementación de cada una de las clases AThread y BThread. En cada una de ellas, además de implementar la interfaz Runnable, se tiene que definir un objeto de la clase Thread y redefinir el método start() para que llame al start() del objeto de la clase Thread pasándole el objeto actual this.

Para finalizar, dos cosas: es posible pasarle un nombre a cada hilo de ejecución para identificarlo, puesto que la clase Thread tiene el constructor sobrecargado para admitir esta opción:

```
public Thread (String nombre);  
public Thread (Runnable destino, String nombre);
```

Siempre es posible recuperar el nombre a través del método:

```
public final String getName();
```

5.9.2. Ciclo de vida de los hilos de ejecución

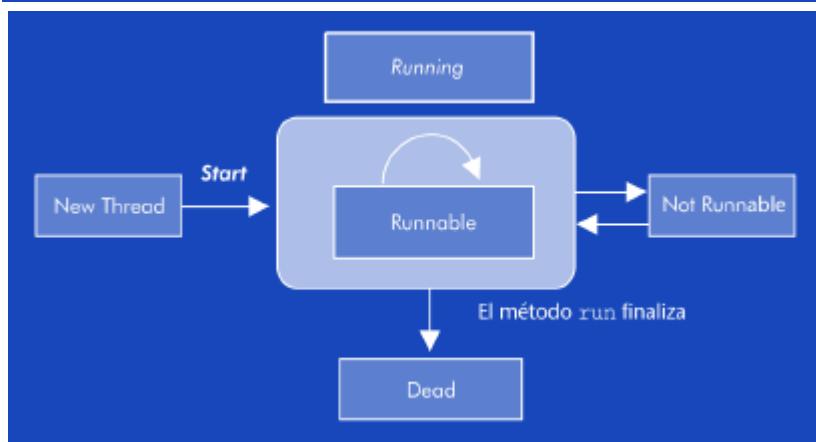
El ciclo de vida de los hilos de ejecución se puede representar a partir de los estados por los que pueden pasar:

- Nuevo (new): el thread se acaba de crear pero todavía no está inicializado, es decir, todavía no se ha ejecutado el método start().
- Ejecutable (runnable): el thread se está ejecutando o está en disposición para ello.
- Bloqueado (blocked o not runnable): el thread está bloqueado por algún mensaje interno sleep(), suspend() o wait() o por alguna actividad interna, por ejemplo, en espera de una entrada de datos. Si está en este estado, no entra dentro de la lista de tareas a ejecutar por el procesador.

Para volver al estado de Ejecutable, debe recibir un mensaje interno resume() o notify() o finalizar la situación que provocaba el bloqueo.

- Muerto (dead): el método habitual de finalizar un *thread* es que haya acabado de ejecutar las instrucciones del método `run()`. También podría utilizarse el método `stop()`, pero es una opción considerada “peligrosa” y no recomendada.

Figura 17.



5.10. Los applets

Un *applet* es una miniaPLICACIÓN Java preparada para ser ejecutada en un navegador de Internet. Para incluir un *applet* en una página HTML, basta con incluir su información por medio de las etiquetas `<APPLET> ... </APPLET>`.

La mayoría de navegadores de Internet funcionan en un entorno gráfico. Por tanto, los *applet* deben adaptarse a él a través de bibliotecas gráficas. En este apartado, se utilizará la biblioteca `java.awt` que es la biblioteca proporcionada originalmente desde sus primeras versiones. Una discusión más profunda entre las diferentes bibliotecas disponibles se verá más adelante en esta unidad.

Las características principales de los *applets* son las siguientes:

- Los ficheros `.class` se descargan a través de la red desde un servidor HTTP hasta el navegador, donde la JVM los ejecuta.
- Dado que se usan a través de Internet, se ha establecido que tengan unas restricciones de seguridad muy fuertes, como por ejem-

plo, que sólo puedan leer y escribir ficheros desde su servidor (y no desde el ordenador local), que sólo puedan acceder a información limitada en el ordenador donde se ejecutan, etc.

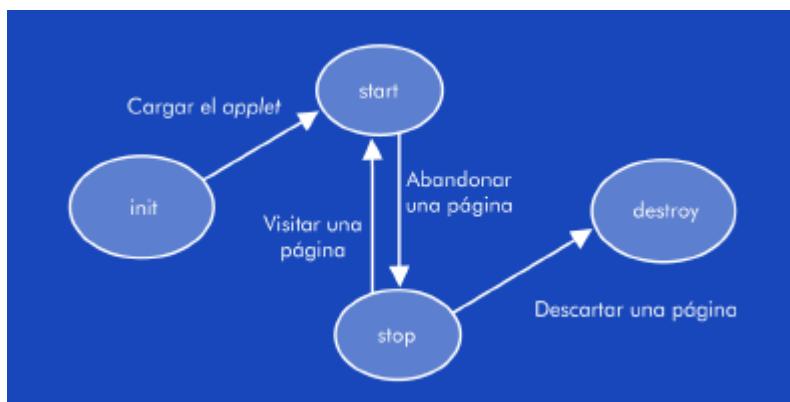
- Los *applets* no tienen ventana propia, que se ejecutan en una ventana del navegador.
- Desde el punto de vista del programador, destacan los siguientes aspectos:
- No necesitan método `main`. Su ejecución se inicia por otros mecanismos.
- Derivan siempre de la clase `java.applet.Applet` y, por tanto, deben redefinir algunos de sus métodos como `init()`, `start()`, `stop()` y `destroy()`.
- También suelen redefinir otros métodos como `paint()`, `update()` y `repaint()` heredados de clases superiores para tareas gráficas.
- Disponen de una serie de métodos para obtener información sobre el *applet* o sobre otros *applets* en ejecución en la misma página como `getAppletInfo()`, `getAppletContext()`, `getParameter()`, etc.

5.10.1. Ciclo de vida de los applets

Por su naturaleza, el ciclo de vida de un *applet* es algo más complejo que el de una aplicación normal. Cada una de las fases del ciclo de vida está marcada con una llamada a un método del *applet*:

- `void init()`. Se llama cuando se carga el *applet*, y contiene las inicializaciones que necesita.
- `void start()`. Se llama cuando la página se ha cargado, parado (por minimización de la ventana, cambio de página web, etc.) y se ha vuelto a activar.
- `void stop()`. Se llama de forma automática al ocultar el *applet*. En este método, se suelen parar los hilos que se están ejecutando para no consumir recursos innecesarios.
- `void destroy()`. Se llama a este método para liberar los recursos (menos la memoria) del *applet*.

Figura 18.



Al ser los *applets* aplicaciones gráficas que aparecen en una ventana del navegador, también es útil redefinir el siguiente método:

- void paint(Graphics g). En esta función se debe incluir todas las operaciones con gráficos, porque este método es llamado cuando el *applet* se dibuja por primera vez y cuando se redibuja.

5.10.2. Manera de incluir applets en una página HTML

Como ya hemos comentado, para llamar a un *applet* desde una página html utilizamos las etiquetas `<APPLET> ... <\APPLET>`, entre las que, como mínimo, incluimos la información siguiente:

- CODE = nombre del *applet* (por ejemplo, miApplet.class)
- WIDTH = anchura de la ventana
- HEIGHT = altura de la ventana

Y opcionalmente, los atributos siguientes:

- NAME = “unnombre” lo cual le permite comunicarse con otros *applets*
- ARCHIVE = “unarchivo” donde se guardan las clases en un .zip o un .jar
- PARAM NAME = “param1” VALUE = “valor1” para poder pasar parámetros al *applet*.

5.10.3. Mi primer applet en Java

La mejor manera de comprender el funcionamiento de los applets es a través de un ejemplo práctico. Para crear nuestro primer applet seguiremos estos pasos:

- 1) Crear un fichero fuente. Mediante el editor escogido, escribiremos el texto y lo salvaremos con el nombre `HolaMundoApplet.java`.

HolaMundoApplet.java

```
import java.applet.*;
import java.awt.*;
/*
 * La clase HolaMundoApplet muestra el mensaje
 * "Hola Mundo" en la salida estándar.
 */
public class HolaMundoApplet extends Applet{
    public void paint(Graphics g)
    {
        // Muestra "Hola Mundo!"
        g.drawString("¡Hola Mundo!", 75, 30 );
    }
}
```

- 2) Crear un fichero HTML. Mediante el editor escogido, escribiremos el texto.

HolaMundoApplet.html

```
<HTML>
<HEAD>
<TITLE>Mi primer applet</TITLE>
</HEAD>
<BODY>
Os quiero dar un mensaje:
<APPLET CODE="HolaMundoApplet.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

- 3) Compilar el programa generando un fichero bytecode.

```
javac HolaMundoApplet.java
```

- 4) Visualizar la página `HolaMundoApplet.html` desde un navegador.

5.11. Programación de interfaces gráficas en Java

La aparición de las interfaces gráficas supuso una gran evolución en el desarrollo de sistemas y aplicaciones. Hasta su aparición, los programas se basaban en el modo texto (o consola) y, generalmente, el flujo de información de estos programas era secuencial y se dirigía a través de las diferentes opciones que se iban introduciendo a medida que la aplicación lo solicitaba.

Las interfaces gráficas permiten una comunicación mucho más ágil con el usuario facilitando su interacción con el sistema en múltiples puntos de la pantalla. Se puede elegir en un momento determinado entre múltiples operaciones disponibles de naturaleza muy variada (por ejemplo, introducción de datos, selección de opciones de menú, cambios de formularios activos, cambios de aplicación, etc.) y, por tanto, múltiples flujos de instrucciones, siendo cada uno de ellos respuesta a eventos diferenciados.



Los programas que utilizan dichas interfaces son un claro ejemplo del paradigma de programación dirigido por eventos.

Con el tiempo, las interfaces gráficas han ido evolucionando y han ido surgiendo nuevos componentes (botones, listas desplegables, botones de opciones, etc.) que se adaptan mejor a la comunicación entre los usuarios y los ordenadores. La interacción con cada uno de estos componentes genera una serie de cambios de estado y cada cambio de estado es un suceso susceptible de necesitar o provocar una acción determinada. Es decir, un posible evento.

La programación de las aplicaciones con interfaces gráficas se elabora a partir de una serie de componentes gráficos (desde formularios hasta controles, como los botones o las etiquetas), que se definen como objetos propios, con sus variables y sus métodos.

Mientras que las variables corresponden a las diferentes propiedades necesarias para la descripción del objeto (longitudes, colores, bloqueos, etc.), los métodos permiten la codificación de una respuesta a cada uno de los diferentes eventos que pueden sucederle a dicho componente.

5.11.1. Las interfaces de usuario en Java

Java, desde su origen en la versión 1.0, implementó un paquete de rutinas gráficas denominadas AWT (*abstract windows toolkit*) incluidas en el paquete `java.awt` en la que se incluyen todos los componentes para construir una interfaz gráfica de usuario (GUI-graphic user interface) y para la gestión de eventos. Este hecho hace que las interfaces generadas con esta biblioteca funcionen en todos los entornos Java, incluidos los diferentes navegadores.

Este paquete sufrió una revisión que mejoró muchos aspectos en la versión 1.1, pero continuaba presentando un inconveniente: AWT incluye componentes que dependen de la plataforma, lo que ataca frontalmente uno de los pilares fundamentales en la filosofía de Java.

En la versión 1.2 (o Java 2) se ha implementado una nueva versión de interfaz gráfica que soluciona dichos problemas: el paquete Swing. Este paquete presenta, además, una serie de ventajas adicionales respecto a la AWT como aspecto modificable (diversos *look and feel*, como Metal que es la presentación propia de Java, Motif propia de Unix, Windows) y una amplia variedad de componentes, que se pueden identificar rápidamente porque su nombre comienza por J.

Swing conserva la gestión de eventos de AWT, aunque la enriquece con el paquete `javax.swing.event`.

Su principal inconveniente es que algunos navegadores actuales no la incluyen inicialmente, con lo cual su uso en los applets queda limitado.

Aunque el objetivo de este material no incluye el desarrollo de aplicaciones con interfaces gráficas, un pequeño ejemplo del uso de la biblioteca Swing nos permitirá presentar sus ideas básicas así como el uso de los eventos.

5.11.2. Ejemplo de applet de Swing

En el siguiente ejemplo, se define un *applet* que sigue la interfaz Swing. La primera diferencia respecto al *applet* explicado anteriormente corresponde a la inclusión del paquete javax.swing.*.

Se define la clase HelloSwing que hereda de la clase Japplet (que corresponde a los *applets* en Swing). En esta clase, se define el método init donde se define un nuevo botón (new JButton) y se añade al panel de la pantalla (.add).

Los botones reciben eventos de la clase ActionEvent y, para su tratamiento, la clase que gestiona sus eventos debe implementar la interfaz ActionListener.

Para esta función se ha declarado la clase GestorEventos que, en su interior, redefine el método actionPerformed (el único método definido en la interfaz ActionListener) de forma que abra una nueva ventana a través del método showMessageDialog.

Finalmente, sólo falta indicarle a la clase HelloSwing que la clase GestorEventos es la que gestiona los mensajes del botón. Para ello, usamos el método .addActionListener(GestorEventos)

HelloSwing.java

```
import javax.swing.*;
import java.awt.event.*;
public class HelloSwing extends Japplet
{
    public void init()
    { //constructor
        JButton boton = new JButton("Pulsa aquí!");
        GestorEventos miGestor = new GestorEventos();
        boton.addActionListener(miGestor); //Gestor del botón
        getContentPane().add(boton);
    } // init
} // HelloSwing

class GestorEventos implements ActionListener
{
    public void actionPerformed(ActionEvent evt)
    {
        String titulo = "Felicidades";
        String mensaje = "Hola mundo, desde Swing";
    }
}
```

```
JOptionPane.showMessageDialog(null, mensaje,  
    titulo, JOptionPane.INFORMATION_MESSAGE);  
  
} // actionPerformed  
} // clase GestorEventos
```

5.12. Introducción a la información visual

Aunque por motivos de simplicidad se han utilizado entornos de desarrollo en modo texto, en la realidad se utilizan entornos de desarrollo integrados (IDE) que incorporan las diferentes herramientas que facilitan al programador el proceso de generación de código fuente y ejecutable (editor, compilador, debugger, etc.).

En ambos casos, no hay ninguna diferencia en el lenguaje de programación: se considera que el lenguaje es “textual”, pues las instrucciones primitivas se expresan mediante texto.

Actualmente, estos entornos de desarrollo proporcionan la posibilidad de trabajar de una forma más visual permitiendo confeccionar las pantallas de trabajo (formularios, informes, etc.), mediante el arrastre de los diferentes controles a su posición final, y después procediendo a la introducción de valores para sus atributos (colores, medidas, etc.) y el código para cada uno de los eventos que puede provocar. No obstante, la naturaleza del lenguaje no varía y se continúa considerando “textual”.

Otro paradigma diferente correspondería a la programación mediante lenguajes “visuales”. Hablamos de *lenguaje visual* cuando el lenguaje manipula información visual, soporta interacciones visuales o permite la programación mediante expresiones visuales. Por tanto, sus primitivas son gráficos, animaciones, dibujos o iconos.

En otras palabras, los programas se constituyen como una relación entre distintas instrucciones que se representan gráficamente. Si la relación fuese secuencial y las instrucciones se expresaran mediante palabras, tal programación sería fácilmente reconocible. En todo caso, ya se ha comentado que la programación concurrente y la que se dirige por eventos no presentan una relación secuencial entre sus instrucciones que, además, suelen ser de alto nivel de abstracción.

Así pues, la programación visual resultaría ser una técnica para describir programas cuyos flujos de ejecución se adapten a los paradigmas anteriormente citados.

Por tanto, a pesar de la posible confusión aportada por los nombres de varios entornos de programación como la familia Visual de Microsoft (Visual C++, Visual Basic, etc.), a estos lenguajes se les debe continuar clasificando como lenguajes “textuales”, aunque su entorno gráfico de desarrollo sí que puede suponer una aproximación hacia la programación visual.

5.13. Resumen

En esta unidad se ha presentado un nuevo lenguaje de programación orientado a objetos que nos proporciona independencia de la plataforma sobre la que se ejecuta. Para ello, proporciona una máquina virtual sobre cada plataforma. De este modo, el desarrollador de aplicaciones sólo debe escribir su código fuente una única vez y compilarlo para generar un código “ejecutable” común, consiguiendo, de esta manera, que la aplicación pueda funcionar en entornos dispares como sistemas Unix, sistemas Pc o Apple McIntosh. Esta filosofía es la que se conoce como “write once, run everywhere”.

Java nació como evolución del C++ y adaptándose a las condiciones anteriormente descritas. Se aprovecha el conocimiento previo de los programadores en los lenguajes C y C++ para facilitar una aproximación rápida al lenguaje.

Al necesitar Java un entorno de poco tamaño, permite incorporar su uso en navegadores web. Como el uso de estos navegadores implica, normalmente, la existencia de un entorno gráfico, se ha aprovechado esta situación para introducir brevemente el uso de bibliotecas gráficas y el modelo de programación dirigido por eventos.

Asimismo, Java incluye de forma estándar dentro de su lenguaje operaciones avanzadas que en otros lenguajes realiza el sistema operativo o bibliotecas adicionales. Una de estas características es la programación de varios hilos de ejecución (*threads*) dentro del mismo proceso. En esta unidad, hemos podido introducirnos en el tema.



5.14. Ejercicios de autoevaluación

1. Ampliad la clase Leer.java para implementar la lectura de variables tipo double.
2. Introducid la fecha (solicitando una cadena para la población y tres números para la fecha) y devolvedlo en forma de texto.

Ejemplo

Entrada: Barcelona 15 02 2003

Salida: Barcelona, 15 de febrero de 2003

3. Implementad una aplicación que pueda diferenciar si una figura de cuatro vértices es un cuadrado, un rectángulo, un rombo u otro tipo de polígono.

Se definen los casos de la siguiente forma:

- Cuadrado: lados 1,2,3 y 4 iguales; 2 diagonales iguales
- Rectángulo: lados 1 y 3, 2 y 4 iguales; 2 diagonales iguales
- Rombo: lados 1,2,3 y 4 iguales, diagonales diferentes
- Polígono: los demás casos

Para ello, se definen la clase Punto2D definiendo las coordenadas x, y, y el método “distancia a otro punto”.

Ejemplo

(0,0) (1,0) (1,1) (0,1) Cuadrado

(0,1) (1,0) (2,1) (1,2) Cuadrado

(0,0) (2,0) (2,1) (0,1) Rectángulo

(0,2) (1,0) (2,2) (1,4) Rombo

4. Convertid el código del ejercicio del ascensor (ejercicio 3 de la unidad 4) a Java.

5.14.1. Solucionario

1.

Leer.java

```
import java.io.*;  
  
public class Leer  
{  
    public static String getString()  
    {  
        String str = "";  
        try  
        {  
            InputStreamReader isr = new  
            InputStreamReader(System.in);  
            BufferedReader br = new BufferedReader(isr);  
            str = br.readLine();  
        }  
        catch(IOException e)  
        { System.err.println("Error: " + e.getMessage());  
        }  
        return str; // devolver el dato tecleado  
    }  
  
    public static int getInt()  
    {  
        try  
        { return Integer.parseInt(getString()); }  
        catch(NumberFormatException e)  
        { return 0; // Integer.MIN_VALUE }  
    } // getInt  
  
    public static double getDouble()  
    {  
        try  
        {  
            return Double.parseDouble(getString());  
        }  
        catch(NumberFormatException e)  
        {  
            return 0; // Double.MIN_VALUE  
        }  
    } // getDouble  
} // Leer
```

2.

construirFecha.java

```
import java.io.*;
public class construirFecha
{
    static String nombreMes(int nmes)
    {
        String strmes;
        switch (nmes)
        {
            case 1: { strmes = "enero"; break; }
            case 2: { strmes = "febrero"; break; }
            case 3: { strmes = "marzo"; break; }
            case 4: { strmes = "abril"; break; }
            case 5: { strmes = "mayo"; break; }
            case 6: { strmes = "junio"; break; }
            case 7: { strmes = "julio"; break; }
            case 8: { strmes = "agosto"; break; }
            case 9: { strmes = "septiembre"; break; }
            case 10: { strmes = "octubre"; break; }
            case 11: { strmes = "noviembre"; break; }
            case 12: { strmes = "diciembre"; break; }
            default: { strmes = " -- "; break; }
        } // switch nmes
        return (strmes);
    } // nombreMes

    public static void main(String args[])
    {
        String poblacion;
        int dia, mes, año;
        String mifecha, strmes;
        System.out.print(" Población: ");
        poblacion = Leer.getString();
        System.out.print(" Dia: ");
        dia = Leer.getInt();
        System.out.print(" Mes: ");
        mes = Leer.getInt();
        System.out.print(" Año: ");
        año = Leer.getInt();
        mifecha = poblacion + ", " + dia;
        mifecha = mifecha + " de " + nombreMes(mes) +
                  " de " + año;
        System.out.print(" La fecha introducida es: ");
        System.out.println(mifecha);
    } // main
} // class
```

3.

Punto2D.java

```
class Punto2D
{
    public int x, y;
    // inicializando al origen de coordenadas
    Punto2D()
    { x = 0; y = 0; }

    // inicializando a una coordenada x,y determinada
    Punto2D(int coordx, int coordy)
    { x = coordx; y = coordy; }

    // calcula la distancia a otro punto
    double distancia(Punto2D miPunto)
    {
        int dx = x - miPunto.x;
        int dy = y - miPunto.y;
        return ( Math.sqrt(dx * dx + dy * dy));
    }
}
```

AppReconocerFigura.java

```
class AppReconocerFigura
{
    static public void main(String args[])
    {
        int i;
        int coordx, coordy;

        // Introducir 4 puntos e
        // indicar cuál es el más cercano al origen.

        Punto2D listaPuntos[];
        listaPuntos = new Punto2D[4];

        // entrar datos
        for (i=0; i<4; i++)
        {
            System.out.println("Entrar el punto (" + i + ")" );
            System.out.print("Coordenada x " );
            coordx = Leer.getInt();
            System.out.print("Coordenada y " );
            coordy = Leer.getInt();
        }
    }
}
```

```

        listaPuntos[i] = new Punto2D(coordx, coordy);
    } //for

    // indicar si los 4 puntos forman un
    // cuadrado: dist1 = dist2 = dist3 = dist4
    // diag1 = diag2
    // rombo: dist1 = dist2 = dist3 = dist4
    // diag1 <> diag2
    // rectangulo: dist1 = dist3, dist2 = dist4
    // diag1 = diag2
    // poligono: otros casos

    double dist[] = new double[4];
    double diag[] = new double[3];

    // calculo de distancias
    for (i=0; i<3; i++)
    {
        dist[i] = listaPuntos[i].distancia(listaPuntos[i+1]);
        System.out.print("Distancia "+i + " " + dist[i] );
    } //for
    dist[3] = listaPuntos[3].distancia(listaPuntos[0]);
    System.out.println("Distancia "+i + " " + dist[3] );

    // calculo de diagonales
    for (i=0; i<2; i++)
    {
        diag[i] = listaPuntos[i].distancia(listaPuntos[i+2]);
    } //for
    if ( (dist[0] == dist[2]) && (dist[1] == dist[3]) )
    {

        // es cuadrado, rectángulo o rombo
        if (dist[1] == dist[2]) {
            // es cuadrado o rombo
            if (diag[0] == diag[1])
                { System.out.println("Es un cuadrado"); }
            else
                { System.out.println("Es un rombo"); } // if
            }
            else
            {
                // es rectangulo
                if (diag[0] == diag[1])
                    { System.out.println("Es un rectángulo"); }
                else
                    { System.out.println("Es un polígono"); } // if
                }
            } // else
            { System.out.println("Es un poligono"); } // if
        } // main
    } // class
}

```

4.

AppAscensor.java

```
import java.io.*;  
  
public class AppAscensor{  
  
    static int n_codigo, n_peso, n_idioma;  
  
    public static void solicitarDatos()  
{  
    System.out.print ("Codigo: ");  
    n_codigo = Leer.getInt();  
  
    System.out.print("Peso: ");  
    n_peso = Leer.getInt();  
  
    System.out.print(  
        "Idioma: [1] Catalán [2] Castellano [3] Inglés ");  
    n_idioma = Leer.getInt();  
} // solicitarDatos  
  
  
public static void mostrarEstadoAscensor(Ascensor nA)  
{  
    nA.mostrarOcupacion();  
    System.out.print(" - ");  
    nA.mostrarCarga();  
    System.out.println(" ");  
    nA.mostrarListaPasajeros();  
} // mostrarEstadoAscensor  
  
public static void main( String[] args)  
{  
    int opc;  
    boolean salir = false;  
    Ascensor unAscensor;  
    Persona unaPersona;  
    Persona localizarPersona;  
  
    opc=0;  
  
    unAscensor = new Ascensor(); // inicializamos ascensor  
    unaPersona = null; // inicializamos unaPersona  
  
    do {  
  
        System.out.print(  
"ASCENSOR: [1]Entrar [2]Salir [3]Estado [0]Finalizar ");  
        opc = Leer.getInt();  
  
        switch (opc)  
        {  
        case 1: { // Opcion Entrar  
            solicitarDatos();  
            switch (n_idioma)
```

```

}

case 1: { //Catalan
    unaPersona = new Catalan (n_codigo, n_peso);
    break;
}
case 2: { //Castellano
    unaPersona = new Castellano (n_codigo, n_peso);
    break;
}
case 3: { //Ingles"
    unaPersona = new Ingles(n_codigo, n_peso);
    break;
}
default: { //Ingles"
    unaPersona = new Ingles(n_codigo, n_peso);
    break;
}
} //switch n_idioma

if (unAscensor.persona_PuedeEntrar(unaPersona))
{
    unAscensor.persona_ElEntrar(unaPersona);
    if (unAscensor.obtenerOcupacion()>1)
    {
        System.out.print(unaPersona.obtenerCodigo());
        System.out.print(" dice: ");
        unaPersona.saludar();
        System.out.println(" "); // Responden las demás
        unAscensor.restoAscensor_Saludar(unaPersona);
    }
} //puede entrar
break;
}
case 2: { //Opcion Salir

    localizarPersona = new Persona(); //Por ejemplo
    unaPersona = null;

    localizarPersona.solicitarCodigo();
    if (unAscensor.persona_Selectionar(localizarPersona))
    {
        unaPersona = unAscensor.obtenerRefPersona();
        unAscensor.persona_Salir( unaPersona );
        if (unAscensor.obtenerOcupacion()>0)
        {
            System.out.print(unaPersona.obtenerCodigo());
            System.out.print(" dice: ");
            unaPersona.despedirse();
            System.out.println(" "); // Responden las demás
            unAscensor.restoAscensor_Despedirse(unaPersona);
            unaPersona=null;
        }
    } else {
        System.out.println(
            "No hay ninguna persona con este código");
    } // seleccionar
}
}

```

```
        localizarPersona=null;
        break;
    }
    case 3: { //Estado
        mostrarEstado(unAscensor);
        break;
    }

    case 0: { //Finalizar
        System.out.println("Finalizar");
        salir = true;
        break;
    }

} //switch opc
} while (! salir);
} // main
} //AppAscensor
```

Ascensor.java

```
import java.io.*;

class Ascensor {
    private int ocupacion;
    private int carga;
    private int ocupacionMaxima;
    private int cargaMaxima;
    private Persona pasajeros[];
    private Persona refPersonaSeleccionada;

    //
    // Constructores y destructores
    //
    Ascensor()
    {
        ocupacion = 0;
        carga=0;
        ocupacionMaxima=6;
        cargaMaxima=500;
        pasajeros = new Persona[6];
        refPersonaSeleccionada = null;
    } //Ascensor()
```

```
// Funciones de acceso
int ObtenerOcupacion()
{ return (ocupacion); }

void ModificarOcupacion(int dif_ocupacion)
{ ocupacion += dif_ocupacion; }

void MostrarOcupacion()
{
    System.out.print("Ocupacion actual: ");
    System.out.print(ocupacion );
}

int obtenerCarga()
{ return (carga); }

void modificarCarga(int dif_carga)
{ carga += dif_carga; }

void mostrarCarga()
{ System.out.print("Carga actual: ");
    System.out.print(carga) ;
}

Persona obtenerRefPersona()
{return (refPersonaSeleccionada);}

boolean persona_PuedeEntrar(Persona unaPersona)
{
    // si la ocupación no sobrepasa el límite de ocupación y
    // si la carga no sobrepasa el límite de carga
    // ->puede entrar

    boolean tmpPuedeEntrar;
    if (ocupacion + 1 > ocupacionMaxima)
    {
        System.out.println(
"Aviso: El ascensor está completo. No puede entrar");
        return (false);
    }

    if (unaPersona.obtenerPeso() + carga > cargaMaxima )
    {
        System.out.println(
"Aviso: El ascensor supera su carga máxima. No puede entrar");
    }
}
```

```
        return (false);
    }
    return (true);
}
boolean persona_Seleccionar(Persona localizarPersona)
{
    int contador;

    // Se selecciona persona entre pasajeros del ascensor.
    boolean personaEncontrada = false;
    if (obtenerOcupacion() > 0)
    {
        contador=0;
        do {
            if (pasajeros[contador] != null)
            {
                if(pasajeros[contador].igualCodigo(localizarPersona))
                {
                    refPersonaSeleccionada=pasajeros[contador];
                    personaEncontrada=true;
                    break;
                }
            }
            contador++;
        } while (contador<ocupacionMaxima);
        if (contador>=ocupacionMaxima)
            {refPersonaSeleccionada=null;}
    }
    return (personaEncontrada);
}

void persona_Estar(Persona unaPersona)
{
    int contador;
    modificarOcupacion(1);
    modificarCarga(unaPersona.obtenerPeso());
    System.out.print(unaPersona.obtenerCodigo());
    System.out.println(" entra en el ascensor ");

    contador=0;
    // hemos verificado anteriormente que hay plazas libres
    do {
        if (pasajeros[contador]==null )
```

```
{  
    pasajeros[contador]=unaPersona;  
    break;  
}  
contador++;  
} while (contador<ocupacionMaxima);  
}  
  
void persona_Salir(Persona unaPersona)  
{  
    int contador;  
  
    contador=0;  
    do {  
        if ((pasajeros[contador]==unaPersona ))  
        {  
            System.out.print(unaPersona.obtenerCodigo());  
            System.out.println(" sale del ascensor ");  
            pasajeros[contador]=null;  
  
            // Modificamos la ocupacion y la carga  
            modificarOcupacion(-1);  
            modificarCarga(-1 * (unaPersona.obtenerPeso()));  
            break;  
        }  
        contador++;  
    } while (contador<ocupacionMaxima);  
    if (contador==ocupacionMaxima)  
    {System.out.println(  
        " No hay persona con este código. No sale nadie ");}  
}  
  
void mostrarListaPasajeros()  
{  
    int contador;  
    Persona unaPersona;  
  
    if (obtenerOcupacion() > 0)  
    {  
        System.out.println("Lista de pasajeros del ascensor:");  
        contador=0;  
        do {  
            if (! (pasajeros[contador]==null ))  
            {  
                System.out.print(pasajeros[contador].obtenerCodigo());  
                System.out.print(" ");  
            }  
        } while (contador<ocupacionMaxima);  
    }  
}
```

```
unaPersona=pasajeros[contador];
System.out.print(unaPersona.obtenerCodigo());
System.out.print("; ");
}
contador++;
} while (contador<ocupacionMaxima);
System.out.println("");
} else {
System.out.println("El ascensor esta vacío");
}
}

void restoAscensor_Saludar(Persona unaPersona)
{
int contador;
Persona otraPersona;

if (obtenerOcupacion() > 0)
{
contador=0;
do {
if (!(pasajeros[contador]==null ))
{
otraPersona=pasajeros[contador];
if (!unaPersona.igualCodigo(otraPersona) )
{
System.out.print(otraPersona.obtenerCodigo());
System.out.print(" responde: ");
otraPersona.saludar();
System.out.println("");
}
}
contador++;
} while (contador<ocupacionMaxima);
}

void restoAscensor_Despedirse(Persona unaPersona)
{
int contador;
Persona otraPersona;

if (obtenerOcupacion() > 0)
```

```

    {
        contador=0;
        do {
            if (! (pasajeros[contador]==null ))
            {
                otraPersona=pasajeros[contador];
                if (! (unaPersona.igualCodigo(otraPersona)))
                {
                    System.out.print(otraPersona.obtenerCodigo());
                    System.out.print(" responde: ");
                    otraPersona.despedirse();
                    System.out.print(" ");
                }
            }
            contador++;
        } while (contador<ocupacionMaxima);
    }
}

} // class Ascensor

```

Persona.java

```

import java.io.*;

class Persona {
    private int codigo;
    private int peso;

    Persona()
    { }

    Persona(int n_codigo, int n_peso)
    {
        codigo = n_codigo;
        peso = n_peso;
    }

    public int obtenerPeso()
    { return (peso); }

    public void asignarPeso(int n_peso)
    { peso = n_peso; }

    public int obtenerCodigo()

```

```
{ return (codigo); }

public void asignarCodigo(int n_codigo)
{ this.codigo = n_codigo; }

public void asignarPersona(int n_codigo, int n_peso)
{
    asignarCodigo( n_codigo );
    asignarPeso( n_peso );
}

void saludar() {};
void despedirse() {};

public void solicitarCodigo()
{
    int n_codigo=0;
    System.out.print ("Codigo: ");
    n_codigo = Leer.getInt();
    asignarCodigo (n_codigo);
}

public boolean igualCodigo(Persona otraPersona)
{
    return
        (this.obtenerCodigo()==otraPersona.obtenerCodigo());
}

} //class Persona
```

Catalan.java

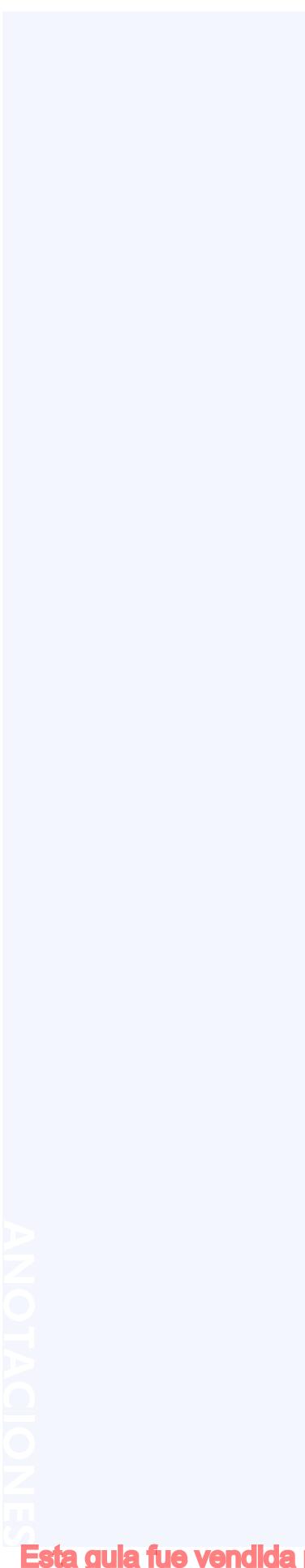
```
class Catalan extends Persona
{

    Catalan()
    { Persona(0, 0); };

    Catalan(int n_codigo, int n_peso)
    { Persona (n_codigo, n_peso); };

    void saludar()
    { System.out.println("Bon dia"); };

    void despedirse()
    { System.out.println("Adéu"); };
}
```

**Castellano.java**

```
class Castellano extends Persona
{
    Castellano()
    { Persona(0, 0); }

    Castellano(int n_codigo, int n_peso)
    { Persona (n_codigo, n_peso); }

    void saludar()
    { System.out.println("Buenos días"); }

    void despedirse()
    { System.out.println("Adiós"); }
}
```

Ingles.java

```
class Ingles extends Persona
{

    Ingles ()
    { Persona(0, 0); }

    Ingles (int n_codigo, int n_peso)
    { Persona (n_codigo, n_peso); }

    void saludar()
    { System.out.println("Hello"); }
    void despedirse()
    { System.out.println("Bye"); }
}
```

GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

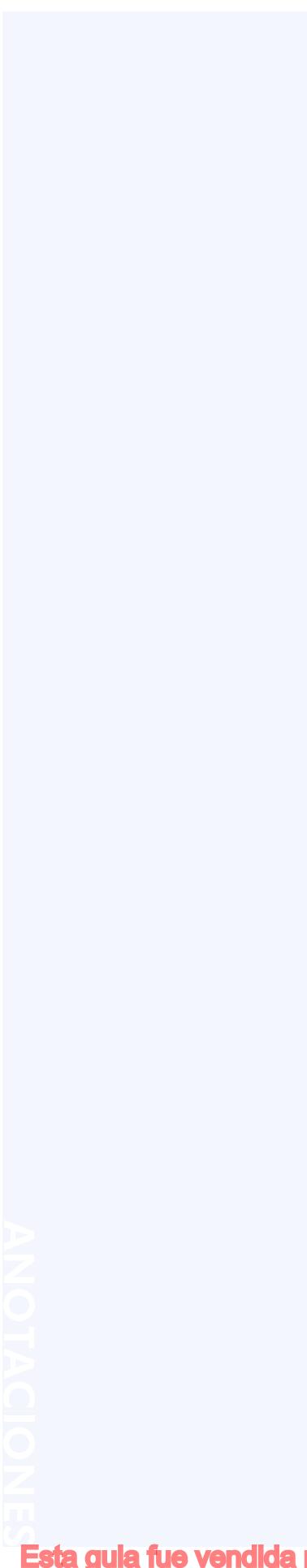
Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.



1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent.

An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition.

Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material.

If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

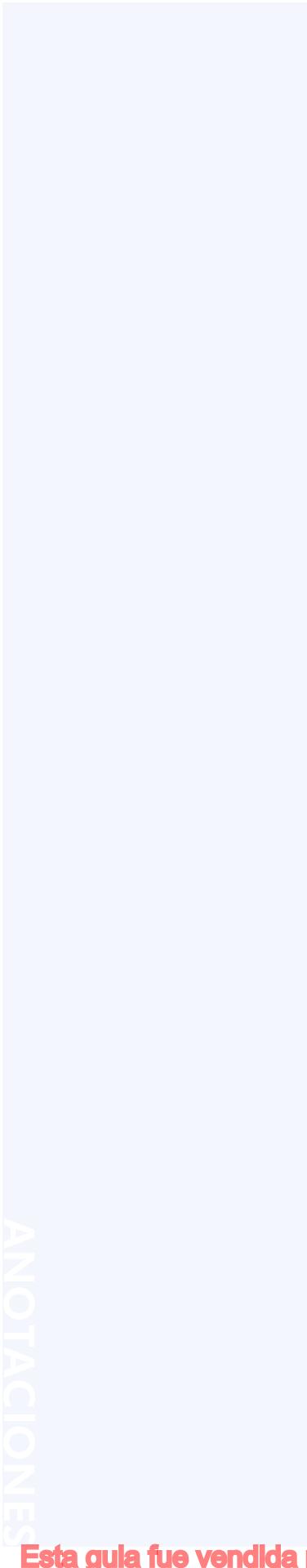
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.



The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit.

When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form.

Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the

translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX • Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX • Editorial Guías México

Lectura 3. Ingeniería de software, análisis y diseño

Capítulo 2: Ingeniería de Software, Análisis y Diseño

En todo desarrollo de sistemas de software es de suma importancia el seguir alguna especificación que permita a los desarrolladores el tener una disciplina que haga que todas las etapas del desarrollo del sistema, desde la pesquisa inicial de requerimientos hasta las pruebas finales del sistema, sean no solo más coherentes sino también más formales.

El desarrollo de software que este proyecto propone, al ser una herramienta que pretende tener aplicación dentro del contexto de un problema real, tiene que seguir un proceso de análisis y diseño que proporcione los cimientos bajo los cuales se va a desarrollar la aplicación conjuntamente. Es por esto que en este capítulo se detallan los procesos de ingeniería de software, análisis, y diseño que se involucran para el desarrollo de una aplicación de software que puede utilizarse como auxiliar al tratamiento del trastorno de lateralidad y ubicación espacial.

El capítulo en sí proporciona una pequeña introducción a lo que es la disciplina de la ingeniería de software, y posteriormente detallará los procesos y principios de análisis y diseño del software que sustentan este proyecto. También se especifican las técnicas de documentación del software que son utilizadas para complementar el desarrollo del sistema que se propone. Aunque el área de estudio y de aplicación de la ingeniería de software abarca también las etapas más complejas de desarrollo y pruebas del software, éstas no se discuten en este capítulo porque se tratarán posteriormente en los capítulos correspondientes.

1.1.Ingeniería de Software

Ingeniería de Software es una disciplina o área de las ciencias de la computación que ofrece métodos y técnicas para desarrollar y mantener software de calidad que resuelve problemas de todo tipo [PRR98].

Ingeniería de Software no es una disciplina que solo debe seguirse para proyectos de software que se encuentren pensados dentro de ciertas áreas, por el contrario, trata con

áreas muy diversas de las ciencias de la computación, tales como construcción de compiladores, sistemas operativos, o desarrollos en Internet como es muy cercanamente el caso de la aplicación de software de esta propuesta. La Ingeniería de Software abarca todas las fases del ciclo de vida del desarrollo de cualquier tipo de sistemas de información aplicables a áreas tales como los negocios, investigación científica, medicina, producción, logística, banca, y – para el caso particular de este estudio – realidad virtual [PRR98].

Un aspecto muy importante de Ingeniería de Software es que proporciona parámetros formales para lo que se conoce como Gestión (o Administración) de Proyectos de Software. Esto se refiere a que Ingeniería de Software proporciona diversas métricas y metodologías que pueden usarse como especificaciones para todo lo referente a la administración del personal involucrado en proyectos de software, ciclos de vida de un proyecto de software, costos de un proyecto, y en si todo el aspecto administrativo que implica el desarrollar software. Por supuesto que estos aspectos no son relevantes para los fines de este proyecto, principalmente porque este proyecto no se desarrolla con fines lucrativos monetariamente hablando.

De acuerdo con Pressman [PRR98], Ingeniería en general es el análisis, diseño, construcción, verificación y gestión de entidades técnicas. En general, todo proceso de ingeniería debe comenzar por contestar las siguientes preguntas: ¿Cuál es el problema a resolver?, ¿Cuáles son las características de la entidad que se utiliza para resolver el problema?, ¿Cómo se realizará la entidad (y la solución)?, ¿Cómo se construirá la entidad?, ¿Cómo va a probarse la entidad?, y ¿Cómo se apoyará la entidad cuando los usuarios finales soliciten correcciones y adaptaciones a la entidad? Para los fines que se desarrolla el software propuesto dentro de este proyecto, podemos contestar estas preguntas en una primera instancia desde un punto de vista global y sin considerar detalles específicos, de tal manera que se pueden establecer los siguientes puntos:

- Desarrollar una aplicación de software que pueda utilizarse como auxiliar en el tratamiento del trastorno de lateralidad y ubicación especial.
- La aplicación de software debe tener características tales que se cumpla con el objetivo del proyecto, es decir, que el software esté perfectamente orientado a sus usuarios para que realmente puede ser aplicado al área problema.

- El software se realizará bajo la siguiente premisa:
 - VRML 2.0 para modelar
 - Java para dar comportamiento
 - El Web para presentar
- La aplicación de software deberá ser probada en un intervalo de tiempo adecuado y lo suficientemente amplio como para poder obtener retroalimentación por parte de los usuarios y hacer las correcciones pertinentes. Deberá ser probada en sujetos reales que padeczan del trastorno de lateralidad y ubicación espacial.
- El software deberá estar documentado adecuadamente para facilitar futuros procesos tales como futuras expansiones ó adaptaciones a nuevas exigencias por parte de los usuarios finales.

Existen diferentes modelos de procesos para la Ingeniería de Software. Cada uno de estos modelos pretende de una manera u otra proporcionar lo más posible de orden al complicado proceso de desarrollar software. Para el caso de esta tesis es necesario apegarse lo más posible a uno de estos modelos con el fin de tener una organización de actividades que se planean en base a una serie de etapas lógicas e interconectadas entre sí. El modelo de ingeniería de software que esta tesis sigue es el Modelo Lineal Secuencial, que será descrito a continuación.

1.1.1. Modelo Lineal Secuencial

El modelo lineal secuencial, también conocido como modelo en cascada, se basa en un enfoque sistemático y secuencial del desarrollo del software que comienza en un nivel de sistemas y progresa con el análisis, diseño, codificación, pruebas, y mantenimiento [PRR98]. La siguiente figura ilustra el modelo lineal secuencial para la ingeniería de software.

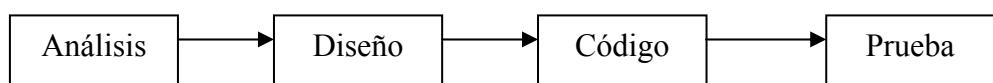


Figura 2.1 Modelo Lineal Secuencial

De acuerdo con Pressman, el modelo lineal secuencial contempla seis actividades que deben llevarse a cabo. A continuación se describen estas actividades, y se aterriza cada una de ella a los fines de este proyecto:

Ingeniería y modelado de Sistemas. El software siempre forma parte de un contexto más grande, que puede ir desde una empresa hasta un sistema. El trabajo comienza estableciendo requisitos de todos los elementos del sistema, y asignando al software algún subgrupo de estos requisitos [PRR98]. En el caso de la herramienta de software que este proyecto propone, queda establecido el hecho de que el software es una aplicación aislada que no se incorpora a un sistema (computacional) más grande, pero si pertenece al contexto de las metodologías de tratamiento contra el trastorno de lateralidad y ubicación espacial, y es por esto que tienen que establecerse requerimientos funcionales y no funcionales que permitan que el software desarrollado pueda ubicarse exitosamente dentro de este contexto. En el capítulo 1, Marco Teórico, ya se establecieron algunos requerimientos generales que un ambiente virtual debe contemplar en sus primeras etapas de desarrollo, más no se han especificado los requerimientos particulares para la aplicación de software propuesta dentro de este proyecto. Estos requerimientos serán considerados posteriormente dentro de este capítulo.

Análisis de los requisitos del software. El proceso de reunión de requisitos se intensifica y se centra especialmente en el software. Dentro del proceso de análisis es fundamental que a través de una colección de requerimientos funcionales y no funcionales, el desarrollador o desarrolladores del software comprendan completamente la naturaleza de los programas que deben construirse para desarrollar la aplicación, la función requerida, comportamiento, rendimiento e interconexión. [PRR98]. En el caso de este proyecto, el proceso de análisis y de obtención de requerimientos se lleva a cabo a través de trabajar conjuntamente con la psicóloga Norma Rodríguez, quien proporciona los parámetros bajo los cuales la aplicación debe desarrollarse para poder de esta manera cumplir con los objetivos de este proyecto. En la sección 2.3 se habla más profundamente de la etapa de análisis.

Diseño. Según Pressman, el diseño del software es realmente un proceso de muchos pasos pero que se clasifican dentro de uno mismo. En general, la actividad del diseño se refiere al establecimiento de las estructuras de datos, la arquitectura general del software, representaciones de interfaz y algoritmos. El proceso de diseño traduce requisitos en una representación de software [PRR98].

Generación de Código. Esta actividad consiste en traducir el diseño en una forma legible por la máquina. En el caso de la aplicación de software de este proyecto, la generación de código se refiere tanto a la parte de generación de los ambientes virtuales, como a la parte en la cuál se añadirá comportamiento a estos ambientes. El lenguaje de programación VRML 2.0 es un lenguaje de modelado en 3D en el cuál se dibuja por medio de generar código de programación de formato y marcado para especificar las características del objeto u objetos que se van agregando a un mundo o entorno virtual. El comportamiento de las escenas virtuales es decir, su funcionalidad, se puede construir a través de algún otro lenguaje de programación, como clases Java o scripts especificados en JavaScript. Todas estas actividades implican generar código.

Pruebas. Una vez que se ha generado código, comienzan las pruebas del software o sistema que se ha desarrollado. De acuerdo con Pressman, el proceso de pruebas se centra en los procesos lógicos internos del software, asegurando que todas las sentencias se han comprobado, y en los procesos externos funcionales, es decir, la realización de las prueba para la detección de errores [PRR98]. En el caso de una herramienta de software para tratar el trastorno de lateralidad y ubicación espacial, es necesario tener etapas de pruebas tanto para la parte funcional del software, como para la parte aplicativa del mismo. Se requiere poder probar el software con sujetos reales que puedan evaluar el comportamiento del software con el fin de proporcionar retroalimentación a los desarrolladores. Es sumamente importante que durante el proceso de desarrollo no se pierda el contacto con los interesados o solicitantes del desarrollo de software, de esta manera los objetivos de proyecto se mantendrán vigentes y se tendrá una idea clara de los aspectos que tienen que probarse durante el periodo de pruebas.

Mantenimiento. El software indudablemente sufrirá cambios, y habrá que hacer algunas modificaciones a su funcionalidad. Es de suma importancia que el software de

calidad pueda adaptarse con fines de acoplarse a los cambios de su entorno externo [PRR98]. Una de las secciones posteriores de este documento se refiere específicamente a posibles expansiones de este proyecto, y por medio de la documentación apropiada y atinada del software se pueden presentar las vías para el mantenimiento y modificaciones al mismo. En el capítulo 3, Desarrollo e Implementación, se describe paso a paso la manera en que se elaboran las escenas virtuales propuestas por este proyecto, de tal manera que este mismo documento puede utilizarse posteriormente como referencia al cómo deben desarrollarse aplicaciones que sigan la metodología de trabajo que este proyecto propone.

1.2.Ingeniería de Software Educativo (ISE)

Como su nombre lo dice, la Ingeniería de Software Educativo es una rama de la disciplina de la ingeniería de software encargada de apoyar el desarrollo de aplicaciones computacionales que tienen como fin implementar procesos de aprendizaje desde instituciones educativas hasta aplicaciones en el hogar. Si lo que se pretende es lograr aplicaciones de software que califiquen como educativas, es necesario que dentro de las fases de análisis y diseño de las mismas se añadan aspectos didácticos y pedagógicos con el fin de poder garantizar la satisfacción de las necesidades educativas en cuestión. Es de suma importancia involucrar efectivamente a los usuarios, para poder identificar necesidades que debe cubrirse durante la etapa de desarrollo [GOR97].

En el caso particular de este proyecto, la aplicación basada en ambientes virtuales que se propone es una herramienta computacional con fines educativos, es por esto que desde la primera etapa del proceso de Ingeniería de Software que se siguió, se involucró a la psicóloga Norma Rodríguez, quien desde un principio proporcionó una serie de consideraciones didácticas que deben atacarse desde el punto de vista de análisis y diseño de la aplicación para que a través de realidad virtual pueda tratarse el Trastorno de Lateralidad y Ubicación Espacial.

Dentro del proceso de Ingeniería de Software Educativo que se utiliza dentro de este proyecto, es también necesario incorporar la distinción de los diferentes elementos que de acuerdo con López, Escalera, y Ledesma conforman un Ambiente Virtual de Aprendizaje [LOEL02]. La identificación y aterrizaje da cada uno de estos elementos

funge un papel sumamente importante en las etapas de análisis y diseño de la herramienta aplicativa que este proyecto propone. Estos elementos son los siguientes:

- **Usuarios.** Son aquellos que van a aprender a través del Ambiente Virtual de Aprendizaje. En el caso de este proyecto, se refiere a aquellas personas que padecen del trastorno de lateralidad y ubicación espacial, y de personas a quienes se les implanta por primera vez educación acerca de las nociones básicas.
- **Contenido.** Es lo que se va a aprender. Para este proyecto, se establece que lo que se va a aprender es un conocimiento general de las nociones básicas de lateralidad y reconocimiento espacial en personas que presentan deficiencias en estas distinciones.
- **Especialistas.** Se refiere al cómo se va a aprender, cómo se van a materializar todos los contenidos educativos que se utilizarán en proceso de aprendizaje. Generalmente el grupo de especialistas consiste en personas con diferentes especialidades, desde el pedagogo hasta los programadores y diseñadores de los entornos virtuales.
- **Acceso,** infraestructura y conectividad. Este elemento se refiere a la arquitectura general de los ambientes virtuales que van a desarrollarse. Como ya se ha mencionado anteriormente, este proyecto se desarrolla principalmente en VRML 2.0 con el fin de aprovechar todas las ventajas de portabilidad que presenta Internet La aplicación no será incorporada a un sistema ya existente y más grande, pero si se contemplará a nivel teórico la manera en que este proyecto podría expandirse para aumentar el rango de aplicación de este proyecto. Estas expansiones se discuten en el capítulo 5.

En 1991, Galvis propone una metodología para la Ingeniería de Software Educativo que se asemeja mucho a la metodología que establece el modelo lineal secuencial descrito anteriormente, y que sigue este proyecto. Esta metodología establece mecanismos de análisis, y diseño educativo y comunicacional de validez comprobaba [GORG97]. La figura 2.2 ilustra este modelo:

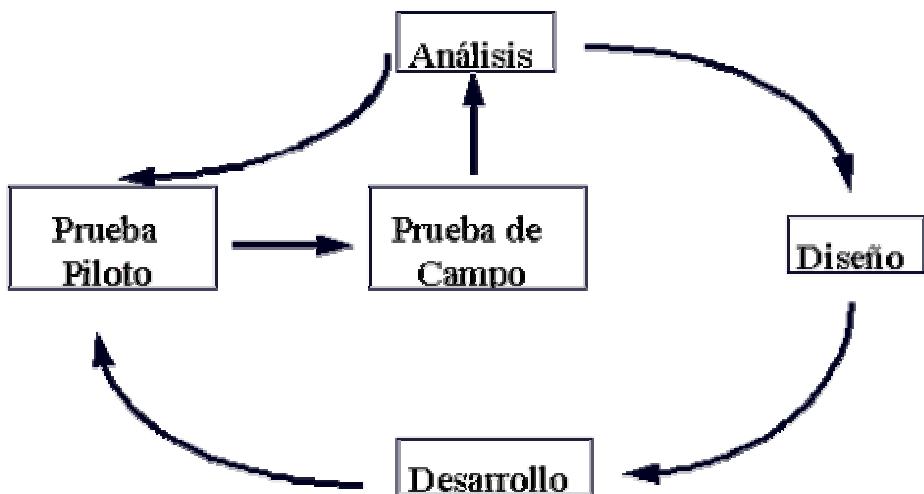


Figura 2.2: Metodología ISE propuesta por Galvis [GORG97].

Como ya se mencionó anteriormente, la Ingeniería de Software Educativo establece actividades que tienen que ser complementadas por medio de la incorporación de los aspectos didácticos y pedagógicos. El como se incorporan estos aspectos al proceso de ingeniería de software que sigue este proyecto se comentará en las secciones siguientes.

1.3. Análisis

Para que el desarrollo de un proyecto de software concluya con éxito, es de suma importancia que antes de empezar a codificar los programas que constituirán la aplicación de software completa, se tenga una completa y plena comprensión de los requisitos del software.

Pressman establece que la tarea del análisis de requisitos es un proceso de descubrimiento, refinamiento, modelado y especificación. Se refina en detalle el ámbito del software, y se crean modelos de los requisitos de datos, flujo de información y control, y del comportamiento operativo. Se analizan soluciones alternativas y se asignan a diferentes elementos del software. El análisis de requisitos permite al desarrollador o desarrolladores especificar la función y el rendimiento del software, indica la interfaz del software con otros elementos del sistema y establece las

restricciones que debe cumplir el software. El análisis de requisitos del software puede dividirse en cinco áreas de esfuerzo, que son:

1. **Reconocimiento del problema.** Reconocer los elementos básicos del problema tal y como los perciben los usuarios finales.
2. **Evaluación y síntesis.** Definir todos los objetos de datos observables externamente, evaluar el flujo y contenido de la información, definir y elaborar todas las funciones del software, entender el comportamiento del software en el contexto de acontecimientos que afectan al sistema.
3. **Modelado.** Crear modelos del sistema con el fin de entender mejor el flujo de datos y control, el tratamiento funcional y el comportamiento operativo y el contenido de la información.
4. **Especificación.** Realizar la especificación formal del software
5. **Revisión.** Un último chequeo general de todo el proceso.

En el caso de una herramienta de software que puede emplearse como auxiliar en el tratamiento de lateralidad y ubicación espacial, el análisis de requerimientos debe llevarse a cabo en base a las necesidades de los terapeutas y de los pacientes. Debido a que el software que este proyecto propone no está basando en algún sistema anterior, la funcionalidad del software debe emular lo más atinadamente posible las actividades de juego con las que se trata este trastorno hoy en día.

Como se mencionó en el capítulo anterior, el tratamiento al trastorno de lateralidad y ubicación espacial en pacientes que tienen de 5 a 14 años de edad se lleva a cabo por medio de actividades de juego que regularmente se presentan al paciente a través de libros de ejercicios y otros impresos en los que se encuentran plasmadas estas actividades. La premisa esencial de este proyecto es entonces, el llevar estas actividades de los libros a entornos virtuales con comportamientos implementados de tal manera que se conserve la eficiencia de las actividades de juego presentadas en papel, pero con las ventajas agregadas que tiene la Terapia de Exposición usando Realidad Virtual.

En el caso de este proyecto el proceso de análisis comenzó con una primera entrevista con la psicóloga Norma Rodríguez. En esta primera entrevista se platicó acerca de lo que es el trastorno de lateralidad y ubicación espacial, como se manifiesta, sus impactos

en los padecientes, y más particularmente de las metodologías de tratamiento que se aplican para ayudar a los pacientes a superar sus problemas consecuentes. En vista de que este proyecto se plantea como un proyecto de aplicación computacional, y no de estudio neurofisiológico, la parte fundamental de la etapa de análisis es la que se realizó alrededor de la metodologías de tratamiento, que es precisamente lo que se pretende modelar dentro de entornos virtuales tridimensionales accesibles a través de Internet.

Los requerimientos de sistema que se definen en la etapa de análisis de un proceso de Ingeniería de Software generalmente se clasifican como requerimientos funcionales y requerimientos no funcionales. Los principales requerimientos funcionales y no funcionales de la herramienta de software que este proyecto propone se definen en las dos secciones siguientes.

1.3.1. Requerimientos Funcionales

Los requerimientos funcionales son los que se encargan de definir lo que la herramienta de software debe hacer. Definen los alcances del sistema en cuanto a las acciones que debe de realizar, y en cuanto a la transferencia de datos entre todas las diferentes funciones del sistema [KII02].

En el caso de este proyecto, los principales requerimientos funcionales son los siguientes:

1. **Sensibilidad a la presencia de los usuarios.** Este proyecto gira en torno a la posibilidad de poder modelar el tratamiento del trastorno de lateralidad y ubicación espacial dentro de ambientes virtuales. Estos ambientes virtuales deben de tener la posibilidad de reaccionar a las acciones de un usuario que se encuentre utilizando el sistema, dado que se debe tener una cierta interacción y comunicación entre los usuarios del ambiente virtual y el ambiente virtual en si mismo. VRML 2.0 proporciona dentro su especificación diferentes nodos que permiten a un mundo virtual, o a un objeto contenido dentro de un mundo

virtual, el disparar ciertas acciones como respuesta a acciones o eventos que el usuario efectúe. Estas acciones y eventos deben ser capturadas y procesadas por el mundo virtual, para producir respuestas al usuario del entorno virtual. La manera en que se implementa e inyecta esta sensibilidad a los ambientes virtuales se discute a detalle en el capítulo 3 (Desarrollo e Implementación.)

2. **Funcionalidad Global.** Esta herramienta de software debe de tener la facultad de desplegar y presentar diferente actividades con ejercicios de los que tradicionalmente se utilizan para tratar el trastorno de lateralidad y ubicación espacial. Las actividades tendrán que ser desplegadas bajo petición de un usuario, y tendrán que tener comportamientos específicos y detallados con respecto a las acciones que el usuario tenga que realizar para resolver el problema que se presente a través del ambiente virtual activado. La gran mayoría de la funcionalidad de los ambientes virtuales estará dada a través de las opciones funcionales que ofrece el lenguaje VRML en su versión 2.0, y en aquellos casos donde se requiera funcionalidad más compleja, se completará VRML a través de Java.
3. **Alcance.** Con anterioridad ya se ha mencionado que el objetivo de este trabajo es modelar dentro de ambientes virtuales el tratamiento al trastorno en cuestión. Sin embargo, es importante mencionar que esta herramienta no tiene como objetivo el funcionar como un asistente para diagnosticar este trastorno. El alcance de este proyecto se limita a presentar dentro de mundos virtuales las actividades de juego que se utilizan para tratar el trastorno de lateralidad y ubicación espacial.

Los mencionados anteriormente son los requerimientos funcionales particulares de este proyecto, sin embargo, en el Capítulo 1, (Marco Teórico), se discuten requerimientos generales que deben contemplarse dentro del desarrollo de todo ambiente virtual. En el caso particular de este proyecto, los requerimientos fundamentales son aquellos establecidos por los requerimientos no funcionales. Estos se discuten en la sección correspondiente.

1.3.2. Requerimientos No Funcionales

Los requerimientos no funcionales son aquellos que definen lo que la herramienta de software debe tener en cuanto a apariencia, sensación, operabilidad, y mantenimiento [KII02].

De acuerdo con Galvis, el objetivo de la etapa de análisis cuando se sigue una metodología de ISE (Ingeniería de Software Educativo), es determinar el contexto en el cual se va a crear la aplicación para poder derivar los requerimientos que deberá atender la solución interactiva como complemento a otras soluciones basadas en uso de otros medios, teniendo en claro el rol de cada uno de los medios educativos seleccionados y la viabilidad de usarlos [GOR97]. Tomando en cuenta esta consideración, y trabajando en conjunto con la psicóloga Norma Rodríguez, es como se derivan los requerimientos no funcionales que debe cubrir la herramienta de software propuesta por este estudio.

Galvis establece que para recolectar este tipo de requerimientos hay que por lo menos tomar en cuenta la siguiente información:

- **Características de la población objetivo.** Se refiere a cuestiones como la edad, características físicas y mentales, experiencias previas, expectativas, actitudes, aptitudes, o intereses. En el caso de este estudio, es muy importante considerar que el rango de edad de los pacientes que van a tratarse es de 5 a 14 años de edad. No todos los pacientes saben leer y escribir. Las características físicas son completamente irrelevantes, y se da por sentado que los pacientes manifiestan problemas en cuanto a sus conocimientos de las nociones básicas. Se asume que ningún paciente tiene experiencia con ambientes virtuales, pero a través de la información que se obtuvo por medio de la psicóloga que apoya este estudio, se sabe que algunos de estos pacientes tienen experiencia con computadoras y con videojuegos. Esto último siendo importante dado que si un paciente tiene experiencia con videojuegos, le será más fácil desenvolverse dentro de los ambientes virtuales que este proyecto propone.
- **Conducta de entrada y campo vital.** Es necesario ubicar la herramienta de software dentro de las áreas bajo las cuales se desenvuelve el paciente. Es importante considerar aspectos como el nivel escolar y desarrollo mental. A

través de una investigación preliminar, para este proyecto se sabe que el nivel escolar es esencialmente de nivel primaria, y que las actividades de juego y tratamiento que pretenden modelarse están diseñadas para pacientes a este nivel educativo.

- **Problema o necesidad a atender.** Como su nombre lo dice, es necesario ubicarse dentro del contexto del problema que pretende atacarse. En el caso de esta tesis el objetivo es modelar el tratamiento al trastorno de lateralidad y ubicación espacial a dentro de ambientes virtuales interactivos.
- **Justificación de los medios interactivos a utilizar.** Galvis establece que el apoyo informático debe ser tomado en cuenta siempre y cuando no exista un mecanismo mejor para resolver el problema. El caso de este proyecto, se tienen varias justificaciones que pueden apreciarse desde distintos puntos de vista. Primero que nada, los desarrollos dentro de las ciencias computacionales se caracterizan por llevar actividades manuales a procesos automatizados, y con esto se facilita el trabajo. En el capítulo 1, se mencionan la grandes ventajas que se han manifestado a través de la utilización de la Terapia de Exposición Usando Realidad Virtual, y son precisamente estas ventajas las que se buscan incorporar al tratamiento del trastorno en cuestión. Otro aspecto importante es que a través de utilizar la plataforma del Web como presentador, el potencial de portabilidad y de crecimiento es mucho mayor que a través de los procesos manuales impresos. Por último, el presentar un tratamiento a través de Realidad Virtual y computadoras, resulta muy llamativo para pacientes con las características definidas para la población objetivo; al ser mas llamativo, la apertura a recibir tratamiento es mayor y con esto se agiliza el tratamiento mismo.

Los puntos anteriores conducen a este proyecto a elaborar una lista de requerimientos no funcionales dentro los cuales destacan los siguientes:

1. **Clasificación de las actividades.** Como ya se mencionó anteriormente, el rango de edad de los pacientes que serán tratados a través de esta herramienta es un poco amplio, por lo que sería absurdo asumir o pensar que una actividad es apropiada para todos los pacientes. La psicóloga Norma Rodríguez especificó que las actividades que conforman el tratamiento a este trastorno se encuentran clasificadas dentro de niveles de dificultad que van de acuerdo a la edad de los

pacientes. A los pacientes más chicos, por ejemplo, no se les pueden introducir conceptos o etiquetas como “derecha” o “izquierda” si aún no saben leer, pero si se les puede hacer distinguir los lados sin necesidad de poner nombres. Este requerimiento se refiere a que esta clasificación de actividades debe respetarse dentro de la herramienta de software aplicativo.

2. **Instrucciones claras.** Es necesario proporcionar a los usuarios instrucciones que describan el objetivo que debe cumplirse dentro de cada una de las actividades que se encuentren modeladas dentro de las escenas virtuales. En vista de que no todos los pacientes saben leer, desplegar las instrucciones escritas en pantalla no es suficiente. Es por esto que la herramienta de software debe de ser capaz de proporcionar las instrucciones de manera clara y precisa, y que todos los usuarios puedan entender cuales son los objetivos que tienen que ser alcanzados. Se hace necesario entonces el uso de archivos de sonidos.
3. **Operabilidad.** En el capítulo 1 se menciona lo importante que es identificar al usuario apropiadamente. Se estableció que este proyecto identifica a sus usuarios como terapeutas y pacientes que no tienen experiencia alguna con Realidad Virtual. Es por esto que los usuarios tienen que poder interactuar con las escenas virtuales sin la necesidad de controles complejos que pueden quitarle el atractivo a esta metodología de tratamiento. Podría decirse que los usuarios finales son los pacientes, ya que en un momento dado el terapeuta puede o no estar presente.
4. **Conservar características de las metodologías tradicionales.** El trastorno de lateralidad y ubicación espacial se trata a través de actividades de juego que tienen ciertas características que las hacen apropiadas para ser utilizadas como metodologías de tratamiento a este padecimiento neurofisiológico. El hecho de que estas actividades vayan a ser presentadas a través de entornos tridimensionales, exige que estas características se conserven.

En la sección Diseño, se discute como la lista de requerimientos funcionales y no funcionales se traduce en una aplicación de software basada en entornos virtuales interactivos.

1.4.Diseño

Anteriormente se mencionó que la etapa de diseño es cuando se traducen los requerimientos funcionales y no funcionales en una representación de software. El diseño es el primer paso en la fase de desarrollo de cualquier producto o sistema de ingeniería. De acuerdo con Pressman, el objetivo del diseño es producir un modelo o representación de una entidad que se va a construir posteriormente [PRR98].

De acuerdo con McGlaughlin [MCG91], hay tres características que sirven como parámetros generales para la evaluación de un buen diseño. Estos parámetros son los siguientes:

1. El diseño debe implementar todos los requisitos explícitos obtenidos en la etapa de análisis.
2. El diseño debe ser una guía que puedan leer y entender los que construyen el código y los que prueban y mantienen el software.
3. El diseño debe proporcionar una idea completa de lo que es el software.

En la sección siguiente se establecen tipos diferentes de diseño que la etapa de diseño del proceso de ingeniería de software produce.

1.4.1. Diseño del Software

El diseño del software desarrolla un modelo de instrumentación o implantación basado en los modelos conceptuales desarrollados durante el análisis del sistema. Implica diseñar la decisión sobre la distribución de datos y procesos [MAJO97].

El diseño es la primera de las tres actividades técnicas que implica un proceso de ingeniería de software; estas etapas son diseño, codificación (en el caso de este proyecto Desarrollo e Implementación) y pruebas. Generalmente la fase de diseño produce un diseño de datos, un diseño arquitectónico, un diseño de interfaz, y un diseño procedural [PRR98].

El diseño de datos esencialmente se encarga de transformar el modelo de dominio de la información creado durante el análisis [PRR98]. En el caso particular de este proyecto el diseño de datos no juega un papel determinante dado que la herramienta de software propuesta, de la manera en que será físicamente desarrollada e implementada, no requiere de estructuras de datos complejas, ni de un esquema de base de datos por ejemplo.

En el diseño arquitectónico se definen las relaciones entre los principales elementos estructurales del programa [PRR98]. Para una herramienta de software basada en el desarrollo e implementación de ambientes virtuales éste es un aspecto fundamental dado que en esta representación del diseño se establece la estructura modular del software que se desarrolla. Dado que este proyecto pretende proponer una metodología de tratamiento al trastorno de lateralidad y ubicación espacial a través de Realidad Virtual, la codificación y generación de ambientes y entornos virtuales es esencial. Cuando se utiliza VRML 2.0 es necesario codificar cada una de las instrucciones que crearán un objeto determinado con sus propias características y atributos. Si se pretendiera codificar por completo toda una escena virtual dentro de un mismo archivo, el archivo crecería superlativamente y su manipulación, adaptación, y mantenimiento se volverían tareas bastante complejas e incomodas. Afortunadamente, a través del nodo *Inline* de la especificación 2.0 de VRML puede darse un alto nivel de modularidad a los mundos virtuales dado que cada objeto puede describirse o codificarse por separado, para posteriormente ser referenciado dentro de la escena virtual contenedora. El nodo *Inline* se detalla en el capítulo siguiente.

El diseño de interfaz describe cómo se comunica el software consigo mismo, con los sistemas que operan con él, y con los operadores que lo emplean [PRR98]. En el caso de la herramienta de software propuesta por este estudio la interfaz del software consigo mismo se lleva a cabo de 2 maneras:

- Nodos de VRML 2.0 se comunican con otro nodos ¹
- Nodos que se comunican con Scripts de comportamiento descritos en Java o en JavaScript.

¹ Este proceso se conoce como ROUTING. Los detalles serán discutidos en el capítulo 3 (Desarrollo e Implementación).

VRML es un lenguaje de modelado diseñado específicamente para integrarse a la plataforma de Internet. Es por este que para los fines de este proyecto se antoja lógico el desarrollar la interfaz con los operadores del software a través de HTML, VRML, JavaScript, o cualquier otra tecnología que puede incorporarse a las especificaciones de esta plataforma.

De acuerdo con Pressman, el diseño procedural transforma elementos estructurales de la arquitectura del programa en una descripción procedural de los componentes del software [PRR98].

1.4.2. Arquitectura del Software

El diseño de la arquitectura del software se refiere a la estructura global del software y las maneras en que esa estructura proporciona integridad conceptual a un sistema [SHA95]. De acuerdo con Pressman, en su forma más simple, la arquitectura es la estructura jerárquica de los módulos del programa, la manera de interactuar de estos componentes, y la estructura de los datos usados por estos módulos [PRR98].

La arquitectura del software que este proyecto propone como una herramienta aplicativa dentro de un contexto real, está pensada de acuerdo a las propiedades que Shaw y Garlan [SHA95] describen como aspectos que deben especificarse como partes de un buen diseño arquitectónico. Estos tres aspectos son: Propiedades estructurales, Propiedades extra-funcionales, y Familias de Sistemas Relacionados. En las siguientes sub-secciones se elaborará con respecto a cada uno de estos tres aspectos respectivamente.

1.4.2.1. Propiedades Estructurales

Este es el aspecto de la representación de software que define los componentes de un sistema, y la manera en que se empaquetan estos componentes e interactúan unos con los otros. Este proyecto propone ambientes virtuales integrados para formar una aplicación conjunta que puede presentarse apropiadamente a los usuarios. La

especificación de la arquitectura del software tiene que respetar fuertemente el concepto de modularidad del software, ya que es necesario poder identificar los componentes individuales que al unirse entre si forman un ambiente virtual.

La manera en que este proyecto de software se encuentra estructurado tiene que ser especificada desde su componente más básico hasta los usuarios finales. De esta manera, se obtiene la tan necesaria modularidad que según Myers [MYE78] es un atributo del software que permite a un programa ser manejable intelectualmente. Un programa grande compuesto de un solo módulo no puede ser entendido fácilmente por un lector. El número de caminos de control, número de variables y la complejidad global harían la comprensión casi imposible. Como ya se mencionó anteriormente, VRML permite modularidad a través de ofrecer la posibilidad de crear los componentes de una escena virtual por separado.

La figura 2.3, un modelo entidad-relación, ilustra la cohesión general que existe entre los diferentes elementos de la herramienta de software que este estudio implementa

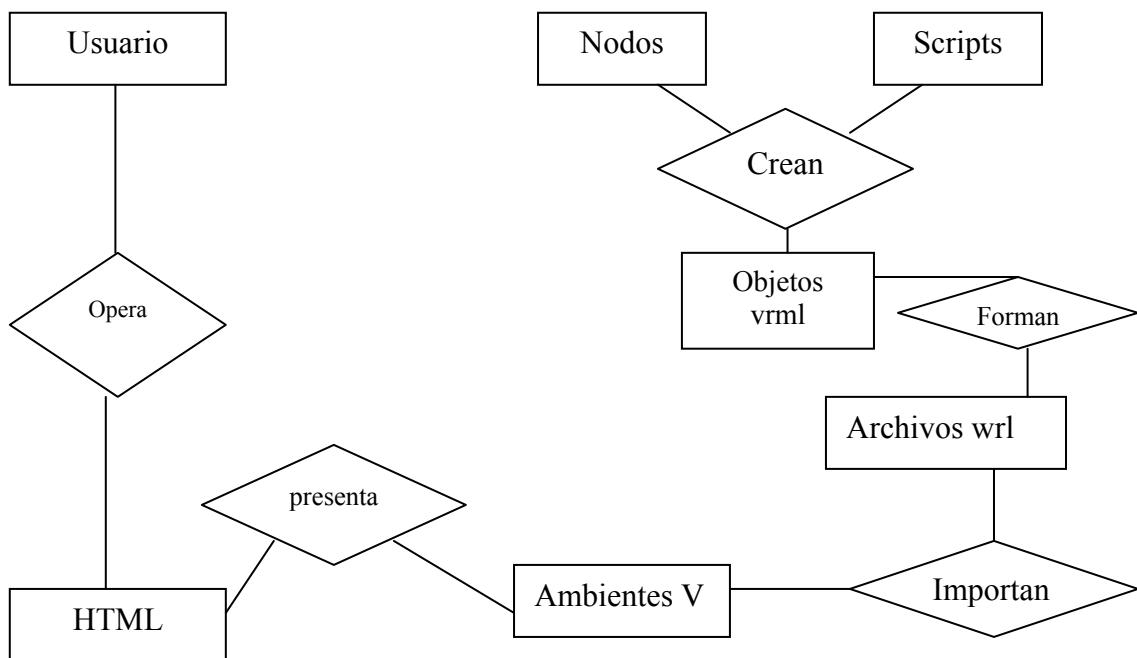


Figura 2.3 Modelo Entidad Relación para la arquitectura del sistema

El modelo entidad relación se explica de la siguiente manera. Los dos componentes más básicos de la herramienta de software que este proyecto propone son los nodos del lenguaje VRML 2.0 y los Scripts de comportamiento en Java y JavaScript. Los nodos y los Scripts crean y definen objetos de VRML que tiene sus propios atributos y características. Un conjunto o agrupación de objetos VRML forman archivos *.wrl que en base a los objetos que los conforman pueden describir elementos más complejos de una escena virtual. Los ambientes virtuales importan una serie de archivos *.wrl que una vez integrados forman el ambiente virtual, que se incrusta en un documento HTML que es lo que da la interfaz con el usuario. De esta manera, se tiene una modularidad total con respecto a la arquitectura del software, ya que para modificar un objeto VRML se redefine ya sea el nodo(s) o el script(s) que lo describe. A su vez, para realizar modificaciones sobre un archivo *.wrl pueden manipularse individualmente los objetos que lo conforman, y de esta manera se puede dar mantenimiento a un ambiente virtual sin necesidad de manipular una cantidad excesiva de líneas de código.

1.4.2.2. Propiedades Extra-funcionales

Esta especificación se refiere al cómo consigue la arquitectura del diseño los requisitos de rendimiento, capacidad, fiabilidad, seguridad, adaptabilidad, y otras características de la herramienta de software [SHA95]. En las secciones correspondientes a los requerimientos funcionales y a los no funcionales se establecieron los principales requerimientos funcionales y no funcionales que se consideran para el desarrollo de la herramienta de software, a continuación se presenta el registro de cada uno de estos requerimientos, y una explicación acerca de como el diseño de la herramienta de software propuesta incorpora dentro de sus características esta lista de requerimientos.

1. Sensibilidad a la presencia de los usuarios

- **ID del requerimiento** – rf01
- **Descripción** – Es necesario que los ambientes virtuales propuestos dentro de este proyecto tengan la posibilidad de captar acciones producidas por los usuarios para generar respuestas. Con esto las escenas virtuales se vuelven interactivas.
- **Tipo** – Requerimiento funcional

- **Caso de Uso** – Esta funcionalidad debe estar presente en todos los ambientes virtuales generados dentro de este proyecto, ya que todos los ambientes deben ser interactivos.
- **Fuente** – Psicóloga Norma Rodríguez, y Maestra Carolina Castañeda.
- **Criterio de Evaluación** – El criterio de evaluación para este requerimiento no es algo cuantificable. La evaluación de este requerimiento se hace en base a si la interacción que tienen los ambientes virtuales contribuye el cumplimiento de los objetivos aplicativos del software.

Implementación – Este requerimiento se implementa dentro del sistema por medio de la inclusión de censores de contacto, proximidad, y tiempo que sean capaces de capturar un evento producido por un usuario para procesarlo y convertirlo en una respuesta que se envía al usuario. Cuando los censores disponibles dentro de VRML 2.0 no son suficientes, se utiliza Java o JavaScript para proporcionar esta funcionalidad adicional.

2. Funcionalidad Global

- **ID del requerimiento** – rf02
- **Descripción** – Esta herramienta de software debe de tener la facultad de desplegar y presentar diferentes actividades con ejercicios de los que tradicionalmente se utilizan para tratar el trastorno en cuestión.
- **Tipo** – Requerimiento Funcional
- **Caso de Uso** – Este requerimiento constituye en sí mismo todo lo referente a los casos de uso del software propuesto, ya que es a través de la presentación de diferentes escenas virtuales que se pretende interactuar con los usuarios.
- **Fuente** – Psicóloga Norma Rodríguez
- **Criterio de Evaluación** – Presentar diversas escenas virtuales interactivas con funcionalidad apropiada al tratamiento de este trastorno.

Implementación – Este requerimiento se implementa a través del modelado de diferentes escenas virtuales con funcionalidades apropiadas y correspondientes.

3. Clasificación de las actividades

- **ID del requerimiento** – rnf01
- **Descripción** – Considerando que el nivel de afectación que presentan los pacientes en base al trastorno, y dado que el rango de edad de la población objetivo es de 5 a 14 años de edad, es necesario tener dentro del software una clasificación de actividades, para que de esta manera haya actividades apropiadas para cada paciente.
- **Tipo** – Requerimiento no funcional.
- **Caso de Uso** – Esta clasificación debe conservarse en todo los casos de uso de esta de la herramienta propuesta por este estudio.
- **Fuente** – Psicóloga Norma Rodríguez
- **Criterio de Evaluación** – Se evaluará en base a si la clasificación de actividades que presenta dentro de los entornos virtuales cumple o no con la clasificación establecida por el tratamiento tradicional al tratamiento de este trastorno.

Implementación – Dentro del ambiente virtual que funcionará como contenedor de todas las demás escenas virtuales, se establecen zonas en la cuales se encuentran los enlaces a las diferentes actividades correspondientes a cada nivel de dificultad. De esta manera, un paciente determinado tendrá la posibilidad de desplazarse hacia la zona donde se encuentren las actividades que le corresponden.

4. Instrucciones Claras

- **ID del requerimiento** – rnf02
- **Descripción** – Como los usuarios de la herramienta de software son principalmente niños, es de suma importancia que se les presenten instrucciones claras y precisas, tanto de manera escrita como en voz acerca de que es lo que tienen que hacer dentro de cada ambiente virtual de actividad.
- **Tipo** – Requerimiento no funcional
- **Caso de Uso** – Cuando los usuarios soliciten instrucciones acerca de lo que se tiene que hacer.
- **Fuente** – Psicóloga Norma Rodríguez

- **Criterio de Evaluación** – Se evaluará en base a la facilidad o dificultad con la que los pacientes entienden las instrucciones que les proporciona el entorno virtual.

Implementación – Existe la opción de presentar las instrucciones al paciente a través de texto. Sin embargo, esto no es suficiente ya que algunos de los pacientes más jóvenes no saben leer. Es por esto que también existe la opción de presentar las instrucciones a través de mensajes de voz.

5. Operabilidad Sencilla

- **ID del requerimiento** – rnf03
- **Descripción** – En vista de que se ubica a los usuarios de este software como terapeutas y pacientes que no tienen experiencia previa con la manipulación e interacción de ambientes virtuales, la operabilidad de los mismos tiene que ser lo mas sencilla posible.
- **Tipo** – Requerimiento no funcional
- **Caso de Uso** – La operabilidad tiene que mantenerse lo más simple posible en todos los casos de uso
- **Fuente** – Psicóloga Norma Rodríguez, Maestra Carolina Castañeda.
- **Criterio de Evaluación** – En base a que tan bien pueden los pacientes operar el sistema.

Implementación – La operabilidad del sistema se hará a través del mouse o ratón y de las flechas de izquierda, derecha, arriba, y abajo del teclado. Como la presentación se hará a través de documentos HTML, el mouse es el dispositivo ideal de navegación. Los VRML browsers tienen controles adicionales, pero estos son un tanto complejos y es por eso que serán deshabilitados para que los usuarios puedan desplazarse dentro de los ambientes virtuales utilizando el mouse y las flechas del teclado.

6. Conservar Características de las Metodologías Tradicionales

- **ID del requerimiento** – rnf04

- **Descripción** – Es necesario que los ambientes virtuales que se utilicen para tratar el trastorno en cuestión conserven los atributos que las actividades tradicionales tienen, ya que son estos atributos los que hacen que las actividades sean o no apropiadas.
- **Tipo** – Requerimiento no funcional
- **Caso de Uso** – Todos los ambientes virtuales de actividad tienen que conservar estas características
- **Criterio de Evaluación** – La psicóloga Norma Rodríguez debe aprobar o rechazar las actividades contenidas de los entornos virtuales

Implementación - Las actividades de juego que se modelan dentro de los ambientes virtuales estarán basadas en los libros de texto que actualmente se utilizan para tratar el trastorno de lateralidad y ubicación espacial, de esta manera se tiene certeza de que los ambientes virtuales modelan un tratamiento útil y con resultados comprados.

En el capítulo 3, Desarrollo en Implementación, de elabora más a detalle con respecto a la implementación de estos requerimientos.

1.4.2.3.Familias de Sistemas Relacionados

Este aspecto se refiere a que el diseño debería tener la capacidad de utilizar bloques de construcción arquitectónica reutilizados [SHA95]. La herramienta de software que este estudio pretende implementar no esta basada en algún sistema interior, ni busca ser incorporada dentro de un sistema más grande. Es por esto que dentro del diseño no pueden contemplarse elementos provenientes de otros sistemas precedentes.

1.4.3. Diseño de la Interfaz

Diseñar la interfaz de usuario es un proceso que empieza con la creación de diferentes modelos de función del sistema. De acuerdo con Rubbin [RUB88], durante este proceso se crean los siguientes modelos:

- Un modelo del diseño del sistema, que incorpora representaciones de datos, arquitectónicos, de interfaces y procedimentales del software. Anteriormente se habló acerca de la estructura arquitectónica que el software de este proyecto implementa.
- Un modelo de usuario que muestra el perfil de los usuarios finales del sistema. En la etapa de análisis de este proyecto se ubica a la población objetivo que en su totalidad esta constituida por niños. Es por esto que los entornos virtuales que se presenten a los usuarios deben ser agradables para niños, y sumamente manipulables sin la necesidad de controles complejos.

Normalmente el proceso del diseño de la interfaz del software es un proceso que se lleva a cabo en colaboración con los usuarios finales del mismo. Este estudio hasta el momento ha establecido que la población objetivo no tiene experiencia previa en cuanto a ambientes virtuales se refiere. Por ende, es válido afirmar que la percepción del sistema por parte de los usuarios finales es muy limitada. El proceso de diseño de la interfaz se basa entonces en tomar en cuenta cuidadosamente las características de la población objetivo, y en crear modelos preliminares que posteriormente son ajustados y modificados en base a las recomendaciones y observaciones que hagan los usuarios finales. Durante el proceso de desarrollo se mantienen reuniones periódicas con un terapeuta, al cual se le van mostrando los avances con el fin de obtener retroalimentación.

Por las mismas herramientas computacionales que se incorporan para desarrollar una herramienta de software con los alcances establecidos por este proyecto, hay algunos elementos de la interfaz con el usuario que no solo son los más convenientes sino también los más apropiados. Las consideraciones técnicas que tienen que ser tomadas en cuenta para desarrollar la interfaz son las siguientes

- VRML 2.0 tiene que ser desplegado por un VRML browser
- Un VRML browser es un plug-in que se integra a un Internet browser regular.
- El VRML browser tiene comandos especiales para manipular las escenas virtuales

- Una escena virtual desarrollada en VRML 2.0 es, esencialmente, un documento presentable en Internet que esta descrito en 3 dimensiones.

En base a estas consideraciones, y tomando en cuenta que uno de los atributos principales de la herramienta de software que este estudio propone es el hecho de que incorpora la portabilidad que la plataforma que Internet ofrece, lo más práctico y conveniente es diseñar una interfaz de usuario a través de un navegador de Internet.

Otro aspecto que es sumamente importante es el hecho de que el VRML browser por si solo tiene controles avanzados para navegar dentro de una escena virtual en 3 dimensiones. Considerando que la población objetivo tiene problemas de ubicación y reconocimiento del espacio, sería una contradicción enorme el pensar que los controles del VRML browser no presentarían problemas de uso a la población objetivo. Es por esta razón que los controles del VRML browser tienen que ser desactivados, para poder introducir como instrumento de navegación el mouse y las flechas (izquierda, derecha, arriba, y abajo) del teclado de la computadora.

En términos generales puede decirse que la interfaz de este proyecto está dada por páginas web que tengan incrustadas las escenas virtuales desarrolladas. Los detalles de cómo se implementa y construye la interfaz están dados en el capítulo siguiente.

1.5.Documentación

La documentación de este proyecto de software se presenta en base a los métodos orientados a objetos propuestos por Martin y Odell en su publicación “Métodos Orientados a Objetos: Consideraciones prácticas” [MAJO97]. Se utilizarán las siguientes técnicas para documentar los componentes más relevantes de la herramienta de software:

- Diagramas de eventos – Para ilustrar la manera en que un usuario del software interactúa con los entornos virtuales
- Diagramas de contexto – Para ubicar el campo de acción que abarcará el software

- Tarjetas CRC – utilizada para representar todas las clases dentro de un diseño. Este proyecto en sí no está apegado estrictamente al concepto de programación o diseño orientado a objetos, sin embargo si se utilizan algunas clases de Java para dar comportamiento a algunos objetos descritos en VRML 2.0.

Referencias a metodologías de como documentar este proyecto, y alguna documentación relevante puede consultarse en el apéndice B incluido en este documento.

Lectura 4. Estándares de desarrollo de sistemas software



Eusko Jaurlaritzaren Informatika Elkartea
Sociedad Informática del Gobierno Vasco

Estándares

Estándares de desarrollo de sistemas software

Fecha: 30/06/2011

Referencia:

EJIE S.A.
Mediterráneo, 14
Tel. 945 01 73 00*
Fax. 945 01 73 01
01010 Vitoria-Gasteiz
Posta-kutxatila / Apartado: 809
01080 Vitoria-Gasteiz
www.ejie.es

Este documento es propiedad de EJIE, S.A. y su contenido es confidencial. Este documento no puede ser reproducido, en su totalidad o parcialmente, ni mostrado a otros, ni utilizado para otros propósitos que los que han originado su entrega, sin el previo permiso escrito de EJIE, S.A.. En el caso de ser entregado en virtud de un contrato, su utilización estará limitada a lo expresamente autorizado en dicho contrato. EJIE, S.A. no podrá ser considerada responsable de eventuales errores u omisiones en la edición del documento.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías Mexico

Control de documentación

Título de documento:

Histórico de versiones

Código:

Versión: 2.0

Fecha: 30/06/2011

Resumen de cambios:

- Se incluyen las actividades y tareas derivadas del modelo estándar de aseguramiento de la calidad.
- Se especifican los roles identificados y se completa el flujo de ejecución de procesos
- Se asignan las tareas más importantes del análisis al personal de EJIE
- Se incorporan las matizaciones y cambios propuestos por DIT
- Se añade la necesidad de validar las aplicaciones en otros navegadores
- Se actualiza para contemplar la metodología de pruebas Probamet y el modelo de aseguramiento de la calidad (modelo SQA)

Cambios producidos desde la última versión

Primera versión.

Control de difusión

Responsable: Ander Martínez

Aprobado por: Begoña Gutierrez

Firma:

Fecha: 30/06/2011

Distribución:

Referencias de archivo

Autor: Ander Martínez

Nombre archivo:

Localización:

Contenido

	Capítulo/sección	Página
1	Introducción	2
2	Objetivo	2
3	Estructura	4
4	Procesos organizativos	5
4.1	Proceso: Preparación y provisión	5
4.1.1.	Actividad: Adquisición	5
4.1.2.	Actividad: Equipamiento	6
4.1.3.	Actividad: Infraestructura técnica	9
4.2	Proceso: Administración del proyecto	10
4.2.1.	Actividad: Gestión del proyecto	11
4.2.2.	Actividad: Gestión del cambio	11
5	Procesos operativos	11
5.1	Proceso: Análisis	12
5.2	Proceso: Diseño	14
5.3	Proceso: Implementación	15
5.4	Proceso: Implantación y aceptación del sistema	18
5.5	Proceso: Aseguramiento de la calidad	19
5.5.1.	Actividad: Verificación documental	20
5.5.2.	Actividad: Auditorías SQA	20
6	Flujo de procesos	23
7	Anexo I. Arquitectura	24
8	Anexo II. Contextos de albergue	26
9	Anexo III Herramientas de desarrollo	27
10	Anexo IV. Sistemas horizontales	28

1 Introducción

En el entorno de GV-EJIE para cada proyecto que es adjudicado y construido el proceso de desarrollo del software viene siendo aplicado bajo distintos criterios acordados entre proveedor y contratante. En las distintas fases de desarrollo del producto o servicio, la sistemática de trabajo y traspaso de los entregables resultado ha podido ser resuelta en muchos casos con distintas herramientas con el mismo cometido, incluso se han aplicado distintos modelos de trabajo que en la práctica han hecho dificultar el propio proceso de desarrollo.

Aunque ya están definidas las distintas normativas a seguir para cada problemática y fase del desarrollo, éstas acusan cierta dispersión y falta de coherencia en su conjunto, no por ello dejando de ser perfectamente válidas.

Desde una perspectiva más tecnológica es conveniente que todas la actividades y tareas necesarias en el proceso de desarrollo del software sean estandarizadas e interconectadas entre si facilitando y acelerando de este modo ya no solo dicho proceso sino también la propia operativa productiva de los sistemas.

2 Objetivo

Es objeto del presente documento:

- Establecer el conjunto mínimo de requerimientos y recomendaciones técnicas que estandaricen el proceso de desarrollo de software en las fases definidas por las metodologías de aplicación.
- Definir una serie de instrucciones de trabajo estandarizadas y coherentes en dicho proceso.
- Proporcionar un marco de referencia de terminología y vocabulario común para el desarrollo del software.
- Identificar las principales expectativas a gestionar en las distintas modalidades de contratación del desarrollo. Con recursos humanos y/o materiales propios y/o ajenos, etc. Se fijará especialmente en los entregables (código, ejecutables, documentación, etc.) a exigir y auditar.
- Estandarizar las herramientas de gestión asociadas a las fases del ciclo de vida del proceso de desarrollo del software.
- Enfatizar las necesidades de gestión de la calidad de los productos a implantar orientadas a minimizar al máximo los fallos del servicio en el entorno de producción.
- Limitar las posibles arquitecturas software y hardware, ajustándolas al modelo de albergue de GV-EJIE.

El estándar contiene un conjunto de procesos, actividades y tareas diseñadas para ser utilizadas en los proyectos de desarrollo software, alineadas en todos los casos con la metodología normalizada.

Se ha tomado como referencia el estándar para los procesos de ciclo de vida del software de la organización ISO, “**ISO/IEC 12207** Information Technology / Software Life Cycle Processes”, el cual establece un proceso de ciclo de vida para el software que incluye procesos y actividades que se aplican desde la definición de requisitos, pasando por la adquisición y configuración de los servicios del sistema, hasta la finalización de su uso. Este estándar tiene como objetivo principal proporcionar una estructura común para que compradores, proveedores, desarrolladores, personal de mantenimiento, operadores, gestores y técnicos involucrados en el desarrollo de software usen un lenguaje común. Este lenguaje común se establece en forma de procesos bien definidos.

El estándar no define explícitamente el contenido, nombre o formato de los entregables a producir.

El estándar será aplicable en la provisión del desarrollo y mantenimiento de un sistema software.

No contendrá el detalle de las normas definidas y aprobadas, sólo será un compendio ordenado de éstas dentro de la secuencia de fases del ciclo de vida del desarrollo software. Las normativas sólo estarán referenciadas pero no descritas íntegramente.

El ámbito de aplicación del presente documento se circunscribe tanto a los desarrollos gestionados desde GV como a los contratados desde EJIE.

El estándar identifica al menos los siguientes roles:

- **Jefe de proyecto de EJIE.** Responsable de la provisión del servicio de desarrollo o de mantenimiento de aplicaciones demandado por el cliente y de la calidad de los productos obtenidos.
- **Cliente.** El cliente es la persona que define y acuerda con el jefe de proyecto de EJIE la provisión del servicio de desarrollo o de mantenimiento de una aplicación. Se refiere al director de servicios del departamento que demanda la prestación.
- **Usuario final.** La persona que utiliza y explota el producto o aplicación obtenido como resultado del servicio contratado.
- **Proveedor.** Tercero responsable de suministrar todo o parte del servicio de desarrollo de aplicaciones solicitado por el cliente.
- **Analista responsable del proyecto de EJIE.** Responsable de la entrega del servicio de desarrollo de aplicaciones contratado por el cliente. Se refiere al técnico de análisis de EJIE que gestiona y participa en la provisión del servicio.
- **Responsable del sistema.** Responsable de la entrega del servicio de desarrollo o de mantenimiento de aplicaciones convenido. Se refiere al jefe de proyecto asignado por el proveedor externo como responsable último del servicio ante el jefe de proyecto de EJIE.
- **Responsable del módulo.** Responsable del desarrollo de un módulo o subsistema en los que se descompone el sistema software. Se refiere al analista o analista-programador asignado por el proveedor externo para la construcción de un módulo.
- **Desarrollador.** Responsable del desarrollo de un conjunto de artefactos que componen un módulo o subsistema. Se refiere al analista-programador o programador asignado por el proveedor externo para la construcción de un módulo.
- **Tester.** Responsable del diseño y ejecución de las pruebas, y del cálculo de los indicadores. Se refiere al equipo de pruebas (responsable de pruebas e ingenieros de pruebas).
- **Oficina técnica de calidad (OTC).** Vela por la ejecución de todas las actividades relacionadas con el aseguramiento de la calidad del producto software a obtener y definidas en el modelo SQA (Software Quality Assurance). Verifica el cumplimiento de las metodologías de aplicación (ARINbide y Probamet), y por ende, de los entregables documentales que estas establecen. Se refiere al equipo de SQA (responsable SQA e ingenieros SQA).
- **Oficina técnica de calidad de EJIE (OTC-EJIE).** Supervisa las actividades realizadas por las oficinas técnicas de calidad de cada proyecto.
- **Oficina de evaluación.** Evalúa los niveles de calidad obtenidos para el producto software obtenido e informa al CAB sobre dichos resultados.
- **CAB (Change advisory board).** Personal que asesora al gerente de cambios en la valoración, priorización y planificación de los cambios.

El lenguaje utilizado en la redacción de los procesos, actividades y tareas de los estándares indicará el grado de exigencia para su conformidad:

	Deberá	Indica un requisito obligatorio para su conformidad.
	Debería	Indica una fuerte recomendación que no es obligatoria para su conformidad.
	Puede	Indica una forma autorizada de cumplir un requisito o de evitar la necesidad de satisfacer la conformidad.
	Si...entonces: deberá debería puede	Indica que el grado de conformidad indicado está sujeto al cumplimiento de ciertas condiciones.



Del mismo modo se remarcán aquellos aspectos que deben ser tratados y cumplidos de manera rigurosa

3 Estructura

El estándar se descompone en:

Procesos organizativos

Engloba los procesos más relacionados con la gestión y preparación, necesarios para el desarrollo del sistema software.

- Proceso: **Preparación y provisión.**
Definir el alcance del desarrollo a realizar, las infraestructuras técnicas necesarias para su ejecución y los acuerdos de gestión del proceso.
- Proceso: **Administración del proyecto.**
Comprende las actividades y tareas necesarias para la correcta administración del proyecto.

Procesos operativos

Define los procesos a realizar desde el punto de vista de la construcción del sistema software.

- Proceso: **Análisis**
Comprende las actividades y tareas dictadas por la metodología de aplicación ARINbide en la fase de *análisis del sistema de información* (ASI), y las definidas por Probamet en la fase de *planificación de pruebas* (PPB1), y parcialmente, en la fase de *análisis y diseño de las pruebas* (APB), indicando además las herramientas de uso en cada actividad y tarea.
- Proceso: **Diseño**
Comprende las actividades y tareas dictadas por la metodología de aplicación ARINbide en la fase de *diseño del sistema de información* (DSI), y parcialmente, las definidas por Probamet en la fase de *análisis y diseño de las pruebas* (APB),, indicando además las herramientas de uso en cada actividad y tarea.
- Proceso: **Implementación**
Comprende las actividades y tareas dictadas por la metodología de aplicación ARINbide en la fase de *construcción del sistema de información* (CSI), y parcialmente, las definidas por Probamet en la fase de *ejecución de pruebas* (EPB), indicando además las herramientas de uso en cada actividad y tarea.
Se indican también las mejores prácticas de aplicación en dicha fase, orientadas a mejorar tanto la ejecución del propio proceso, como a optimizar la tarea de entrega del sistema software.

- Proceso: **Implantación y aceptación del sistema**
Comprende las actividades y tareas dictadas por la metodología de aplicación ARINbide en la fase de implantación y aceptación del sistema (IAS), y parcialmente, las definidas por Probamet en la fase de ejecución de pruebas (EPB).
- Proceso: **Aseguramiento de la calidad**
Asegurará que los productos obtenidos y los procesos del ciclo de vida del proyecto cumplen con los requerimientos y los planes establecidos.

4 Procesos organizativos

- ↑ En el proceso de desarrollo de los sistemas software se **deberá** aplicar:
- La metodología de desarrollo de aplicaciones ARINbide.
 - La metodología de pruebas Probamet.
 - El modelo de aseguramiento de la calidad estándar de GV-EJIE (Modelo SQA)

4.1 Proceso: Preparación y provisión

Definir el alcance del desarrollo a realizar, de las infraestructuras técnicas necesarias para su ejecución y de los acuerdos de gestión del proceso.

Este proceso se descompone en las actividades descritas a continuación.

4.1.1. Actividad: Adquisición

- ↑ El jefe de proyecto de EJIE **deberá** definir las necesidades del sistema a construir, adquirir o mantener, indicando además en el pliego de bases técnicas la obligatoriedad de adscripción a los presentes estándares, incluyendo entonces la exigencia en la aplicación de las metodologías ARINbide y Probamet, y del modelo de aseguramiento de la calidad del software (modelo SQA).

- ↑ Se **deberá** indicar también el valor NAC¹ (nivel de aseguramiento de la calidad) asignado al proyecto.



- ↑ El jefe de proyecto de EJIE **deberá** señalar en el pliego de bases técnicas las implicaciones emanadas de las siguientes obligaciones y recomendaciones relativas a las actividades marcadas por la metodología de desarrollo ARINbide:
- La fase de *Requisitos de usuario* que consta de las actividades de *Definición del Sistema* (ASI 1) y la de *Establecimiento de Requisitos* (ASI 2) ↑ **deberán** ser realizadas por el analista responsable del proyecto de EJIE
 - Y al menos las actividades de *Identificación de subsistemas de análisis* (ASI 3), *Análisis de casos de uso* (ASI 4) y la de *Análisis de clases* (ASI 5), ➔ **deberían** también ser realizadas por el analista responsable del proyecto de EJIE

- ↑ El jefe de proyecto de EJIE **deberá** decidir la idoneidad de la contratación de un equipo adicional que actúe como Oficina Técnica de Calidad para el proyecto, por lo tanto, distinto al que realizará las actividades de ingeniería, y al que realizará las tareas de pruebas. Incluso se podría decidir que las tareas de pruebas pasen a ser responsabilidad de dicha Oficina Técnica de Calidad. En cualquier caso se **deberá** asegurar que dicho rol queda cubierto.

¹ El valor NAC (Nivel de Aseguramiento de la Calidad) permite determinar el grado de calidad necesario para un proyecto de desarrollo software, y en consecuencia, identificar qué controles mínimos serán de obligado cumplimiento, las tipologías y niveles de pruebas a ejecutar, y los umbrales admitidos para los indicadores identificados en el modelo de aseguramiento de la calidad.

- ↑ El jefe de proyecto de EJIE **deberá** realizar el proceso de selección del proveedor, acordando los modelos de prestación del servicio más adecuados a las necesidades demandadas por el cliente, y proponiendo así la contratación del proveedor más apropiado.

4.1.2. Actividad: Equipamiento

Proveer de los bienes básicos para la ejecución del proyecto de desarrollo de software. Comprende tanto el hardware, el software, las herramientas, los estándares y las facilidades para el desarrollo, operación y mantenimiento.

Este proceso se descompone en las tareas descritas a continuación.

Tarea: Provisión del puesto de trabajo.

- ↑ Sea cual sea la modalidad de la contratación, los miembros de los equipos de desarrollo y mantenimiento **deberán** proveerse del mismo conjunto de herramientas y recursos hardware homologados en el entorno de GV-EJIE.

- ⚡ O **si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces** éste **debería** proveerse del mismo conjunto de herramientas y hardware que las especificadas para GV-EJIE, o bien de aquellas que considere suficientes para cubrir adecuadamente el proceso de desarrollo software para el que ha sido contratado, facilitando y acelerando así la entrega de los productos construidos.

Recursos software:

- ↑ Se ha establecido el conjunto mínimo de herramientas que se **deberá** disponer en base al perfil del usuario asignado. Todos los perfiles (jefe de proyecto, analista, o desarrollador) contarán con un mismo conjunto básico de herramientas, ampliéndose éste para el perfil de desarrollador.

Herramientas comunes para jefe de proyecto, analista o desarrollador:

- Software de base:
 - Windows XP
 - Microsoft Office 2003 u OpenOffice
- Navegadores:
 - Internet Explorer
 - Mozilla Firefox
 - Chrome
 - Opera
- Diseño de imágenes:
 - Gimp
- Utilidades para desarrollo:
 - Framework .NET. Necesario para ejecución aplicaciones .NET en local
 - Java JRE: Necesario para ejecución aplicaciones ·Java en local
- Bases de datos Oracle:
 - Cliente Oracle. Herramientas administrativas cliente de Oracle
 - Oracle SQL Developer. Gestión de base de datos Oracle (v>=9)

- Bases de datos SQL-Server:
 - Cliente SQL-Server. Herramientas administrativas cliente de SQL-Server
- Herramientas de desarrollo y pruebas:
 - Microsoft Project. Planificación de proyectos.
 - Enterprise Architect. Modelado UML, diagramación, documentación, e ingeniería inversa.
 - XML Spy. Edición, depuración y tratamiento de XML y tecnologías asociadas.
 - Subversion: Sistema open-source para control de versiones.
 - AIS web accesibility (AWA) Toolbar para Internet Explorer. Utilidades para análisis de normas WAI, XHTML, CSS
 - Web Developer extensión (WDE) Toolbar para Firefox. Utilidades para análisis de normas WAI, XHTML, CSS
 - Badboy y Selenium. Utilidad para generación de scripts de navegación para un sitio web.
 - JMeter. Pruebas de rendimiento y stress sobre aplicaciones web (sólo para cliente)
 - KeyToolUI: Herramienta gráfica para la manipulación de formatos criptográficos. (Certificados, keystores, firmas, claves en diversos formatos).
 - Xolido Sign: Herramienta gráfica de firma electrónica.
 - SoapUI: Herramientas para testeo de servicios web.
 - Portal SQA: Aplicación web corporativa de EJIE de gestión y consulta de resultados del Modelo SQA
- Gestión de errores e incidencias: Registro, control y gestión de las incidencias generadas en un proyecto:
 - Mantis: Aplicación web para el registro y gestión de bugs.
- Gestión de pruebas: Registro, control y gestión de pruebas en un proyecto:
 - TestLink: Aplicación web para la gestión de pruebas (requisitos, planes de pruebas, casos de prueba).
- Gestión de librerías y dependencias: control de uso de librerías de aplicación:
 - Maven: repositorio de librerías y utilidades para la descarga y publicación
 - Archiva: herramienta con interfaz gráfica para gestión de repositorios propios (departamental o de aplicaciones horizontales)

Solo para perfil desarrollador:

- Software de desarrollo:
 - Firebug: Extensión de Firefox compuesta por un paquete de utilidades para editar, monitorizar y depurar el código fuente, CSS, HTML y JavaScript de una página web.
 - Fiddler: proxy para capturar y analizar el tráfico de las peticiones web
- Herramientas de desarrollo y pruebas:
 - QuickREx: Plugin de ayuda a la creación de expresiones regulares.

Solo para perfil desarrollador J2EE/JEE:

- Software de desarrollo:
 - Entorno desarrollo java. Oracle Weblogic Server
 - JDK
 - Eclipse OPE. Entorno integrado de desarrollo J2EE

- Plugin UDA: plugin corporativo para desarrollo rápido de aplicaciones
- Hibernate Tools: plugin de Eclipse que permite generar el modelo de datos a partir de una base de datos relacional
- QuickREx: gestión de expresiones regulares
- Subversive: El proyecto Subversive se ocupa de facilitar la integración de Subversion para Eclipse.
- JUnit. Plugin para Eclipse, para pruebas unitarias.
- Checkstyle. Plugin para Eclipse, para pruebas de chequeo de código, en base a reglas de codificación.
- PMD. Plugin para Eclipse, para pruebas de chequeo de código, detectando código muerto, código repetido, etc.
- FindBugs. Plugin para Eclipse de detección de errores de código Java.
- jQuery: librería Javascript para desarrollo de aplicaciones web enriquecidas

Solo para perfil desarrollador .NET:

- Software de desarrollo:
 - Visual Studio .NET. Entorno integrado de desarrollo .NET
 - AnkhSVN: Plug-in para Visual Studio para trabajo desde puesto local de desarrollo con el repositorio de versiones Subversion.
- Herramientas de desarrollo y pruebas:
 - NUnit: Permite realizar pruebas unitarias para cualquier lenguaje de .Net.

Usuarios

➊ Se **deberá** solicitar la creación de los usuarios de red y configuraciones XLNetS necesarios en el entorno de desarrollo, y en el de pruebas cuando se estime conveniente.

Tarea: Provisión del canal de acceso.

⚡ **Si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces** se ➊ **deberá** proveer el canal de comunicaciones entre dicho puesto y los servidores de desarrollo a los que sea preciso acceder.

➋ En este supuesto, el jefe de proyecto de EJIE **deberá** asegurar que el proveedor dispone de acceso desde sus oficinas a otras herramientas corporativas de desarrollo tales como el repositorio de modelos UML “Enterprise Architect”, el de gestión de versiones “Subversion”, la herramienta de gestión de errores “Mantis” (incluido el repositorio de conocimiento de EJIE), y la de gestión de pruebas “TestLink”.

➌ Preferentemente se **deberá** optar por una conexión VPN entre el proveedor y EJIE con las siguientes prestaciones mínimas recomendadas:

- Tipo de conexión: ADSL Simétrico con IP fija
- Velocidad: 2 MB
- Software de cliente VPN (proporcionado por EJIE)

➍ Esta conexión vía VPN **deberá** condicionarse con un acuerdo previo de compromisos entre EJIE y la empresa colaboradora, e incluyendo además la provisión por parte de EJIE del software de cliente VPN y la creación de los usuarios de conexión que se hayan acordado.

Tarea: Emulación.

💡 Si se acordó que el puesto de trabajo estuviese en los locales del proveedor, entonces el adjudicatario deberá implantar en sus oficinas las soluciones de servidor que permitan agilizar el desarrollo del sistema software y su implantación posterior en los servidores de EJIE.

Existen diversas maneras de facilitar los desarrollos de aplicaciones. En general éstas pasan por intentar emular el entorno existente en EJIE, así como sus aplicaciones horizontales (emulando-virtualizando máquinas o bien con otras técnicas).

Algunas de estas sugerencias se detallan a continuación:

- **Emulación – virtualización de máquinas**

Se reduce a intentar emular los sistemas básicos existentes en el entorno de EJIE; para ello se exponen dos de las posibilidades que existen: las imágenes virtuales y las distribuciones arrancables.

- Imágenes virtuales: Se trataría de un fichero con una máquina virtual que simulase el entorno de desarrollo en servidor de EJIE; este fichero contendría todos los sistemas utilizados en EJIE con su misma configuración.
- Distribución arrancable: El sistema que estaría implantado en esta distribución arrancable sería el mismo que la imagen virtual con la diferencia de que no se necesitaría el programa base de la imagen virtual para su ejecución sino que se ejecutaría en el arranque del sistema.

- **Emulación de aplicaciones horizontales**

Esta sugerencia consiste en buscar una fórmula que permita emular de alguna manera las aplicaciones horizontales para evitar la dependencia exclusiva de estos sistemas de EJIE y que puedan provocar retrasos en los desarrollos.

Una de las posibilidades sería utilizando clases “huecas” que devuelvan unos valores predefinidos acordes con los valores válidos que retornarían esas aplicaciones horizontales y que permitan no tener que modificar el sistema a la hora de implantar la aplicación en los servidores de desarrollo de EJIE; un ejemplo práctico puede ser emular la API de XLNets mediante unas clases “huecas” y un xml que emule un token de sesión que es interrogado desde dichas clases.

4.1.3. Actividad: Infraestructura técnica

Establecer y mantener la infraestructura técnica necesaria para el resto de procesos. Comprende tanto el hardware, el software, las herramientas, los estándares y las facilidades para el desarrollo, operación y mantenimiento, desde la perspectiva del uso de los sistemas de infraestructura y de su albergue.

Este proceso se descompone al menos en las tareas descritas a continuación.

Tarea: Sistema de infraestructura XLNetS y PKI

- ↑ Los procesos de autenticación de usuarios mediante pares usuario-password o mediante certificado electrónico, así como la gestión de autorizaciones de acceso a los recursos de la aplicación deberán valerse del sistema de seguridad corporativo XLNetS.
- ↑ Se deberá solicitar la creación de los recursos y configuraciones XLNetS del entorno de desarrollo, pruebas, y producción necesarios para el sistema software (cadena de conexión a BD, usuarios, unidades orgánicas, inclusión de los PCs de trabajo en dominios, etc.)

- ↑ Se **deberá** seguir el procedimiento de administración de usuarios definido para la organización; SCP-34.
- ↑ Los certificados electrónicos de uso de la aplicación **deberán** solicitarse a Izenpe.

Tarea: Sistema de infraestructura PLATEA-Tramitación

- ↑ Se **deberá** solicitar la creación de los recursos PLATEA-Tramitación del entorno de desarrollo, pruebas, y producción necesarios para el sistema software (alta del procedimiento en el catálogo de procedimientos, alta en el catálogo de trámites y tareas, etc.)

Tarea: Sistema de infraestructura PLATEA-Presencia en Internet

- ↑ Se **deberá** solicitar la creación de los recursos PLATEA-Presencia en Internet del entorno de desarrollo, pruebas y producción necesarios para el sistema software (workareas de aplicación, usuarios Interwoven y Open Deploy, etc.)

Tarea: Albergue

- ↑ Para el sistema a construir, y en base a la combinación de colectivos usuarios y sus posibles canales de acceso, se **deberá** determinar el contexto o contextos de albergue así como su disposición en una o varias aplicaciones. Se **deberá** definir además que aplicación o aplicaciones deben desplegarse en los contextos de Internet, Intranet, Extranet-Jakinaplus o Extranet-Jaso. El proceso de desarrollo de cada aplicación **deberá** adscribirse en cada caso a los modelos de despliegue establecidos y a los libros de estilo vigentes.

- ↑ En cualquier caso se **deberán** considerar las reglas de visibilidad y acceso impuestas por el sistema de seguridad de red definido en GV, tanto para el caso de invocaciones entre distintos contextos como para peticiones dentro del mismo. Cualquier requerimiento de arquitectura de sistema que no cumpla el modelo de seguridad de red **deberá** ser estudiado y aprobado, si procede, por EJIE, siempre como paso previo al inicio de cualquiera de los procesos operativos.

Tarea: Documentación, normativas y notas técnicas

- ↑ El jefe de proyecto de EJIE **deberá** facilitar al proveedor el conjunto de las normativas vigentes, y de toda la documentación que se considere de interés en la construcción del sistema software, así como de todas aquellas notas técnicas internas que sean publicadas. **Deberá** prestar especial atención a las actualizaciones que vayan surgiendo de modo que el proveedor esté continuamente informado de las novedades o cambios que se produzcan.
- ↑ **Si** se acordó la provisión del puesto de trabajo en los locales del proveedor **entonces** el jefe de proyecto de EJIE **deberá** asegurar que éste dispone de acceso al repositorio de conocimiento técnico de EJIE, al cual podrá acudir ante cualquier eventualidad técnica que surja.

4.2 Proceso: Administración del proyecto

Comprende las actividades y tareas necesarias para la correcta administración del proyecto.

Este proceso se descompone en las actividades descritas a continuación.

4.2.1. Actividad: Gestión del proyecto

Comprende las actividades y tareas dictadas por la metodología de aplicación ARINbide en la fase de *gestión del proyecto* (GPR), las definidas por Probamet en su fase de *seguimiento de pruebas* (PPB2), y las marcadas por el modelo de aseguramiento de la calidad (Modelo SQA) en su fase de *ejecución y seguimiento de las actividades* SQA (SQA 3).

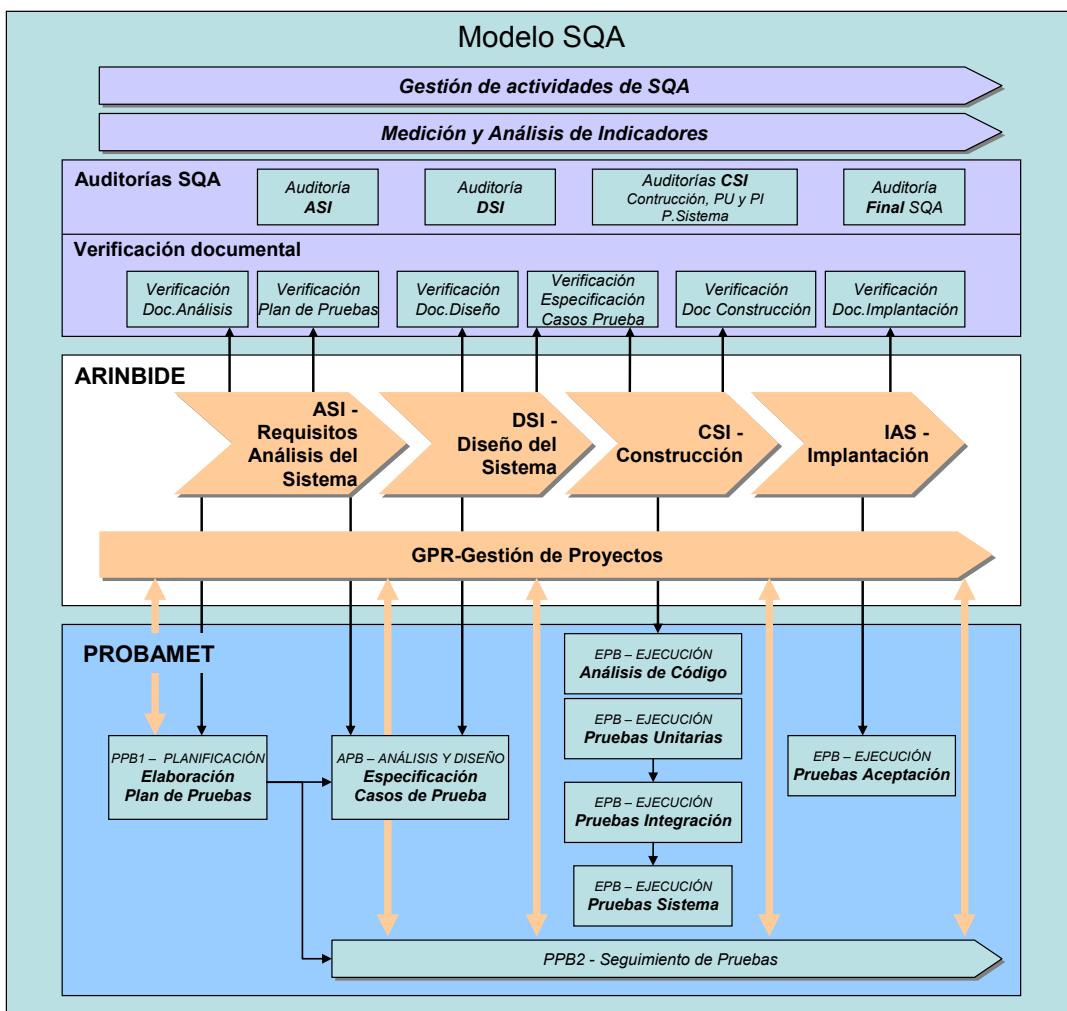
- ➊ Se **deberán** ejecutar las actividades y tareas definidas por ARINbide en el módulo de *gestión de proyectos* (GPR), así como las indicadas por Probamet en su fase de *seguimiento de pruebas* (PPB2), y las establecidas por el Modelo SQA en su fase de *ejecución y seguimiento de las actividades* SQA (SQA 3).
- ➋ Se **deberá** seguir el procedimiento de gestión de proyectos definido para la organización; SCP-60 para el caso de contrataciones gestionadas por EJIE.
- ➌ Para la generación de la documentación se **deberán** utilizar las herramientas ofimáticas homologadas en GV-EJIE, Microsoft Office u OpenOffice.
- ➍ Para elaborar los entregables de planificación se **deberá** utilizar Microsoft Project o cualquier otra herramienta que permita su importación posterior en ésta.

4.2.2. Actividad: Gestión del cambio

- ➊ Se **deberá** seguir el procedimiento de gestión de cambios definido para la organización.
- ➋  Los traspasos entre entornos (desde desarrollo a pruebas, y desde pruebas a producción) **deberán** ser planificados y propuestos con antelación suficiente para que los servicios de gestión del cambio y gestión del despliegue de EJIE puedan planear adecuadamente los trabajos a realizar.
- ➌ Para la generación de la documentación se **deberá** proveer de Microsoft Office o de OpenOffice como herramienta ofimática homologada en GV-EJIE.

5 Procesos operativos

- ➊ Se **deberá** aplicar:
 - La metodología de desarrollo de aplicaciones ARINbide.
 - La metodología de pruebas Probamet.
 - El modelo de aseguramiento de la calidad estándar de GV-EJIE (Modelo SQA)



5.1 Proceso: Análisis

Comprende las actividades y tareas dictadas por la metodología de aplicación ARINbide en la fase de *análisis del sistema de información* (ASI), y las definidas por Probamet en la fase de *planificación de pruebas* (PPB1), y parcialmente, en la fase de *análisis y diseño de las pruebas* (APB). Se indica para cada caso las herramientas de uso en cada actividad y tarea.

Puesto que el proceso se corresponde con las fases de ARINbide y de Probamet ya indicadas, no se detallan las actividades y tareas en las que se descompone, sólo se remarcán ciertos aspectos especialmente importantes que **deberán** o **deberían** cumplirse.

- ① Se **deberán** ejecutar las actividades y tareas definidas por ARINbide en esta fase, así como las dictadas por Probamet para este ciclo. Estas se **deberán** realizar considerando el uso de sólo las herramientas, librerías y sistemas especificados en el documento de estándares tecnológicos publicado por la Dirección de Informática y Telecomunicaciones del GV. Ante cualquier otra nueva necesidad, EJIE **deberá** realizar un estudio detallado determinándose la idoneidad o no de la propuesta.
- ② Las actividades de *Definición del Sistema* (ASI 1) y la de *Establecimiento de Requisitos* (ASI 2) definidas por la metodología de desarrollo ARINbide en su fase de *Requisitos de usuario*, ① **deberán** ser realizadas por

el analista responsable del proyecto de EJIE.

- ⌚ Y al menos las actividades de *Identificación de subsistemas de análisis* (ASI 3), *Análisis de casos de uso* (ASI 4) y la de *Análisis de clases* (ASI 5), también ⌚ **deberían** ser realizadas por el analista responsable del proyecto de EJIE.
- ⌚ Para la generación de la documentación se **deberá** utilizar Microsoft Office u OpenOffice como herramienta ofimática homologada en GV-EJIE. ⚡ Y **si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces** éste ⚡ **puede** utilizar cualquier otra herramienta que permita generar los formatos compatibles que puedan ser gestionados por alguna de estas.
- ⌚ Para la creación de los modelos UML y de sus descripciones detalladas a incluir en la documentación, se **deberá** utilizar "Enterprise Architect". ⚡ Y **si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces** éste ⚡ **puede** utilizar cualquier otra herramienta que permita generar el formato estándar XML, de modo que dichos modelos resultado puedan ser gestionados por esta.
- ⌚ Se **deberá** utilizar un único repositorio de modelos UML. Si el proyecto es de cierta envergadura dicho repositorio se **deberá** crear en el servidor remoto habilitado a tal efecto, facilitando así el trabajo en equipo
- ⌚ En la actividad de definición de interfaces de usuario, los prototipos **deberán** construirse con un software de prototipado al uso (como por ejemplo Pencil o Microsoft Web Express) si se opta por la opción de posibilidad de navegación, o con GIMP si se prefieren modelos más estáticos. Y **si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces** éste ⚡ **puede** utilizar cualquier otra herramienta que permita generar los formatos compatibles con estas.
- ⌚ Se **deberá** poner especial atención al proceso de revisión y aprobación de requisitos definidos y especificados para el sistema software.



- ⌚ El valor NAC asignado al proyecto determinará las tipologías y niveles de pruebas a ejecutar. Así, el plan de pruebas **deberá** contemplar la ejecución del conjunto de:
- Pruebas de aceptación. Que deberá realizar el usuario en el entorno de pruebas, y que permitirán determinar la aceptación o no del sistema construido.
 - Pruebas de sistema. Conjunto de pruebas del sistema software que aseguren la cobertura completa de la especificación de requisitos.
 - Pruebas funcionales. Que permitan asegurar que se cumplen los requisitos funcionales definidos para cada uno de los módulos de análisis del sistema.
 - Pruebas no funcionales. Los ensayos de instalación y configuración, de consistencia de datos, de seguridad, de prestaciones (rendimiento, escalabilidad, capacidad, carga, estrés, volumen y estabilidad), de fallo y recuperación del sistema, de accesibilidad y de usabilidad.
- ⌚ El plan de pruebas **deberá** identificar también:
- Pruebas de integración. Se **deberá** establecer al menos la estrategia de pruebas de integración a realizar.
 - Pruebas unitarias. En esta fase se **deberán** identificar al menos los caminos y procesos críticos (riesgos) del sistema a partir de los cuales en la fase de implementación se crearán los artefactos de pruebas básicas de componentes.
- ⌚ En esta fase se **deberán** definir al menos a alto nivel los conjuntos de casos de prueba

emanados de las pruebas de aceptación y de sistema.

5.2 Proceso: Diseño

Comprende las actividades y tareas dictadas por la metodología de aplicación ARINbide en la fase de *diseño del sistema de información* (DSI), y parcialmente, las definidas por Probamet en la fase de *análisis y diseño de las pruebas* (APB). Se indican para cada caso las herramientas de uso en cada actividad y tarea.

Puesto que el proceso se corresponde con las fases de ARINbide y de Probamet ya indicadas, no se detallan las actividades y tareas en las que se descompone, sólo se remarcán ciertos aspectos especialmente importantes que **deberán** o **deberían** cumplirse.

- ➊ Se **deberán** ejecutar las actividades y tareas definidas por ARINbide en esta fase, así como las dictadas por Probamet para este ciclo. Éstas se **deberán** realizar considerando el uso de sólo las herramientas, librerías y sistemas especificados en el documento de estándares tecnológicos publicado por la Dirección de informática y telecomunicaciones del GV. Ante cualquier otra nueva necesidad, EJIE **deberá** realizar un estudio detallado determinándose la idoneidad de la propuesta.
- ➋ Para la generación de la documentación se **deberá** utilizar Microsoft Office u OpenOffice como herramienta ofimática homologada en GV-EJIE. ⚡ Y **si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces** éste **puede** utilizar cualquier otra herramienta que permita generar los formatos compatibles que puedan ser gestionados por alguna de estas.
- ➌ Para la creación de los modelos UML y de sus descripciones detalladas a incluir en la documentación, se **deberá** utilizar "Enterprise Architect". ⚡ Y **si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces** éste **puede** utilizar cualquier otra herramienta que permita generar el formato estándar XMI, de modo que dichos modelos resultado puedan ser gestionados por esta.
- ➍ Se **deberá** utilizar el mismo repositorio de modelos UML y definiciones detalladas que el solicitado en la fase de análisis.



- ➊ Se **deberá** completar exhaustivamente el plan de pruebas establecido en el proceso de análisis y almacenado en el repositorio "TestLink", ampliando y detallando además el conjunto de casos de prueba a realizar.
- ➋ Si se trata de un aplicativo con interfaz web que será desplegado en el contexto de internet, las pruebas del sistema **deberán** contemplar los ensayos necesarios para garantizar que el sistema se comporta correctamente en los navegadores más utilizados (Internet Explorer, Firefox, Chrome, Opera, y Safari)². Mientras que si el contexto de despliegue es intranet (con PCs gestionados), el sistema **deberá** soportar los navegadores homologados por los estándares tecnológicos publicados por la Dirección de informática y telecomunicaciones del GV para dicho contexto (Internet Explorer y Firefox). Para el caso de extranet, la lista de navegadores a soportar se **deberá** decidir en función de los colectivos usuarios de la aplicación y del uso o no de PCs gestionados.
- ➌ Se **deberá** establecer perfectamente la matriz de trazabilidad requisito – casos de prueba que permitirá validar que éstos se cumplen y se satisfacen adecuadamente.

² Se han incluido los navegadores más utilizados a nivel mundial a fecha de Enero de 2010. No obstante, y con objeto de dar cobertura al máximo número de usuarios en el momento de realizar el proyecto, se recomienda consultar las estadísticas de uso de los navegadores web en el contexto de los interesados de la aplicación que se va a construir, para así ampliar o reducir para cada caso la lista de navegadores a soportar.

↑ Con objeto de minimizar el esfuerzo a realizar, y haciendo uso del conjunto de herramientas homologadas, se **deberá** sistematizar al máximo el proceso de ejecución de las pruebas que se hayan definido.

↑ Se **deberán** tener en consideración las restricciones impuestas por el modelo de seguridad de red y por ende las arquitecturas lógicas y físicas que este impone. Para más información consultar el *Anexo I. Arquitectura*.

↑ Se **deberá** considerar especialmente la información necesaria a reflejar en el manual de implantación del sistema software. Para más información consultar el *proceso de implantación y aceptación del sistema (5.4)*

↑ Las tareas relacionadas con la gestión de pruebas establecidas por Probamet **deberán** realizarse con la herramienta de gestión de pruebas estándar, TestLink, conteniendo por lo tanto el conjunto de requisitos definidos para el proyecto, así como el detalle de los casos de prueba que los validan.

5.3 Proceso: Implementación

Comprende las actividades y tareas dictadas por la metodología de aplicación ARINbide en la fase de *construcción del sistema de información* (CSI), y parcialmente, las definidas por Probamet en la fase de *ejecución de pruebas* (EPB). Se indica para cada caso las herramientas de uso en cada actividad y tarea.

Puesto que el proceso se corresponde con las fases de ARINbide y de Probamet ya indicadas, no se detallan las actividades y tareas en las que se descompone, sólo se remarcan ciertos aspectos especialmente importantes que **deberán** o **deberían** cumplirse.

Se indican también las mejores prácticas de aplicación en dichas fases, orientadas a mejorar tanto la ejecución de los propios procesos como a optimizar la tarea de entrega del sistema software.

↑ Se **deberán** ejecutar las actividades y tareas definidas por ARINbide en esta fase, así como las dictadas por Probamet para este ciclo. Éstas se **deberán** realizar considerando el uso de sólo las herramientas, librerías y sistemas especificados en el documento de estándares tecnológicos publicado por la Dirección de informática y telecomunicaciones del GV. Ante cualquier otra nueva necesidad, EJIE **deberá** realizar un estudio detallado determinándose la idoneidad de la propuesta.

↑ Para la generación de la documentación se **deberá** utilizar Microsoft Office u OpenOffice como herramienta ofimática homologada en GV-EJIE. ⚡ Y **si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces** éste **puede** utilizar cualquier otra herramienta que permita generar los formatos compatibles que puedan ser gestionados por alguna de estas.

↑ Para la creación de los modelos UML a incluir en la documentación se **deberá** utilizar "Enterprise Architect". ⚡ Y **si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces** éste **puede** utilizar cualquier otra herramienta que permita generar el formato estándar XMI, de modo que dichos modelos resultado puedan ser gestionados por esta.

↑ Se **deberá** utilizar el mismo repositorio UML que el solicitado en el proceso de análisis (5.1).

↑ El procedimiento de catalogación y gestión de errores que puedan producirse durante el proceso de implementación de cualquiera de las áreas del sistema software (código ejecutable, elementos estáticos, scripts de creación de BD, ejecución del plan de pruebas, etc.) **deberán** tratarse utilizando la herramienta "Mantis".

Previamente se **deberá** crear dicho repositorio y asignar los roles (informador, validador, resolutor, etc.) y usuarios que se consideren necesarios.

- ↑ Las tareas relacionadas con la gestión de pruebas establecidas por Probamet **deberán** realizarse con la herramienta de gestión de pruebas estándar, TestLink, conteniendo por lo tanto el conjunto de requisitos definidos para el proyecto, el detalle de los casos de prueba que los validan, y los resultados obtenidos para cada caso de prueba.
- Y El tratamiento de los ficheros “planos” se **puede** realizar con la herramienta “UltraEdit32”.
- Y El diseño de los ficheros XML se **puede** realizar con la herramienta “XMLSpy”.
- ↑ El acceso al servidor de desarrollo de EJIE se **deberá** realizar con la herramienta “SecureCRT”, o con “SecureFX” si lo que se desea es hacer una carga FTP. ⚡ Y **si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces** éste **puede** utilizar cualquier otra herramienta compatible con la configuración del servidor.



↑ Se **deberán** generar y ejecutar intensivamente los casos de prueba definidos en los procesos anteriores para el sistema software.

Contenidos estáticos

- ↑ El diseño de las interfaces gráficas en formato HTML junto con el resto de artefactos web (CSS, javascript, etc.) se **deberá** realizar con “Microsoft Web Designer”
- ↑ Las imágenes se **deberán** crear con la herramienta “GIMP”. ⚡ Y **si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces** éste **puede** utilizar cualquier otra herramienta que permita generar los formatos compatibles con esta.
- ↑ Las entregas de contenidos estáticos se **deberán** realizar siempre a través del repositorio de versionado “Subversion”, en la rama de “estatico” .
- ↑ Para sistemas a desplegar en el contexto de Internet, los contenidos estáticos **deberán** ser cargados en la “workarea” de aplicación del servidor de contenidos “Interwoven”. Posteriormente se **deberán** desplegar en los servidores web Apache con la herramienta de “Open Deploy”. Este proceso se repetirá en los entornos de pruebas y de producción.
- ↑ En el resto de contextos (Intranet y Extranet) se **deberán** copiar estos contenidos en el servidor web “Apache” del entorno correspondiente haciendo una carga FTP. Y para el paso a producción se **deberá** utilizar la aplicación M26 para su despliegue definitivo en dicho entorno.

Código, configuración y pruebas

- ↑ **Si** se trata de un sistema con tecnología JEE **entonces** se **deberá** utilizar “UDA” como utilidad de generación de código básico a partir del cual se construirá el sistema.
- ↑ Como herramienta de desarrollo IDE en entorno J2EE se **deberá** utilizar “Eclipse” junto con el plugin “UDA”.

deberá



↑ El proceso de codificación y el de generación del conjunto de pruebas unitarias se **deberá** realizar **en el PC local**. **No se permite** desplegar en el servidor de desarrollo ningún código que no haya sido perfectamente probado y validado previamente en el entorno de PC local.

↑ **No se permite** utilizar el servidor de desarrollo como única herramienta o entorno de compilación y pruebas unitarias.

↑ Las entregas de código, de los ficheros de configuración en el servidor de desarrollo y de los conjuntos de pruebas automatizadas (unitarias, integración y sistema) se **deberán** realizar siempre a través del repositorio de versionado “Subversion” en la rama de “codigo”, en la de “config”, y en las de “test” (test_unit, test_integration y test_system) respectivamente. Es decir, desde el PC de desarrollo se alimentará el repositorio, y desde éste y haciendo uso de las tareas habilitadas a tal efecto se descargará en el servidor de desarrollo para su compilación y despliegue.

↑ Haciendo uso de la herramienta para la automatización de la ejecución de pruebas, “Hudson”, el desarrollador **deberá** lanzar el proceso de análisis estático del código, así como todas aquellas pruebas unitarias y de integración automáticas que se hayan configuradas. Este proceso se podrá ejecutar tantas veces como se considere necesario. Los resultados obtenidos se recogerán automáticamente en la herramienta de gestión de pruebas “TestLink” y las incidencias producidas se darán de alta en la herramienta de gestión de errores “Mantis”.

↑ Sólo el responsable del módulo **podrá** realizar el despliegue de dicho módulo en el servidor de desarrollo. Esta acción se **deberá** hacer de manera programada y coordinada con el resto de responsables de módulos, idealmente un máximo de una vez al día hasta que el módulo testeado se considere correcto.



↑ Se **deberá** reducir por tanto al máximo el número de despliegues a realizar en el servidor de desarrollo.

↑ Cualquier código que esté generando problemas de estabilidad en el servidor de desarrollo **deberá** ser descargado y subsanado en el entorno PC local, nunca en el servidor.

✓ **Una vez desplegado el módulo:**

↑ Haciendo uso de la herramienta de generación de scripts automáticos de pruebas de sistema-funcionales, “Selenium”, el responsable del módulo **deberá** generar los escenarios de validación y pruebas de cada uno de los módulos contenidos en su ámbito de responsabilidad.

↑ El responsable del módulo y/o el desarrollador **deberá** generar también el conjunto de pruebas de sistema no funcionales: de instalación y configuración, de consistencia de datos, de seguridad, de prestaciones (rendimiento, escalabilidad, capacidad, carga, estrés, volumen y estabilidad), de fallo y recuperación del sistema, de accesibilidad y de usabilidad. Se **deberá** utilizar el conjunto de herramientas estandarizadas para cada caso (“Ais Web Accessibility For Internet Explorer”, soapUI, etc.). Para más información consultar el Anexo III. *Herramientas de desarrollo*.

↑ Haciendo uso de la herramienta para la automatización de la ejecución de pruebas, “Hudson”, el responsable del módulo **deberá** lanzar, para cada uno de sus módulos, el proceso de validación de todas aquellas pruebas de sistema automáticas que se hayan configurado. Este proceso se podrá ejecutar tantas veces como se considere necesario. Los resultados obtenidos se **deberán** recoger en la herramienta de gestión de pruebas “TestLink” y las incidencias producidas se **deberán** dar de alta en la herramienta de gestión de errores “Mantis”.

- ↑ Los procesos de ejecución de pruebas que no sea posible automatizar con el uso de las herramientas **deberán** realizarse de manera “manual”.
- ✓ **Sólo cuando ya estén validados los distintos módulos de la aplicación:**
- ↑ De modo similar al estadio anterior (pruebas de módulos), el responsable del sistema y/o el desarrollador **deberán** generar, configurar y ejecutar las pruebas de integración entre módulos, y las de sistema (funcionales y no funcionales), pero esta vez para todo el sistema.

Datos

- ↑ Las tareas de gestión de los datos albergados en base de datos se **deberán** realizar con “SQLDeveloper” si la base de datos es Oracle y con la “Consola SQLServer” si la base de datos es SQLServer.
- ↑ Las entregas de scripts de creación de base de datos se **deberán** realizar siempre a través del repositorio de versionado “Subversion” en la rama de “scripts”, y los datos en la rama de “datos”.
- ↑ Se **deberá** definir adecuadamente la volumetría de datos del sistema, cumplimentando para ello la hoja de cálculo estándar en EJIE.
- ⚡ En lo referente al código fuente, los contenidos estáticos y scripts de creación de BD, **si** se acordó que el puesto de trabajo estuviese en los locales del proveedor, **entonces**, y tras acuerdo por ambas partes, éste **puede** utilizar cualquier conjunto de herramientas de sus elección siempre y cuando permitan entregar a EJIE los formatos de soporte definidos para los servidores de desarrollo de GV-EJIE.

5.4 Proceso: Implementación y aceptación del sistema

Comprende las actividades y tareas dictadas por la metodología de aplicación ARINbide en la fase de implantación y aceptación del sistema (IAS), y parcialmente, las definidas por Probamet en la fase de ejecución de pruebas (EPB). Se indica para cada caso las herramientas de uso en cada actividad y tarea.

Puesto que el proceso se corresponde con las fases de ARINbide y de Probamet ya indicadas, no se detallan las actividades y tareas en las que se descompone, sólo se remarcán ciertos aspectos especialmente importantes que **deberán** o **deberían** cumplirse.

- ↑ Se **deberán** ejecutar las actividades y tareas definidas por ARINbide en esta fase, así como las dictadas por Probamet para este ciclo. Éstas se **deberán** realizar considerando el uso de sólo las herramientas, librerías y sistemas especificados en el documento de estándares tecnológicos publicado por la Dirección de informática y telecomunicaciones del GV. Ante cualquier otra nueva necesidad, EJIE **deberá** realizar un estudio detallado determinándose la idoneidad de la propuesta.
- ↑ De modo similar al estadio anterior (pruebas de sistema), el usuario final y/o el responsable del sistema **deberán** ejecutar las pruebas de aceptación de usuario que se hayan definido. Con el fin de realizar estas pruebas en un entorno más similar al de producción, estas **deberán** realizarse obligatoriamente en el entorno de pruebas.
- ↑ Las incidencias y posibles mejoras detectadas por el usuario se **deberán** registrar y tratar a través de la herramienta “Mantis”, siguiendo siempre el proceso de resolución del ciclo completo de creación de código.



↑ Como paso previo al despliegue definitivo del sistema en el entorno de producción, y obligatoriamente en el entorno de pruebas, se **deberán** ejecutar el conjunto total de pruebas (análisis estático del código, pruebas unitarias, de integración y de sistema), vaidando que el resultado de dichas pruebas es el esperado, y que por lo tanto el sistema cumple con los indicadores de calidad establecidos.

- ↑ Se **deberá** reflejar en el manual de implantación del sistema software:
- Necesidades de visibilidad y acceso entre los componentes del sistema y entre máquinas físicas y otros sistemas de GV o externos (DDFF, Ministerios –red SARA-, etc.), puertos de escucha (si es que se conocen)
 - Protocolos de comunicación utilizados (HTTP, HTTPS sobre SSL one-way o two-way, FTP, NFS, IMAP, POP3, etc.)
 - Software o librerías homologadas necesarias junto con sus versiones
 - Parámetros de configuración de:
 - Pool (JDBC) de conexión a BD
 - Descriptores de despliegue de los EJBs de la aplicación
 - Colas JMS
 - Etc.
 - Relación de scripts de creación de objetos de BD u otros, que se deben ejecutar.
 - Configuraciones a realizar en otros sistemas de infraestructura o de plataforma como puede ser PLATEA, correo Exchange, etc.
 - Otras necesidades especiales

En definitiva, todo aquello que se considere relevante para la correcta instalación del sistema en los distintos entornos. Se **deberán** elaborar entonces:

- Esquema detallado de la arquitectura física del sistema
- Esquema detallado en el que se refleje su arquitectura lógica

- ↑ En consonancia con la actividad de gestión del cambio (4.2.2) definida como parte del proceso de administración del proyecto (4.2), se **deberá** elaborar detalladamente el plan de implantación del sistema, que **deberá** contemplar las especificaciones y requisitos definidos en el manual de implantación. Se asegura así la correcta instalación del producto en los entornos de pruebas y de producción.

5.5 Proceso: Aseguramiento de la calidad

Asegurará que los productos obtenidos y los procesos del ciclo de vida del proyecto cumplen con los requerimientos y los planes establecidos.

- ↑ Se **deberá** aplicar el modelo de aseguramiento de la calidad estándar en GV-EJIE (Modelo SQA).
- ! El proceso de obtención de los indicadores que realmente determinan el grado de calidad del producto software se **deberá** realizar en el entorno de pruebas, puesto que se entiende que este entorno cuenta con una configuración y recursos más similares a los disponibles en el entorno de producción.
- ↑ Todo proyecto de construcción o mantenimiento de un sistema software **deberá** incluir obligatoriamente el rol de Oficina Técnica de Calidad.
- ↑ El proceso de aseguramiento de la calidad **deberá** ser ejecutado por la Oficina Técnica de Calidad del proyecto, que **deberá**:
 - Asegurar que se ejecutan todas las actividades relacionadas con el aseguramiento de la calidad

del producto software definidas en el modelo SQA

- Verificar el cumplimiento de las metodologías de aplicación: ARINbide y Probamet, y por ende, que los entregables documentales que estas establecen son correctos y suficientes.
- Verificar que se superan los umbrales establecidos por los indicadores de calidad.

➊ El Jefe de proyecto de EJIE, como responsable último del proyecto, **deberá** asegurar que el sistema software construido cumple con los estándares de calidad definidos.

➋ El Modelo SQA se fundamenta en un calendario de entregas **versionadas** del producto (documentación y ejecutables) y en el **plan SQA** que **deberá** ser elaborado y acordado previamente entre los equipos de desarrollo y de pruebas, y la oficina técnica de calidad.

Este proceso se descompone en las actividades descritas a continuación.

5.5.1. Actividad: Verificación documental

En el tiempo, se ejecutará en paralelo a cada uno de los proceso de desarrollo y pruebas del sistema software.

En función del NAC asignado al proyecto se habrán determinado ya:

- Los controles de calidad (SQA) a realizar durante el desarrollo del sistema.
- Las tipologías de pruebas a ejecutar.
- Los indicadores estándar y sus umbrales permitidos

➊ En las entregas de la documentación asociada al proyecto se **deberá** asegurar que se cumplen los controles de calidad (SQA) definidos, garantizando así el cumplimiento de la metodología de desarrollo, ARINbide, y de pruebas, Probamet, así como la calidad de dicha documentación:

- Verificación documentación de análisis
- Verificación del pan de pruebas
- Verificación documentación de diseño
- Verificación de la especificación de casos de prueba
- Verificación documentación de construcción
- Verificación documentación de implantación

➋ Los resultados de la verificación realizada **deberán** recogerse en el “portal SQA” de EJIE, formando parte entonces de los indicadores de calidad calculados para el sistema software.

➌ El analista responsable del proyecto de EJIE **deberá** asegurar que se cumplen los controles de calidad (SQA) definidos.

5.5.2. Actividad: Auditorías SQA

En el tiempo, se ejecutará en paralelo a cada uno de los proceso de desarrollo y pruebas del sistema software.

➊ Al finalizar cada una de las fases del desarrollo y pruebas del sistema software indicadas por ARINbide (ASI; DSI, CSI e IAS) y por Probamet se **deberá** ejecutar la auditoría de fin de fase, obteniendo así el grado de calidad del sistema según los indicadores recogidos:

- Indicadores de pruebas
 - Análisis estático de código

- Pruebas unitarias
- Pruebas de integración
- Pruebas de sistema
 - Pruebas basadas en requisitos (tanto funcionales como no funcionales)
 - Rendimiento (prestaciones):
 - ✓ Cumplimiento global indicadores prestaciones
 - ✓ Indicadores servidor web
 - ✓ Indicadores servidor de aplicaciones
 - ✓ Indicadores servidor bbdd
 - Seguridad grado cumplimiento owasp top 10
 - Usabilidad grado cumplimiento evaluación heurística
 - Accesibilidad:
- Indicadores de calidad SQA
 - Indicadores de Fin de Fase



- ➊ Se **deberá** prestar especial atención a los indicadores de rendimiento obtenidos (uso de la memoria del servidor, de la CPU, tiempo de respuesta de las peticiones http, parámetros de configuración de pool de conexión a base de datos, descriptores de despliegue de EJBS, colas JMS, etc.) los cuales normalmente tienden a descuidarse.
- ➋ En el entorno de pruebas, haciendo uso de la herramienta de pruebas de carga “LoadRunner” se **debería** asegurar que el sistema software construido cumple con las especificaciones de rendimiento definidas. Pero **si** se entiende que el sistema debe soportar una gran carga de trabajo, bien por el tipo de proceso que resuelve, bien por el número de usuarios concurrentes o por cualquier otra circunstancia, **entonces** las pruebas de carga **deberán** realizarse de manera obligatoria.

- ➌ Los resultados obtenidos **deberán** recogerse en el portal SQA de EJIE, formando parte entonces de los indicadores de calidad calculados para el sistema software.
- ➍ Las incidencias y posibles mejoras detectadas se **deberán** registrar y tratar a través de la herramienta “Mantis”. Estas **deberán** ser solventadas en los PCs de desarrollo, nunca en el servidor y siguiendo siempre el proceso de resolución del ciclo completo del código.
- ➎ Ante cualquier incidente externo al sistema software, se **deberá** seguir el procedimiento de gestión de incidencias definido para la organización; SCP-32.
- ➏ El analista responsable del proyecto de EJIE **deberá** asegurar que se cumplen los controles de calidad (SQA) definidos, garantizando así el cumplimiento de los indicadores de calidad establecidos y por tanto, con los requisitos del sistema.
- ➐ Durante la fase de construcción del sistema (CSI) en los PCs locales se **deberá** asegurar que el código construido cumple con las reglas de calidad de código estático definidas por las herramientas estandarizadas: “CheckStyle”, “PMD” y “FindBugs”.

Traspaso al entorno de Producción

- ➑ Una vez que el usuario final acepta el sistema software construido, y siguiendo la normativa de traspasos vigente, se **deberá** realizar la implantación del aplicativo en el entorno de producción.
- ➒ Como paso previo al entorno de producción se **deberá** ejecutar la auditoría final SQA el cual permitirá consolidar los resultados de las actividades de calidad realizadas en el proyecto y el grado de calidad obtenido

en los productos generados. Esta auditoría establece además conclusiones y recomendaciones para el CAB en la toma de decisiones para el paso a producción del sistema.

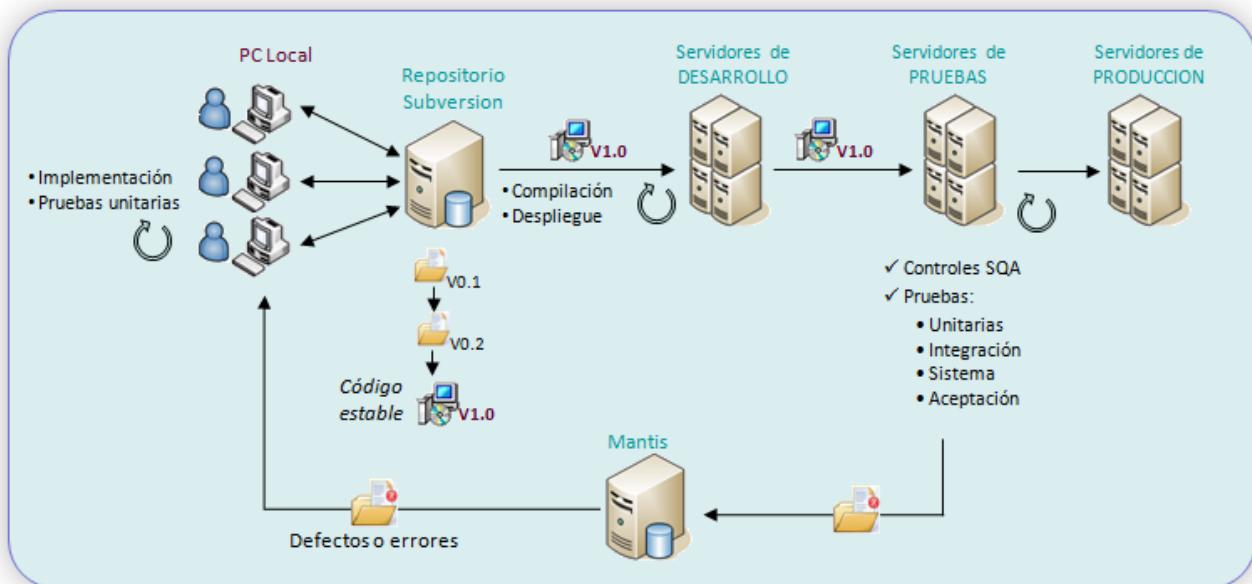
- ↑ El analista responsable del proyecto de EJIE **deberá** crear en “Subversion” una versión estable del sistema software para el entorno de producción.
- ↑ El sistema software **deberá** ser traspasado desde el entorno de pruebas ajustando las configuraciones óptimas calculadas y especificadas en el manual de implantación
 - ⚠ Durante los dos primeros meses en los que el sistema esté operativo en el entorno de producción se **deberá** realizar un seguimiento exhaustivo del mismo asegurando así que éste funciona de manera adecuada. Se **deberá** comprobar especialmente todo aquello que pueda provocar la inestabilidad del sistema y por extensión, de los servidores en los que se encuentren desplegados
- ↑ Transcurrido el período de depuración y ajuste final en el entorno de producción, si no se tiene previsto que el sistema software sufra ningún cambio, adaptación o mejora, este **deberá** deshabilitarse del servidor de desarrollo y de pruebas, evitando así un consumo innecesario de recursos.

6 Flujo de procesos

- ➊ El flujo de ejecución de procesos **deberá** seguir el siguiente esquema:



Y para el proceso de implementación:



- ⚡ Si se acordó la provisión del puesto de trabajo en los locales del proveedor **entonces** éste **deberá** entregar el sistema software debidamente probado y validado previamente en sus instalaciones. No obstante se **deberán** realizar entregas parciales por módulos o fases para poder validar y asegurar de manera temprana y continua la calidad del sistema que se está construyendo. Todas las entregas de contenidos estáticos, código, scripts de base de datos, etc. **deberán** realizarse sobre el repositorio de versiones "Subversion", y por tanto correctamente etiquetados e identificados, evitando siempre la carga directa por SFTP sobre los servidores de desarrollo.

- ↑ El equipo de desarrollo **deberá** realizar la implementación del sistema y la codificación de las pruebas unitarias en los PCs locales.
- ↑ Puesto que el código generado se deberá compilar y desplegar en el servidor de desarrollo, este se **deberá** descargar siempre desde el repositorio “Subversion” al disco del servidor, para así poder ser compilado y desplegado. Un vez realizado se podrán ejecutar todas las pruebas que se considere necesarias.
- Y El desarrollador **podrá** dejar en el repositorio de “Subversion” tantas versiones como considere necesario.
- ↑ Se **deberá** realizar la subida y ejecución de los scripts del modelo de datos en los servidores de back-end.
- ↑ Se **deberá** realizar la subida del contenido estático mediante la aplicación M26 (aplicación intranet) o a través de Interwoven y Open Deploy (aplicación Internet)
- Y En el entorno de desarrollo, una vez realizadas el conjunto de pruebas básicas que garanticen al menos el funcionamiento elemental del sistema, este se **podrá** promocionar al entorno de pruebas.
- ↑ Se **deberán** definir y generar los scripts de simulación de escenarios, pudiendo ejecutar así las pruebas de sistema.
- ↑ En el entorno de pruebas, el equipo de pruebas **deberá** obtener los valores de las métricas marcadas por los indicadores definidos por el modelo de aseguramiento de la calidad (análisis estático del código, pruebas unitarias, pruebas de integración y pruebas de sistema) reflejando dicha información en el “Portal SQA”.
- Y En el entorno de pruebas se **deberían** realizar las pruebas de carga del sistema que se consideren necesarias.
- ↑ Se **deberá** asegurar también que la documentación asociada al proyecto cumplen los controles de calidad definidos, garantizando así el cumplimiento de la metodología de desarrollo, ARINbide, y de pruebas, Probamet.
- ↑ La Oficina técnica de calidad deberá velar por la ejecución de todas las actividades relacionadas con el aseguramiento de la calidad del producto software a obtener y definidas en el modelo SQA, y comprobar que los resultados obtenidos están dentro de los umbrales aceptados.
- ↑ El usuario final **deberá** realizar las pruebas del sistema. Los defectos o errores, y por tanto las modificaciones, **deberán** seguir el mismo proceso que el resto de incidencias detectados en las fases de desarrollo del sistema.
- ↑ En todos los casos, las incidencias o errores **deberán** reflejarse y tramitarse a través de “Mantis”, asignándose así la subsanación al desarrollador correspondiente
- ↑ Una vez aceptado el sistema software en el entorno de pruebas por parte del usuario final, se **deberá** establecer como versión para producción la que tenga actualmente el entorno de pruebas, es decir, la aceptada por el usuario, siendo por lo tanto la que se traspasará al entorno de producción.
- ↑ Se **deberá** realizar el traspaso del sistema software al entorno de producción.
- ↑ Como paso previo a la implantación definitiva en el entorno de producción, la Oficina de evaluación **deberá** informar al CAB al respecto de las conclusiones y recomendaciones obtenidas en lo relativo a los niveles de calidad del sistema software a implantar.

7 Anexo I. Arquitectura

En el ámbito de GV-EJIE, la arquitectura física y por ende la arquitectura lógica de alto nivel es idéntica para cada uno de los entornos existentes, desarrollo, pruebas y producción.

En el entorno de desarrollo se realizará la primera integración de lo desarrollado en PC local a la arquitectura y el entorno tecnológico que sustentará el aplicativo. En el entorno de pruebas se realizarán las pruebas de usuario planteadas como requisitos del aplicativo como paso previo al entorno productivo. El entorno de producción será el entorno final donde la aplicación se ejecutará con los usuarios finales y los datos reales.

El paso entre los diferentes entornos se realizará mediante una sistemática de traspasos.

Plataforma multicapa

La arquitectura física impuesta por el modelo de seguridad de red define un modelo de tres capas. Dicho modelo presenta además una distribución de componentes software, distinguiéndose tres niveles lógicos con cometidos distintos: uno para los servicios de usuario, otro para los servicios de la lógica de negocio (la lógica principal de la aplicación), y otro nivel para los servicios de datos.

Estas capas están físicamente distribuidas de tal manera que únicamente poseen conectividad uno a uno, es decir, la capa de presentación solo tiene acceso a la de negocio, y ésta sólo tendrá acceso a la de datos.

• **Presentación**

La lógica de presentación es básicamente la encargada de proporcionar la interfaz necesaria para presentar la información y recoger las solicitudes de proceso de los usuarios. Es la parte de funcionalidad de la aplicación que permite al usuario interactuar con la misma.

En la capa de presentación se engloban al menos los contenidos estáticos necesarios para la aplicación (CSSs, javascript, imágenes, etc.) que se desplegarán en el servidor web Apache del contexto y entorno correspondiente.

• **Negocio o aplicación**

La lógica de negocio es el puente entre el usuario y los servicios de datos. Estos servicios de negocio responden a peticiones del usuario, y de otros servicios de negocio, implementando procedimientos y tratamientos sobre los datos que comprenden las principales funcionalidades y objetivos de la aplicación.

Como medio para la implementación y soporte de esta capa lógica, así como para el funcionamiento de toda esta arquitectura, estará el servidor de aplicaciones Oracle Weblogic Server para tecnología Java, y Microsoft IIS para tecnología .NET. El servidor proporciona los servicios necesarios para gestionar el tratamiento de los componentes de negocio desarrollados así como su control transaccional.

• **Datos o Back-end**

La capa de datos representa los servicios de datos para la aplicación. Esta capa es la responsable de proporcionar medios para recuperar y actualizar la información y mantener la integridad de datos.

La capa de datos estará sustentada en un sistema de gestión de base de datos.

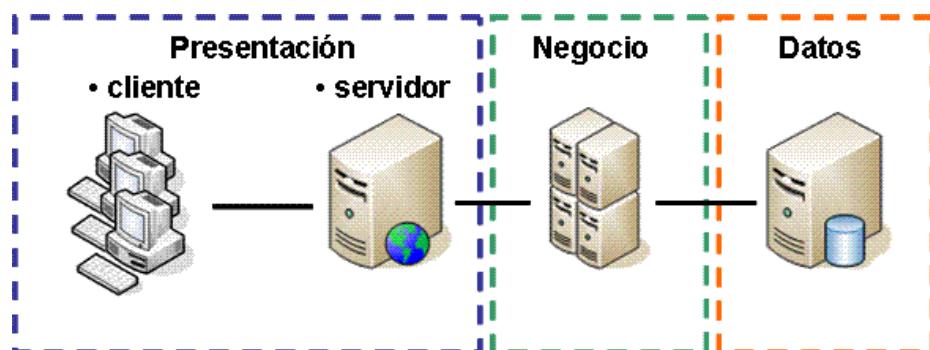
Esta capa sustentará aquellos procesos batch programados para su ejecución con Control-M y gestionados por la aplicación K31/O75

• **Integración**

Se añade una capa de integración como:

- Intermediario y agregador de funciones de interés general suministrados por distintos sistemas para su exposición y consumo
- Suministrador de servicios con formato de servicio web
- Ámbito de despliegue de procesos de negocio inter-sistemas (orquestación de servicios)
- Proveedor de adaptadores que faciliten el acceso a distintas tecnologías subyacentes
- Propuesta de ámbitos de publicación y consumo de eventos con un único punto de entrada pero con diversas tecnologías de publicación y recolección
- Acelerador de reutilización de sistemas y de interoperabilidad técnica

Todos estos componentes se distribuyen entonces en capas lógicas, pero con disposición en forma de servicios para aquellas funciones de interés general (SOA)



En el proceso de desarrollo del software en el ámbito de GV-EJIE, estos componentes *deberán* de implantarse en el entorno de desarrollo para posteriormente ir traspasándose a los entornos de pruebas y de producción. El paso de desarrollo a pruebas se hará a través de las zonas de traspaso, mientras que para el paso a producción se utilizará lo existente en las zonas de pruebas, es decir, todo paso a producción *deberá* ser transferido y validado previamente en el entorno de pruebas.

8 Anexo II. Contextos de albergue

Los sistemas software o aplicaciones se van a clasificar en los diferentes contextos de albergue según sus características de negocio.

Una aplicación podrá tener módulos en diferentes contextos en función de las necesidades existentes aunque el tratamiento para los procesos de desarrollo e implantación será como si de aplicaciones independientes se tratara.

Estos contextos de albergue se corresponden con los contextos de Internet, de Intranet y de Extranet.

- El contexto de Internet está asociado a aquellas aplicaciones que requieren un acceso de los usuarios desde Internet. Este contexto tiene connotaciones especiales en lo referente a la publicación de los contenidos estáticos y estilos.
- El contexto de Intranet está asociado a aquellas aplicaciones que requieren únicamente un acceso de los usuarios de la intranet de GV. En este contexto estará ubicada la plataforma de Integración que actúa como intermediario y agregador de funciones de interés general suministrados por distintos sistemas para su exposición y consumo.
- El contexto de Extranet está asociado a aquellas aplicaciones que requieren exponer todo o parte de su negocio a un grupo limitado y definido de usuarios apoyándose en otras redes que no se corresponden con la intranet de GV, habitualmente Internet. Actualmente existen dos tipos de extranet; la extranet de “jakinaplus” y la extranet de “Jaso”.

En lo referente a la arquitectura de los diferentes contextos, se trata de una arquitectura replicada para las capas de presentación y de negocio compartiendo la capa de Back-End. Las diferencias básicas entre los diferentes contextos de albergue se limitan, sobre todo, en los aspectos referentes a la seguridad y la conectividad.

9 Anexo III Herramientas de desarrollo

- **AWA:** Ais Web Accessibility es una barra de herramientas que ha sido desarrollada para ayudar a examinar las páginas Web de forma manual para una gran variedad de aspectos de accesibilidad.
- **WDE:** Web Developer es una extensión para el navegador Mozilla Firefox, que consta de una barra de herramientas que ha sido desarrollada para ayudar a examinar las páginas Web de forma manual contemplando una gran variedad de aspectos de accesibilidad.
- **Firebug:** Firebug es una extensión (add-on) de Firefox creada y diseñada especialmente para desarrolladores y programadores web. Es un paquete de utilidades con el que se puede analizar (revisar velocidad de carga, estructura DOM), editar, monitorizar y depurar el código fuente, CSS, HTML y JavaScript de una página web de manera instantánea e "inline".
- **BadBoy:** Permite efectuar el testeo de aplicaciones web, incluyendo una interfaz simple, fácil e intuitiva, mediante los métodos de captura y repetición, siendo una gran ayuda para la prueba de carga de gran alcance, informes detallados, gráficos, etc...
- **Selenium:** Permite efectuar pruebas funcionales de aplicaciones web, mediante la grabación de escenarios funcionales (navegación), facilitando además la automatización de pruebas contra distintas plataformas y navegadores web
- **Enterprise Architect:** Herramienta CASE para el diseño y construcción de sistemas software. Permite definir y gestionar la creación de modelos UML.
- **GIMP:** Editor de imágenes similar a Adobe Photoshop o Corel Photopaint.
- **Mantis:** Aplicación web para el registro y control de bugs.
- **TestLink:** Herramienta de gestión de las pruebas. Permite crear diferentes proyectos de pruebas gestionando para cada uno de ellos el plan de pruebas, los requisitos, casos de prueba y defectos y el análisis de los resultados.
- **Hudson:** Herramienta para la integración y automatización de tareas. Permite automatizar el despliegue y compilación de aplicaciones y la ejecución de pruebas en servidor.
- **Sonar:** Herramienta para validación de calidad del software. Ofrece un Cuadro de mando para visualización de métricas de calidad
- **Oracle SQL Developer:** Herramienta gráfica para el desarrollo en bases de datos Oracle. Permite visualizar objetos de base de datos, ejecutar sentencias SQL y scripts SQL, y editar y depurar sentencias PL/SQL. También permite ejecutar informes ya proporcionados o los creados y salvados por el usuario. SQL Developer simplifica y mejorar la productividad a la hora de desarrollar sobre bases de datos Oracle.
- **Subversion:** Sistema open-source escalable de control de versiones, muy potente, usable y flexible, que ha sido diseñado para sustituir a CVS. Para ello trata de proporcionar funcionalidades similares al CVS preservando su filosofía de desarrollo y de dar solución a los principales defectos del CVS.
- **Tortoise SVN:** Cliente gratuito de código abierto para el sistema de control de versiones Subversion. Está desarrollado bajo la Licencia Pública General GNU (GPL).
- **CollabNet Subversion Command-Line Client:** Cliente de código abierto para el sistema de control de versiones Subversion, desarrollado bajo la Licencia Pública General GNU (GPL). Aporta la posibilidad de trabajar desde línea de comandos.
- **XMLSpy:** Altova XMLSpy es el estándar en el entorno de desarrollo para el modelado, edición, depuración y transformación de todas las tecnologías XML. Por eso XMLSpy es ideal para los desarrolladores de J2EE,

.NET, Eclipse y de bases de datos que necesiten estas tecnologías.

- **CheckStyle:** Herramienta de desarrollo que ayuda a los programadores a escribir código Java adscrito a estándares de codificación establecidos, facilitando para ello la automatización del proceso de chequeo del código generado.
- **KeyToolUI:** Herramienta gráfica para la manipulación de formatos criptográficos. (Certificados, keystores, firmas, claves en diversos formatos).
- **Xolido Sgn:** Herramienta gráfica para la manipulación de formatos criptográficos. (Certificados, keystores, firmas, claves en diversos formatos).
- **JUnit:** Paquete Java utilizado para automatizar los procesos de prueba. Mediante la creación de Tests, JUnit realizará pruebas unitarias del código que indique el usuario.
- **HP Diagnostics:** herramienta utilizada para monitorizar el rendimiento y consumos de recursos de las aplicaciones en servidor.
- **JMeter:** Apache JMeter es una herramienta de carga diseñada para realizar Pruebas de Rendimiento y Pruebas Funcionales sobre Aplicaciones Web.
- **PMD:** PMD es una herramienta de auditoría y verificación de código estático, que permite detectar errores potenciales en las aplicaciones, en base a un conjunto de reglas parametrizables.
- **Subversive:** El proyecto Subversive se ocupa de facilitar la integración de Subversion para Eclipse. El plugin instalado se integra en el entorno de desarrollo y nos permite trabajar con los repositorios Subversion. Se podrán utilizar los repositorios Subversion casi de la misma manera que se utilizan los repositorios CVS mediante el plugin CVS incluido en la distribución estándar de Eclipse.
- **SoapUI:** Herramienta para testeo de servicios web. Permite la realización de tests funcionales, de carga, simulación de servicios web (mock webservices), integración con herramientas de generación de código, etc. a través de una interfaz gráfica. Está escrito en java, es multiplataforma e integrable en los entornos de desarrollo más comunes(eclipse, netbeans e IntelliJ).
- **QuickRex:** Plug-in para Eclipse cuyo objetivo es facilitar la creación de expresiones regulares.
- **NUnit:** Alternativa a JUnit para .Net. Permite realizar pruebas unitarias para cualquier lenguaje de .Net.
- **AnkhSVN:** Plug-in para Visual Studio para trabajo desde puesto local de desarrollo con el repositorio de versiones Subversion.
- **Eclipse:** Herramienta para el desarrollo de aplicaciones Java y J2EE (IDE de desarrollo).
- **OEPE:** Conjunto gratuito plug-ins que permiten a los desarrolladores de WebLogic trabajar con Java EE y Servicios Web Estándares. Oracle Enterprise Pack para Eclipse permite el desarrollo bases de datos, Java SE, Java EE, Web Services, XML, y Spring.
- **JReport:** Aplicación para la generación de informes para aplicaciones JAVA.
- **JasperReports:** Librería para la generación de informes para aplicaciones JAVA.
- **Maven:** Sistema de gestión de repositorio de librerías de aplicación y utilidades para la descarga y publicación
- **Archiva:** herramienta con interfaz gráfica para gestión de repositorios propios

10 Anexo IV. Sistemas horizontales

- **XLNETS:** Sistema general de autenticación de Gobierno Vasco basado en LDAP.
- **PLATEA:** Plataforma para la e-administración de Gobierno Vasco.
- **Control M:** Aplicación para la planificación y ejecución de procesos batch.
- **K31:** Gestor de procesos batch.
- **O75:** Interfaz web y API para interactuar con el gestor de procesos batch (K31).
- **Pasarela de pagos:** Plataforma que integra la gestión de los pagos en Gobierno Vasco.
- **NORA:** Aplicativo para la gestión actualizada de los datos de localización, incluyendo utilidades geográficas como Visor GIS y Geolocalizador
- **GIS Corporativo:** Plataforma de información geográfica y servicios GIS
- **DOKUSI (Sistema Integral de Gestión Documental):** Plataforma para la gestión de todos los documentos almacenados electrónicamente por el Gobierno Vasco.
- **SGA (Sistema de Gestión de Archivos):** Plataforma para la gestión de todos los documentos archivados por el Gobierno Vasco.
- **SMS Corporativo:** Sistema para la gestión y envío de mensajes a móviles, basado en la plataforma Latinia
- **UDA (Utilidades de desarrollo de aplicaciones):** Conjunto de utilidades, herramientas, librerías, plugins, guías, y recomendaciones funcionales y técnicas que permiten acelerar el proceso de desarrollo de sistemas software con tecnología Java.

Lectura 5. Análisis, diseño y mantenimiento del software

Análisis, Diseño y Mantenimiento del Software

Manuel Arias Calleja

Dpto. de Inteligencia Artificial - ETSI Informática - UNED



Actualizada en Noviembre de 2010

Índice general

Guía de estudio de la asignatura	V
Presentación y objetivos	V
Contexto y conocimientos previos	VI
Esquema y resumen de contenidos	VIII
Material y medios de estudio	X
1. Contexto de la asignatura en la Ingeniería de Software	1
2. Fase de especificación	5
3. Fase de diseño	7
4. Fase de implementación	9
5. Fases de pruebas	13
6. Fase de entrega y mantenimiento	15
7. Metodologías de desarrollo	17
8. Herramientas de desarrollo y validación	19
Bibliografía	21

Guía de estudio de la asignatura correspondiente al primer parcial

Autores: Manuel Arias Calleja. Basado en una versión del año 2003 por Manuel Arias Calleja y José Ramón Álvarez Sánchez.

Revisado: Noviembre de 2010 por Manuel Arias Calleja y Ángeles Manjarrés Riesco

Revisión de los elementos del UML: Ángeles Manjarrés y Manuel Arias Calleja

Revisión de los principios de orientación a objetos: Manuel Arias

Asignatura: Análisis, Diseño y Mantenimiento de Software

Código: 55-402-4 (4º curso de la titulación: “Ingeniero Informático”)

Breve descripción:

Análisis y definición de requisitos. Diseño, propiedades y mantenimiento del software.

Presentación y objetivos

El objetivo principal de la Ingeniería del software (IS) es el desarrollo y mantenimiento de software de forma sistemática y productiva, asegurando su calidad, fiabilidad y facilidad de uso. Los enfoques más comunes de la docencia de la IS, se centran en el análisis de los procesos de desarrollo y mantenimiento, así como de las técnicas integrales y de apoyo al servicio de la obtención de productos de alta calidad que satisfagan al usuario.

Esta Guía didáctica está diseñada para conducir el estudio de la asignatura en sus aspectos puramente técnicos, que serán los evaluados en las pruebas presenciales. En la Guía de Curso se detallan objetivos docentes adicionales que serán cubiertos mediante actividades NO obligatorias que los alumnos podrán seguir en los cursos virtuales a la vez que estudian el temario convencional.

Contexto y conocimientos previos

Contexto académico previo

Ingeniería del software es una materia troncal del segundo ciclo que en la titulación de Ingeniero en Informática, impartida por la ETSII de la Universidad Nacional de Educación a Distancia (Resolución de 21 de marzo de 2001, BOE 10 de abril), se ha dividido en dos asignaturas obligatorias y anuales del cuarto curso, cada una de las cuales tiene asignados 9 créditos (5 teóricos y 4 prácticos). Las denominaciones que se han dado a estas dos asignaturas complementarias son Análisis, Diseño y Mantenimiento del software (descrita como “Análisis y definición de requisitos. Diseño, propiedades y mantenimiento del software”) y Análisis y Gestión del Desarrollo del Software (descrita como “Análisis de aplicaciones. Gestión de configuraciones. Planificación y gestión de proyectos informáticos”).

Ambas asignaturas se hayan fuertemente vinculadas, en tanto que se complementan para proporcionar una visión general del proceso completo de la producción de software, tanto desde un punto de vista técnico como humano, vertientes que caracterizan todo proceso de ingeniería.

El programa de la asignatura Análisis y Gestión del Desarrollo del Software está estructurado en dos cuatrimestres. El primer cuatrimestre se dedica al Proceso Software Personal (PSP), cuyo objetivo es la adquisición de una correcta disciplina personal para el desarrollo de un software de calidad en los plazos y costes comprometidos. El segundo cuatrimestre está dedicado a la gestión global del proceso de desarrollo software en el que intervienen decenas o centenares de ingenieros. Su objetivo es también obtener un software de calidad en los plazos y costes planificados, si bien en este caso se destacan la problemática y las técnicas asociadas al trabajo en equipo. Las asignaturas de Ingeniería del software deberían, idealmente, ser cursadas en paralelo por los alumnos, dado su carácter mutuamente complementario. En definitiva, el objeto de la asignatura Análisis y Gestión del Desarrollo del Software es mostrar cómo se evalúan las técnicas de ingeniería estudiados en la asignatura Análisis, Diseño y Mantenimiento del software, y cómo se aplican en un contexto profesional.

Otras asignaturas relacionadas

Son muchas y diversas las asignaturas de la titulación relacionadas con la asignatura Análisis, Diseño y Mantenimiento del Software. El haber cursado algunas de estas asignaturas (o bien las asignaturas equivalentes de carreras impartidas en otras universidades) es requisito imprescindible para su adecuado seguimiento.

En concreto, es razonable esperar que el alumno matriculado en Ingeniería del software haya seguido previamente las asignaturas obligatorias del primer ciclo Ingeniería del software e Ingeniería del Software de Gestión, como iniciación a la materia objeto

de esta memoria (dada la extensión de los contenidos, es conveniente que el alumno parta de una visión introductoria y general de la ingeniería de software, conozca los modelos de ciclo de vida básicos y alguna metodología estructurada).

Por otro lado, el alumno sólo podrá dar pleno sentido al contenido de esta materia si ha cursado las asignaturas que se encuadran en el componente que en la sección anterior se ha etiquetado por Fundamentos de la computación. Entre tales asignaturas se cuentan las relacionadas con la teoría de la programación (Programación I, Programación II, Programación III, Estructura de Datos y Algoritmos y Lenguajes de Programación, todas ellas asignaturas obligatorias del primer ciclo), y las que estudian la teoría de la computabilidad (Teoría de autómatas I, Teoría de autómatas II). (Obviamente, la comprensión de las mencionadas asignaturas a su vez ha exigido del alumno los conocimientos matemáticos que se proporcionan en las asignaturas Lógica matemática, Análisis Matemático, Álgebra, Ampliación de Matemáticas y Matemática discreta).

Resulta asimismo altamente recomendable que el alumno haya cursado también algunas asignaturas relacionadas con los temas anteriores pero de una orientación más práctica y específica. Es el caso de las optativas del tercer curso Programación Concurrente, Programación Declarativa y Programación Orientada a la Inteligencia Artificial (orientadas a la programación), y Diseño y Evaluación de Configuraciones, y Configuración, Diseño y Gestión de Sistemas Informáticos.

Es también de interés, si bien no imprescindible para la asimilación de los contenidos, que el alumno disponga de conocimientos del campo de la Inteligencia Artificial. En particular, la asignatura Sistemas basados en conocimiento I, asignatura optativa del tercer curso, aborda el estudio de las diferentes fases del ciclo de desarrollo de un sistema experto, coincidentes en lo esencial con las fases de desarrollo de los productos software convencionales. Sin duda el alumno que haya estudiado esta asignatura dispondrá de una visión más abierta, y la reflexión sobre los paralelismos y discrepancias entre ambas disciplinas le ayudará a profundizar en la materia. De hecho, a medida que aumenta la complejidad de las aplicaciones software en dominios no tradicionalmente abordados por la Inteligencia artificial, la ingeniería del conocimiento y la IS convencional han ido eliminando progresivamente sus difusas fronteras, y comenzado a beneficiarse mutuamente de sus respectivos avances. En concreto, el uso de las técnicas OO se ha generalizado en todos los ámbitos, y las técnicas de formalización de requisitos se ven enriquecidas por los interesantes resultados obtenidos en el campo del procesamiento del lenguaje natural, tradicionalmente estudiado en la Inteligencia Artificial. El conocimiento de los principios básicos de la Inteligencia Artificial, que se estudian en la asignatura obligatoria del segundo curso Introducción a la Inteligencia Artificial, es también sin duda de utilidad en este contexto.

En el programa del 4º curso existe un grupo de asignaturas que guarda una evidente relación con la Ingeniería del software, y que resulta interesante que el alumno estudie en paralelo con las asignaturas de Ingeniería del software, tratando de establecer las

oportunas conexiones. Es particularmente el caso de la asignatura obligatoria Lógica computacional, donde se estudian los fundamentos de las técnicas formales de desarrollo software y de la asignatura Inteligencia Artificial e Ingeniería del Conocimiento, que redunda en el estudio de las técnicas de la ingeniería del conocimiento. Finalmente, es preciso destacar que los conocimientos adquiridos en estas asignaturas son esenciales para el seguimiento de una buena parte de las asignaturas del quinto curso, en particular, de las obligatorias Sistemas Informáticos I, Sistemas Informáticos II y Sistemas Informáticos III, más las optativas Aprendizaje y Personalización del Software, Diseño de Sistemas de Trabajo Cooperativo (CSCW) y Calidad de Software; y sirven de introducción a las asignaturas optativas Sistemas en tiempo real, Seguridad en las comunicaciones y en la información y Sistemas de comunicación de datos. Asimismo son obviamente indispensables para la realización del Proyecto de Fin de Carrera, ejercicio de desarrollo de un proyecto informático que debe llevarse a cabo con el rigor de un proyecto real, y que es obligatorio para la obtención del título. Por supuesto será fundamental para la formación posterior del alumno titulado y para el ejercicio de su práctica profesional.

Esquema y resumen de contenidos

A continuación exponemos un resumen de los temas que componen el contenido de la asignatura que se expandirá más adelante. Estos temas se podrían agrupar en tres partes que se corresponden con los objetivos presentados anteriormente:

- Parte I: Introducción. Tema 1.
- Parte II: Fases de Construcción. Temas 2 al 6.
- Parte III: Metodologías y Herramientas. Temas 7 y 8.

La primera parte es preparatoria e incluye la introducción y ubicación de los elementos que van a conformar la asignatura. En la segunda parte se van describiendo las distintas fases del desarrollo y mantenimiento del software. La parte final incluye un conjunto de metodologías donde se recopilan y organizan de diferentes formas las fases, junto con algunas herramientas de desarrollo.

Esta asignatura es anual y por lo tanto se divide en dos parciales. A los efectos de exámenes parciales se considera que los temas del 1 al 4 pertenecen al primer parcial y los temas del 5 al 8 pertenecen al segundo parcial.

Esquema:

- Tema 1: Contexto de la Asignatura en la IS

- Necesidad de una metodología
- Ciclo de vida del software
- Notaciones de especificación y diseño
- Tema 2: Fase de Requisitos
 - Obtención de requisitos
 - Análisis de requisitos
 - Representación de requisitos
 - Validación de requisitos
 - Bases de documentación
- Tema 3: Fase de Diseño
 - Conceptos y elementos del diseño
 - Métodos de diseño
 - Validación y confirmación del diseño
 - Documentación: especificación del diseño
- Tema 4: Fase de Implementación
 - Iteración de pruebas y depuración
 - Manuales técnico y de usuario
- Tema 5: Fases de Pruebas
 - Verificación y validación a lo largo del ciclo de vida
 - Técnicas y métodos de prueba
 - Documentación de pruebas
- Tema 6: Fase de Entrega y Mantenimiento
 - Finalización del proyecto
 - Planificación de revisiones y organización del mantenimiento
 - Recopilación y organización de documentación
- Tema 7: Metodologías de Desarrollo
 - Proceso Unificado de Rational
 - Método Extreme Programming
 - Métodos de software libre: “catedral” vs. “bazar”

- Tema 8: Herramientas de Desarrollo y Validación
 - Herramientas CASE
 - CVS (Concurrent Versioning System)
 - Entornos de desarrollo de interfaces

Material y medios de estudio

Estructura de esta Guía Didáctica y libro base

En los siguientes capítulos de esta guía de estudio se detallan los contenidos con la información que el alumno de educación a distancia necesita para poder estudiar la asignatura. Al principio de cada capítulo se incluye un “*Tutorial previo*” con los elementos que describen el contexto, conocimiento previo, objetivos y guía de estudio para ese capítulo. En esta asignatura se utilizará como material básico de estudio para el curso 2010-2011 el libro [Pre05] o bien con la edición siguiente [Pre10]:

- Roger S. Pressman. *Ingeniería de Software: un Enfoque Práctico*. McGraw-Hill, 2005
- Roger S. Pressman. *Ingeniería de Software: un Enfoque Práctico*. McGraw-Hill, 2010

Este libro base cubre la mayor parte del temario. En esta guía los contenidos de cada capítulo se sustituirán por la referencia (entre corchetes como [Pre05, sec. ... y cap ...]) a los apartados del libro base, o bien se incluirán desarrollados directamente en esta guía si no existe una correspondencia directa con el libro base. Se adjuntará la referencia a los contenidos correspondientes a la séptima edición como [Pre10, sec. ... y cap ...]. Al final de cada capítulo se incluye en esta guía un “*Tutorial posterior*” que contiene ejercicios o actividades propuestas adicionales a las que aparecen en el libro base, para que el alumno pueda comprobar si ha logrado los objetivos del capítulo, y también información adicional para la extensión de conocimientos para los alumnos interesados en profundizar en el tema, junto con referencias alternativas sobre los mismos contenidos. Al final de esta guía didáctica se incluye en un apéndice el glosario de términos habituales en la Ingeniería de Software que pueden aparecer en el contenido¹, también se incluye una recopilación de referencias bibliográficas (recomendadas o referidas en el material), más un índice alfabético de referencia para conceptos y términos que aparecen en el material.

¹ Además al final del libro base [Pre05] o [Pre10] hay un apéndice que recopila todas las siglas en inglés y castellano usadas profusamente en Ingeniería del Software.

Medios Adicionales

Adicionalmente a esta guía, el alumno dispondrá de los medios de comunicación habituales con su Profesor Tutor en el Centro Asociado o a través de las Tutorías Telemáticas (o Cursos Virtuales) de la UNED <http://virtual.uned.es/>, y también con el Equipo Docente en la Sede Central (en las direcciones, teléfonos y horarios indicados en la Guía de Curso). Esto se complementa con los canales de comunicación y recopilación de información tanto en soporte físico (CDROM) como en línea a través de la página de Internet de la asignatura en la UNED <http://www.ii.uned.es/superior/cuarto/ADMSoft.html>. En todos estos medios se incluirá la información particular de referencias y contenidos que se detallan en los capítulos de esta guía, con la ventaja adicional de que en los medios en línea los enlaces de direcciones en Internet y otros materiales se irán ampliando y actualizando más frecuentemente. Se recomienda encarecidamente el seguimiento de los cursos virtuales. Además del libro base (que contiene al final de cada capítulo “otras lecturas y fuentes de información”) y del material incluido en esta guía, también se recomiendan como bibliografía complementaria general los libros [Som98] (o la edición en inglés más reciente [Som01]) y [RBP⁺96]:

- Ian Sommerville. *Ingeniería de Software*. Addison-Wesley Iberoamericana, 1998
- James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Modelado y diseño orientado a objetos. Metodología OMT y OMT II*. Prentice Hall, 1996

Evaluación

La evaluación oficial de la asignatura se hará por medio de las pruebas presenciales habituales de la UNED. Se harán dos pruebas parciales, una para cada cuatrimestre. Las pruebas subjetivas consistirán en una parte de preguntas teóricas breves sobre aspectos relevantes del temario correspondiente, más posiblemente otra parte práctica compuesta de ejercicios con supuestos prácticos que describirán parcialmente un problema de diseño de software sobre el cual se pedirá al alumno completar o extender algunos aspectos relacionados con el temario correspondiente. La puntuación correspondiente a cada pregunta se especificará en el enunciado. En la nota final se tendrá en cuenta la compensación entre preguntas dentro de un mismo examen parcial así como la compensación de ambos parciales. Los parciales compensan a partir de 4 y se guardan hasta septiembre.

En algunos capítulos puede haber propuestas para realizar prácticas por el propio alumno, estas serán totalmente voluntarias (y que por tanto no podrán ser tenidas en cuenta para la puntuación de la nota final), pero se recomienda al alumno su realización para ayudarle a una mejor comprensión de los temas.

Capítulo 1

Contexto de la asignatura en la Ingeniería de Software

Tutorial previo

Introducción

Al inicio de la asignatura es conveniente repasar algunos conceptos generales sobre Ingeniería de Software e introducir los temas que se van a estudiar.

En el desarrollo de sistemas software es necesario planificar el uso de recursos y temporizar adecuadamente las diferentes actividades para no sobrepasar los límites económicos y de tiempo dedicados a un proyecto. Es principalmente con este objetivo que se han ido desarrollando el conjunto de técnicas que reciben el nombre de Ingeniería del Software. Gracias a estas técnicas el desarrollo del software se sistematiza en la medida de lo posible, de modo que el resultado sea previsiblemente aceptable, es decir, cumpla las expectativas planteadas en términos de tiempo de desarrollo, precio final, mantenibilidad y eficiencia.

Una metodología de desarrollo reúne un conjunto de prácticas que se ejercen en las diferentes fases del ciclo de vida de un producto software. Esas fases son: especificación, diseño, codificación, pruebas y mantenimiento. Para dar soporte a las diferentes etapas, y particularmente a las etapas de análisis y diseño, se han definido notaciones de modelado tales como el difundido estándar UML.

En este primer capítulo se muestra la utilidad de seguir una metodología en el desarrollo de software. A continuación se presentan los aspectos más generales del ciclo de vida del software y se introducen las distintas fases del desarrollo en que se profundizará en los capítulos del bloque II (temas 2 al 6). Finalmente, se describe el lenguaje UML, de uso extendido y soportado actualmente por varias herramientas CASE capaces de generar código a partir de los diagramas de esta notación.

CAPÍTULO 1. CONTEXTO DE LA ASIGNATURA EN LA INGENIERÍA DE SOFTWARE

Relación con otros temas

Con este tema se pretende asegurar que el alumno dispone de base suficiente para abordar el estudio del resto de la asignatura. Es recomendable el repaso de las asignaturas previas de la titulación en que se estudiaron las bases de la Ingeniería del Software y el desarrollo de programas.

Al mismo tiempo este tema es importante por proporcionar una visión general de las diferentes fases de un ciclo de desarrollo de software y de su coordinación en metodologías.

Para hacerse una composición de lugar sobre lo que se habla en este capítulo y en la asignatura en general es recomendable haber desarrollado algún proyecto en un lenguaje de programación.

Objetivos del tema

Este tema es introductorio. El alumno debe comprobar que dispone de los conocimientos básicos necesarios para afrontar el resto de la asignatura, entendiendo la estructura general del desarrollo del software, así como las implicaciones y relaciones con la planificación y gestión de proyectos informáticos. Para ello debe: comprender la necesidad de utilizar una metodología en el desarrollo de aplicaciones, comprender las ventajas e inconvenientes de los diferentes ciclos de vida para discernir cuando es adecuado usar uno u otro, y saber utilizar (en un nivel medio) la notación UML para la especificación y diseño de sistemas.

Guía de estudio y esquema

Es conveniente realizar una lectura inicial del tema para identificar sus contenidos y repasar los conceptos que considere necesarios, bien acudiendo a textos de asignaturas ya cursadas, bien estudiando las referencias proporcionadas en el apartado de “extensión de conocimientos”. El capítulo tiene tres partes:

- La explicación acerca de la necesidad de una metodología se incluye directamente en la adenda y, adicionalmente, se debe estudiar en el libro base [Pre05, cap. 1, y secs. 2.1 y 2.2]. En la séptima edición [Pre10, cap. 1] excluyendo la sección 1.2 y [Pre10, cap. 2, sec. 2.1 hasta 2.1.2] inclusive.
- Los ciclos de vida. Se da una descripción de cada uno y una lista de ventajas e inconvenientes. También este apartado está en la adenda y se debe extender su estudio en el libro base [Pre05, secs. 3.1 a 3.5.3 y 3.7]. En la séptima edición [Pre10, secs. 2.3, 2.4 y 2.9].
- La notación de especificación y diseño UML. La mejor forma de asimilar esta parte consiste en estudiar la teoría e ir haciendo pequeños ejemplos. En la adenda

CAPÍTULO 1. CONTEXTO DE LA ASIGNATURA EN LA INGENIERÍA DE SOFTWARE

se incluye este apartado. Además de los ejemplos de la adenda se pueden consultar los que aparecen en el libro base en [Pre05, secs. 7.5 y 8.3 a 8.8]. En la séptima edición [Pre10, Apéndice 1].

De los apartados anteriores nos interesan fundamentalmente los ejemplos; la teoría de dichos apartados se referencia en el siguiente capítulo de esta guía. Advertimos que este material de estudio del UML se refiere a la primera versión del lenguaje. Hemos creído conveniente formar al alumno en esta primera versión debido a que aún la segunda es relativamente reciente y es de prever que ambas versiones convivan en el mundo empresarial durante un tiempo significativo. (Las novedades introducidas en la segunda versión se estudiarán en una adenda a esta guía que se publicará el primero de noviembre en la Web de la asignatura).

**CAPÍTULO 1. CONTEXTO DE LA ASIGNATURA EN LA INGENIERÍA DE
SOFTWARE**

Capítulo 2

Fase de especificación

Tutorial previo

Introducción

Una vez que el proyecto ha superado tanto la decisión de desarrollarlo como su estudio de viabilidad (estudiado en otras asignaturas) empieza la fase de especificación, donde se emplean un conjunto de técnicas para captar requisitos y representarlos de un modo útil para la fase posterior de diseño.

Se dice que la especificación es la fase importante mientras que el diseño es la difícil. La importancia se debe al alto coste económico y de tiempo que supone un requisito mal entendido. La realización de una buena especificación no es tan sencillo como se puede pensar en un principio, pues muchas veces el propio cliente no tiene una imagen clara del sistema final o surgen nuevas necesidades a mitad del desarrollo. Este problema puede afrontarse de varias formas: usando técnicas de comunicación efectivas, estudiando el dominio del problema, creando modelos del problema real y, por último, revisando la especificación creada.

Se deben documentar y registrar debidamente todas las actividades anteriores para que en caso de que surja algún problema se pueda seguir su traza hasta los requisitos.

Relación con otros temas

La extracción, modelado, análisis y representación de requisitos o especificaciones es un problema muy relacionado con la Ingeniería del Conocimiento, por tanto sería conveniente que el alumno repasara los temas correspondientes en las asignaturas previas relacionadas. Es necesario ejercitarse la capacidad de abstracción para poder expresar lo que pide el cliente en una representación formal que responda a sus expectativas.

Objetivos del tema

Es necesario en esta fase separar y entender los conceptos propios de cómo especificar un problema y cómo hacer útil esas especificaciones para posteriores fases del desarrollo. El alumno debe ser capaz de extraer y representar un conjunto de requisitos expresados en lenguaje natural a una representación que sirva de entrada a la fase siguiente de diseño.

Guía de estudio y esquema

En el tema se exponen los contenidos teóricos genéricos. El alumno debe identificar cuáles son los elementos correspondientes a la obtención y el análisis de los requisitos en esos ejemplos para generalizarlo a otros problemas.

1. Obtención de requisitos: se estudia directamente en el material en esta guía. Se deben extender conocimientos en [Pre05, secs. 7.1 a 7.4]. En la séptima edición [Pre10, secs. 5.1 a 5.3].
 2. Construcción del modelo de análisis y negociación de requisitos: este apartado se puede estudiar en [Pre05, secs. 7.6 y 7.7]. En la séptima edición [Pre10, secs. 5.5 y 5.6].
 3. Modelado del análisis y de datos: el contenido se estudiará en [Pre05, secs. 8.1 a 8.3]. En la séptima edición [Pre10, secs. 6.1 y 6.4].
 4. Análisis orientado a objetos: el contenido se estudiará en [Pre05, sec. 8.4]. En la séptima edición no existe explícitamente, pero se puede consultar el punto 2.4 de la adenda.
 5. Modelado basado en escenarios: el contenido se estudiará en [Pre05, sec. 8.5]. En la séptima edición [Pre10, secs. 6.2].
 6. Modelado de clases: el contenido se estudiará en [Pre05, sec. 8.7]. En la séptima edición [Pre10, sec. 6.5].
 7. Modelado del comportamiento: el contenido se estudiará en [Pre05, sec. 8.8]. En la séptima edición [Pre10, sec. 7.3].
 8. Análisis estructurado (en el libro se denomina “Modelado orientado al flujo”): el contenido se estudiará en [Pre05, sec. 8.6]. En la séptima edición [Pre10, sec. 7.1 y 7.2].
 9. Validación de requisitos: en [Pre05, sec. 7.8] junto con el apartado correspondiente en esta guía. En la séptima edición [Pre10, sec. 5.7].
 10. Bases de documentación: en el contenido del apartado correspondiente de esta guía.
-

Capítulo 3

Fase de diseño

Tutorial Previo

Introducción

Una vez que se han identificado los requisitos para el problema, es necesario idear y componer la forma de la solución para el problema. El diseño es la fase en la que se estudia el problema, se identifican las características que tendría una solución y se analiza a su vez cada una de esas características. En la fase de diseño es más efectivo utilizar patrones y modelos conocidos para ir resolviendo partes del problema, es decir modularizar el problema y proyectarlo en módulos en la solución. Aunque, el proceso de diseño es en gran medida *ad hoc*, es decir, no está tan formalizado como las otras fases y por tanto se apoya bastante en la experiencia e intuición de los diseñadores.

En este tema se van a proponer las formas de diseño convencionales, una comparación de los más utilizados y la validación o comprobación de su adecuación, junto con la parte correspondiente de documentación de todo el proceso. Principalmente hay dos tipos de diseño:

- Diseño funcional: parte de una vista de alto nivel que se va refinando progresivamente. En este caso la atención se focaliza en la forma de procesar los datos.
- Diseño orientado a objetos: se entiende el sistema como un conjunto de objetos. Para distinguirlos se identifican los tipos de datos, sus operaciones y atributos asociados. Los objetos se comunican entre ellos enviándose mensajes.

La validación del diseño comprueba si se siguen las especificaciones del diseño y se cumplen requisitos de calidad. La mejor forma de redactar la documentación consiste en seguir una plantilla de un documento estándar rellenando varias secciones.

Relación con otros temas

Para la comprensión de este tema es conveniente tener en mente los diferentes modelos de programación que han ido estudiando en la carrera y que se utilizarán después en la fase del tema siguiente. También es necesario haber preparado el tema de requisitos para entender cuáles son los elementos de entrada en el diseño. Es necesario saber: descomponer un problema en otros más pequeños, identificar las estructuras de datos necesarias e identificar flujos de datos entre subsistemas y cuáles son las entradas y salidas necesarias para cada subsistema.

Objetivos del tema

Se deben obtener los conocimientos sobre las técnicas de diseño más efectivas y utilizadas, junto con la comprensión de las diferencias entre esas técnicas que permitan conocer cuál es el método más apropiado en cada situación. Se pretende en este tema introducir los dos tipos de métodos que existen de diseñar un sistema y proporcionar un método genérico para redactar la documentación.

Guía de estudio y esquema

A la vez que se estudia el contenido teórico del tema es conveniente ir siguiendo los ejemplos propuestos y realizar algunos ejercicios simples con otros problemas, utilizando los problemas supuestos por el alumno como ejercicios en el tema 2.

Como se ha dicho antes, la tarea de diseño es en gran medida *ad hoc* y depende de la experiencia de los diseñadores. El mejor método para aprender a diseñar es realizar todos los ejercicios propuestos (tanto explícitos, en el tutorial posterior, como implícitos o genéricos mencionados anteriormente).

- Conceptos generales de diseño: este apartado se estudiará en [Pre05, cap. 9]. En la séptima edición [Pre10, cap. 8], excluyendo las secciones 8.2.2 y 8.3.9. Se incluye en la adenda una pequeña introducción a los patrones y al diseño orientado a objetos, también se puede leer [Pre10, sec. 12.1 y 12.2], pero tan sólo por encima.
- Diseño arquitectónico. Este apartado se estudiará en [Pre05, cap. 10]. En la séptima edición [Pre10, cap. 9]. La sección [Pre05, sec. 10.6] trata el diseño estructurado. En la séptima edición [Pre10, sec. 9.6].
- Diseño al nivel de componentes. Este apartado se estudiará en [Pre05, cap. 11]. En la séptima edición [Pre10, cap. 10], excluyendo [Pre10, secs. 10.4 y 10.6].
- Validación y confirmación del diseño. Este apartado se incluye en la adenda.

Capítulo 4

Fase de implementación

Tutorial Previo

Introducción

Una vez que se dispone de un diseño para la solución del problema, incluso en una fase temprana o provisional del diseño, ya se puede comenzar a plasmar ese diseño en el código que permita realizarlo o implementarlo. En esta fase el programador recibe las especificaciones del diseño y transforma esas especificaciones, que pueden estar en formatos mezclados formales, semiformales o manuales, en un programa o módulo que las efectúe. Esta tarea de modificación puede estar semi-automatizada en algunos casos o realizarse de forma completamente manual. En cualquier caso se requiere que esa transformación se haga de manera sistemática y coherente. Las particularidades propias de lenguajes de programación concretos se estudian en otras asignaturas y por tanto no se incluirán en este tema.

Durante la implementación se escribe el código de la aplicación. Existen una serie de "vicios" en el estilo de codificación que pueden tener como consecuencia que el código sea confuso para terceras personas o para uno mismo pasado un tiempo y en consecuencia difícil de mantener; además, algunas prácticas pueden inducir errores. Es especialmente recomendable para esta parte seguir las recomendaciones del PSP (*Personal Software Process*). Para realizar el trabajo este debe ser dividido en pequeños módulos que se reparte entre individuos o pequeños grupos atendiendo a un gráfico de dependencias entre tareas y con la idea en mente de paralelizar tanto trabajo como sea posible. A medida que se van haciendo los módulos hay que comprobar que cumplen con una cierta calidad, para ello existen varios tipos de pruebas. En este capítulo se estudia la necesidad de comprobar la ejecución correcta del módulo, por tanto interesan las pruebas que se hacen a nivel de módulo, también llamadas pruebas unitarias. Además

de todo ello hay dos tipos de manuales importantes que hay que producir en esta etapa: el manual de usuario y el manual técnico.

Relación con otros temas

En este capítulo el alumno debe recordar los conocimientos sobre programación que ha ido adquiriendo a lo largo del estudio de otras asignaturas en la carrera para entender y generalizar la idea de los estilos de codificación y la depuración. Para comprender la problemática asociada a la programación es necesario conocer muy bien al menos un lenguaje de programación y tener alguna experiencia con él. Se puede considerar que con las prácticas de programación de cursos pasados es suficiente.

Objetivos del tema

Los objetivos de este tema son resaltar la importancia de que la codificación de los programas se haga de una forma ordenada, sistemática y documentada, así como de la necesidad de realizar pruebas y depuración de errores propios de la implementación. Se busca aprender algunas recomendaciones para producir código de calidad, poder entregar el código en tiempo mínimo, poder dividir el trabajo resultante del diseño entre un grupo de personas y tener una plantilla para redactar documentación para los manuales de usuario y técnico.

Guía de estudio y esquema

Después del estudio del contenido teórico del tema, se deben realizar pruebas de codificación con distintos estilos basándose en los mismos ejemplos mostrados, utilizando el lenguaje de programación que mejor conozca o prefiera el alumno. En paralelo con el estudio del tema podría ser de interés que se desarrollara un pequeño proyecto de programación donde se pusieran en práctica los conceptos desarrollados en el capítulo.

Para conseguir los objetivos es necesario no sólo memorizar el contenido del capítulo, sino desarrollar hábitos que permitan utilizarlo en la práctica de la programación, lo cual es una tarea que requiere constancia. La redacción de una documentación de calidad no es trivial, son necesarias aptitudes pedagógicas, de estilo de escritura, etc. De todas formas, aunque puede resultar difícil las primeras veces, se puede considerar una tarea “burocrática”, es decir, al final lo que hay que hacer es seguir un manual de redacción de manuales.

- Técnicas de depuración. El contenido de este apartado se encuentra en [Pre05, sec. 13.7]. En la séptima edición [Pre10, 17.8].

CAPÍTULO 4. FASE DE IMPLEMENTACIÓN

- Documentación del código. El contenido se encuentra en la adenda.

Capítulo 5

Fases de pruebas

Tutorial Previo

Introducción

Este tema incluye en su denominación “Fases”, en plural, debido a que realmente no hay una única fase de pruebas, sino que las pruebas se van realizando en todas las demás fases. Las pruebas en este caso consisten en la comprobación de que la salida obtenida en cada fase corresponde a las especificaciones de entrada correspondientes. Las pruebas consumen mucho tiempo, pero deben hacerse de un modo sistemático para asegurar que el resultado cumple con el grado de calidad exigido (densidad de errores por KLDC, etc). Para medir esta calidad existen algunas métricas. En esta parte del desarrollo se trata de encontrar errores, no sólo de codificación, sino también los relativos a la especificación o el diseño, en este sentido se puede distinguir entre verificación y validación. La *verificación* trata de comprobar si se está construyendo el producto correctamente, la *validación* si es el producto correcto. Las pruebas que se van haciendo durante el ciclo de vida son: ingeniería del sistema (prueba del sistema), especificación (prueba de validación), diseño (prueba de integración) y codificación (prueba de unidad). Los tipos de pruebas tienen naturaleza diferente y en consecuencia, las técnicas para cada una de ellas son diferentes; también se hará un recorrido por cada una de ellas. Inevitablemente también hay que añadir la correspondiente documentación de las pruebas realizadas.

Relación con otros temas

Este tema está muy relacionado con los anteriores temas 3 al 5 correspondientes a las fases de requisitos, diseño e implementación, donde se deben distribuir las pruebas correspondientes. Es recomendable, aunque no necesario, haber construido alguna vez un módulo que pruebe a otro dando entradas y comprobando si las salidas que proporciona el módulo probado se corresponden con lo esperado.

Objetivos del tema

Se debe extraer una idea clara de que los principios de ingeniería requieren la comprobación y verificación de todos los elementos intermedios en el proceso de desarrollo, en este caso, del software. También se deben conocer técnicas que indiquen los errores que se comenten, tanto de concepción de la tarea a realizar como de las funcionalidades que se implementen y que esta detección ocurra lo antes posible.

Guía de estudio y esquema

Es conveniente recapitular y repasar los temas anteriores 3 al 5 para ir identificando dentro de cada una de las fases cuáles son las pruebas necesarias para la validación que se explican en este capítulo.

- Estrategia, técnicas y métodos de prueba clásicos. Este apartado se debe estudiar en [Pre05, secs. 13.1 a 13.6 y 14.1 a 14.6]. En la séptima edición [Pre10, secs. 17.1 a 17.7 y 18.1 a 18.6].
- Pruebas orientadas a objetos. Este apartado se debe estudiar en [Pre05, secs. 14.7 a 14.9]. En la séptima edición [Pre10, cap. 19].
- Pruebas de entornos especializados. Este apartado se debe estudiar en [Pre05, sec. 14.10]. En la séptima edición [Pre10, cap. 18.8].
- Patrones de prueba. Este apartado se debe estudiar en [Pre05, sec. 14.11]. En la séptima edición [Pre10, sec. 18.9].
- Documentación de pruebas. El material está incluido directamente en la adenda.

Capítulo 6

Fase de entrega y mantenimiento

Tutorial Previo

Introducción

Como etapa final en el ciclo de vida del software se debe realizar la entrega de la primera versión al cliente y considerar las posibles modificaciones posteriores de mantenimiento. Dentro del mantenimiento se deben incluir no sólo las correcciones de errores detectados posteriormente por el cliente, sino también las modificaciones necesarias para la actualización, e incluso las peticiones de cambios por parte del cliente. Una vez que se entrega el producto, no ha acabado el trabajo, en realidad, es cuando empieza (y de hecho, existen organizaciones que viven de ello, por ejemplo las que dan soporte para GNU/Linux y para otras aplicaciones de libre distribución). Existen varios tipos de mantenimiento, corregir errores es uno de ellos, otros son adaptar el sistema a nuevos entornos o para proporcionarle nuevas funcionalidades. Es interesante medir el esfuerzo que se gasta en mantenimiento, para lo que también existen sus correspondientes métricas.

La documentación describe el sistema desde la especificación de requisitos hasta la aceptación por parte del usuario. Esta información debe estar estructurada y ser legible. Existen herramientas CASE que automatizan esta parte (hasta donde es posible).

Relación con otros temas

Este tema presenta los elementos finales del ciclo de vida del software y está relacionado con la planificación general del mismo que se presentó en el primer tema y con las fases de desarrollo en los anteriores.

Objetivos del tema

Determinar cuál es la finalización del desarrollo del software y la extensión de la vida del software desarrollado. Comprender la problemática asociada a mantener una aplicación. Dar una guía de genérica de como realizar el mantenimiento y dar estrategias viables para afrontarlo.

Guía de estudio y esquema

El contenido del tema es principalmente teórico por lo que la metodología está más orientada al estudio de los contenidos y la reflexión sobre los ejemplos.

- Finalización del proyecto. Este material se incluye directamente en la agenda.
- Reingeniería. El material correspondiente a este apartado está en [Pre05, cap. 31]. En la séptima edición [Pre10, cap. 29] exceptuando las secciones 29.1 y 29.2.
- Recopilación y organización de documentación. Este apartado está incluido en esta guía didáctica.

Capítulo 7

Metodologías de desarrollo

Tutorial Previo

Introducción

Una vez que hemos visto las fases más habituales del proceso de análisis, diseño y mantenimiento del software, es necesario estudiar algunas formas de agrupar, organizar, secuenciar y distribuir temporalmente las tareas estudiadas, según los detalles de diferentes metodologías. Una metodología constituye, en definitiva, el manual o guía que realmente se pone en práctica al abordar la construcción de un sistema. Las metodologías de desarrollo puede decirse que consisten en “poner orden” en todo lo que se ha ido viendo hasta ahora, es decir, utilizan un ciclo de vida determinado y siguen las fases de especificación, diseño, etc. de un modo concreto; algunas incluso están apoyadas por herramientas hechas a medida (por ejemplo el método Rational).

El tipo de metodologías que se van a ver están orientadas a objetos que son del tipo que demanda el mercado actualmente y además dan buenos resultados. Se estudia en primer lugar el “Proceso Unificado de Rational” por su amplia difusión y consideración de metodología tradicional. A continuación se presenta una metodología alternativa muy reciente, llamada “Extreme Programming”, que tiene características muy interesantes. A continuación se presenta la metodología de planificación, desarrollo y mantenimiento de sistemas de información del Ministerio de las Administraciones Públicas denominada Métrica (versión 3). Finalmente se hace una aproximación hacia nuevos enfoques de desarrollo de software surgidos a raíz del movimiento del Software Libre.

Relación con otros temas

Es necesario que el alumno haya estudiado previamente los temas 3 al 7, para que comprenda cuáles son los elementos que conforman todo el proceso de desarrollo del

CAPÍTULO 7. METODOLOGÍAS DE DESARROLLO

software. En este tema el alumno podrá retomar también los conceptos globales que se dieron en el primer tema donde encajan las diferentes fases.

Objetivos del tema

Es necesario que el alumno entienda las relaciones y organizaciones posibles entre las diferentes fases del desarrollo del software que dependen del tipo de entorno y de la aplicación que se desarrolla. En este tema solamente se detallan unas cuantas ilustrativas. Se trata de aprender métodos actuales, realistas y prácticos de como construir un sistema desde su concepción hasta su mantenimiento.

Guía de estudio y esquema

Este capítulo es principalmente teórico, pero tiene una vertiente práctica (ver apartado de actividades) que se recomienda realizar, al menos de forma simulada imaginando los diferentes escenarios, ya que al igual que ocurría en el capítulo dedicado a la implementación, la única manera de asimilar realmente este capítulo es realizar un proyecto donde se ponga en práctica alguna metodología. No se debe pensar que se domina realmente una metodología hasta que se ha puesto en práctica dentro de un **grupo** de desarrollo. El problema que tiene el aprendizaje de una metodología es que no se puede afrontar fácilmente como un esfuerzo individual (como con el PSP).

- Proceso unificado de Rational. El material correspondiente es un resumen de esta metodología en la adenda.
- Método “Extreme Programming”. Incluido en la adenda. Adicionalmente se puede leer [Pre05, cap. 4]. En la séptima edición [Pre10, cap. 3].
- Métrica 3. El material correspondiente es un resumen de esta metodología en la adenda.
- Métodos de software libre: “catedral” vs. “bazar”. Incluido en la adenda.

Capítulo 8

Herramientas de desarrollo y validación

Tutorial Previo

Introducción

Existen varias herramientas informáticas que facilitan las técnicas de la ingeniería del software en diferentes aspectos. En este tema se estudian en primer lugar las herramientas CASE, posteriormente veremos una herramienta muy genérica para el desarrollo de versiones de ficheros de forma concurrente (CVS) y finalmente algunos entornos genéricos de programación.

En el mercado hay varios tipos de herramientas CASE (Computer Aided Software Engineering) para múltiples propósitos: planificación de proyectos, herramientas de análisis y diseño, de documentación, etc. Algunas sólo tratan de un tema concreto y otras abarcan todas las fases de una metodología.

CVS es un sistema de almacén de ficheros (*repository*) centralizado en un servidor. El propósito de introducir este apartado aquí es dar a conocer una herramienta que se usa en el control de configuración. Veremos también la herramienta Subversion.

Por otra parte, hoy en día entre el 50 y el 60 por ciento de las líneas de código de una aplicación son relativas a la interfaz de usuario, es lógico por tanto que existan algunas herramientas dedicadas solo a este fin. Las herramientas de desarrollo de interfaces son en realidad herramientas CASE especializadas.

Relación con otros temas

Aquí conviene recordar tanto las diferentes fases del desarrollo que se han estudiado en los temas 3 al 7, puesto que esas fases reflejarán la especificidad de las herramientas.

De la misma forma algunos entornos estarán influidos por alguna metodología concreta de las estudiadas en el tema 8.

Para entender CVS es conveniente conocer un poco el sistema UNIX y tener algunas nociones de teleinformática. Para manejar una herramientas CASE, sobre todo si está pensada para una metodología concreta como por ejemplo Rational Rose, es necesario conocer esa metodología y es imprescindible conocer el UML.

Objetivos del tema

Es necesario conocer los diferentes elementos y posibilidades que proporcionan los entornos y herramientas informáticas para ayudar precisamente en la construcción de aplicaciones informáticas. Se pretende poder manejar un conjunto básico de comandos de CVS. Dar una introducción a las herramientas CASE y de construcción de interfaces.

Guía de estudio y esquema

En este tema se hace una revisión somera de algunas de los tipos de herramientas existentes. Es conveniente hacer una primera lectura directa, para después hacer pruebas con algunas de las herramientas que estén disponibles para el alumno, que de esta forma se puede hacer una idea más clara de cuáles son las posibilidades que aporta y cuáles son los ámbitos de aplicación de cada una. Al ser este capítulo de contenido eminentemente práctico, es recomendable probar o “jugar” con las herramientas recomendadas en el mismo para hacerse una idea de lo que se está hablando.

- Herramientas CASE. El material de este apartado está incluido en la adenda.
- Gestión de la configuración. El material de este apartado está incluido en la adenda.
- Entornos de desarrollo de interfaces. Este apartado está incluido directamente en la adenda y también procede mayormente de la documentación respectiva de los entornos.

Bibliografía

- [FW94] Neville J. Ford and Mark Woodroffe. *Introducing Software Engineering*. Prentice-Hall, 1994.
- [Pre05] Roger S. Pressman. *Ingeniería de Software: un Enfoque Práctico*. McGraw-Hill, 2005.
- [Pre10] Roger S. Pressman. *Ingeniería de Software: un Enfoque Práctico*. McGraw-Hill, 2010.
- [RBP⁺96] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Modelado y diseño orientado a objetos. Metodología OMT y OMT II*. Prentice Hall, 1996.
- [Sch01] Stephen R. Schach. *Object oriented and classical software engineering*. Mc GrawHill, 2001.
- [Som98] Ian Sommerville. *Ingeniería de Software*. Addison-Wesley Iberoamericana, 1998.
- [Som01] Ian Sommerville. *Software Engineering*. Addison-Wesley, 2001.

Lectura 6. Metodologías para desarrollar software seguro

Metodologías para desarrollar software seguro

Carlos Joaquín Brito Abundis
Universidad Autónoma de Zacatecas
carlosbreeto@gmail.com

Resumen: La seguridad ha pasado de ser un requerimiento no funcional, que podía implementarse como parte de la calidad del software a un elemento primordial de cualquier aplicación. Los hackers y grupos criminales evolucionan día a día y se han convertido expertos en explotar las vulnerabilidades de las aplicaciones y sitios en internet. Para hacer frente a estas amenazas, es necesaria la implementación de metodologías que contemplen en su proceso de desarrollo de software la eliminación de vulnerabilidades y la inclusión de la seguridad como un elemento básico en la arquitectura de cualquier producto de software. Este trabajo revisa algunas de las metodologías que contemplan la seguridad en su proceso.

Palabras clave: metodologías, desarrollo de software, procesos ágiles, seguridad, vulnerabilidades.

Methodologies for software security development

Abstract: Security has changed from a non-functional requirement, which could be implemented as a part of software quality, to a key element in any software application. Hackers and criminal groups evolve every day and they have become expert in exploiting vulnerabilities in applications and websites. To address these threats, it is necessary that organizations implementing methodologies that include activities focused on eliminating vulnerabilities and

integrating security as a basic element in the software development process. This paper reviews some of the methodologies that provide security activities in the software development process.

Keywords: methodologies, software development, agile processes, security, vulnerabilities.

1. Introducción

La sociedad se encuentra vinculada innegablemente a la tecnología; sin embargo, su mismo uso tan amplio hace que existan grandes riesgos en cada una de las aplicaciones de la tecnología. Además, los riesgos evolucionan y aumentan con la tendencia en el uso de tecnologías como el cloud computing, Web 2.0 y aplicaciones para dispositivos móviles (Laskowski, 2011). En una organización se pierden aproximadamente \$6.6 millones de dólares por cada brecha de seguridad de TI, estas actividades ilícitas tienen ganancias de hasta 114 billones de dólares anualmente y 431 millones de víctimas anuales, es decir, 14 víctimas de cibercrimen cada segundo (Norton, 2012). Un escenario mundial que no puede pasar desapercibido por la gente relacionada al desarrollo de software y empezar a optar por metodologías que garanticen el lugar que le corresponde a la seguridad.

El objetivo de este trabajo es presentar dos de las metodologías de desarrollo que permiten producir productos de software seguros. Después se presenta un análisis de las metodologías que permiten producir productos de software seguros, una comparación entre ellas y por último las conclusiones y los trabajos futuros que se desprenden de éste trabajo.

2. Metodologías de desarrollo tradicionales

La clave de un software seguro, es el proceso de desarrollo utilizado. En el proceso, es donde se produce el producto que pueda resistir o sostenerse ataques ya anticipados, y recuperarse rápidamente y mitigar el daño causado por los ataques que no pueden ser eliminados o resistidos. Muchos de los defectos relacionados con la seguridad en software se pueden evitar si los desarrolladores estuvieran mejor equipados para reconocer las implicaciones de su diseño y de las posibilidades de implementación.

La manera en que se desarrolla software ha evolucionado, desde la forma de “code and fix” (codificar y después arreglar), en la que un equipo de desarrollo tiene la idea general de lo que quiere desarrollar, pasando a metodologías formales (RUP, Proceso Unificado, PSP/TSP, entre otras) en las que existe una planeación detallada del desarrollo, desde la elicitation, documentación, requerimientos, diseño de alto nivel y la inspección (Vanfossen, 2006). Los métodos ágiles permiten tener un desarrollo iterativo, con ciclos de entrega continuos, un contacto con el cliente permanente; permitiendo que se incluyan como parte de los stakeholders a los administradores de riesgo de la empresa, certificadores y al personal responsable de las políticas de seguridad. Los métodos ágiles se rigen bajo el manifiesto de los procesos ágiles, que menciona que la prioridad más alta es la satisfacción del cliente a través de entregas tempranas y continúas de un producto con valor (Fowler & Highsmith, 2001).

Sin embargo; las metodologías mencionadas, tienden enfocarse en mejorar la calidad en el software, reducir el número de defectos y cumplir con la funcionalidad especificada (Davis, 2005); pero en la actualidad, también es necesario entregar un producto que garantice tener cierto nivel de seguridad. Hay metodologías que contemplan durante su proceso, un conjunto de actividades específicas para remover vulnerabilidades detectadas en el diseño

o en el código, la aplicación de pruebas que aportan datos para la evaluación del estado de seguridad, entre otras actividades relacionadas para mejorar la seguridad del software.

3. Metodologías enfocadas al desarrollo de software seguro

Existen varias metodologías que establecen una serie de pasos en búsqueda de un software más seguro y capaz de resistir ataques. Entre ellas se encuentran Correctness by Construction (CbyC), Security Development Lifecycle (SDL), Digital Touchpoints, Common Criteria, Comprehensive, Lightweight Application Security Process (CLASP), TSP-Secure. El presente artículo estudiará las características de las dos primeras, detallando las fases que las conforman, destacando sus particularidades y al final se realiza una comparativa de ambas.

3.1 Correctness by Construction (CbyC)

Es un método efectivo para desarrollar software que demanda un nivel de seguridad crítico y que además sea demostrable. La empresa Praxis ha utilizado CbyC desde el año 2001 y ha producido software industrial con taza de defectos por debajo de los 0.05 defectos por cada 1000 líneas de código, y con una productividad de 30 líneas de código por persona al día.

Las metas principales de ésta metodología son obtener una taza de defectos al mínimo y un alta resiliencia al cambio; los cuales se logran debido a dos principios fundamentales: que sea muy difícil introducir errores y asegurarse que los errores sean removidos tan pronto hayan sido inyectados. CbyC busca producir un producto que desde el inicio sea correcto, con requerimientos

rigurosos de seguridad, con definición muy detallada del comportamiento del sistema y un diseño sólido y verificable (Croxford & Chapman, 2005).

3.1.1 Fases de la Metodología CbyC

CbyC combina los métodos formales con el desarrollo ágil; utiliza notaciones precisas y un desarrollo incremental que permite mostrar avances para recibir retroalimentación y valoración del producto. La Figura 1, muestra las fases propuestas por ésta metodología para el desarrollo de software.

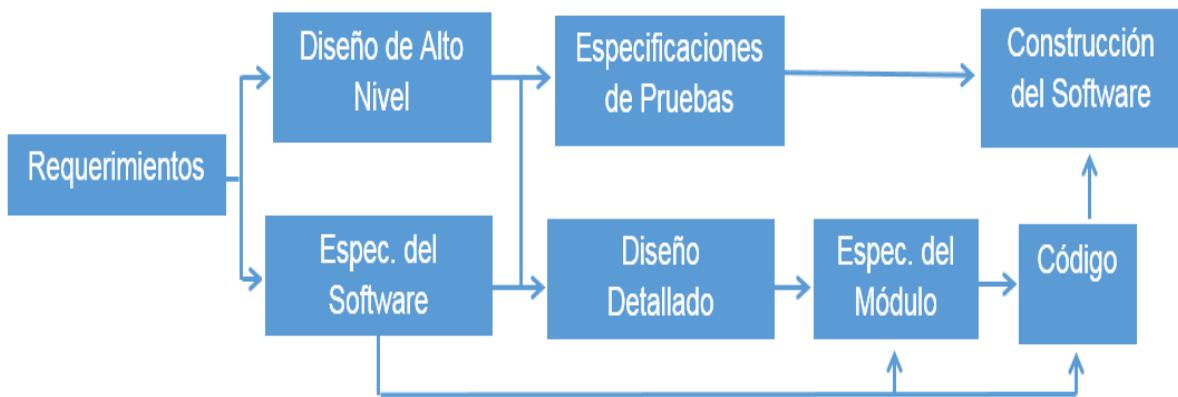


Figura 1.Proceso de desarrollo propuesto por el CbyC (Amey, 2006).

3.1.1.1 Fase de Requerimientos

En la fase de requerimientos se especifica el propósito, las funciones y los requerimientos no funcionales. Se escriben los requerimientos de usuario con sus respectivos diagramas contextuales, diagramas de clase y definiciones operativas. Cada requerimiento debe pasar por un proceso de trazabilidad, así como requerimiento de seguridad a la amenaza correspondiente.

3.1.1.2 Fase de Diseño de Alto Nivel

Se describe la estructura interna del sistema, (distribución de la funcionalidad, estructura de las bases de datos, mecanismos para las transacciones y comunicaciones) la manera en que los componentes colaboran y especifican con mayor énfasis los requerimientos no funcionales de protección y seguridad. Se utilizan los métodos formales para definir el diseño de alto nivel y obtener una descripción, un comportamiento y un modelo preciso. Los métodos formales han probado ser exitosos para especificar y probar el nivel de ‘correctness’ y en la consistencia interna de las especificaciones de funciones de seguridad (Jarzombek & Goertzel, 2006).

3.1.1.3 Fase de Especificación del Software

En ésta fase se documentan las especificaciones de la interfaz de usuario (se define el “look and feel” del sistema), las especificaciones formales de los niveles superiores y se desarrolla un prototipo para su validación.

3.1.1.4 Fase de Diseño Detallado

Define el conjunto de módulos y procesos y la funcionalidad respectiva. Se utiliza notación formal para eliminar ambigüedad en la interpretación del diseño.

3.1.1.5 Fase de Especificación de los Módulos

Se define el estado y el comportamiento encapsulado en cada módulo del software tomando en cuenta el flujo de información. Los módulos deberán de

tener el enfoque de bajo acoplamiento y alta cohesión; así los efectos serían mínimos en caso de producirse una falla en el flujo de información.

3.1.1.6 Fase Codificación

El concepto de evitar cualquier ambigüedad posible, también se aplica en el código; por ello se sugiere la utilización de SPARK (lenguaje formalmente definido y matemáticamente comprobable, basado en el lenguaje de programación Ada, utilizado en sistema de aeronáutica, sistemas médicos y control de procesos en plantas nucleares) en las partes críticas del sistema por estar diseñado para tener una semántica libre de ambigüedades, permitir realizar análisis estático y ser sujeto de una verificación formal (Brito, 2010). Se conducen pruebas de análisis estático al código para eliminar algunos tipos de errores y en caso de ser necesario, se realizará una revisión al código.

3.1.1.7 Fase de las Especificaciones de las Pruebas

Para obtener las especificaciones de las pruebas, se toman en cuenta las especificaciones del software, los requerimientos y el diseño de alto nivel. Se efectúan pruebas de valores límites, pruebas de comportamiento y pruebas para los requerimientos no funcionales; pero CbyC no usa pruebas de caja blanca ni pruebas de unidad; todas las pruebas son a nivel de sistema y orientado a las especificaciones.

3.1.1.8 Fase de Construcción del Software

CbyC utiliza el desarrollo de tipo ágil; en la primera entrega, se tiene el esqueleto completo del sistema con todas las interfaces y mecanismos de

comunicación; con una funcionalidad muy limitada que se irá incrementando en cada iteración del ciclo.

CbyC es compatible con los principios de los procesos de Personal Software Process/Team Software Process (PSP/TSP) y al combinar resulta en una tasa reducción en la taza de defectos. El enfoque técnico de CbyC complementa a PSP para aumentar la capacidad de remoción de defectos en etapas tempranas del desarrollo. (Croxford & Chapman, 2005) . La Figura 2, muestra las tasas de defectos en los cinco niveles del CMM y la tasa cercana cero perteneciente a la metodología CbyC, dejando en claro el beneficio de la implementación de ésta un proceso de desarrollo.

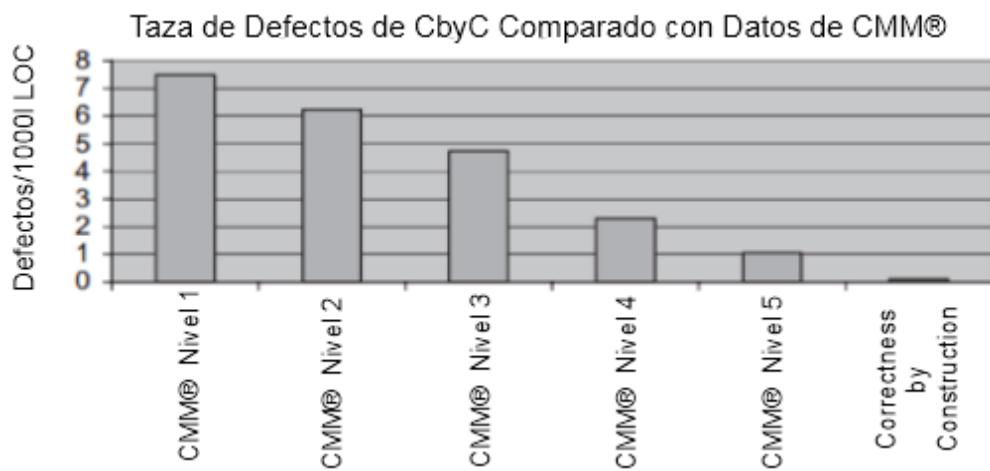


Figura 2.Comparativa de taza de defectos CMMI contra CbyC (Hall, 2007).

3.2 Security Development Lifecycle (SDL)

Es un proceso para mejorar la seguridad de software propuesto por la compañía de Microsoft en el año 2004; con dieciséis actividades enfocadas a mejorar la seguridad del desarrollo de un producto de software (Peterson, 2011).

Las prácticas que propone SDL van desde una etapa de entrenamiento sobre temas de seguridad, pasando por análisis estático, análisis dinámico, fuzz testing del código hasta tener plan de respuesta a incidentes. Una de las características principales de SDL es el modelado de amenazas que sirve a los desarrolladores para encontrar partes del código, donde probablemente exista vulnerabilidades o sean objeto de ataques (Korkeala, 2011).

3.2.1 Fases de la Metodología SDL

Existen dos versiones del SDL, la versión rígida y la orientada al desarrollo ágil. Las diferencias versan en que la segunda desarrolla el producto de manera incremental y en la frecuencia de la ejecución de las actividades para el aseguramiento de la seguridad. La versión rígida del SDL es más apropiada para equipos de desarrollo y proyectos más grandes y no sean susceptibles a cambios durante el proceso. SDL ágil es recomendable para desarrollos de aplicaciones web o basados en la web (Wood & Knox, 2012). En la Figura 3, se encuentra el flujo de las fases en la metodología de SDL cabe resaltar la existencia de una etapa previa a los requerimientos, enfocada al entrenamiento en seguridad y la última etapa del proceso que se encarga de darle seguimiento al producto en caso de algún incidente de seguridad.

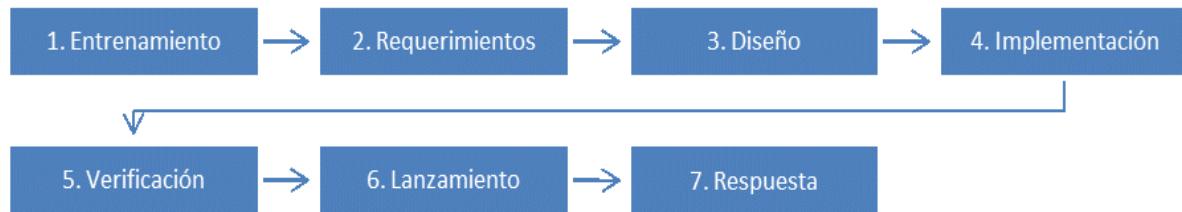


Figura 3. Proceso del desarrollo software del SDL (Microsoft Corporation, 2007).

Las dieciséis actividades de SDL, que consolidan la seguridad, se detallan en la Tabla 1.

Actividades del SDL para la Seguridad	
1. Entrenamiento	5. Verificación
Entrenamiento de seguridad básica	Ánálisis dinámico
2. Requerimientos	Fuzz Testing
Establecer requerimientos de seguridad	Revisión de la superficie de ataques
Crear umbrales de calidad y límites de errores	6. Lanzamiento
Evaluación de los riesgos de seguridad y privacidad	Plan de respuesta a incidentes
3. Diseño	Revisión de seguridad final
Establecer requerimientos de diseño	Aprobar y archivar lanzamiento
Análisis de la superficie de ataques	7. Respuesta
Modelado de amenazas	Ejecutar el plan de respuesta a incidentes
4. Implementación	
Utilizar herramientas aprobadas	
Prohibir funciones no seguras	
Ánálisis estático	

Tabla 1. Actividades del SDL para el aseguramiento de la seguridad (Microsoft Corporation, 2007)

3.2.1.1 Fase de Entrenamiento

Todos los miembros de un equipo de desarrollo de software deben recibir una formación apropiada con el fin de mantenerse al corriente de los conceptos básicos y últimas tendencias en el ámbito de la seguridad y privacidad. Las personas con roles técnicos (desarrolladores, evaluadores y administradores de programas) que están directamente implicadas en el desarrollo de programas de software deben asistir como mínimo una vez al año a una clase de formación en materia de seguridad en conceptos fundamentales como: defensa en profundidad, principio de privilegios mínimos, modelos de riesgos,

saturaciones de búfer, inyección de código SQL, criptografía débil, evaluación de riesgos y procedimientos de desarrollo de privacidad.

3.2.1.2 Fase de Requerimientos

En la fase de requerimientos, el equipo de producción es asistido por un consultor de seguridad para revisar los planes, hacer recomendaciones para cumplir con las metas de seguridad de acuerdo al tamaño del proyecto, complejidad y riesgo. El equipo de producción identifica como será integrada la seguridad en el proceso de desarrollo, identifica los objetivos clave de seguridad, la manera que se integrará el software en conjunto y verificarán que ningún requerimiento pase desapercibido.

3.2.1.3 Fase de Diseño

Se identifican los requerimientos y la estructura del software. Se define una arquitectura segura y guías de diseño que identifiquen los componentes críticos para la seguridad aplicando el enfoque de privilegios mínimos y la reducción del área de ataques. Durante el diseño, el equipo de producción conduce un modelado de amenazas a un nivel de componente por componente, detectando los activos que son administrados por el software y las interfaces por las cuales se pueden acceder a esos activos. El proceso del modelado identifica las amenazas que potencialmente podrían causar daño a algún activo y establece la probabilidad de ocurrencia y se fijan medidas para mitigar el riesgo.

3.2.1.4 Fase de Implementación

Durante la fase de implementación, se codifica, prueba e integra el software. Los resultados del modelado de amenazas sirven de guía a los desarrolladores para generar el código que mitigue las amenazas de alta prioridad. La codificación es regida por estándares, para evitar la inyección de fallas que conlleven a vulnerabilidades de seguridad. Las pruebas se centran en detectar vulnerabilidades y se aplican técnicas de fuzz testing y el análisis de código estático por medio de herramientas de escaneo, desarrolladas por Microsoft. Antes de pasar a la siguiente fase, se hace una revisión del código en búsqueda de posibles vulnerabilidades no detectadas por las herramientas de escaneo.

3.2.1.4 Fase Verificación

En ésta parte el software, ya es funcional en su totalidad y se encuentra en fase beta de prueba. La seguridad es sujeta a un “empujón de seguridad”; que se refiere a una revisión más exhaustiva del código y a ejecutar pruebas en parte del software que ha sido identificado como parte de la superficie de ataque.

3.2.1.5 Fase de Lanzamiento

En el lanzamiento, el software es sujeto a una revisión de seguridad final, durante un periodo de dos a seis meses previos a la entrega con el objetivo de conocer el nivel de seguridad del producto y la probabilidad de soportar ataques, ya estando liberado al cliente. En caso de encontrar vulnerabilidades que pongan en riesgo la seguridad, se regresa a la fase previa para enmendar esas fallas.

3.2.1.6 Fase de Respuesta

Sin importar la cantidad de revisiones que se haga al código o las pruebas en búsqueda de vulnerabilidades, no es posible entregar un software cien por ciento seguro; así que, se debe de estar preparado para responder a incidentes de seguridad y a aprender de los errores cometidos para evitarlos en proyectos futuros (Lipner, 2004).

La compañía Microsoft utilizó el proceso de SDL para hacer el Windows Vista, reduciendo en un 50% las vulnerabilidades comparadas con Windows XP SP2. Después lanzó Microsoft Office 2007, con una reducción del 90% de las vulnerabilidades, a causa de dos requerimientos de SDL: del requerimiento de integer overflow y el requerimiento de fuzzing (Microsoft Corporation, 2007).

4. Comparativa de las metodologías CbyC y SDL

Las metodologías presentadas en éste artículo, tienen como objetivo establecer una forma de desarrollar software que sea más seguro y capaz de soportar ataques maliciosos. En la Tabla 2 se hace una comparativa entre las características de ambas.

Atributo/Metodología	CbyC	SDL
Utilización de métodos formales	X	-
Utiliza desarrollo iterativo	X	-
Entrenamiento en temas de seguridad	-	X
Establece requerimientos de seguridad	X	X
Estudia la trazabilidad de los requerimientos	X	-
Asistencia de personal especializado en la seguridad	-	X
Realiza modelado de amenazas	-	X
Análisis estático	X	X
Análisis dinámico	-	X
Fuzz testing	-	X
Revisión de código	X	X
Desarrolla un plan en caso de incidentes de seguridad	-	X

Tabla 2. Comparativa entre las metodologías CbyC y SDL

De la tabla anterior se podría llegar a pensar que SDL representa mejor seguridad en el producto final. Sin embargo CbyC enfoca su proceso en crear un producto que sea correcto y con el menor número de defectos posibles. SDL tiene varias actividades durante todo el proceso para educar, para descubrir vulnerabilidades en el código y para reducir al área de ataque maliciosos. Las dos tienen limitaciones respecto a los lenguajes de programación, ya que por una parte CbyC se debe comprar la versión del lenguaje SPARK que soporta exigencias críticas de seguridad; y por otro lado SDL siendo impulsada por Microsoft utiliza lenguajes y suite de programación familiares a la compañía, al igual que las herramientas proporcionadas únicamente corren bajo ambiente del sistema operativo Windows.

5. Conclusiones

La seguridad ha dejado de ser un atributo de calidad, se encuentra en cada capa de la arquitectura de software, y por lo tanto no se puede dejar como un elemento aislado, sino que es transversal y multidimensional. Los hackers parecen que siempre están dos pasos más delante de las organizaciones, si se continúa haciendo software de la manera tradicional; esa brecha será aprovechada y seguirán explotando vulnerabilidades que pudieron haber sido evitadas utilizando metodologías como las que en éste documento se citaron.

Este trabajo presentó de manera general, procesos desarrollo de software que tienen actividades enfocadas a la mejora de la seguridad. La utilización de una metodología de éstas características es imperativa para mitigar y tratar de evitar los ataques que día a día, las organizaciones y las personas son víctimas, causando pérdidas millonarias. Recordando que la mejor metodología es aquella que se adapta al contexto del producto a desarrollar, eso da la pauta para escoger la metodología idónea. En un estudio posterior, se podría formular la integración de la versión ágil del SDL con Scrum y aplicarlo en un caso práctico.

Referencias

Amey, P. (2006). Correctness by Construction. Consultado el 29 de septiembre del 2013, en <https://buildsecurityin.us-cert.gov/articles/knowledge/sdLC-process/correctness-by-construction>

Brito, E. (2010). A (Very) Short Introduction to SPARK: Language , Toolset , Projects , Formal Methods & Certification (pp. 479–490). Portugal: INForum 2010 - II Simpósio de Informática.

Croxford, M., & Chapman, R. (2005). Correctness by Construction : A Manifesto for High-Integrity Software. The Journal of Defense Software Engineering, 18(12), 5–8.

- Davis, N. (2005). Secure Software Development Life Cycle Processes: A Technology Scouting Report (pp. 14–20).
- Fowler, M., & Highsmith, J. (2001). The Agile Manifesto. Consultado el 07 de julio del 2013, en <http://www.pmp-projects.org/Agile-Manifesto.pdf>
- Hall, A. (2007). Realising the Benefits of Formal Methods. *Journal of Universal Computer Science*, 13(5), 669–678.
- Korkeala, M. (2011). Integrating SDL for Agile in an ongoing software development project. *Cloud Software Finland*, 1–17.
- Laskowski, J. (2011). Agile IT Security Implementation Methodology (primera ed., pp. 13–21). Birmingham, Reino Unido: Packt Publishing Ltd.
- Lipner, S. (2004). The Trustworthy Computing Security Development Lifecycle. Annual Computer Security Applications Conference, pp. 2 – 11.
- Microsoft Corporation. (2007). The Trustworthy Computing Security Development Lifecycle The Microsoft SDL Team. Consultado en <http://www.microsoft.com/en-us/download/details.aspx?id=12379>
- Norton. (2012). Norton Cybercrime Report 2012 (pp. 1–9). Consultado en <http://us.norton.com/cybercrimereport>
- Peterson, G. (2011). Security Architecture Blueprint. Dublin: Secure Application Development. Consultado en <http://secappdev.org/handouts/2011/Gunnar Peterson/ArctecSecurityArchitectureBlueprint.pdf>
- Vanfossen, T. (2006). Plan-driven vs . Agile Software Engineering and Documentation: A Comparison from the Perspectives of both Developer and Consumer Submitted for the PhD Qualifying Examination. CiteSeerX
- Wood, C., & Knox, G. (2012). Guidelines for Agile Security Requirements Engineering. *Software Requirements Engineering* (pp. 1 –5). Rochester, Nueva York.

Notas biográficas



Carlos Joaquín Brito Abundis tiene el grado de Licenciado en Derecho por la Universidad Autónoma de Durango, actualmente cursa el último semestre de la carrera de Ingeniería en Software en la Universidad Autónoma de Zacatecas. Participó con el proyecto de investigación “Evaluación de la Seguridad de la Red Universitaria mediante Pruebas de Penetración” en el Primer Foro de Jóvenes Investigadores realizado en la ciudad de Fresnillo, Zacatecas. Asimismo, elaboró el poster “Enfoques de Desarrollo para Software Seguro” en el marco del Segundo Congreso Internacional de Mejora de Procesos de Software. Los temas de su interés son pruebas de penetración en dispositivos móviles y la evaluación de la seguridad en las empresas.



Esta obra está bajo una licencia de Creative Commons
Reconocimiento-NoComercial-CompartirIgual 2.5 México.

Lectura 7. Entornos de programación, concepto, funciones y tipos

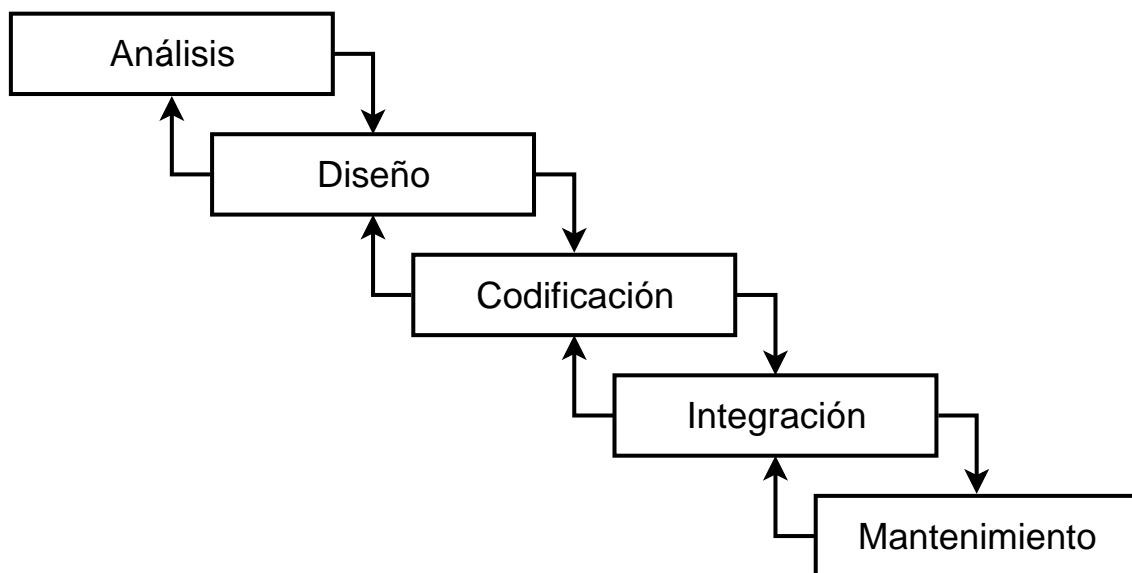
Asignatura: Entornos de programación

Entornos de programación

Concepto, funciones y tipos

1. Actividades de desarrollo de software

En Ingeniería de Software se denomina "ciclo de vida" a una determinada organización en el tiempo de las actividades de desarrollo de software. Las principales actividades son las siguientes:



La figura representa el denominado "ciclo de vida en cascada", donde las flechas indican el orden en que se van realizando las actividades. Este modelo está en desuso, pero sigue siendo adecuado para identificar las actividades principales y el orden natural entre ellas.

Análisis de requisitos

Se estudian las necesidades de los usuarios, se decide qué debe hacer la aplicación informática para satisfacerlas en todo o en parte, y se genera un *Documento de Requisitos*.

Diseño de la arquitectura

Se estudia el Documento de Requisitos y se establece la estructura global de la aplicación, descomponiéndola en partes (módulos, subsistemas) relativamente independientes. Se genera un *Documento de Diseño*.

Diseño detallado

En esta segunda parte de la actividad de diseño se fijan las funciones de cada módulo, con el detalle de su interfaz. Se genera el código de declaración (o especificación) de cada módulo.

Codificación

Se desarrolla el código de cada módulo.

Pruebas de unidades

Como complemento de la codificación, cada módulo o grupo de módulos se prueba por separado. En las pruebas se comprueba si cada módulo cumple con su especificación de diseño detallado.

Pruebas de integración

Se hace funcionar la aplicación completa, combinando todos sus módulos. Se realizan ensayos para comprobar que el funcionamiento de conjunto cumple lo establecido en el documento de diseño.

Pruebas de validación

Como paso final de la integración se realizan nuevas pruebas de la aplicación en su conjunto. En este caso el objetivo es comprobar que el producto desarrollado cumple con lo establecido en el documento de requisitos, y satisface por tanto las necesidades de los usuarios en la medida prevista.

Fase de mantenimiento

No hay actividades diferenciadas de las anteriores. El mantenimiento del producto exige rehacer parte del trabajo inicial, que puede corresponder a cualquiera de las actividades de las etapas anteriores.

2. Entornos de desarrollo de Software

Un **entorno de desarrollo de software** es una combinación de herramientas que automatiza o soporta al menos una gran parte de la tareas (o fases) del desarrollo: análisis de requisitos, diseño de arquitectura, diseño detallado, codificación, pruebas de unidades, pruebas de integración y validación, gestión de configuración, mantenimiento, etc. Las herramientas deben estar bien integradas, pudiendo interoperar unas con otras.

Están formados por el conjunto de instrumentos (*hardware, software, procedimientos, ...*) que facilitan o automatizan las actividades de desarrollo. En el contexto de esta asignatura se consideran básicamente los instrumentos software.

- **CASE:** *Computer-Aided Software Engineering*
 - Con este término genérico se denominan los productos software que dan soporte informático al desarrollo
 - Sería deseable automatizar todo el desarrollo, pero normalmente se automatiza sólo en parte
 - Productos CASE: son cada uno de los instrumentos o herramientas software de apoyo al desarrollo
- La tecnología CASE da soporte para **actividades verticales**
 - Son actividades verticales las específicas de una fase del ciclo de vida: análisis de requisitos, diseño de la arquitectura, edición y compilación del código, etc.
- También se necesita soporte para **actividades horizontales**
 - Son actividades horizontales las actividades generales: documentación, planificación, gestión de configuración, etc.

En [2] se expone una visión práctica de los que es un entorno de desarrollo.

3. Productos CASE en general

Los productos CASE facilitan el desarrollo organizado del software aplicando técnicas de **Ingeniería de Software**. En sentido amplio podemos englobar en la tecnología CASE toda la variedad de herramientas aplicables en el desarrollo de software: herramientas de análisis y diseño; editores de código, documentos, diagramas, etc.; compiladores y montadores de código ejecutable (linkers); depuradores; analizadores de consistencia; herramientas para obtención de métricas; generadores de código o de documentación; etc., etc.

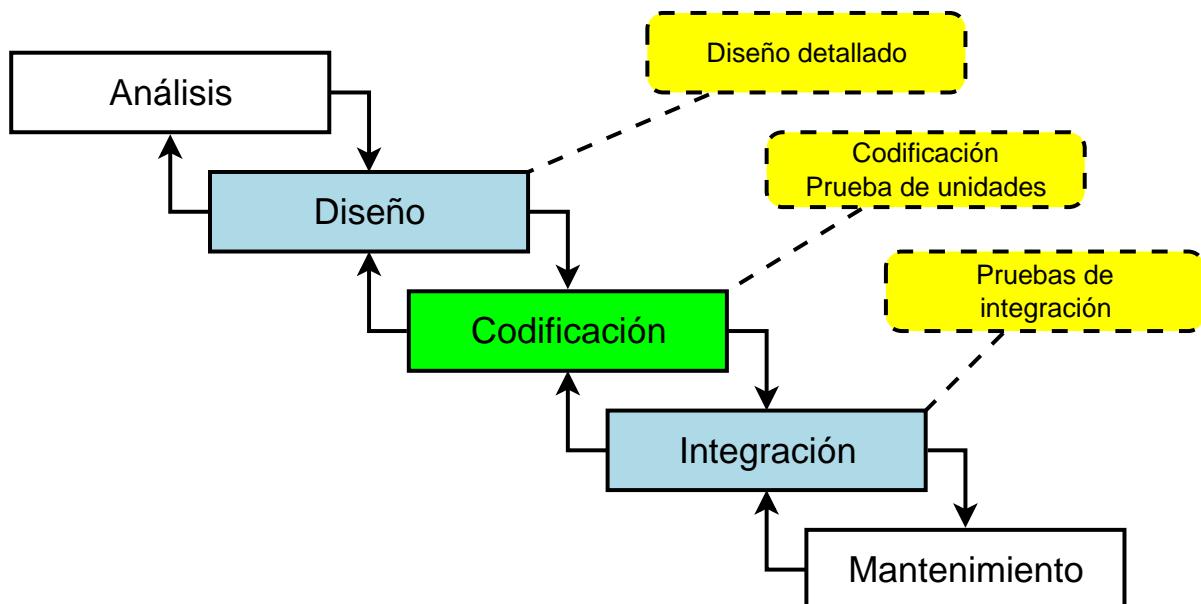
Debido a esa enorme variedad de productos, se han realizado diversos intentos para clasificarlos. Un punto de vista para su clasificación es el nivel de las funciones que realiza un producto determinado. En [3] (Table I) se sugiere la siguiente terminología para los **niveles funcionales**:

- **Servicio** (*service*): realiza automáticamente una determinada operación (atómica o unitaria).
Ejemplo: compilación de un programa
- **Herramienta** (*tool*): ofrece los servicios necesarios para dar soporte a una tarea determinada (lo que hace un miembro del equipo de desarrollo en un momento dado).
Ejemplo: edición de código fuente.
- **Banco de trabajo** (*workbench*): da soporte a todas las actividades correspondientes a un rol o perfil profesional propio de uno de los miembros del equipo de desarrollo. A veces se le llama también "herramienta" (*tool*)
Ejemplo: "herramienta" CASE de análisis y diseño (OO, UML, ...)
- **Entorno o factoría** (*environment, factory*): da soporte a todo el proceso de desarrollo.
A veces se le llama también "banco de trabajo" (*workbench*)

4. Entorno de programación

Las actividades mejor soportadas por herramientas de desarrollo son normalmente las centrales: codificación y pruebas de unidades. El conjunto de herramientas que soportan estas actividades constituyen lo que se llama un **entorno de programación**. A veces se utilizan las siglas **IDE** (*Integrated Development Environment*) para designar estos entornos, aunque no son un entorno de desarrollo completo, sino sólo una parte de él.

- Siguiendo la terminología anterior, de niveles funcionales, es el **banco de trabajo del programador**
- Da soporte a las actividades de la fase de codificación (preparación del código y prueba de unidades)
- Los mismos productos sirven también para el diseño detallado y para las pruebas de integración.
- Se sitúa, por tanto, en la parte central del ciclo de desarrollo



5. Funciones de un Entorno de Programación

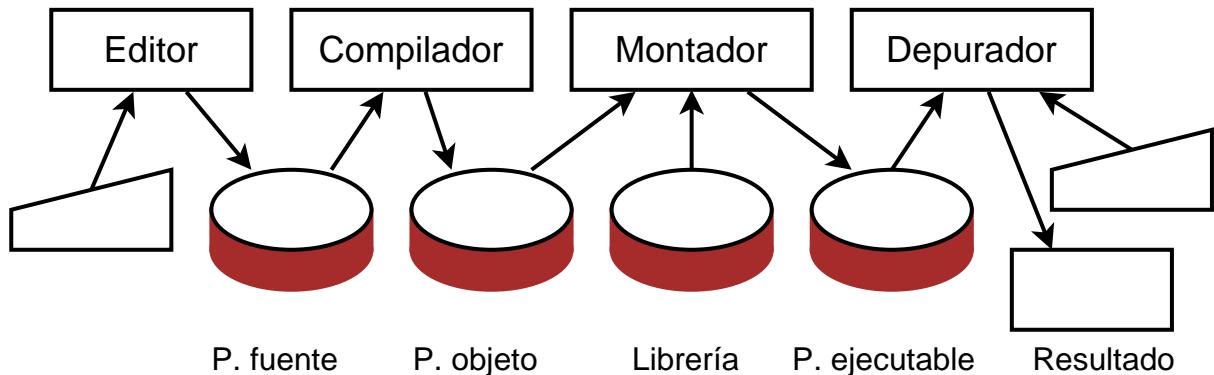
Como se ha dicho, la misión de un Entorno de Programación es dar soporte a la preparación de programas, es decir, a las **actividades de codificación y pruebas**.

- Las tareas esenciales de la fase de codificación son:
 - Edición (creación y modificación) del código fuente
 - Proceso/ejecución del programa
 - Interpretación directa (código fuente)
 - Compilación (código máquina) - montaje - ejecución
 - Compilación (código intermedio) - interpretación
- Otras funciones:
 - Examinar (hojear) el código fuente
 - Analizar consistencia, calidad, etc.
 - Ejecutar en modo depuración
 - Ejecución automática de pruebas
 - Control de versiones
 - Generar documentación, reformar código
 - ... y otras muchas más ...

6. Tipos de Entornos de Programación

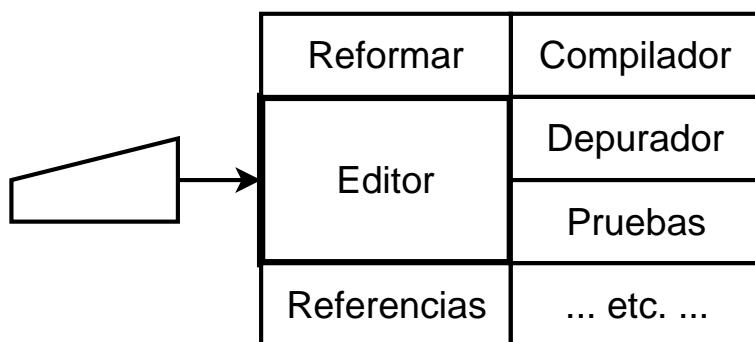
Un entorno de programación puede estar concebido y organizado de maneras muy diferentes. A continuación se mencionan algunas de ellas.

- En las primeras etapas de la informática la preparación de programas se realizaba mediante una cadena de operaciones tales como la que se muestra en la figura para un lenguaje procesado mediante compilador. Cada una de las herramientas debía invocarse manualmente por separado. En estas condiciones no puede hablarse propiamente de un entorno de programación



- El editor es un editor de texto simple
- El compilador traduce cada fichero de código fuente a código objeto
- El montador (*linker / builder / loader*) combina varios ficheros objeto para generar un fichero ejecutable
- El depurador maneja información en términos de lenguaje de máquina
- Un entorno de programación propiamente dicho combina herramientas como éstas, mejoradas y mejor integradas. A veces se nombra con las siglas **IDE** (*Integrated Development Environment*).

I.D.E.



- Los componentes cuya evolución ha sido más aparente son los que realizan la interacción con el usuario:
 - El editor ya no es un simple editor de texto, sino que tiene una clara orientación al lenguaje de programación usado (reconoce y maneja determinados elementos sintácticos)

- El depurador no presenta información en términos del lenguaje de máquina, sino del lenguaje fuente
 - El editor está bien integrado con las demás herramientas (se posiciona directamente en los puntos del código fuente en los que hay errores de compilación, o que se están ejecutando con el depurador en un momento dado).
 - No es fácil establecer una clasificación dentro de la variedad de entornos de programación existentes. En algún momento [1] se describieron las siguientes clases de entornos, no excluyentes, usando un criterio esencialmente pragmático:
 - Entornos **centrados en un lenguaje**
 - Entornos **orientados a estructura**
 - Entornos **colección de herramientas**
- En [4] y [5] se exponen otras clasificaciones basadas en criterios más o menos formales.

6.1 Entornos centrados en un lenguaje

Presentan las siguientes características generales:

- Son específicos para un lenguaje de programación en particular
- Están fuertemente integrados. Aparecen como un todo homogéneo
- Se presentan como una herramienta única
- El editor tiene una fuerte orientación al lenguaje
- Son relativamente cómodos o fáciles de usar
- A veces son poco flexibles en lo referente a la interoperación con otros productos o a la ampliación de sus funciones
- Se basan en representar el código fuente como texto

Podemos encontrar ejemplos de estos entornos para todo tipo de lenguajes

- Lenguajes funcionales con interpretación directa
 - (Inter)Lisp, Haskell, etc.
- Lenguajes compilados a código de máquina nativo
 - Delphi, Visual C++, AdaGide/GNAT, GPS, etc.
- Lenguaje ejecutados sobre máquina virtual
 - Java (Visual Age, Eclipse), C# (Visual Studio .NET)
- Ejemplos especiales:
 - Entornos Ada (Stoneman, Cais, Asis)
 - Entornos Smalltalk
 - Entornos Oberon, Component Pascal

6.2 Entornos orientados a estructura

Podrían considerarse incluidos en la clase anterior, ya que suelen ser específicos para un lenguaje de programación, pero están concebidos de manera diferente:

- El editor de código fuente no es un editor de texto, sino un editor de estructura (editor sintáctico)
- Se basan en representar internamente el código fuente como una estructura:
 - Árbol de sintaxis abstracta: AST
- La presentación externa del código es en forma de texto
 - Plantillas (elementos sintácticos no terminales)
 - Texto simple (elementos terminales - a veces "frases" para expresiones)
- Compilación incremental (en algunos casos)
- Para desarrollo personal, no en equipo
- Ejemplos:
 - [The Cornell Program Synthesizer](#) (subconjunto de PL/I)
 - [Mentor](#) (Pascal)
 - [Alice Pascal](#)
 - [Gandalf](#) (intenta ser un entorno de desarrollo completo, para todo el ciclo de vida)

Estos entornos estuvieron de moda en los años 80. Los desarrollos fueron fundamentalmente académicos, y quedaron en desuso. En la actualidad los lenguajes de mercado (XML) pueden ser una buena forma de representar la estructura del código fuente con vistas a su manipulación. Existen editores y procesadores XML que podrían ser la base de nuevos entornos de programación orientados a estructura.

6.3 Entornos basados en combinación de herramientas

Consisten en una combinación de diversas herramientas capaces de interoperar entre ellas de alguna manera. Se denominan *entornos toolkit*. Presentan las siguientes características:

- Presentan integración débil
- Son un conjunto de elementos relativamente heterogéneos
- Son fáciles de ampliar o adaptar mediante nuevas herramientas
- Pueden ser construidos en parte por el propio usuario (programador): éste es más o menos el estilo UNIX original
- Ofrecen poco control de uso de cada herramienta
- El elemento frontal (*front-end*) para interacción con el usuario suele ser un editor configurable, con llamadas a herramientas externas. A veces estos editores configurables se designan también con las siglas IDE (que debería reservarse para el entorno completo)
- Ejemplos de editores configurables
 - [Emacs](#), [Vim](#), [Gvim](#)
 - [Med](#), [SciTE](#), [jEdit](#)
 - [Eclipse](#) (algo más que un editor)

6.4 Entornos multilenguaje

Hay aplicaciones que combinan piezas de código fuente escritas en diferentes lenguajes de programación. Algunas posibilidades de combinación son las siguientes:

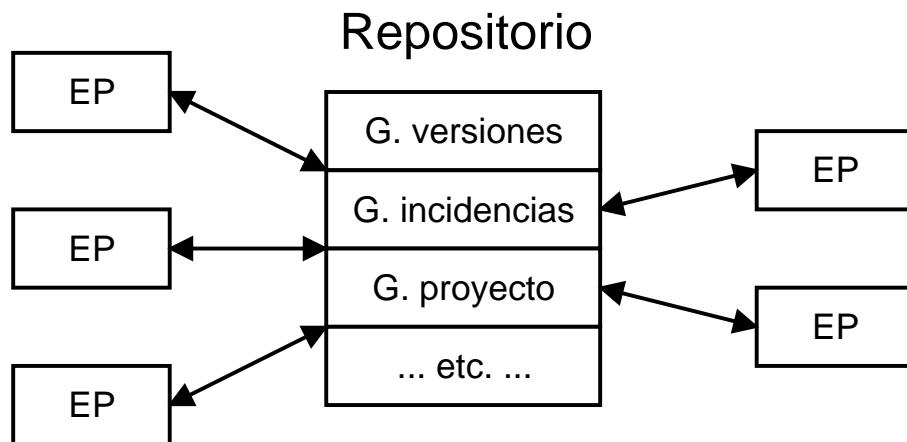
- Entornos genéricos
 - No se combinan lenguajes en un mismo programa. Hay varios programas, cada uno en su propio lenguaje
 - Bastaría combinar las herramientas correspondientes a cada lenguaje (compiladores, etc.)
 - Se podría usar un frontal común: editor personalizable que soporte los lenguajes concretos
 - Ejemplos:
 - Emacs (con diferentes "modos")
 - Eclipse (con diferentes "plug-ins")
- Entornos específicos
 - Para una combinación concreta de lenguajes
 - Vienen a ser como los entornos centrados en un lenguaje, sólo que admiten más de uno
 - Usan un formato binario compatible que permite combinar en un mismo programa partes escritas en los diferentes lenguajes
 - Ejemplo: GPS permite combinar módulos en Ada y C++
- Lenguajes ejecutados sobre máquina virtual
 - La máquina virtual establece el formato del código binario
 - Pueden combinarse módulos escritos en diferentes lenguajes para los que exista el compilador apropiado
 - Cada lenguaje puede tener su entorno de programación separado, o bien existir un entorno de programación único
 - Ejemplos:
 - JVM (Java Virtual Machine). El lenguaje original es Java. El intérprete es el JRE (Java Runtime Environment). Hay compiladores a JVM para otros lenguajes además de Java: Ada, Fortran, Component Pascal (Oberon), etc. (incluso C#)
 - .Net (Microsoft). El lenguaje original es C#. El intérprete es el CLR (Common Language Runtime). Hay compiladores a .Net para otros lenguajes además de C#: Ada, Fortran, Component Pascal (Oberon), etc. (incluso Java)

6.5 Entornos para ingeniería de software

Un entorno de programación, tal como se ha definido anteriormente, serviría para dar soporte a las tareas de desarrollo de software realizadas por una persona. Para desarrollar proyectos de software no triviales se necesita trabajar en equipo usando las recomendaciones de la ingeniería de software.

Cada miembro del equipo de desarrollo puede disponer de una estación de trabajo con un entorno de programación adecuado para realizar su trabajo individual, y se necesita además algún medio de combinar los trabajos individuales en una labor de conjunto, debidamente organizada.

Una manera intuitiva de organizar el entorno general de desarrollo es basarlo en un repositorio central de información, dotado de un sistema de gestión de configuración, y añadirle sistemas de mensajería, de gestión de incidencias, herramientas de modelado para análisis y diseño, de gestión del proyecto, etc.



Por ejemplo, hay plataformas generales que ofrecen este soporte como servicios web, incluso de manera gratuita para el desarrollo de software libre: [SourceForge](#), [Google Code](#), etc.

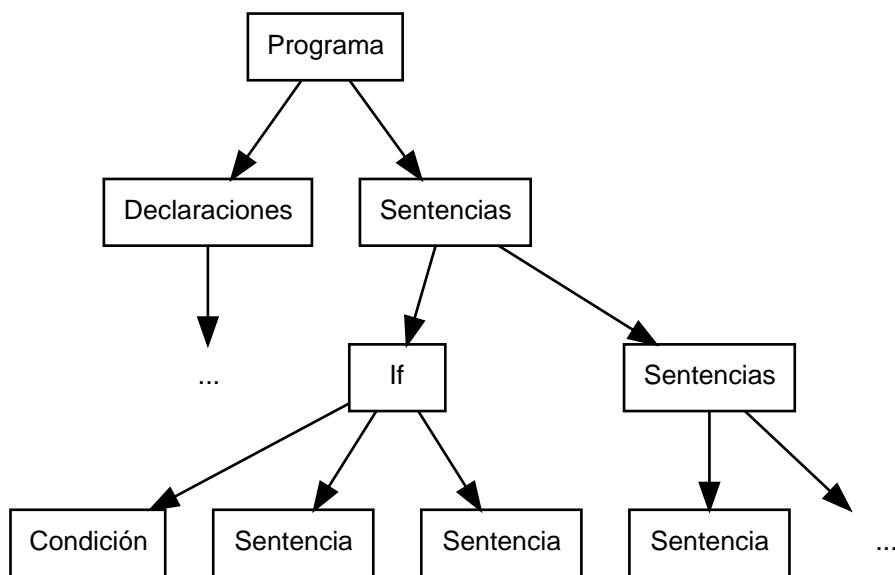
7. Entornos orientados a estructura

La idea de que un programa no es equivalente al texto de su código fuente, sino que lo esencial es la estructura lógica del cómputo que describe, ha llevado a la creación de los llamados entornos de programación orientados a estructura^[1], en los que se manipula directamente la estructura lógica del código y no su representación como texto. Incluso hay casos en que el código del programa no se representa como texto sino en forma gráfica.

7.1 Editores de estructura

Los editores de estructura de código, llamados también editores sintácticos o más frecuentemente editores dirigidos por sintaxis (*syntax-directed editors*), permiten editar el código fuente manipulando directamente una representación interna de su estructura. A diferencia de la edición del código como texto, la edición de la estructura se hace sobre elementos sintácticos tales como expresiones, sentencias o funciones y no sobre elementos textuales tales como caracteres, palabras o líneas.

La representación habitual de la estructura del código es la de su árbol de sintaxis abstracta (AST). Ejemplo:



Los entornos de programación basados en un editor de estructura se denominan entornos orientados a estructura. Suelen tener las siguientes características:

- Soportan un único lenguaje de programación.
- Garantizan que el código es sintácticamente correcto.
- La compilación se realiza de manera incremental, a medida que se edita el código.
- Permite la ejecución inmediata del código editado, incluso aunque esté incompleto.
- Soportan el desarrollo de software a nivel individual, pero no el desarrollo en equipo a gran escala.

La mayoría de estos entornos se desarrollaron a finales del años 70 y a lo largo de los 80. Se emplearon habitualmente en ambientes académicos. Algunos ejemplos de entornos orientados a estructura son:

- El **Cornell Program Synthesizer (CPS)**. Es quizá el ejemplo más conocido y el más referenciado en la literatura [4]. Ha servido de ejemplo para desarrollar otros. Permite programar en un subconjunto del lenguaje PL/I denominado PL/CS.
- **Mentor** [7] es un entorno de programación en Pascal.
- **Gandalf** [8] es un conjunto de varios subproyectos. Su objetivo principal fue crear un entorno completo de desarrollo de software, y no sólo un entorno de programación.
- **Alice Pascal** [9] es otro entorno de programación en lenguaje Pascal compatible con TurboPascal. Sigue las ideas del CPS.
- **SDS** es un entorno de programación en Modula-2. Es un producto comercial desarrollado por Interface Technologies. Ha desaparecido.

Como complemento se han llegado a desarrollar también generadores de entornos similares a los generadores de compiladores. En particular existe el **Synthesizer Generator** [10], capaz de generar entornos similares al sintetizador de Cornell para otros lenguajes de programación a partir de una descripción de la sintaxis y semántica del lenguaje mediante una gramática de atributos.

7.2 Lenguajes y entornos visuales

Este es un caso especial de entornos orientados a estructura. La representación externa del código fuente no es en forma de texto, sino gráfica. El editor permite ir construyendo el grafo que representa la estructura del código. El programa construido de esta manera se ejecuta directamente mediante un intérprete, o bien se exporta como texto en un lenguaje formal para ser compilado o interpretado externamente. Algunos ejemplos de este tipo de entornos son:

- **Prograph**
- **Projector** (parte del meta-CASE DOME)
- **VFPE**

Tanto Prograph como Projector son lenguajes de flujo de datos. Un programa basado en flujo de datos se representa como un grafo en el que los nodos son operadores y los arcos son flujos de datos que conectan la salida de ciertos operadores con las entradas de otros. Una operación se ejecuta cuando hay datos presentes en todas las entradas requeridas. En ese momento se producen resultados que se transmiten por los arcos de salida, pudiendo entonces ejecutarse otras operaciones.

VFPE es un editor gráfico de la estructura (árbol sintáctico) de un programa funcional. El programa editado puede ejecutarse directamente o exportarse como código Haskell.

7.3 Discusión

Los entornos orientados a estructura presentan innegables ventajas respecto a los entornos basados en la edición del texto fuente. Entre ellas:

- Evitan los errores sintácticos
- Evitan tener que escribir los elementos fijos del código: Palabras clave, puntuación, etc.

- Presentan el código con un estilo uniforme, bien encolumnado.
- Guían al programador indicando qué elementos pueden insertarse en cada punto y recordándole la sintaxis de cada sentencia estructurada.
- Facilitan la reorganización del código al permitir la selección directa de secciones de código: funciones, bucles, etc.
- Facilitan trabajar con estructuras lógicas no contempladas directamente en el lenguaje de programación.

Si embargo estos entornos, tal como se concibieron inicialmente, no han llegado a utilizarse en la práctica habitual de desarrollo de software, ya que presentaban claros inconvenientes:

- No permitían el trabajo en equipo. En muchos casos sólo trabajaban con programas monolíticos.
- Era difícil realizar algunas operaciones de edición que resultan triviales sobre el texto.
- Exigen un cambio en la mentalidad del programador [11].
- Es difícil editar la estructura a nivel de grano fino (expresiones: operadores, operandos, ...)

Muchos de los productos citados como ejemplo han desaparecido. Su interés es fundamentalmente histórico. No obstante, la idea de trabajar directamente sobre la estructura del código sigue resultando atractiva, y parece posible aplicarla de nuevo siendo conscientes de sus dificultades y aprovechando los avances tecnológicos actuales.

8. Evolución de los entornos de programación

Otra circunstancia que quizá ha colaborado a impedir la difusión de los entornos orientados a estructura es el hecho de que los entornos de programación convencionales basados en la manipulación del código fuente como texto han ido mejorando a lo largo de los años, ofreciendo algunas funciones similares a las que se pretendían conseguir con los entornos orientados a estructura. A continuación se analizan algunas de estas funciones.

8.1 Lectura y navegación del código

Los entornos basados en texto fuente han ido incrementando su orientación al lenguaje, ofreciendo facilidades tales como las siguientes:

- **Resaltado de sintaxis.** Se destacan los elementos léxicos con diferentes colores o tipo de letra. También se pueden destacar los paréntesis o llaves que se emparejan simplemente poniendo el cursor sobre uno de ellos. El reconocimiento de estos elementos se basa en un sencillo análisis de cada línea de texto, que se realiza sobre la marcha. En la mayoría de los casos el resultado es totalmente satisfactorio, pero a veces se producen confusiones si se emplean construcciones sintácticas complejas. Un análisis sintáctico más completo para garantizar el correcto funcionamiento en todos los casos sería demasiado costoso.
- **Plegado/desplegado.** Se puede controlar el nivel de detalle con el que se presenta el texto fuente. Determinadas secciones de código pueden presentarse de manera abreviada, para facilitar la visión de conjunto. Por ejemplo, un subprograma puede reducirse a la línea de cabecera junto con una indicación visual de que parte del contenido está oculto. Ocasionalmente puede tener los inconvenientes indicados en el punto anterior.
- **Acceso directo entre elementos relacionados.** Son facilidades equivalentes a hipertexto para saltar rápidamente entre la declaración, definición, implementación y uso de un determinado identificador. Puede extenderse incluso a funciones de librería, y también al recorrido de todos los puntos en que usa un identificador. En ocasiones se dispone de una ventana emergente que muestra la cabecera de declaración de una función al poner el cursor sobre un punto en que se usa dicha función. Estas facilidades resultan extraordinariamente útiles para la lectura y comprensión del código.
- **Vistas múltiples.** Se usan varias ventanas para presentar simultáneamente diferentes vistas del código fuente. Es habitual tener un panel lateral con la estructura en árbol del código, bien del fichero en edición o de todo el proyecto. También se pueden tener a la vez los ficheros fuente de interfaz y de implementación del mismo módulo. Y ventanas adicionales con los resultados de aplicar determinadas herramientas: lista de errores de compilación, lista de puntos en que usa un identificador, mensajes de diagnóstico de una herramienta de análisis, representaciones gráficas, etc.

Las últimas facilidades de la lista pueden tener un coste elevado. En bastantes casos hay que construir unas tablas internas, bastante voluminosas, que describen la estructura del código y permiten indexar la aparición de cada identificador en el texto fuente.

8.2 Generación de documentación

Si se incluyen en el código fuente comentarios de descripción de cada elemento importante es posible generar automáticamente documentación de referencia a base de extraer las declaraciones de dichos elementos junto con el texto de descripción. Por ejemplo, se pueden generar las páginas del manual de referencia de una librería, incluyendo índices alfabéticos o temáticos, o incluso incluir todo el código fuente en forma de páginas web enlazadas para consultarlas de manera muy cómoda.

Algunos ejemplos de esta tecnología son:

- **Doxygen**: Genera documentación de código C/C++ en forma de páginas web, incluyendo diversos índices, el código fuente coloreado e indexado, e incluso diagramas de dependencia entre módulos.
- **Javadoc**: Para lenguaje Java. Usa un formato prefijado de comentario para las descripciones, incluyendo marcas HTML embebidas y palabras clave introducidas con el símbolo @. Se generan documentos en forma de páginas web.
- **AdaBrowse/AdaDoc/Ada2html**: Son herramientas similares a las anteriores para código en lenguaje Ada. Igualmente generan páginas web.

A continuación se reproduce un ejemplo tomado de la documentación disponible en el sitio web de Sun Microsystems, Inc [12]. El fragmento de código fuente es:

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

A partir de él se genera el siguiente fragmento de documentación (*convertido a XHTML*):

getImage

```
public Image getImage(URL url,
                      String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute [URL](#). The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

`url` - an absolute URL giving the base location of the image
`name` - the location of the image, relative to the `url` argument

Returns:

the image at the specified URL

See Also:

[Image](#)

8.3 Análisis estático

En estas técnicas se analiza la estructura sintáctica y/o semántica del programa para realizar comprobaciones, extraer información y generar informes. Algunos ejemplos interesantes son:

- **Análisis de consistencia:** Se realizan comprobaciones adicionales a las que habitualmente hace un compilador. De esta manera se comprueba que el código no contiene irregularidades, ambigüedades, código inalcanzable, construcciones desaconsejadas, etc. Ejemplo en lenguaje C: uso de una asignación como condición, sentencia simple en un 'if', etc:

```
if (x = 3)
    printf( "ejemplo" );
    printf( "\n" );
```

- **Comprobación de estilo:** Se comprueba si el código cumple determinadas reglas de presentación o codificación. Puede realizarse conjuntamente con el análisis del punto anterior.
- **Cálculo de métricas:** Existen medidas bien establecidas de la complejidad o tamaño del código. Pueden utilizarse como indicadores de calidad o de productividad en el desarrollo del proyecto. Consisten habitualmente en recuentos de determinados elementos (número de líneas de código, de sentencias, de funciones, etc.) y obtención de parámetros estadísticos (número de líneas por función, porcentaje de comentarios, número de métodos por clase, etc.).
- **Seccionamiento (*slicing*) de programas:** Consiste en extraer la parte del código que interviene en el cálculo de una determinada variable. Esto ayuda a comprender el funcionamiento del programa, diagnosticar fallos, detectar dependencias entre partes del código o determinar el impacto de un posible cambio.
- **Otras ayudas a la comprensión del código:** En muchos casos se traducen en diagramas que muestran de manera clara determinadas características del código. Por ejemplo, el grafo de flujo de llamadas, un diagrama de estructura modular, etc.

Las operaciones de análisis mencionadas tienen partes en común, por lo que muchas veces una misma herramienta puede realizar simultánea o alternativamente varias de ellas.

8.4 Técnicas avanzadas de construcción de código

El desarrollo de metodologías de programación ha dado como resultado diversas técnicas y herramientas de ayuda a la construcción de nuevo código. Algunas de ellas se han incorporado a los entornos de programación integrados y otras están soportadas mediante herramientas independientes. A continuación se mencionan algunas de ellas.

- **Asistentes (Assistants, Wizards):** Ayudan al programador, bien guiando las operaciones de construcción, o bien generando automáticamente parte del código. Por ejemplo, los entornos de programación orientada a objetos pueden disponer de un *Class Wizard* que presenta un índice de las clases existentes con sus miembros, y facilita la creación de nuevos elementos generando un esqueleto de código para la nueva clase o miembro.
- **Ingeniería inversa:** No es una técnica de generación de nuevo código, sino de reconstrucción de información de diseño a partir de código heredado no documentado.
- **Reingeniería:** Complementa la anterior, facilitando la reorganización del código antiguo para facilitar su mantenimiento.
- **Refactorización:** Viene a ser una forma de reingeniería continua del código en desarrollo, recomendada por la metodología de Programación Extrema (*Extreme Programming - XP*). El desarrollo se hace mediante cambios progresivos, y en cada uno de ellos se reorganiza el código, si es conveniente. Es decir, se modifica el diseño y no sólo el código fuente.
- **Desarrollo basado en componentes:** Es una forma intensiva de reutilización. Se construye una aplicación combinando componentes ya existentes en lugar de desarrollar nuevos elementos partiendo de cero. Por supuesto, sólo puede aplicarse de la forma indicada si existe previamente una base de componentes para el dominio de la aplicación. En general exige desarrollar algo de código para configurar la aplicación y a veces para actuar como intermediario entre componentes (*wrapper*, *middleware*), si estos no pueden conectarse directamente.
- **Composición invasiva:** A veces la simple combinación de componentes no es posible, sino que el código interno de los componentes debe ser modificado parcialmente para obtener la nueva aplicación. En este caso debería haber una herramienta que realice los cambios de acuerdo con una cierta especificación, en lugar de realizar los cambios manualmente. La acción de composición debe ser repetible, para facilitar el mantenimiento.
- **Programación orientada a aspectos (AOP):** Es una forma de composición invasiva en que el código base se amplía para incluir nuevos "aspectos" inicialmente no tenidos en cuenta. Existen lenguajes para definir los nuevos fragmentos de código e indicar en qué puntos deben insertarse. La herramienta de soporte se denomina "tejedor" (*weaver*) y genera el código modificado a partir del código original y la descripción de los nuevos "aspectos".

- **Lenguajes específicos de un dominio (DSL):** En lugar de crear librerías de funciones para implementar las operaciones frecuentes en un dominio de aplicación e invocarlas desde un lenguaje de programación de uso general, lo que se puede hacer es crear un lenguaje de programación especializado que incorpore directamente dichas funciones como predefinidas. Este lenguaje especializado puede facilitar la productividad de los programadores al trabajar a un nivel de abstracción superior y usar una notación más compacta o específica, que facilite la claridad del código. Es frecuente que estos lenguajes especializados no se procesen con un compilador específico, sino mediante un traductor o generador de código que produce código fuente en un lenguaje de programación de uso general, que posteriormente se compila de la forma habitual. Esto facilita además combinar en un mismo proyecto partes codificadas con el lenguaje del dominio y con el lenguaje de uso general.
- **Programación letrada (Literate Programming - LP):** Se usa poco en la práctica, aunque puede resultar muy eficiente de cara al mantenimiento del software. La idea es redactar el código como si fuera un documento técnico, explicando cada parte. Los fragmentos de código aparecen entremezclados con las explicaciones, en el orden adecuado para entender el conjunto. No hay que seguir el orden del código final, sino que cada parte se introduce en el momento apropiado para facilitar la lectura y comprensión del documento. Se utiliza un lenguaje de autor especializado (denominado genéricamente *web*), y dos herramientas: una que genera el documento en formato legible con una buena presentación (*weave*), y otra que genera el código de la aplicación listo para ser compilado (*tangle*).

8.5 Discusión

Las técnicas y facilidades incorporadas en los entornos de programación y desarrollo a lo largo de estos años resultan extraordinariamente útiles para abordar la construcción de grandes sistemas de software, con cientos de miles o millones de líneas de código. Pero su aplicación sobre la base convencional de representar y manipular el código fuente como texto tiene un coste enorme en esfuerzo para la creación de las herramientas y en recursos para su aplicación. Cada herramienta o función necesita analizar el texto de código para reconocer su estructura lógica. Para facilitar los tratamientos es frecuente que en el entorno se tengan a la vez una representación de código como texto y unas tablas o índices con su estructura lógica, que si es detallada ocupa un espacio muy superior al del mismo código. Además a medida que se modifica el código hay que mantener sincronizadas ambas representaciones, lo que exige una gran potencia de cálculo.

Los antiguos entornos orientados a estructura no llegaron a desarrollar las facilidades ahora disponibles, pero tenían un enfoque mucho más simple, con una sola representación del código, más eficiente y sencilla de manipular. Recientemente han surgido desarrollos en los que se recuperan en parte algunas ideas o técnicas de orientación a estructura.

9. Referencias

1. S.A. Dart, R.J. Ellison, P.H. Feiler, A.N. Habermann: *Software Development Environments*. IEEE Computer, Vol.20 No.11 pp.18-28, Nov.1987.
2. M.B. Doar: *Practical Development Environments* (el Cap.2: *Project basics*, está disponible como muestra, así como una *vista parcial* del libro). O'Reilly Media, 2005.
3. C. Fernström, K-H Närfelt, L. Ohlsson: *Software Factory Principles, Architecture, and Experiments*. IEEE Software, Vol.9 No.2 pp.36-44, Mar.1992.
4. A. Fuggetta: *A Classification of CASE Technology*. IEEE Computer, Vol.26 No.12 pp.25-38, Dic.1993.
5. D.E. Perry, G.E. Kaiser: *Models of Software Development Environments*. IEEE Trans.Softw.Eng., Vol.17 No.3, pp.283-295, Mar.1991.
6. T. Teitelbaum, T. Repps: *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment*. Comm. ACM, V.24 N.9 pp.563-673, Sep.1981.
7. V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang: *Programming environments based on structured editors: the MENTOR experience*. In *Interactive Programming Environments*, McGraw-Hill, 1984.
8. A. Habermann, D. Notkin: *Gandalf: Software development environments*. IEEE Transactions on Software Engineering SE-12, pp 1117-1127, 1986.
9. B. Templeton: *Alice Pascal website* (<http://www.templetons.com/brad/alice.html>)
10. T. Reps, T. Teitelbaum: *The Synthesizer Generator*. ACM Softw.Engin.Notes, May 1984.
11. L. R. Neal: *Cognition-Sensitive Design and User Modeling for Syntax-Directed Editors*. In Proceedings CHI+GI 1987 (Toronto, April 5-9, 1987) ACM, New York, pp 27-32.
12. Sun Microsystems: *Javadoc website* (<http://java.sun.com/j2se/javadoc/writingdoccomments/index.html>).

Lectura 8. Pruebas e implementación de la aplicación

CAPÍTULO 4

PRUEBAS E IMPLEMENTACIÓN DE LA APLICACIÓN

CONCEPTOS DE PRUEBAS DE APLICACIÓN

El departamento de “Testing” se encarga de diseñar, planear y aplicar el rol de pruebas a los sistemas que el PROVEEDOR tiene a su cargo, con el fin de asegurar la calidad y seguridad de los productos que ofrece a sus clientes. Aunque el equipo de desarrollo no tuvo participación en esta etapa del desarrollo del sistema LMP, aplicamos varias técnicas de testing de software, que aunque no fueron a nivel de las que aplica el departamento de “Testing”, nos permitieron validar la operación del software e ir mejorando la aplicación.

Probar es el proceso de ejecución de software con la intención de encontrar y corregir errores. Ya que los sistemas web residen en red y son accesibles desde muchos sistemas operativos, navegadores de varios dispositivos, plataformas de hardware la búsqueda de errores representa un reto significativo. Las pruebas se enfocan en contenido, función, estructura, usabilidad, navegabilidad, rendimiento, compatibilidad, interacción, capacidad y seguridad. Estas pruebas ocurren mientras se diseña y desarrolla la aplicación y pruebas que se llevan a cabo una vez que se implementa.

Como parte de las pruebas al sistema LMP se examinaron las siguientes dimensiones de calidad.

Contenido: Evalúa tanto en el nivel sintáctico como en el semántico. En el primero, se valora el vocabulario, puntuación y gramática de los textos en la aplicación. En el segundo se valora la consistencia de la información.

Función: Se prueba para descubrir errores que indican falta de conformidad con los requerimientos del cliente.

Estructura: Se valora para garantizar que entrega adecuadamente el contenido y la función de la aplicación.

Usabilidad: Se prueba para asegurar que la interfaz soporta cada tipo de usuario.

Navegabilidad: Se prueba para asegurar que toda la sintaxis y la semántica de navegación se ejecutan para descubrir cualquier error de navegación (por ejemplo, vínculos muertos, inadecuados y erróneos).

Rendimiento: Se prueba bajo condiciones operativas, configuraciones y cargas diferentes a fin de asegurar que el sistema responde a la interacción con el usuario y que maneja la carga operativa con un nivel aceptable.

Compatibilidad: Se prueba al ejecutar la aplicación en varias configuraciones del cliente como en el servidor.

Interoperabilidad: Se prueba para garantizar que la aplicación tiene la interfaz adecuada con otras aplicaciones y/o bases de datos.

Seguridad: Se prueban al valorar las vulnerabilidades potenciales e intenta explotar cada una. Cualquier intento de penetración exitoso se estima como un fallo de seguridad.

El procedimiento básico de pruebas de aplicaciones Web se basa en definir entradas (información a partir de la cual la aplicación generara una serie de contenidos Web) y realizar comprobaciones sobre las salidas (contenidos Web generados: documentos HTML, JavaScript, documentos XML, etc.). Se trata, por tanto, de pruebas funcionales que se realizan mediante un enfoque de caja negra, es decir, los casos de prueba se construyen sin tener en cuenta la estructura interna de la aplicación sino únicamente sus entradas y salidas esperadas. Estas entradas y salidas son típicamente peticiones HTTP y documentos Web respectivamente.

En el caso de la aplicación LMP se realizaron diversas pruebas por parte del equipo de testing de la empresa, mi tarea en esta etapa como tal no fue realizar estas pruebas puntualmente pero mientras se desarollo la aplicación utilizamos herramientas como:

Zero Day Scan es un servicio online gratuito que detecta las vulnerabilidades de una Web, como por ejemplo el Cross-Site Scripting (XSS), inyección SQL, y otras. A partir de este análisis, genera un resumen de resultados. Requiere la confirmación del propietario de la página.

Características:

- ∞ No requiere instalación, es un servicio gratuito.
- ∞ Detecta ataques de Cross Site Scripting (XSS)
- ∞ Detecta directorios ocultos y archives de backup.
- ∞ Comprueba vulnerabilidades de seguridad conocidas.
- ∞ Busca vulnerabilidades de SQL injections.
- ∞ Realiza Website Fingerprinting

Además de Powerfuzzer que permite crear tests personalizados para de aplicaciones Web con la intención de detectar agujeros de seguridad. En esencia, se trata de un simple escáner de aplicaciones. Algunos de los tests que ofrece son los siguientes:

- ∞ Código de páginas cruzadas (XSS)
- ∞ Inyecciones (SQL, LDAP, código, comandos, y XPATH)
- ∞ CRLF
- ∞ Estados HTTP 500

Por ejemplo en la Figura 4.1 se muestra cómo la aplicación Powerfuzzer intenta hacer diversos ataques a la aplicación, para detectar alguna vulnerabilidad en el sistema de login, sin éxito.

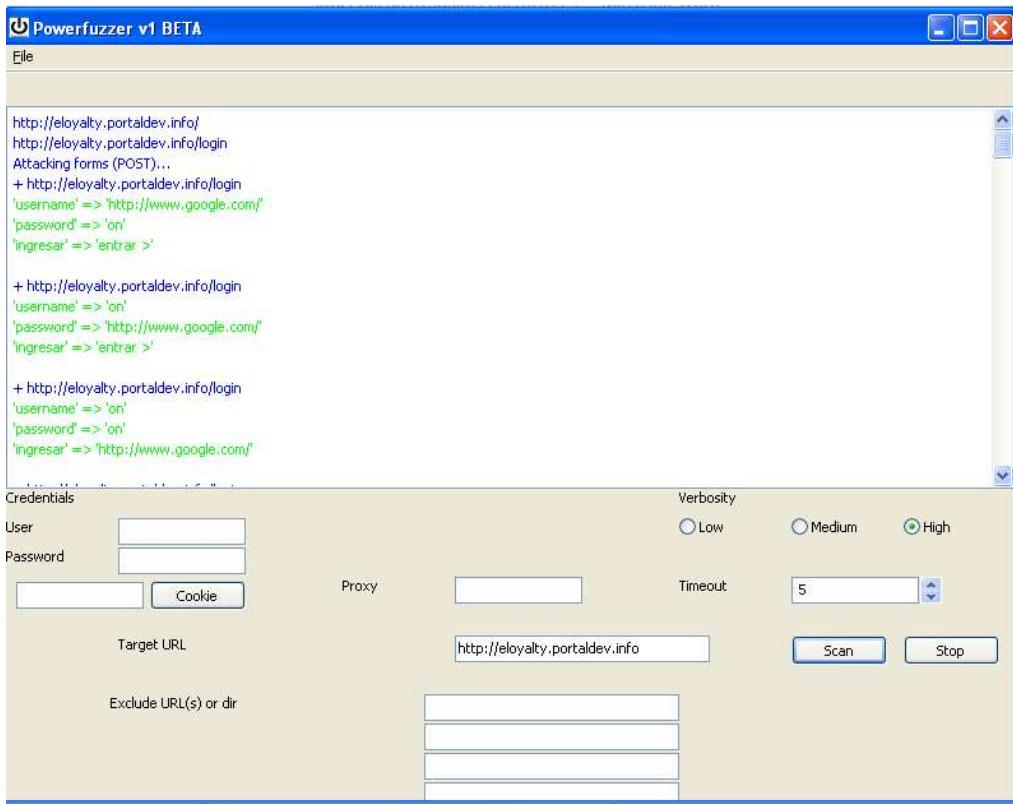


Figura 4.1 Test de seguridad a login del sistema LMP

En el Anexo 4 se listan una serie de herramientas que nos ayudan a realizar pruebas a un sistema.

PRUEBAS DE CAJA NEGRA

El sistema de pruebas de caja negra no considera la codificación dentro de sus parámetros a evaluar, es decir, que no están basadas en el conocimiento del diseño interno del programa. Estas pruebas se enfocan en los requerimientos establecidos y en la funcionalidad del sistema.

Mientras se desarrolló la aplicación realizamos pruebas de caja negra por ejemplo en la pantalla de login mostrar los mensajes de error correctamente como se muestra en la Figura 4.2. En la que se muestra el mensaje de error en caso de ingresar con un usuario/contraseña incorrectos.



Figura 4.2 Error de login en sistema LMP

PRUEBAS DE CAJA BLANCA

Al contrario de las pruebas de caja negra, éstas se basan en el conocimiento de la lógica interna del código del sistema. Las pruebas contemplan los distintos caminos que se pueden generar gracias a las estructuras condicionales, a los distintos estados del mismo, etc.

Debido a que el equipo de desarrollo conoce todas las condiciones del sistema LMP podemos realizar pruebas con diferentes escenarios y tipos de datos de entrada de información en el sistema por ejemplo en el alta de usuarios podemos probar el ingreso de la información con diferentes textos de entrada y en caso de tener algún error corregirlo de manera rápida y ágil. Por ejemplo la Figura 4.3 muestra los errores posibles en el alta de un usuario.

Sucursal

▼

- Este es un campo obligatorio

Nombre de usuario

Contraseña (6-12)

- La confirmación no coincide
- La contraseña debe contener entre 6 y 12 dígitos
- La contraseña debe contener al menos un carácter en mayúscula
- La contraseña debe contener al menos un carácter en minúscula

Confirme contraseña

Firma (6-12)

- Este es un campo obligatorio

Figura 4.3 Errores en alta de Usuarios en el Sistema LMP

IMPLEMENTACIÓN Y MANTENIMIENTO DE SOFTWARE

Después de realizar las pruebas en el sistema, el PROVEEDOR realizó la instalación de la aplicación en producción con el equipo de software y hardware solicitado al CLIENTE. Durante esta etapa el equipo de instalación realizó el deployment de la aplicación ya previamente probada, aun así se realizaron pruebas de caja negra en la aplicación final para asegurar su correcto funcionamiento.

Éste comienza casi de inmediato. El software se libera a los usuarios finales y, en cuestión de días, los reportes de errores se filtran de vuelta hacia la organización de ingeniería de software. En semanas una clase de usuarios indica que el software debe cambiarse de modo que pueda ajustarse a las necesidades especiales de su entorno. Y en meses, otro grupo corporativo, que no quería saber nada del software cuando se liberó, ahora reconoce que puede ofrecerle beneficios inesperados. Necesitará algunas mejoras para hacer que funcione en su mundo.

El reto del mantenimiento del software comienza. Uno se enfrenta con una creciente lista de corrección de errores, peticiones de adaptación y mejoras categóricas que deben planearse, calendarizarse y, al final de cuentas, lograrse. Mucho antes, la fila creció bastante y el trabajo que implica amenaza con consumir los recursos disponibles.

Otra razón del problema de mantenimiento de software es la movilidad del personal. Es probable que el equipo o la persona responsable del software que hizo el trabajo original ya no esté más por ahí. O peor aun otras generaciones de personal de software modificaron el

sistema y se mudaron. Y puede ser que ya no quede alguien que tenga algún conocimiento del sistema heredado.

En el caso de LMP desde que se puso en producción fue sujeto de modificaciones y adaptaciones a procesos que otras áreas de la empresa del cliente necesitaba. La ventaja de la aplicación LMP es que está diseñada para poder realizar cambios de forma rápida y legible para los desarrolladores debido al uso del modelo MVC proporcionado por ZF.

En esta etapa de mantenimiento a la aplicación el equipo del PROVEEDOR da soporte a la aplicación a la fecha debido a que han surgido requerimientos de nuevas funcionalidades además de cuidar la estabilidad del mismo.

Durante este periodo he tenido la oportunidad de aplicar mis conocimientos en desarrollo de software especialmente para aplicar mejoras de funcionalidad las cuales se han acoplado a las nuevas necesidades de la empresa.

Lectura 9. Ingeniería de software I. Manual de prácticas

UNIVERSIDAD VERACRUZANA

Facultad de Estadística e Informática
Licenciatura en Informática

Ingeniería de Software I
Manual de Prácticas

Elaboraron:

Dr. Juan Manuel Fernández Peña
Dra. María de los Ángeles Suman López

Este material fue utilizado en los periodos:

- febrero – julio 2010, agosto 2010 – enero 2011,
- febrero – julio 2011, agosto 2011 – enero 2012
- febrero – julio 2012, agosto 2012 – enero 2013

En el plan 2002 de la Licenciatura en Informática

Agosto 2011

Contenido

<i>Introducción</i>	1
Práctica de herramientas semiautomáticas de apoyo al desarrollo de software.....	1
Modelos para el Análisis de Software	1
Obtención de Métricas de Software	2
Planeación de pruebas se Sistema	2
Modelado de Diseño de Software	2
Planeación y Aplicación de Pruebas de Unidad.....	2
<i>Práctica 1. Introducción a un IDE</i>	3
<i>Práctica 2. Delphi y Bases de Datos</i>	8
<i>Práctica 3. Utilización de una Herramienta CASE</i>	10
<i>Práctica 4. Creación del Modelo Ambiental</i>	14
<i>Práctica 5. Creación del Modelo de Comportamiento</i>	19
<i>Práctica 6. Creación del Modelo de Implementación del Usuario</i>	24
<i>Práctica 7. Cálculo de las Métricas de Análisis</i>	25
<i>Práctica 8. Planeación de Pruebas de Aceptación de Sistema</i>	27
<i>Práctica 9. Creación de Modelos de Diseño</i>	29
<i>Práctica 10. Plan de Pruebas de Integración</i>	31
<i>Práctica 11. Planeación de Pruebas de Unidad</i>	32
<i>Práctica 12. Utilización de una Herramienta para Pruebas de Unidad</i>	33
<i>Práctica 13. Cálculo de Métricas de Diseño</i>	39
<i>Bibliografía</i>	41

Introducción

Dentro del curso de Ingeniería de Software I, por acuerdo de maestros que forman la Academia de Ingeniería de Software I, se incluyeron los siguientes saberes heurísticos:

- Práctica en herramientas de apoyo al desarrollo de software como: IDE y CASE.
- Aplicación de la metodología escogida para la obtención de los modelos de análisis y diseño de un sistema de software utilizando el CASE e IDE practicados.
- Cálculo de métricas de calidad aplicadas a sistemas de software.
- Planeación de Pruebas de sistema aplicadas al sistema de software en desarrollo.
- Planeación y aplicación de pruebas de unidad para software realizado por cada alumno utilizando una herramienta automática para la aplicación de la prueba.

Cada saber heurístico ocupa uno o varios saberes teóricos y se adquieren realizando trabajos prácticos. El presente manual describe las prácticas correspondientes a la adquisición de los saberes mencionados.

Las prácticas que se incluyen en este manual son de suma importancia para el futuro profesional de un ingeniero de software.

Práctica de herramientas semiautomáticas de apoyo al desarrollo de software

El desarrollo moderno de software utiliza herramientas automáticas y semiautomáticas que apoyan la labor del Ingeniero de Software. Entre ellas están:

- IDE (Integrated Development Environment) que ayudan en la preparación de y codificación de los programas que conforman un sistema, ello mediante un ambiente que contiene elementos tales como: lista de componentes reutilizables, editor de código, revisión sintáctica, depuradores, formación de proyectos y una variedad de herramientas que varía de un IDE a otro.
- CASE (Computer Aided Software Engineering) que facilitan el desarrollo de modelos de desarrollo de software.
- Herramientas de prueba.

Modelos para el Análisis de Software

Al finalizar la serie de prácticas de esta parte del curso el alumno:

- Habrá aprendido a realizar el análisis de sistemas de software basados en la metodología “Análisis Estructurado Moderno” de Edward Yourdon [Yourdon, E. (1993)]
- También será consciente de todos los elementos que se deben considerar para entender lo que se pide en el desarrollo de software.

Las prácticas que forman esta parte del curso se llevan a cabo de manera paralela y arrojan una serie de modelos que son: esencial, de comportamiento y de implantación del usuario.

Obtención de Métricas de Software

Esta competencia está directamente relacionada con la calidad de un software ya que son las métricas las que de alguna manera van ayudando a entender tanto los procesos como la calidad que del software se requiere. En este manual se plantea el cálculo de las siguientes métricas:

- Métrica Bang. Utilizada, principalmente, para revisar la completez de los modelos de análisis.
- Métrica de Puntos de Función. Sirve para establecer tanto los restricciones del software que se va a realizar como el esfuerzo que se requerirá para realizar su implementación.
- Métricas Card y Glass para estructura
- Métricas de módulos: Cohesión, acoplamiento y complejidad. Todas ellas dan información sobre un módulo como qué tan integrado está, si es independiente de otros módulos y cuántas deberán las pruebas mínimas que se le deberán aplicar a un módulo.

Planeación de pruebas se Sistema

En la FEI las actividades asociadas a la prueba de software se llevan a cabo conforme se va avanzando en el desarrollo del software, primero se realiza la planeación de las pruebas y luego se aplican. En el nivel de prueba de sistema la planeación ocurre paralelamente con el análisis. Lo anterior lleva varias ganancias: ayuda a entender con mayor precisión qué se quiere del software y el cliente tiene elementos que a futuro le servirán para revisar que el software entregado es lo que pidió.

Modelado de Diseño de Software

El diseño de software es la actividad ingenieril por excelencia dentro del desarrollo de software, durante el diseño se deberán considerar aspectos tales como las plataformas de hardware y software donde correrá el nuevo software; las estructuras generales y detalladas y la forma de cumplir con las restricciones impuestas.

Planeación y Aplicación de Pruebas de Unidad

Una de las últimas tareas que comprende el desarrollo de software es la aplicación de pruebas de unidad. Como la unidad es la parte más pequeña de la estructura del software, generalmente hay muchas, por ello es importante que el desarrollador se apoye en herramientas automáticas que le permitan acelerar la tarea de prueba. En este manual se presenta la práctica de una de tales herramientas: DUnit.

Práctica 1. Introducción a un IDE

I. Objetivo: El alumno entenderá la forma de trabajo de un IDE y lo manejará para construir una pequeña aplicación si incluir archivos y Bases de Datos.

II. Equipo necesario:

- Computadora con MS-Windows.
- Una herramienta IDE, preferentemente Delphi de Borland.
- Libros del lenguaje, preferentemente Object Pascal, y la herramienta IDE que lo manipula.

III. Material de apoyo:

- Ejemplo con las unidades que constituyen el la aplicación.
- Otros ejemplos de programas que vienen con el IDE

IV. Procedimiento:

1. Abrir el IDE dando clic en su ícono en el escritorio o buscándolo en *Todos los programas* del menú *Inicio* de MS-Windows
2. Escoger crear un nuevo proyecto que incluya una unidad de tipo Forma: VCL Project (Projet>New>VCL Project).
3. Agregar los componentes necesarios como: campos de edición, botones, menús, para crear una ventana como en la Figura 1. Oprimir la tecla F12 para ver el código, resultará parecido al de la Figura 2 (no lo cambie).
4. Modificar los atributos de cada componente utilizando la barra de atributos que aparece a la izquierda, después de dar un clic sobre la componente deseada.
 - Escribir en Caption de Button1 “Verificar”
 - Escribir en Caption de Button2 “Calcular”
5. Dar doble clic en el botón Verificar e incluir el código de la Figura 3
6. Dar doble clic en el botón Calcula e incluir su funcionalidad asociada como en la Figura 4.
7. Crear una nueva unidad: Project>New>Unit
8. Usar el código de la Figura 5 e incluirlo en la unidad recién creada.

V. Resultados esperados:

Programa pequeño sin uso de archivos que calculará el impuesto (ISR) de una cantidad dada.

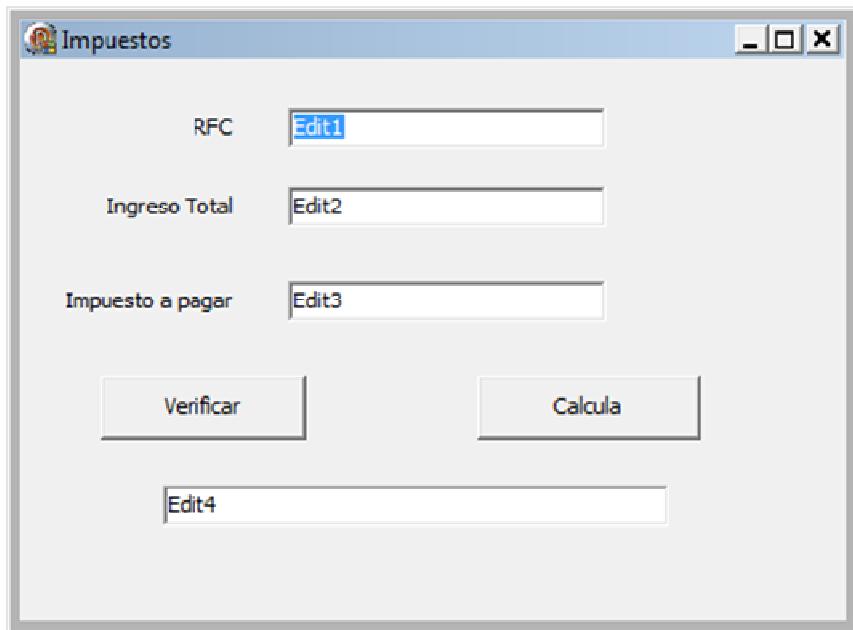


Figura 1. Ventana para el Programa sobre el cálculo de ISR

```

unit IUImpuestos;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm2 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Button1: TButton;
    Button2: TButton;
    Edit4: TEdit;
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form2: TForm2;

implementation
uses ucalcula;

{$R *.dfm}
// AQUI VA EL CÓDIGO DE LOS DOS BOTONES (FIGURAS 2-3 Y 2-4)

end.

```

Figura 2. Código de la Ventana que inserta Delphi automáticamente

```

procedure TForm2.Button1Click(Sender: TObject);
var calcimp:pagoimpue; resp:integer;
begin
  calcimp := pagoimpue.create;
  Edit4.Text := 'Validando ...';
  resp:=calcimp.vale(Edit1.Text);
  if (resp=0) then Edit4.Text := 'RFC parece válido'
    else
      if (resp=1) then Edit4.Text := 'RFC inválido; faltan o sobran caracteres'
        else
          if (resp=2) then Edit4.Text := 'RFC inválido; revise letras iniciales'
            else
              if (resp=3) then Edit4.Text := 'RFC inválido; revise fecha de nacimiento';
end;

```

Figura 3. Código del Botón 2 (Verificar)

```

procedure TForm2.Button2Click(Sender: TObject);
var calcimp:pagoimpue; impudebido: single;
begin
  calcimp := pagoimpue.create;
  if (StrToFloat(Edit2.Text)<0) then
    Edit4.Text := 'El total de ingresos debe ser positivo'
  else
  begin
    impudebido := calcimp.impuesto(StrToFloat(Edit2.Text));
    Edit3.Text := Format('%f',[impudebido] );
    Edit4.Text := '';
  end;
end;

```

Figura 4. Código asociado al Botón 1 (Calcula)

```

unit ucalcula;

interface
  type pagoimpue = class
    function vale(rfc:String):integer;
    function impuesto(tot:single):single;
    constructor Create;
  end;
implementation
uses sysutils;
  var Tabla: array[0..7,0..3] of single;
  // Constructor: prepara la tabla de impuestos
  constructor pagoimpue.Create;
  begin
    //cargar tabla 2009
    Tabla[0,0] := 0.01; Tabla[0,1] := 5952.84 ; Tabla[0,2] := 0.0; Tabla[0,3] := 0.0192;
    Tabla[1,0] := 5952.85; Tabla[1,1] := 50524.92; Tabla[1,2] := 114.24; Tabla[1,3] := 0.0640;
    Tabla[2,0] := 50524.93; Tabla[2,1] := 88793.04; Tabla[2,2] := 2966.76; Tabla[2,3] := 0.1088;
    Tabla[3,0] := 88793.05; Tabla[3,1] := 103218.00; Tabla[3,2] := 7130.88; Tabla[3,3] := 0.1600;
    Tabla[4,0] := 103218.01; Tabla[4,1] := 123580.20; Tabla[4,2] := 9438.60; Tabla[4,3] := 0.1792;
    Tabla[5,0] := 123580.21; Tabla[5,1] := 249243.48; Tabla[5,2] := 13087.44; Tabla[5,3] := 0.1994;
    Tabla[6,0] := 249243.49; Tabla[6,1] := 392841.96; Tabla[6,2] := 38139.60; Tabla[6,3] := 0.2195;
    Tabla[7,0] := 392841.97; Tabla[7,1] := 0.0; Tabla[7,2] := 69662.40; Tabla[7,3] := 0.2800;
  end;
  // Verificación de rfc
  function pagoimpue.vale.rfc(String ):integer;
  var err, ix:integer; ll:integer; lim :integer; rfcu:string;
  begin
    err := 0;
    rfcu := UpperCase(rfc);
    ll := Length(rfcu);
    if (ll<12) or (ll>13) then
      err :=1;
    if (ll=12) then lim := 3 else lim := 4;
    for ix := 1 to lim do
      if (not (rfcu[ix] in ['A'..'Z'])) then
        err := 2;
    for ix := lim+1 to ll do
      if (not (rfcu[ix] in ['0'..'9'])) then
        err :=3;
    if (err <3) then

```

```

begin
  if (rfcu[lim+3] in ['2'..'9']) or (rfcu[lim+5] in ['4'..'9']) then
    err := 3;
  end;
  vale :=err;
end;
//
// Cálculo de impuestos
//
function pagoimpue.impuesto(tot:single):single;
var iz:integer; ix:integer;
begin
  //calculo
  ix := 7;
  for iz:= 6 downto 0 do
    if (tabla[iz,1]>=tot) then
      ix := iz;
  impuesto := Tabla[ix,2] + (tot - Tabla[ix,0])*Tabla[ix,3];
end;
end.

```

Figura 5. Código de las Funciones para Cálculo del Impuesto

Práctica 2. Delphi y Bases de Datos

I. Objetivo. En esta práctica se desarrollará una aplicación que permite consultar una base de datos que incluye imágenes, usando elementos predeterminados de Delphi y una base de datos proporcionada por Borland.

II. Equipo necesario:

- Computadora con MS-Windows.
- Una herramienta IDE, preferentemente Delphi de Borland, versión de 7, 8, Studio o Turbo.
- Libros del lenguaje, preferentemente Object Pascal, y la herramienta IDE que lo manipula.

III. Material de apoyo:

Opcionalmente, una imagen en formato BMP.

IV. Procedimiento:

Se procederá como sigue:

- a) Se crea un proyecto VCL (vea práctica anterior); se le hacen adaptaciones al título de la forma en el Object Inspector (ver ventana izquierda inferior).
- b) Se agrega un componente *menú general* de la Tool Palette (ver ventana inferior derecha) y dando doble clic en el componente agregar dos opciones del mismo nivel: *Tabla* y *Cerrar*
- c) En la opción Tabla se agregan dos subopciones: Actualizar y Buscar.
- d) Desde el SO cree una carpeta con el nombre del proyecto y en ésta salve todas sus archivos usando la opción Save All del menú File de Delphi vaya nombrando los archivos como sigue:
 - a. La forma con el nombre *MenúInicial* y
 - b. El Proyecto con el nombre *Practica2*.
- e) Se crea una nueva unidad del tipo Data Module File>New -> Other -> Data Module. Se agregan la componentes TTable y se actualizan sus atributos como sigue:
 - a. TTable se conecta, usando el Object Inspector y el atributo Database>DataBasename a una base de datos llamada DBDEMONS localizada en: C:\Archivosdeprogramas\ArchivosComunes\BorlandShared\Data
 - b. Se escoge la tabla: Animals.dbf poniendo este nombre en el atributo Tablename;
 - c. Se usa como campo índice NAME y
 - d. Se inicia el atributo Active en true (ver Object Inspector).
- f) También se agrega al DataModule la componente DataSource y se actualizan sus atributos:
 - a. DataSource, se conecta a Table1 usando el atributo DataSet.
 - b. Se salva el DataModule con el nombre *Datos*.
- g) Se crea una forma (File>new -> Form) que se empleará de ventana para la consulta. Se le agrega un componente DBNavigator, un DBGrid y un DBImage. En Uses (dentro del código de la unidad) se agrega el nombre

- del módulo de datos (*Datos*) y se conectan los tres elementos al Data Access usando el Object Inspector. Se salva con el nombre *Consulta*.
- h) De la misma manera se crea una forma de ventana para búsqueda, pero a ésta se le agregan: un campo Edit, un botón y un campo DBImage. En *Uses* se agrega el nombre del módulo de datos y se conecta el DBImage con el acceso. Se salva con el nombre *Busca*.
 - i) En el código de la ventana principal (Form1) se agregan, en *Uses*, los nombres de las dos unidades (*Consulta* y *Busca*). En los menús correspondientes se generan eventos que hacen visibles a las ventanas (cada una por separado). En el menú cerrar se cierra la forma. Se salva.
if not (Datos.DataModule1.Table1.FindKey([Edit1.Text]) then Edit1.Text:= 'No se encuentra ese animal');
 - j) En la ventana de búsqueda se crea un evento asociado al botón que permita leer un nombre del campo Edit y luego lo busque en la base de datos con la siguiente instrucción y se salva.

V. Resultados esperados:

Un programa ejecutable hecho en Object Pascal y que muestra el uso de diversas componentes como: un menú con las opciones de actualizar archivo, buscar registro, cerrar; un navegador de una tabla (archivo), una rejilla. Además se verá como se enlazaron las unidades con la clases tipo FORM.

Práctica 3. Utilización de una Herramienta CASE

I. Objetivo:

Conocer el manejo de una herramienta CASE mediante la introducción de modelos de análisis y diseño de un sistema de software de ejemplo.

II. Equipo necesario:

- Computadora con MS-Windows.
- Una herramienta CASE, preferentemente MS-Visio.

III. Material de apoyo:

- Diagramas que representan los modelos de análisis y diseño de un sistema de software, mismos que proporcionará el profesor.

IV. Procedimiento:

1. Buscar la herramienta MS-Visio en el menú Inicio y de allí en Microsoft Office y abrirlo
2. Para iniciar cada uno de los diagramas que le dará el profesor haga:
 - Para el DFD (Diagrama de Flujo de Datos, escoger: Archivo>New>Diagrama de flujo>Diagrama de flujo de datos).
 - Para el DE (Diagrama de Estructura), escoger: Archivo>New>Software y Base de Datos>Estructura de Programas
 - Para el DE (Diagrama de Estructura), escoger: Archivo>New>Software y Base de Datos>Modelo de Bases de Datos
3. Una vez elegido el tipo de diagrama que realizará, se deben utilizar sólo los iconos que aparecen en la ventana *Formas* (a la izquierda de la pantalla), posicionándose con el ratón sobre el símbolo o ícono deseado y arrastrarlo a la ventana central donde aparece una página cuadrícula donde se conformará el dibujo.
4. Al finalizar cada diagrama de la orden de guardar como en cualquier herramienta de MS-Office.

V. Resultados esperados:

Una serie de diagramas introducidos a la herramienta CASE utilizada y el conocimiento del empleo de la misma.

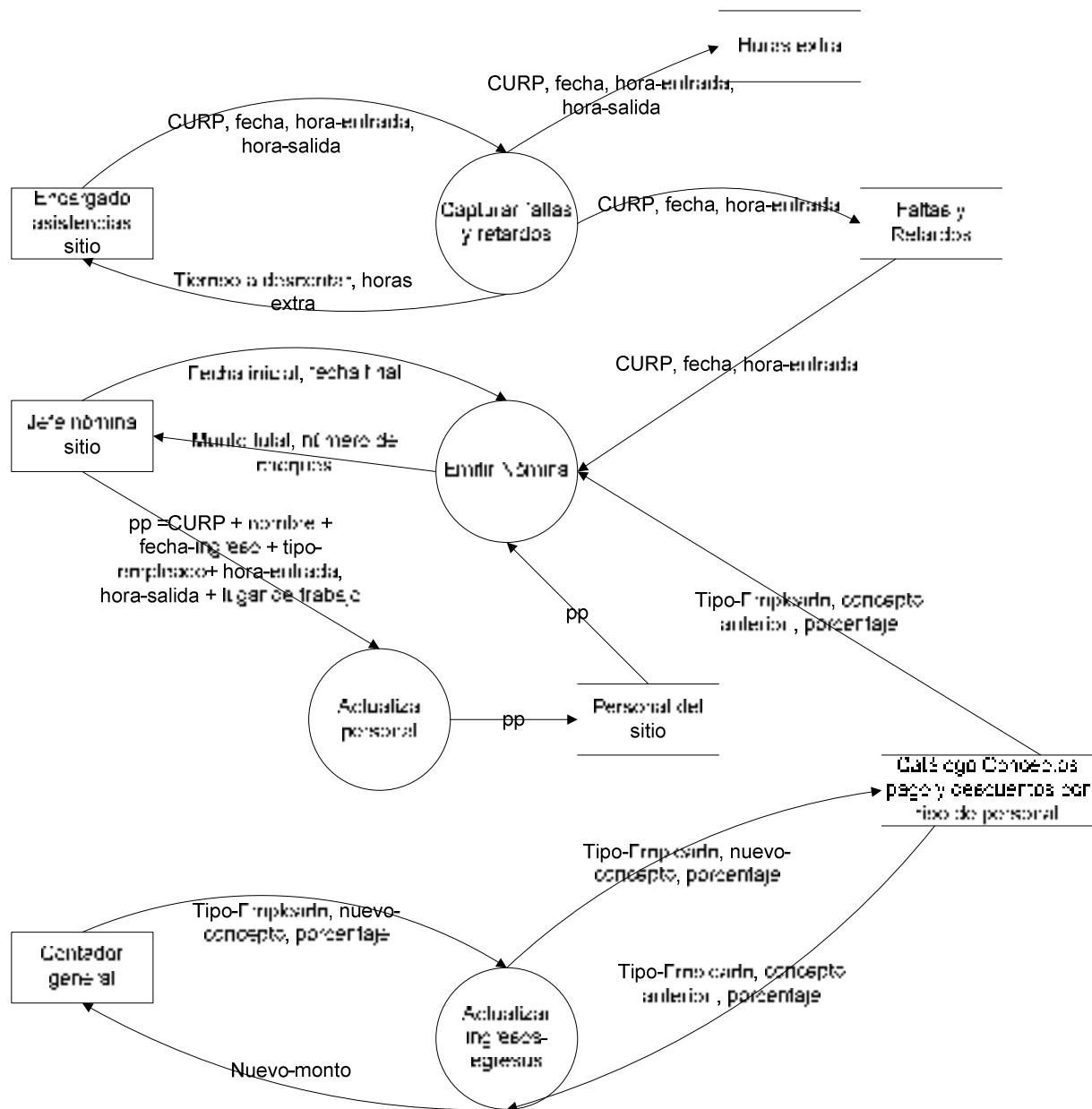


Figura 6. Ejemplo de Diagrama de Flujo de Datos

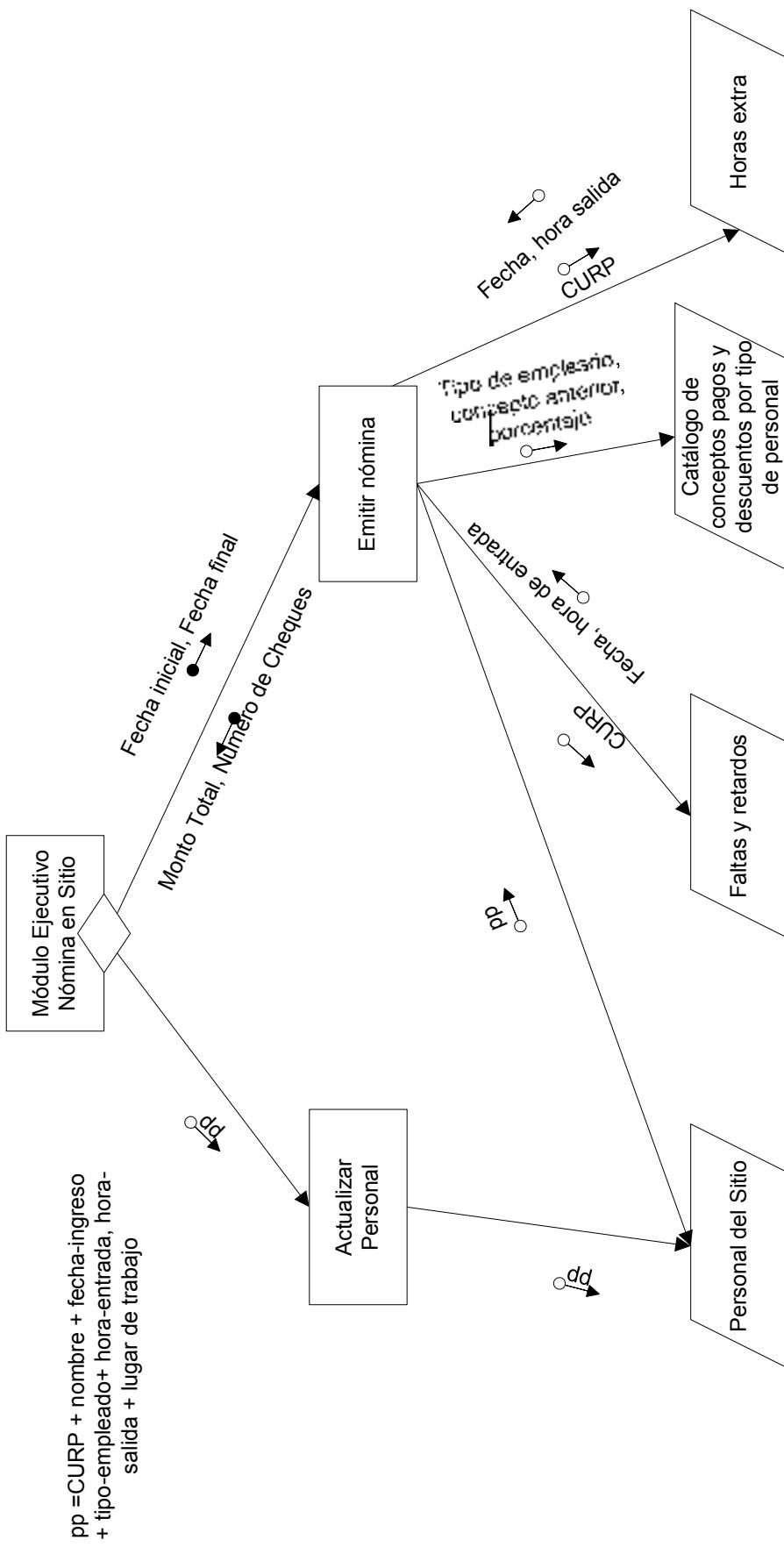


Figura 7. Ejemplo de Diagrama de Estructura

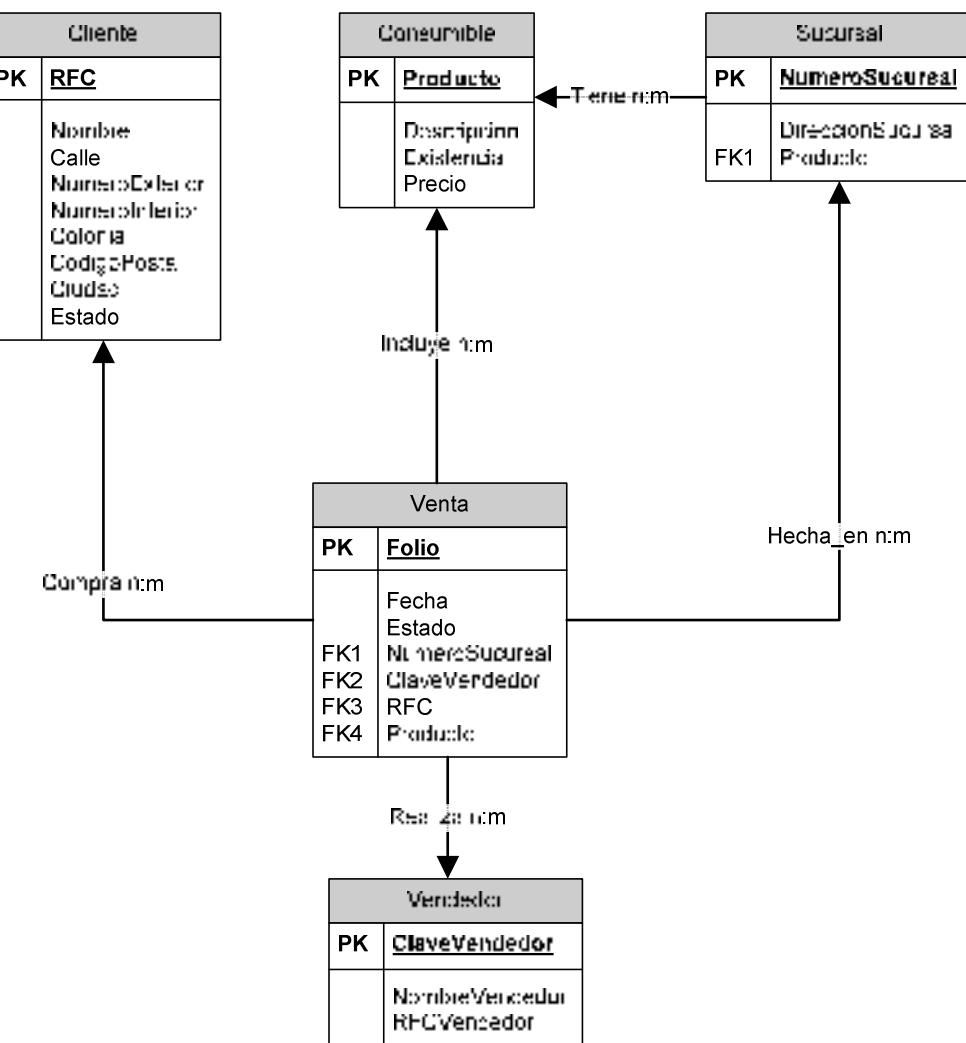


Figura 8. Modelo Entidad – Relación del Sistema de venta de Consumibles

Práctica 4. Creación del Modelo Ambiental

I. Objetivo:

Crear y poner en un archivo de documento el *modelo ambiental* de un sistema de software.

II. Equipo necesario:

- Una PC con el SO MS-Windows
- MS-Office, incluidos Word y Visio

III. Material de apoyo:

- La explicación oral, por parte del profesor, de un problema para ser resuelto mediante un sistema automatizado
- El libro de la metodología estudiada [Yourdon, E. (1993)]
- Las diapositivas sobre la metodología que se encuentran en la página www.uv.mx/asumano
- Un cuaderno para apuntes.
- Ejemplos de otros modelos ambientales dados por el profesor

IV. Procedimiento:

NOTA: Esta práctica puede llevar más de una sesión y éstas pueden ser en el salón de clase o en el Centro de Cómputo.

1. En su cuaderno realice los tres elementos que conforman el modelo ambiental
2. Revise que sean considerados correctamente los tres elementos
 - a. **Para la declaración de propósitos.** Que en un párrafo se listen todas las funciones que se desean del nuevo software, se entienda quienes serán los clientes y usuarios del nuevo sistema y cual sería el papel de cada uno dentro de éste. Además debe incluir los posibles sistemas externos con que interactuará el mismo.
 - b. **Para el Diagrama de Flujo de Datos.** Dibujar una sola burbuja con el nombre del nuevo sistema. A partir de la burbuja conectar flujos (flechas) que incluyan paquetes de datos y que entren o salgan a terminadores (usuarios o sistemas externos) o archivos (a este nivel como bases de datos que incluyan gran parte de la información que se manejará).
 - c. **Para la lista de acontecimientos.** Cada enunciado de la lista debe tener la forma: Usuario + función + objeto directo + complemento.
3. Abra MS-Visio, escoja la opción de DFD y dibuje el diagrama de contexto.
4. Usando el modelo ambiental que se hizo previamente en el salón de clase y el dibujo del DFD de contexto hecho en Visio:
 - a. Abra MS-Word y escriba la declaración de propósitos,
 - b. Copie de MS-Visio el DFD de contexto y péguelo en MS-Word usando Edición>pegado especial>Imagen (metarchivo mejorado) y luego, de ser necesario, dar Formato>Imagen>Diseño>en línea con el texto, con lo que se asegura que la imagen no cambié de posición al modificar el documento.

c. Finalmente, agregue en Word a lista de acontecimientos.

V. Resultados esperados:

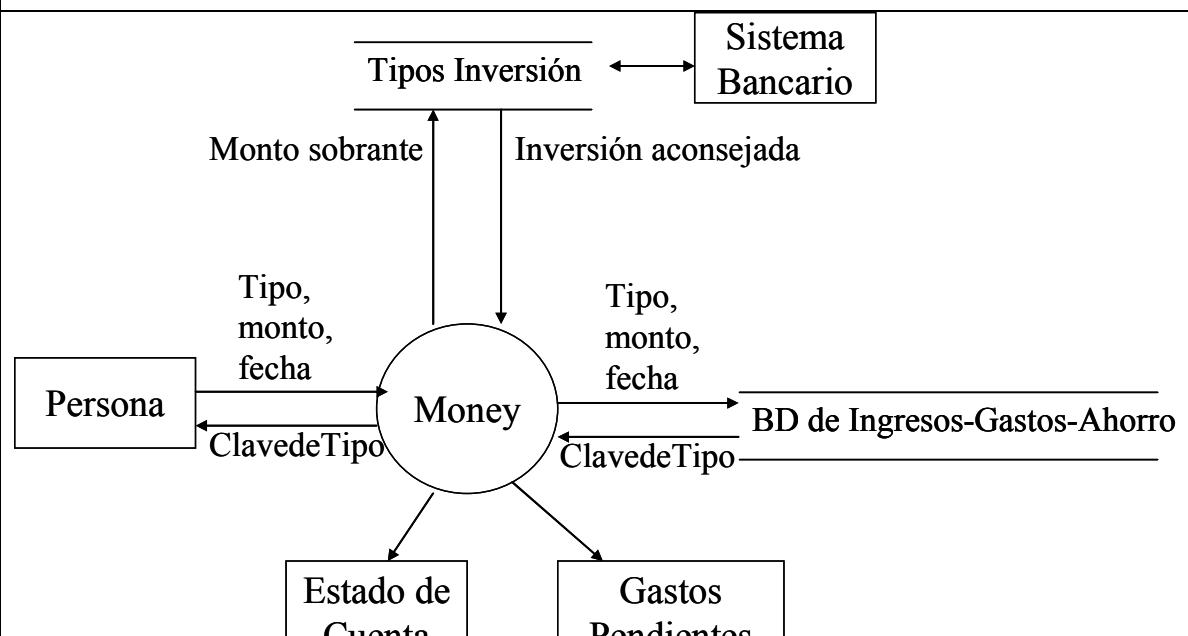
Un documento Word con los elementos que conforman el modelo ambiental: declaración de propósitos, DFD de contexto y Lista de Acontecimientos.

Ejemplos de modelo ambiental

Ejemplo 1 Sistema Money

Declaración de propósitos:

- Se requiere hacer un sistema que ayude a las personas a llevar un control de sus gastos e ingresos y que, además, le calcule que ahorro le quedó y en donde le conviene invertir éste con base en los tipos de inversión que reporta el Banco de México a través de su Sistema Bancario. La forma en que el usuario del sistema Money utilizará la opción de inversión será utilizando Internet, mientras que el registro de ingresos y egresos se realizaría de manera local.



Lista de Acontecimientos:

1. Persona registra ingreso
2. Persona registra egresos o gastos.
3. Persona actualiza tipos de ingresos y gastos.
4. Persona solicita cálculo de cantidad disponible para ahorro.
5. Persona solicita tipo de inversión con base en su ahorro del mes.

Ejemplo 2. Sistema de cajero Automático

Declaración de propósitos:

Se quiere realizar un sistema que permita al *Banco de la Ilusión*, prestar servicios de consulta de saldo y retiro de efectivo de sus cajeros automáticos (ATM). Éstos mismos servicios los deberá brindar a usuarios de otros bancos con un cargo adicional, mismo que se le cobrará de manera inmediata de la cuenta que se esté manejando.

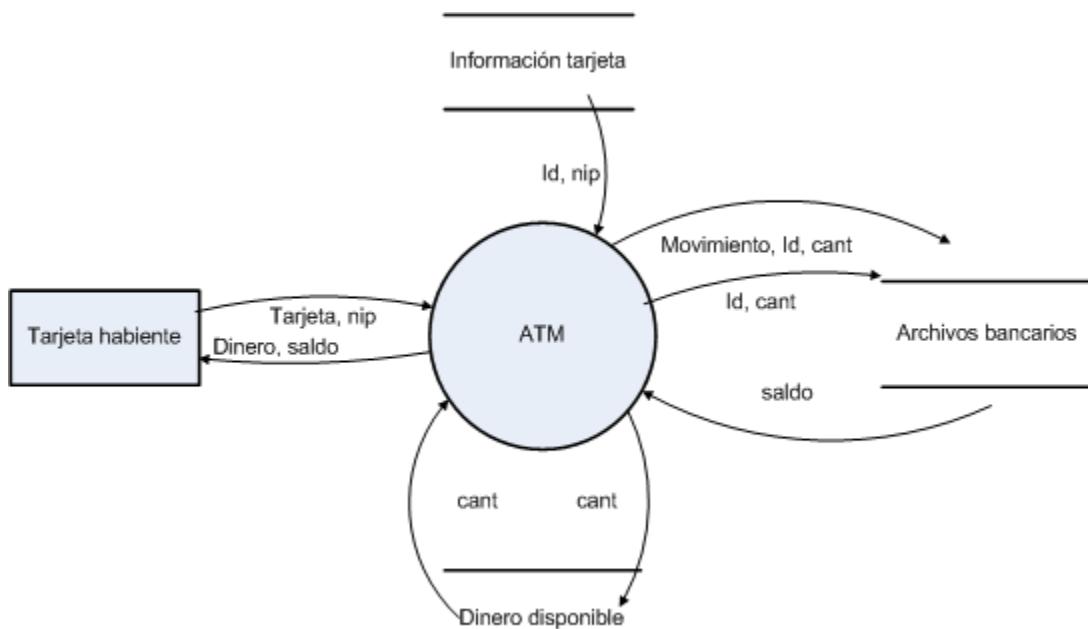


Figura 10. Diagrama de Contexto del ATM

Lista de Acontecimientos:

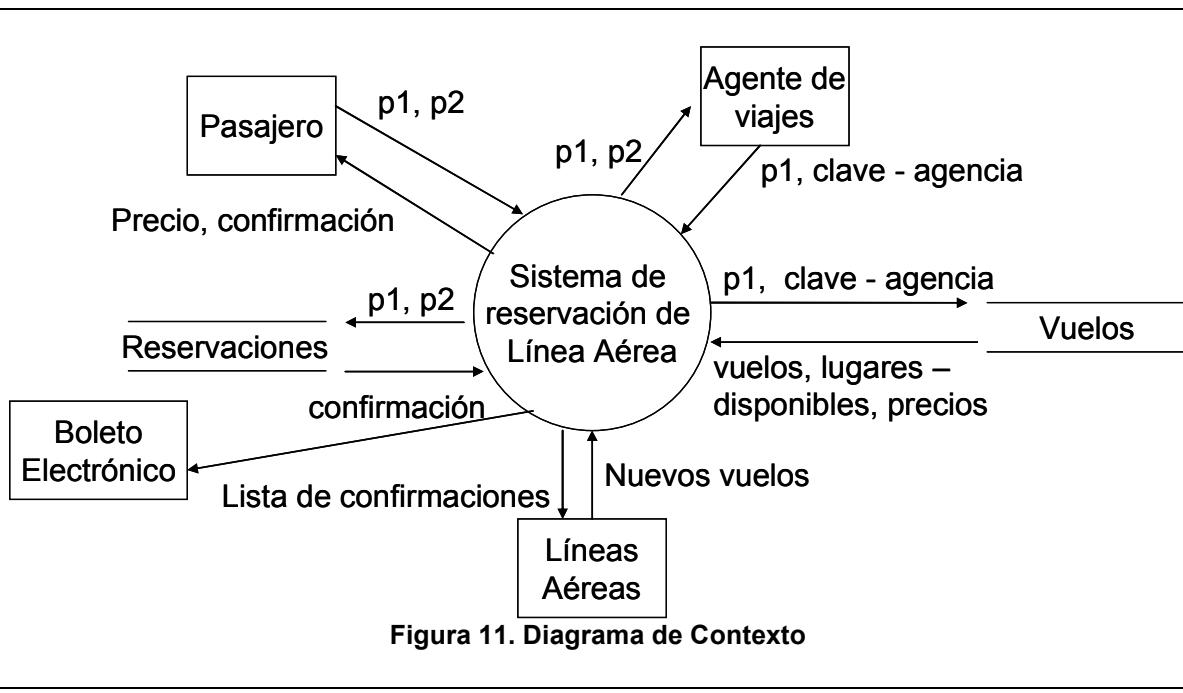
1. Usuario ingresa tarjeta bancaria
2. Usuario se autentifica tecleando su NIP
3. Usuario solicita saldo
4. Usuario solicita retiro
5. Banco obtiene saldo
6. Banco autoriza movimiento
7. Banco registra movimiento

Ejemplo 3. Diccionario del Español Mexicano

Tabla 1

Declaración de propósitos:

La Asociación Internacional de Vuelos Comerciales (AIVC) ha decidido realizar un sistema de apartado y venta de boletos por Internet (SAVBI). Para ello, se permite que un pasajero elija la fecha, hora y destino deseado (p1) y lo envíe vía Internet a un Agente de Viajes (AV) de su país que esté registrado en la AIVC y que debe tener un servidor automático de recepción de peticiones, mismo que a su vez manda la petición del usuario al servidor de la AIVC que revisa si existen vuelos y lugares disponibles de acuerdo a la petición del pasajero y los envía al servidor de la AV. Una vez que el pasajero está conforme con el vuelo y precio, éste debe enviar lo contenido en p2 que es: número de boletos deseados, datos de su tarjeta de crédito (tipo - VISA o MasterCard -, fecha de caducidad y número; después de ello el pasajero recibirá su confirmación mediante un boleto electrónico que el pasajero debe imprimir para llevar el día de su vuelo al aeropuerto donde le será canjeado por pases de abordar. El SAVBI debe tener actualizado los vuelos y su información asociada todo el tiempo y por ello las líneas aéreas deben estarlo actualizando.



Lista de Acontecimientos:

1. Pasajero solicita viaje con datos necesarios
2. Agencia de viajes revisa disponibilidad
3. Líneas aéreas responden disponibilidad del viaje
4. Pasajero manda datos para pagar boleto
5. Pasajero imprime boleto
6. Agencia de viaje actualiza disponibilidad
7. Líneas aéreas actualizan nuevos vuelos.

Práctica 5. Creación del Modelo de Comportamiento

I. Objetivo:

Crear y poner en un archivo de documento el *modelo de comportamiento* de un sistema de software.

II. Equipo necesario:

- Una PC con el SO MS-Windows
- MS-Office, incluidos Word y Visio

III. Material de apoyo:

- Un problema para ser resuelto mediante un sistema automatizado
- El modelo ambiental del sistema
- El libro de la metodología estudiada [Yourdon, E. (1993)]
- Las diapositivas sobre la metodología que se encuentran en la página www.uv.mx/personal/asumano
- El cuaderno para apuntes que sobre los modelos han realizado los alumnos.
- Dos ejemplos completos que permitirán ver como se desarrollarán los sistemas de los alumnos.

IV. Procedimiento:

NOTA: Esta práctica puede llevar más de una sesión y éstas pueden ser en el salón de clase o en el Centro de Cómputo.

1. En su cuaderno realice los elementos necesarios para su modelo de comportamiento
 - a. DFD nivel cero que será generado a partir de la lista de acontecimientos descrita en el modelo ambiental.
 - b. El DFD nivel uno
 - c. DER asociado al DFD de nivel uno.
 - d. DTE del sistema
 - e. Diccionario de Datos (DD).
2. Revise que sean considerados los puntos de la Tabla 2-2.
3. Usando MS-Visio dibuje los DFD, DER y DTE necesarios
4. Abra el documento Word que contiene el modelo ambiental y agregue los diagramas del modelo de comportamiento.
5. Para cada diagrama escriba una explicación que permita al usuario entender lo que se está planteando en ellos.
6. Realice o actualice el DD.

Tabla 1. Elementos a revisar en los diagramas del Modelo de Comportamiento

Elemento del MC	Características a revisar
Burbujas del DFD nivel cero	Deben ser tantas burbujas como enunciados de la lista de acontecimientos. Cada burbuja se numerará a partir de uno se nombrará describiendo la respuesta que el sistema debe dar al acontecimiento asociado
Burbujas del DFD nivel 1	Deben numerarse de acuerdo a la lista de acontecimientos seguidos de punto y otro número secuencial. Ejemplo: para el acontecimiento 1, las burbujas tendrá los números 1.1, 1.2, 1.3, etc.
Diagrama Entidad Relación	Sólo debe haber entidades que aparecen en el DFD nivel 1. Debe marcarse la cardinalidad y el nombre de las relaciones que haya entre entidades.
Flujos de datos	Cada flecha que entra o sale de una burbuja debe tener encima los datos que transporta (en paquete o desglosados)
Almacenes	Cada almacén se nombrará con una frase nominal que exprese lo que almacenan y que coincida con alguna entidad o relación del DER
DTE	Tiene estado inicial y final, se han definido todos los estados, se puede llegar y salir a o de todos los estados, cada estado responde al sistema adecuadamente a todas las condiciones posibles.

V. Resultados esperados:

Un documento Word con los elementos que conforman los modelos: Ambiental corregido y de Comportamiento.

Ejemplos de Modelos de comportamiento: Sistema Money

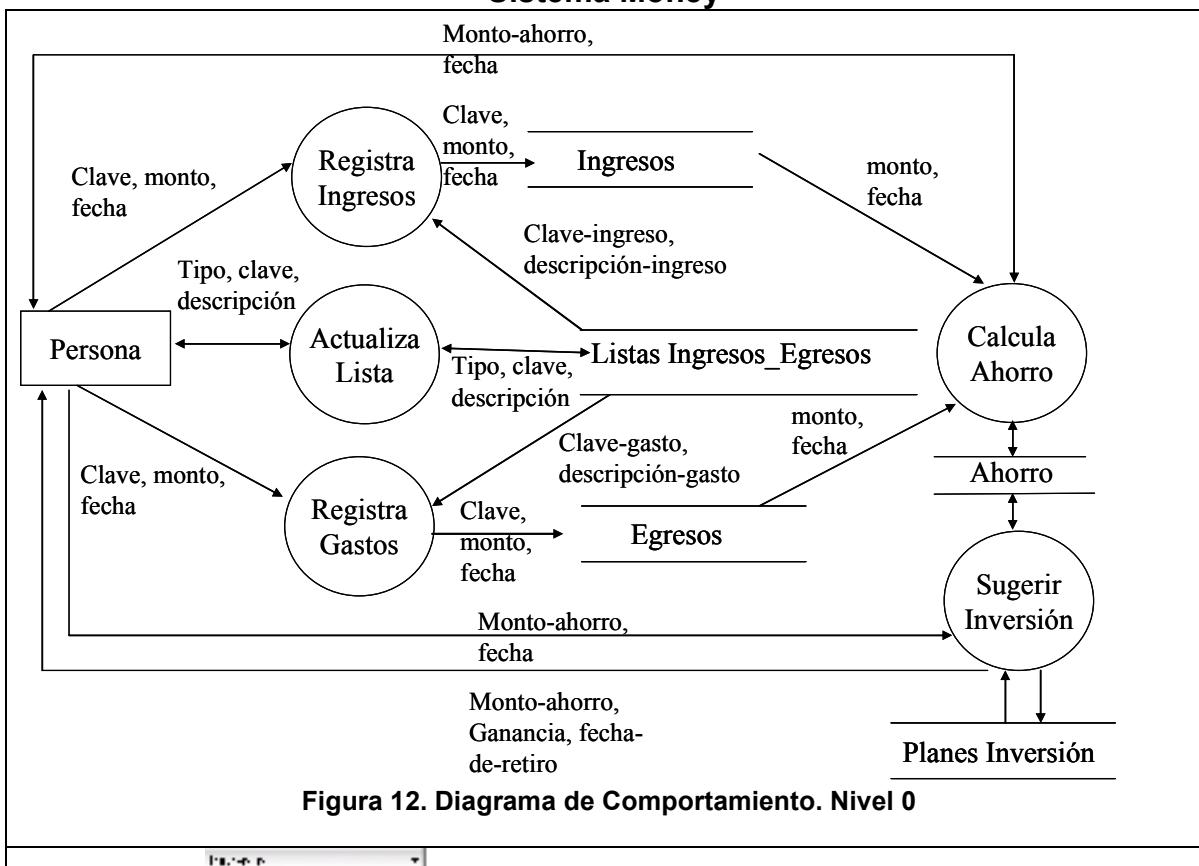


Figura 12. Diagrama de Comportamiento. Nivel 0

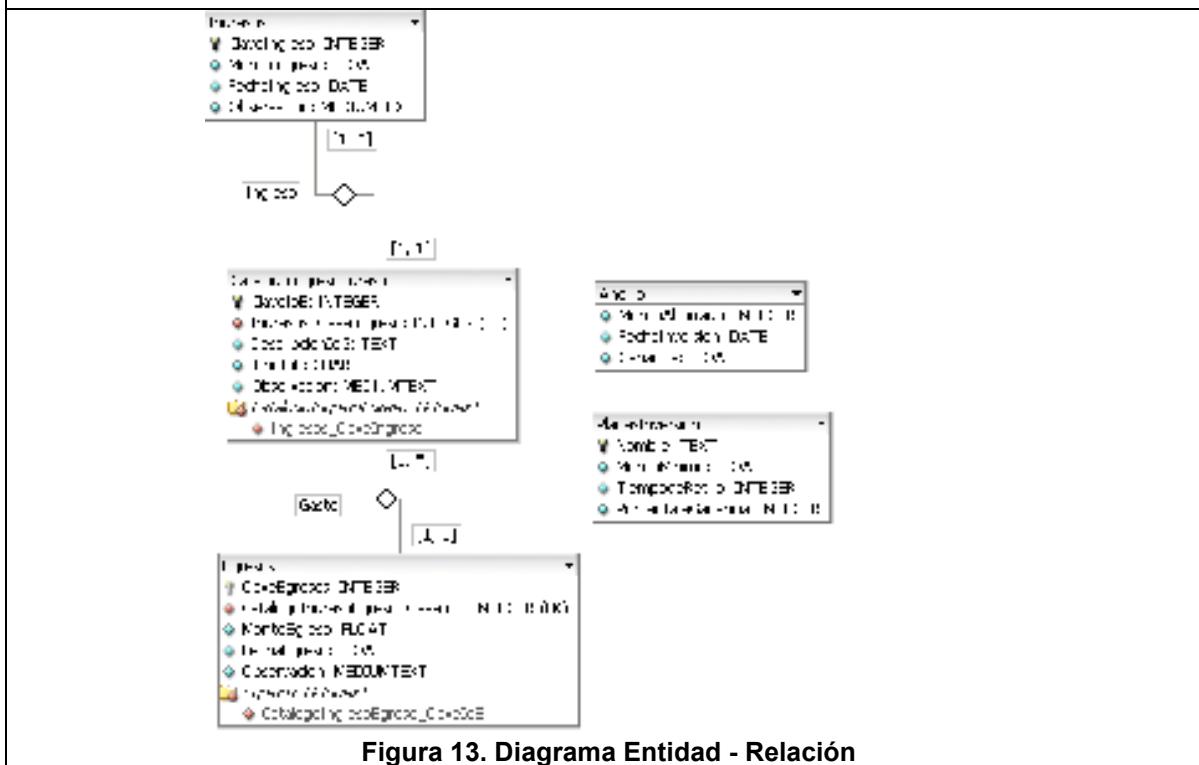
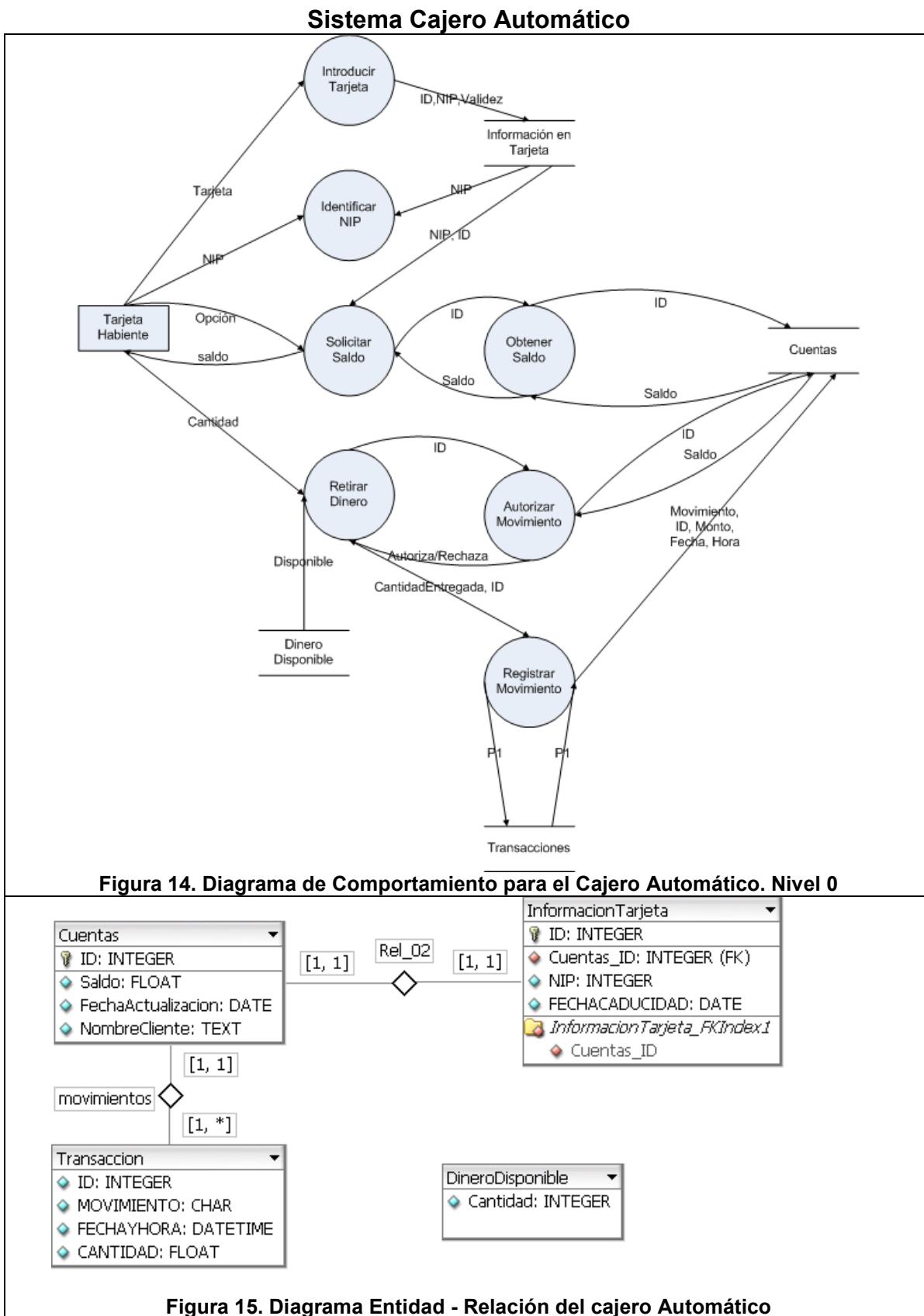
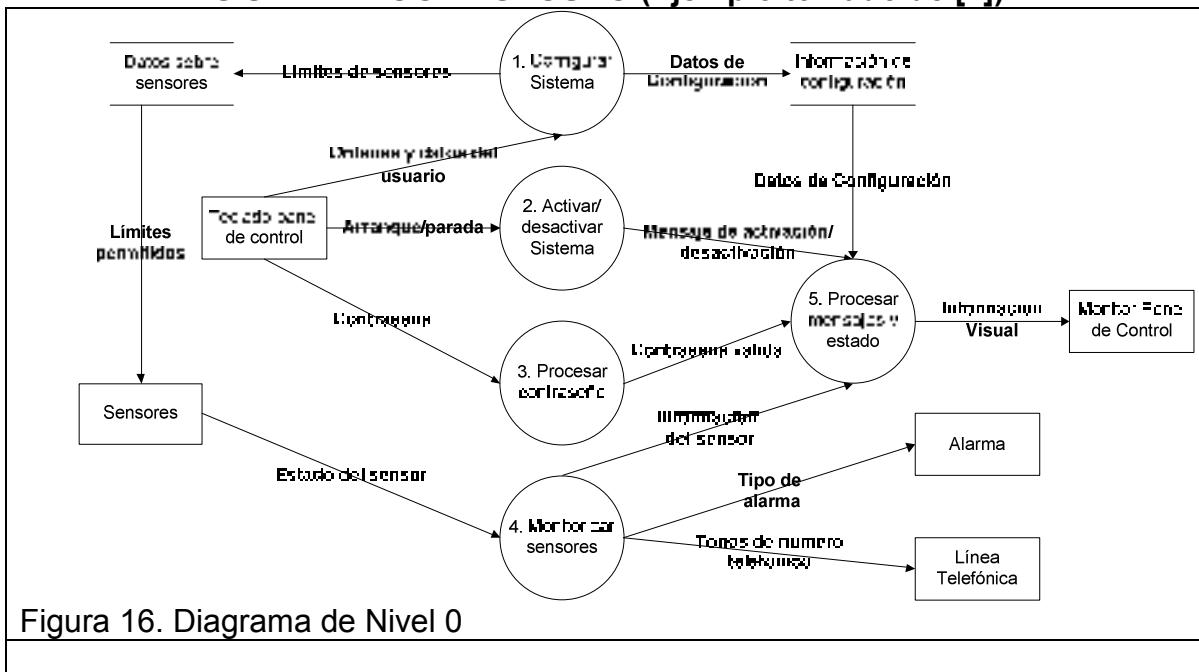


Figura 13. Diagrama Entidad - Relación



SISTEMA HOGAR SEGURO (Ejemplo tomado de [4])



Práctica 6. Creación del Modelo de Implementación del Usuario

I. Objetivo:

Realizar usando un archivo de documento el *modelo de implementación del usuario* de un sistema de software en versión de Manual Preliminar del Usuario.

II. Equipo necesario:

- Una PC con el SO MS-Windows
- Conexión a Internet
- MS-Office, incluidos Word
- IDE Delphi (cualquier versión)

III. Material de apoyo:

- El modelo de comportamiento del software que se está desarrollando.
- El libro de la metodología estudiada [Yourdon, E. (1993)]
- Las diapositivas sobre la metodología que se encuentran en la página www.uv.mx/personal/asumano
- El cuaderno para apuntes que sobre los modelos han realizado los alumnos.
- Sistemas de software semejantes que existen ya sea proporcionados por el profesor o en Internet.

IV. Procedimiento:

NOTA: Esta práctica puede llevar más de una sesión y éstas pueden ser en el salón de clase o en el Centro de Cómputo.

1. En su cuaderno realice los elementos necesarios para su modelo de comportamiento
 - a. Ventanas que representarán la forma en que el sistema interactuará con el usuario. Utilice las facilidades que ofrece una GUI (Graphic User Interface) hecha utilizando Delphi.
 - b. Para cada ventana poner su explicación de uso.
 - c. Lista de posibles fallos a los que se puede ver expuesto el sistema.

Ejemplos:

- i. Se pierde la conexión de la red.
- ii. No están la Base de Datos
- iii. Se desconecta el suministro eléctrico
- iv. El hardware falla
- v. El usuario introduce mal los datos

2. Revise que su Manual Preliminar del Usuario contenga:

- a. Introducción que incluya: Cuales son las características de su software (suponga que ya lo hizo) y que lo hace especial para que las personas decidan utilizarlo en lugar de otro.
- b. Todas las funcionalidades que describió en su modelo de comportamiento en el DFD nivel 1.
- c. Aquellas actividades manuales que deben hacerse asociadas al sistema de software que está realizando.
- d. Que en cada ventana se definan los datos que allí aparecen según el DD.
- e. Su lista de fallos posibles y la forma en que el usuario debe afrontarlos.

Práctica 7. Cálculo de las Métricas de Análisis

I. Objetivo:

Calcular dos métricas sobre los modelos de análisis con el fin de tener elementos sobre tamaño y características de calidad de éstos.

II. Equipo necesario:

- Lápiz, papel y calculadora.

III. Material de apoyo:

- El modelo de comportamiento del software que se está desarrollando.
- Las diapositivas sobre el cálculo de la métricas que se encuentran en la página www.uv.mx/personal/asumano
- Apuntes sobre Puntos de Función, sección 2.7 del libro de Áncora [Suman, A. (2006)].
- Ejemplos sobre el cálculo de las métricas: Bang y Puntos de Función que se adjuntan dos ejemplos de sistemas de software con el cálculo de las métricas.
- Archivo Excel de apoyo al cálculo de Puntos de Función

IV. Procedimiento:

1. Para la métrica Bang:
 - a. Usando el DFD nivel 1, calcule la primitiva funcional PFu
 - b. Usando el DER, calcule la primitiva de relaciones RE
 - c. Saque el coeficiente RE / PFu
 - d. Si su aplicación es:
 - Dominio de datos y el coeficiente RE / PFu es menor de 1.5, revise y modifique, en su casa, sus DER y DFD.
 - Dominio de función y el coeficiente RE / PFu es mayor de 0.6, revise y modifique, en su casa, sus DER y DFD.
 - Híbrida y el coeficiente RE / PFu es mayor de 1.4 o menor de 0.8, revise y modifique, en su casa, sus DER y DFD.
 - e. Volver a calcular coeficiente RE / PFu hasta que se ajuste a la cantidad adecuada.
2. Para la métrica de PF:
 - a. Haga una lista de indicadores de datos (ALI, AIE) y de transacciones (EE, SE y CE)
 - b. Ingrese los datos pedidos en la hoja Excel para que ésta calcule la complejidad de cada indicador y los punto de función sin ajustar (T).
 - c. Haga la lista de catorce modificadores (Estimadores) y elija los que tendrán influencia en su sistema. Redacte dos renglones por cada modificador donde se justifique por qué lo eligió.
 - d. Asigne el grado de influencia (de cero a cinco) de cada modificador escogido usando las descripciones que vienen en las notas.
 - e. Sume los grados de influencia y llame M a la suma.

- f. Observe los puntos de función que se calcularon en la hoja Excel utilizando la fórmula.
- g. Haga una interpretación sobre su resultado sobre: complejidad y posible costo.

V. Resultado esperado:

- Un coeficiente RE / PFu que le permitirá darse cuenta de la validez del modelo de comportamiento.
- El cálculo de PF que le servirá para estimar esfuerzo y como medida histórica que **permite entender nuevos sistemas**.

Práctica 8. Planeación de Pruebas de Aceptación de Sistema

I. Objetivo:

Planificar cómo el cliente y usuarios podrán verificar que el software que se les está construyendo hará lo que ellos desean, para lo cual tendrán una serie de casos de prueba que incluirán diversas formas de verificar el comportamiento del software desde el normal o esperado hasta los posibles fallos que se pudieran cometer y a los cuales el software debe responder con ayudas y prevención de defectos.

NOTA: Esta práctica no se terminará en un par de horas ni en un día, trate de avanzar con una función del software en construcción y avance de función en función. Como lo marca un modelado iterativo e incremental.

II. Equipo necesario:

- Computadora con MS-Office.

III. Material de apoyo:

- Los modelos de análisis del software que se está desarrollando, incluyendo el Diccionario de Datos y Manual Preliminar del Usuario.
- Los apuntes sobre prueba de software, Capítulos 1 y 2, que se encuentran en la página www.uv.mx/personal/jfernandez. Observe bien los ejemplos.
- El cuaderno donde empezó a establecer sus dominios y particiones y a realizar sus casos de prueba.

IV. Procedimiento:

1. Definir dominios de cada pieza elemental del DD.
 - a. Use la Tabla del DD (ver el ejemplo de Money), agregue dos columnas, llámelas Dominio y Particiones y proceda como sigue:
 - i. En la columna Dominio especifique, usando notación de conjuntos, rango o con lenguaje natural, los dominios de las piezas elementales de información de flujo de datos y en los almacenes de datos, mismas que deben aparecer en el DD.
 - ii. En la columna Particiones, describa las clases de equivalencia dictadas por el comportamiento de la función en que interviene cada variable de entrada.
2. Para cada funcionalidad (acontecimiento)
 - a. Establecer los valores de entrada con los que se establecerán los casos de prueba de cada función del software en desarrollo. Identifique las entradas y salidas; luego escoja:
 - i. Un valor de cada clase de equivalencia para cada variable de entrada.
 - ii. Escoja dos valores que sean los extremos cada clase de equivalencia.
3. Identifique las salidas correspondientes y escriba los casos de prueba, uno por cada combinación de los valores escogidos en las diversas variables. No varíe todos los valores de las variables a la vez.

V. Resultado esperado:

Un conjunto de casos de prueba que permitirán revisar y aprobar o rechazar cada una de las funcionalidades que se prometieron en la fase de análisis.

Práctica 9. Creación de Modelos de Diseño

I. Objetivo:

Crear algunos de los modelos que conforman el diseño dentro de la metodología: Análisis Estructurado Moderno de Yourdon, a saber: modelo de procesador y modelo de implantación de programas

II. Equipo necesario:

- Computadora con MS-Office, incluido MS-Visio.

III. Material de apoyo:

- Los modelos de análisis del software que se está desarrollando, incluyendo el Diccionario de Datos y Manual Preliminar del Usuario.
- Las diapositivas sobre la metodología utilizada que se encuentran en la página www.uv.mx/personal/asumano.
- El cuaderno donde empezó a establecer sus modelos de diseño.
- Los ejemplos: Money y ATM que se le entregaron previamente.

IV. Procedimiento:

1. Definir modelo de procesador. En una Tabla de tres columnas en MS-Word
 - a. A la columna uno póngale de título la palabra “Procesador” y agregue un renglón por cada conjunto de procesadores equivalentes.
 - b. A la columna dos póngale como título “Procesos”, escriba cuáles serán los procesos (burbujas del DFD nivel 1) que correrán en cada uno de los procesadores que se definieron.
 - c. A la columna tres titúlela como “Almacenes”, escriba la lista de almacenes (entidades y relaciones del DER) que se guardarán en cada tipo de máquina (procesador).
2. Para cada procesador se debe realizar un DE.
 - a. Ponga un módulo ejecutivo llamado como el procesador
 - b. Si una burbuja no tiene burbujas derivadas, póngala como módulo del primer nivel del DE.
 - c. Las burbujas derivadas, póngalas en el segundo nivel asociadas a la burbuja padre.
 - d. Si percibe que la cohesión de las burbujas no es correcta o que la burbuja es muy general, dibuje más módulos derivados.
 - e. Incluya los almacenes que cada módulo de último nivel empleará.
 - f. Incluya los parámetros de entrada salida: de control - aquellos que modificarán el comportamiento del módulo que lo está recibiendo- o de datos -aquellos que se obtienen de un almacén o que son datos necesarios para el cálculo o consulta que realiza el módulo que lo recibe-.
3. Para cada módulo, escriba su pseudocódigo.

V. Resultado esperado:

Los modelos de diseño: de procesador (tabla con procesadores, procesos y almacenes), de implantación de programas (DE por cada procesador, pseudocódigo de cada módulo).

Práctica 10. Plan de Pruebas de Integración

I. Objetivo:

Realizar el Plan de Pruebas de Integración del Sistema de Software que se está realizando en el curso que servirá para que Administrador del Proyecto de Software e Ingeniero de Pruebas puedan tanto planificar los casos de prueba como su aplicación.

II. Equipo necesario:

- Computadora con MS-Office.

III. Material de apoyo:

- Los modelos de diseño del software que se está desarrollando, incluyendo el Modelo de Procesador y Modelo de Implementación de Programas.
- Las diapositivas sobre prueba de integración de software que se encuentran en la página www.uv.mx/personal/jfernandez. Observe bien los ejemplos.
- El cuaderno donde empezó a establecer su plan de prueba.

IV. Procedimiento:

Usando el estándar IEEE 829 para especificar Planes de Prueba haga:

1. Abra un documento de MS-Word
2. Escriba el título: *Plan de Prueba de Integración del Sistema <<nombre de su sistema de software>>*
3. Ingrese una tabla de dos columnas y doce renglones.
4. Escriba los apartados de cada renglón como sigue:

1. Identificación	
2. Elementos a probar	
3. Enfoque	
4. Criterio aceptación	
5. Criterio suspensión	
6. Productos a entregar	
7. Tareas	
8. Necesidades ambientales	
9. Responsabilidades	
10. Personal	
11. Calendario	
12. Riesgos y contingencias	

5. Llene cada renglón con lo que se necesita.

V. Resultado esperado:

Una tabla para cada Diagrama de Estructura de cada procesador.

Práctica 11. Planeación de Pruebas de Unidad

I. Objetivo:

Crear una serie de casos de prueba de unidad sobre el algoritmo que le tocó realizar y que corresponde a su sistema de software que está realizando.

II. Equipo necesario:

- Una PC con el SO MS-Windows

III. Material de apoyo:

- Un módulo codificado en Pascal o pseudocódigo
- Las notas sobre el método de caminos básicos en la página: www.uv.mx/personal/jfernandez
- Un cuaderno para apuntes.

IV. Procedimiento:

1. Realice el grafo de control empleando MS-Visio y un diagrama de flujo de datos.
2. Calcule usando las tres fórmulas la complejidad ciclomática
3. Abra MS-Word
4. Escriba el título: "Casos de Prueba para el módulo: <nombre del módulo que usted codificó>"
5. En el siguiente renglón escriba: "Autor: <su nombre>"
6. Cree una tabla de dos columnas como sigue:

Tabla 0-1. Valores a considerar para la prueba del módulo: <su módulo>

Dato de entrada	Dominio	Particiones

7. Ingrese una Tabla de cuatro columnas y a cada columna póngale el nombre que sigue:

Tabla 0-2. Tabla para Casos de Prueba de Unidad

Camino básico	Num. de caso de prueba	Datos de entrada	Salida esperada

8. Para cada camino básico haga, al menos, un caso de prueba.

V. Resultados esperados:

Un reporte de prueba de unidad que deberá ser enviado por Internet al profesor con:

1. El código en Pascal o pseudocódigo
2. El grafo de control que representa el algoritmo y su complejidad ciclomática
3. La lista numerada de caminos básicos
4. La tabla de valores de entrada con los posibles valores a considerar
5. La tabla de casos de prueba.

Práctica 12. Utilización de una Herramienta para Pruebas de Unidad

I. Objetivo:

Conocer el manejo de una herramienta para la prueba de unidad y aplicar las pruebas de unidad a un módulo codificado previamente.

II. Equipo necesario:

- Una PC con el SO MS-Windows
- Delphi 2006 o Delphi 7

III. Material de apoyo:

- Su módulo que codificó en Pascal bajo el ambiente Delphi
- Notas sobre DUnit en: www.uv.mx/personal/jfernandez.

IV. Procedimiento:

1. Para Delphi 2006:
 - a. Entre a la opción Archivo (File)
 - b. Escoja crear un nuevo proyecto (New)
 - c. De clic en la opción otros (other)
 - d. Pida la opción UnitTest
 - e. Escoja Test Project
 - f. Ponga el nombre del proyecto, por ejemplo: *Prueba1*
2. Además, debe crear una Unit con el código del programa que realizó. Póngale un nombre, por ejemplo: *Mi-modulo*
3. Se debe crear una nueva unidad que contendrá los casos de prueba :
 - a. Escoja a File>new>other>Test Unit
 - b. De el nombre de la unidad a probar (source file, ejemplo: *Mi-modulo*)
 - c. Ponga nombre a su unidad de prueba.
4. Codifique los casos de prueba como en el ejemplo del *máximo común divisor* (ver notas sobre DUnit).
5. Corra la aplicación (F9).

V. Resultados esperados:

Un proyecto de prueba de unidad DUnit con tres archivos de Delphi: el proyecto, la unidad a probar y la unidad de prueba. Además se deberá ver la corrida de los casos de prueba ingresados como lo muestran las notas sobre DUnit.

EJEMPLO DE UN PROYECTO DUnit

CÓDIGO DEL PROYECTO:

```
program ProyPruebaAritmetica;  
{
```

Delphi DUnit Test Project

```
This project contains the DUnit test framework and the GUI/Console test  
runners.
```

```
Add "CONSOLE_TESTRUNNER" to the conditional defines entry in the  
project options
```

```
to use the console test runner. Otherwise the GUI test runner will be used  
by  
default.
```

```
}
```

```
{$IFDEF CONSOLE_TESTRUNNER}  
{$APPTYPE CONSOLE}  
{$ENDIF}
```

```
uses
```

```
Forms,  
TestFramework,  
GUITestRunner,  
TextTestRunner,  
Aritme in 'Aritme.pas',  
TestAritme in 'TestAritme.pas';
```

```
{$R *.RES}
```

```
begin
```

```
Application.Initialize;  
if IsConsole then  
  TextTestRunner.RunRegisteredTests  
else  
  GUITestRunner.RunRegisteredTests;  
end.
```

**CÓDIGO DE LA UNIDAD A PROBAR: PROGRAMA QUE CALCULA EL
MÁXIMO COMÚN DIVISOR ENTRE DOS ENTEROS**

```
unit Aritme;

interface
type Aritmetica=class
    function maxcomdiv(x,y:integer):integer;
end;
implementation
function Aritmetica.maxcomdiv(x,y:integer):integer;
begin
    if ((x<=0) or (y<=0)) then
        maxcomdiv:=-1 //indica error
    else
        begin
            while (x<>y) do
                if (x>y) then x:=x-y
                else y:=y-x;
            maxcomdiv:=x;
        end;
    end;
end.
```

CASOS DE PRUEBA Y CÓDIGO DE LA UNIDAD QUE CONTIENE LOS CASOS DE PRUEBA

Tabla 0-1. Casos de Prueba de Aritmética.maxcomdiv

Número de caso	Entrada	Salida esperada	Comentario
1	0 y 8	- 1	Un parámetro es ilegal
2	6 y -2	- 1	Un parámetro es ilegal
3	10 y 3	1	No hay factor común excepto 1
4	4, 28	4	El menor es el factor común
5	30 y 30	30	Números iguales
6	42 y 231	21	El divisor común es un valor medio diferente a ambos

```
nit TestAritme;  
{
```

Delphi DUnit Test Case

This unit contains a skeleton test case class generated by the Test Case Wizard.

Modify the generated code to correctly setup and call the methods from the unit being tested.

```
}
```

```
interface
```

```
uses
```

```
TestFramework, Aritme;
```

```
type
```

```
// Test methods for class Aritmetica
```

```
TestAritmetica = class(TTestCase)
```

```
strict private
```

```
FAritmetica: Aritmetica;
```

```
public
```

```
procedure SetUp; override;
```

```
procedure TearDown; override;
```

```
published
```

```

procedure Testmaxcomdiv1;
procedure Testmaxcomdiv2;
procedure Testmaxcomdiv3;
procedure Testmaxcomdiv4;
end;

implementation

procedure TestAritmetica.SetUp;
begin
  FArimetica := Aritmetica.Create;
end;

procedure TestAritmetica.TearDown;
begin
  FArimetica.Free;
  FArimetica := nil;
end;

procedure TestAritmetica.Testmaxcomdiv1;
var
  ReturnValue: Integer;
  y: Integer;
  x: Integer;
begin
  // TODO: Setup method call parameters
  // Caso de prueba 1
  x:=0; y:=8;
  ReturnValue := FArimetica.maxcomdiv(x, y);
  Check(ReturnValue=-1,'no debe ser menor o igual a cero');
end;

Procedure TestAritmetica.Testmaxcomdiv2;
var
  ReturnValue: Integer;
  y: Integer;
  x: Integer;
begin
  // Caso de prueba 2
  x:=6; y:=-2;
  ReturnValue := FArimetica.maxcomdiv(x, y);
  Check(ReturnValue=-1, 'no debe ser menor o igual a cero');
end;

Procedure TestAritmetica.Testmaxcomdiv3;
var
  ReturnValue: Integer;

```

```

y: Integer;
x: Integer;
begin
// Caso de prueba 3
x:=10; y:=3;
ReturnValue := FAritmetica.maxcomdiv(x, y);
Check(ReturnValue=1, 'segundo menor sin factor común');
end;

Procedure TestAritmetica.Testmaxcomdiv4;
var
ReturnValue: Integer;
y: Integer;
x: Integer;
begin
// Caso de prueba 4
x:=4; y:=28;
ReturnValue := FAritmetica.maxcomdiv(x, y);
Check(ReturnValue=4, 'el primero menor, exacto');
end;

initialization
// Register any test cases with the test runner
RegisterTest(TestAritmetica.Suite);
end.

```

Práctica 13. Cálculo de Métricas de Diseño

I. Objetivo:

Medir los elementos estructurales que conforman los modelos de diseño del sistema que se está elaborando.

NOTA: Esta práctica puede llevarle más de dos sesiones y parte de ella se realiza en equipo y otra parte es personal.

II. Equipo necesario:

- Computadora con Windows XP o superior y, opcionalmente conectada a Internet
- Lápiz, papel y calculadora.

III. Material de apoyo:

- El IDE Delphi 7 ó superior
- Hoja de cálculo como MS-Excel
- Los modelos de implantación de programas del software que se está desarrollando: Diagramas de estructura de cada procesador y pseudocódigo de todos los módulos.
- Las diapositivas sobre el cálculo de métricas que se encuentran en la página www.uv.mx/personal/asumano. Fijarse en los ejemplos sobre el cálculo de las métricas: Complejidad Total del Sistema, Complejidad Relativa del Sistema, cohesión y acoplamiento
- Las notas sobre el método de prueba de Caminos Básicos que están en la página www.uv.mx/personal/jfernandez. Fijarse en los ejemplos sobre el cálculo de la complejidad ciclomática $v(G)$.

IV. Procedimiento:

1. Para las métricas Complejidad Total del Sistema y Complejidad Relativa del Sistema (esta parte de la práctica es por equipo):
 - a. Construya en una hoja de cálculo, como en Excel, una Tabla de 5 columnas y llámelas así:

Tabla 0-1. Tabla para el cálculo de métricas CTS y CRS

Módulo	$f_{out}(i)$	$v(i)$	$S(i)$	$D(i)$
--------	--------------	--------	--------	--------

- b. Para cada DE de cada procesador haga lo siguiente:
 - i. Agregue una fila a la Tabla 2-3 por cada módulo de los DE
 - ii. Anote las medidas $f_{out}(i)$ y $v(i)$ de cada módulo
 - iii. Incluya la fórmula para $S(i)$ y $D(i)$
- c. Agregue un renglón al final de la Tabla 2-3 y en la celda intersección columna $S(i)$ agregue la función suma (Σ) de toda la columna.
- d. En la celda intersección columna $D(i)$ agregue la función suma (Σ) de toda la columna.

- e. Ya con las sumas calcule CTS y CRS.
- 2. Para las métricas de nivel componente (módulo, subrutina o procedimiento) haga (esta parte de la práctica es personal y es sólo para un módulo):
 - a. Codifique un módulo en Pascal bajo el ambiente Delphi. El módulo a codificar debe devolver un resultado numérico y sus entradas pueden ser numéricas o alfanuméricas, NO debe realizar ningún tipo de acceso a almacenes (tablas, archivos).
 - b. Compile y corra su módulo utilizando una interfaz gráfica que convierta los campos Edit.text en cantidades numéricas. Esta interfaz será provisional y servirá para prueba de la funcionalidad.
 - c. Para calcular la complejidad ciclomática:
 - i. Haga el grafo de control del módulo codificado.
 - ii. Calcule $v(G)$ con las tres formas. Comprueba que sean resultados iguales; de no ser así, vuelva a calcular.
 - d. Para calcular la cohesión.
 - i. Realice la lista de tokens.
 - ii. Haga las rebanadas de cada parámetro de salida.
 - iii. Cuente los adhesivos y superadhesivos.
 - iv. Haga las divisiones para el cálculo de la cohesión funcional fuerte.
 - v. Haga las divisiones para el cálculo de la cohesión funcional débil.
 - e. Para calcular el acoplamiento
 - i. Cuente las variables de dato y de control de entrada (d_i y c_i).
 - ii. Cuente las variables de dato y de control de salida (d_o y c_o).
 - iii. Revise si hay variables globales y cuéntelas (g_d y g_c).
 - iv. A la variable w asígnele el número de módulos que manda llamar su módulo en estudio (puede ser cero).
 - v. A la variable r asígnele el número de módulos que llaman a su módulo en estudio (al menos debe ser uno).
 - vi. Calcule el indicador de acoplamiento m_c .

V. Resultado esperado:

- Medida de estructura arquitectónica Complejidad Total del Sistema y Complejidad Relativa del Sistema (CRS). Note que si CRS es mayor a 26.5 debe simplificar su modelo.
- Medidas a nivel componente:
 - Complejidad ciclomática $v(G)$.
 - Cohesión. Si el cálculo de la CFF ó CFD resultan una fracción cercana a uno su módulo es cohesivo, lo cual es bueno, si están muy cercanos a cero, modifique su diseño, tal vez su módulo necesite dividirse.
 - Acoplamiento. Si el indicador de acoplamiento es cercano a uno tiene un acoplamiento bajo, de lo contrario, revise su codificación para lograr mayor generalidad.

Bibliografía

1. Fernández Peña, J.M., Diapositivas sobre prueba para el curso de Ingeniería de Software I, <http://www.uv.mx/personal/jfernandez>
2. Sumano López, M.A., Diapositivas del curso de Ingeniería de Software I, <http://www.uv.mx/personal/asumano>
3. Yourdon, E. (1993), Análisis Estructurado Moderno, Prentice-Hall
4. Pressman, R. (2005), Ingeniería de software, un enfoque práctico; quinta edición, McGraw-Hill

Lectura 10. Metodología de Ingeniería del Software para el desarrollo y mantenimiento de sistemas de información

Metodología de Ingeniería del Software para el desarrollo y mantenimiento de sistemas de información del Gobierno de Extremadura

Índice del Documento

1.- Introducción	Página 4
2.- Propuesta de metodología	Página 4
3.- Ingeniería del software	Página 6
3.1. Fase de Estudio de Viabilidad	Página 6
3.2. Fase de Análisis	Página 7
3.3. Fase de Diseño	Página 10
3.4. Fase de Construcción del Sistema	Página 13
3.5. Fase de implantación y aceptación del sistema	Página 17
4. Arquitectura	Página 19
5.- Lenguajes de programación	Página 22
6. Integración con el proyecto e-GobEx	Página 22

Control de versiones

Versión	Responsable	Observaciones	Fecha
0	SDP	Revisión del documento Plataforma Corporativa de Aplicaciones Informática PCDAI.	Junio 2014
Borrador	SDP	Versión borrador del documento	Junio 2014
1.0	SDP	Versión inicial del documento	Julio 2014

1.-Introducción

El presente documento tiene como objetivo establecer la metodología a aplicar en la ingeniería del software para el desarrollo y mantenimiento de sistemas de información el Gobierno de Extremadura en las distintas arquitecturas presentes en la actualidad.

Como característica principal de esta metodología cabe destacar su concepción como metodológica basada en el paradigma de orientación a objetos. Este paradigma por lo tanto nos hará abandonar las concepciones metodológicas y los lenguajes de programación tradicionales del desarrollo estructurado y orientado a datos o funciones.

2.- Propuesta de metodología

La Metodología en cuestión constará de cuatro aspectos diferenciados sobre los que hay que definir los criterios propios a aplicar: Ingeniería del Software, Arquitectura, Lenguajes de programación y pautas para la integración con la Plataforma de Administración Electrónica de la Junta de Extremadura. Se introducen de manera genérica cada uno de estos aspectos que requerirán un detalle en capítulos posteriores de este documento:

Ingeniería del Software

La ingeniería que se propone estará basada en METRICA v.3 utilizando para el proceso de modelado el lenguaje de modelado unificado UML (Unified Modeling Language) propiciado por la OMG (Object Management Group) sobre sus diagramas específicos de representación del modelado. Se pretende que el ciclo de vida de desarrollo se encuentre íntegramente concebido sobre el paradigma de orientación a objetos.

Arquitectura

La Arquitectura definirá la forma de abordar el diseño de los componentes definidos por la Ingeniería, en este caso serán las clases identificadas. Mientras que en la Ingeniería definimos los pasos que hay que seguir, en la Arquitectura se definirán aspectos relacionados con la ordenación e interrelación entre las clases. Estará basada en la arquitectura de múltiples capas especializadas en tareas específicas.

Lenguajes de programación

Dado que todo el desarrollo de aplicaciones estará basado en el paradigma de orientación a objetos, se hace necesario la utilización de lenguajes concebidos para la programación orientada a objetos o que la soporten debidamente.

Integración con la plataforma e-GobEx

En los proyectos en los que así se determine en orden a los requerimientos funcionales establecidos en los documentos técnicos, serán de uso obligatorio los servicios corporativos para la integración con el proyecto de administración electrónica (e-GobEx)

3.- Ingeniería del Software

La ingeniería describe el procedimiento de trabajo que se debe llevar a cabo para el desarrollo y mantenimiento de sistemas de información, haciéndolo independiente a su tipología. Describe los productos a generar en las distintas fases estandarizando el proceso y con objeto de garantizar la calidad del producto final.

La Ingeniería del software propuesta se divide en cinco fases secuenciales que forman el ciclo de vida de los sistemas de información: Fase de Estudio de Viabilidad, Fase de Análisis, Fases de Diseño, Fase de Implementación y Fase de Despliegue.

3.1.- Fase de estudio de viabilidad

En esta fase se define el problema que se quiere resolver a través de la especificación de requisitos que debe cumplir la aplicación. Obteniendo como salida el Modelo de Requisitos.

Para su implementación utilizaremos los siguientes componentes:

Modelo de Requisitos

Este modelo no tiene una representación específica en UML, por lo cual ha sido diseñado específicamente y representado al final de esta guía. No obstante, en este modelo se deben recoger todos los requisitos especificados por el usuario que deba cumplir la aplicación en el momento de abordarla, plasmándolos en un documento de requerimientos que deberá ser aceptado con su firma por los gestores proponentes. Con el tiempo los requisitos cambiarán y por tanto este modelo también lo hará.

Modelo de casos de uso de alto nivel

Se recogen en este modelo los distintos casos de uso de la aplicación y como se resuelven estos a través de la interacción de los distintos actores, identificados en este paso, y el sistema, que se va perfilando ya. Se utilizará para confeccionar este modelo el Diagrama de Casos de Uso establecido dentro de la metodología UML.

Prototipo de interfaces de usuario

Desde el primer momento se tendrá un esbozo de la salida e interfaz de la aplicación específica para cumplir con los requisitos aportados por los usuarios y las necesidades de intercambio de información detectadas. Para este paso no hay representación recomendada en UML por lo que se deberá recurrir a los procedimientos tradicionales de representación de entradas/salidas con formato de representación de información.

3.2.- Fase de Análisis

En esta fase se aborda el análisis exhaustivo de las clases que llevarán a cabo la realización de los casos de uso. Se partirá desde el Modelo de Requisitos.

Establecimiento de los requisitos del sistema.

En esta actividad se lleva a cabo la definición, análisis y validación de los requisitos a partir de la información facilitada por el usuario, completándose el modelo de requisitos obtenido en la actividad anterior. El objetivo de esta actividad es obtener un catálogo detallado de los requisitos, a partir del cual se pueda comprobar que los productos generados en las actividades de modelización se ajustan a los requisitos de usuario.

Se propone como técnica de obtención de requisitos, la especificación de los casos de uso de la orientación a objetos. Dicha técnica ofrece un diagrama simple y una guía de especificación en las sesiones de trabajo con el usuario.

Tarea		Productos	Técnicas y Prácticas	Participantes
ASI 2.1	Obtención de Requisitos	<ul style="list-style-type: none"> - Catálogo de Requisitos - Modelo de Casos de Uso 	<ul style="list-style-type: none"> - Sesiones de Trabajo - Catalogación - Casos de Uso 	<ul style="list-style-type: none"> - Usuarios Expertos - Analistas
ASI 2.2	Especificación de Casos de Uso	<ul style="list-style-type: none"> - Catálogo de Requisitos - Modelo de Casos de Uso - Especificación de Casos de Uso 	<ul style="list-style-type: none"> - Sesiones de Trabajo - Catalogación - Casos de Uso 	<ul style="list-style-type: none"> - Usuarios Expertos - Analistas
ASI 2.3	Análisis de Requisitos	<ul style="list-style-type: none"> - Catálogo de Requisitos - Modelo de Casos de Uso - Especificación de Casos de Uso 	<ul style="list-style-type: none"> - Sesiones de Trabajo - Catalogación - Casos de Uso 	<ul style="list-style-type: none"> - Usuarios Expertos - Analistas
ASI 2.4	Validación de Requisitos	<ul style="list-style-type: none"> - Catálogo de Requisitos - Modelo de Casos de Uso - Especificación de Casos de Uso 	<ul style="list-style-type: none"> - Sesiones de Trabajo - Catalogación - Casos de Uso 	<ul style="list-style-type: none"> - Usuarios Expertos - Analistas

Ilustración 1: Tareas recomendadas por Metrica V3.

Análisis de casos de uso

En este paso y por cada caso de uso, se especificará como resuelven los casos de uso las diferentes clases de análisis identificadas hasta ese momento, pudiendo surgir nuevas clases de análisis que se incorporarán al catálogo anteriormente descrito. En particular para cada caso de uso se deberá confeccionar un Diagrama de Secuencia de UML que represente la realización del caso de uso. También son recomendables los diagramas de clase y de objetos. Todos los diagramas que se realicen en este paso se deberán especificar a tres capas, es decir, para realizar un caso de uso, interactuarán con el actor las clases de presentación que se apoyarán en los servicios de las clases de negocio y éstas a su vez obtendrán los datos necesarios para su operativa desde las clases de datos.

Tarea		Productos	Técnicas y Prácticas	Participantes
ASI 4.1	Identificación de Clases Asociadas a un Caso de Uso	<ul style="list-style-type: none"> - Modelo de Clases de Análisis 	<ul style="list-style-type: none"> - Diagrama de Clases 	<ul style="list-style-type: none"> - Analistas
ASI 4.2	Descripción de la Interacción de Objetos	<ul style="list-style-type: none"> - Análisis de la Realización de los Casos de Uso 	<ul style="list-style-type: none"> - Diagrama de Interacción de Objetos (secuencia o colaboración) 	<ul style="list-style-type: none"> - Analistas

Ilustración 2: Tareas recomendadas por Metrica V3.

Análisis de clases

En este modelo, que no es un diagrama de clases al uso, se detallan cada una de las clases que van apareciendo en la realización del diagrama de los casos de uso. Por tanto, este paso y el siguiente se realizarán en paralelo. No hay una representación estándar para este modelo. No obstante las clases serán distinguidas y agrupadas en las tres capas de la arquitectura: Capa de Presentación, Capa de Negocio y Capa de Datos, teniendo por tanto como salida de este paso la localización y posterior definición de las clases de presentación, de las clases de negocio y de las clases de datos.

Tarea		Productos	Técnicas y Prácticas	Participantes
ASI 5.1	Identificación de Responsabilidades y Atributos	- Modelo de Clases de Análisis - Comportamiento de Clases de Análisis	- Diagrama de Clases - Diagrama de Transición de Estados	- Analistas
ASI 5.2	Identificación de Asociaciones y Agregaciones	- Modelo de Clases de Análisis	- Diagrama de Clases	- Analistas
ASI 5.3	Identificación de Generalizaciones	- Modelo de Clases de Análisis	- Diagrama de Clases	- Analistas

Ilustración 3: Tareas recomendadas por Metrika V3.

Especificación del plan de pruebas

En esta actividad se inicia la definición del plan de pruebas, el cual sirve como guía para la realización de las pruebas, y permite verificar que el sistema de información cumple las necesidades establecidas por el usuario, con las debidas garantías de calidad.

El plan de pruebas es un producto formal que define los objetivos de la prueba de un sistema, establece y coordina una estrategia de trabajo, y provee del marco adecuado para elaborar una planificación paso a paso de las actividades de prueba. El plan se inicia en la fase de análisis, definiendo el marco general, y estableciendo los requisitos de prueba de aceptación, relacionados directamente con la especificación de requisitos.

Dicho plan se va completando y detallando a medida que se avanza en los restantes procesos del ciclo de vida del software

Se plantean los siguientes niveles de prueba:

- Pruebas unitarias.

- Pruebas de integración.
- Pruebas del sistema.
- Pruebas de implantación.
- Pruebas de aceptación.

En esta actividad también se avanza en la definición de las pruebas de aceptación del sistema. Con la información disponible, es posible establecer los criterios de aceptación de las pruebas incluidas en dicho nivel, al poseer la información sobre los requisitos que debe cumplir el sistema, recogidos en el modelo de requisitos.

Tarea	Productos	Técnicas y Prácticas	Participantes
ASI 10.1	Definición del Alcance de las Pruebas	- Plan de Pruebas	- Sesiones de Trabajo
ASI 10.2	Definición de Requisitos del Entorno de Pruebas	- Plan de Pruebas	- Sesiones de Trabajo
ASI 10.3	Definición de las Pruebas de Aceptación del Sistema	- Plan de Pruebas	- Sesiones de Trabajo

Ilustración 4: Tareas recomendadas por Metrica V3.

3.3.- Fase de Diseño

Pasamos a definir en esta fase las clases que habrán de ser implementadas. Mientras que en la fase anterior nos preocupábamos de cubrir los requisitos mediante la realización de los casos de uso identificados en la Fase de Estudio de viabilidad y a través de las clases de análisis, en esta fase se detallarán las clases anteriores, debiendo surgir nuevas clases auxiliares de apoyo para la programación, así como interfaces internas para la interacción entre clases. En suma, se detallarán los métodos y atributos en su totalidad de

cada una de las clases para su posterior implementación en cualquier lenguaje POO.

Diseño de casos de uso reales

Se abordará nuevamente la realización de los casos de uso utilizando para ello los Diagramas de Secuencia de UML por cada caso de uso. Las nuevas clases que vayan surgiendo de apoyo o complementarias se incorporarán al catálogo de las clases de diseño listas para ser programadas.

Tarea	Productos	Técnicas y Prácticas	Participantes	
DSI 3.1	Identificación de Clases Asociadas a un Caso de Uso	- Diseño de la Realización de los Casos de Uso <ul style="list-style-type: none">o Especificación Detallada	- Diagrama de Interacción de Objetos	- Equipo del Proyecto
DSI 3.2	Diseño de la Realización de los Casos de Uso	- Diseño de la Realización de los Casos de Uso <ul style="list-style-type: none">o Especificación Detallada	- Diagrama de Interacción de Objetos	- Equipo del Proyecto
DSI 3.3	Revisión de la Interfaz de Usuario	- Diseño de Interfaz de Usuario: <ul style="list-style-type: none">o Formatos Individuales de Interfaz de Pantalla Gráficao Catálogo de Controles y Elementos de Diseño de Interfaz de Pantalla Gráficao Modelo de Navegación de Interfaz de Pantalla Gráficao Formatos de Impresióno Prototipo de Interfaz de Pantalla Gráfica	- Catalogación <ul style="list-style-type: none">- Diagrama de Transición de Estados- Diagrama de Interacción de Objetos- Prototipado	- Equipo del Proyecto - Usuarios Expertos
DSI 3.4	Revisión de Subsistemas de Diseño e Interfaces	- Diseño de la Realización de los Casos de Uso <ul style="list-style-type: none">o Definición a Nivel de Subsistemas e Interfaz	- Diagrama de Interacción de Objetos	- Equipo del Proyecto - Equipo de Arquitectura

Ilustración 5: Tareas recomendadas por Metraca V3.

Diseño de las clases

Sin representación específica, se pretende detallar las clases de diseño a través de un diagrama de clases. Sin embargo, como consecuencia del modelo a tres capas, se distinguirán y agruparán las clases de presentación (que formarán la interfaz de usuario definitiva), las clases de negocio y las clases de

datos. Se tienen en cuenta las decisiones tomadas sobre el entorno tecnológico y el entorno de desarrollo elegido para la implementación.

Otro de los objetivos del diseño de las clases es identificar para cada clase, los atributos, las operaciones que cubren las responsabilidades que se identificaron en el análisis, y la especificación de los métodos que implementan esas operaciones, analizando los escenarios del Diseño de Casos de Uso Reales. Se determina la visibilidad de los atributos y operaciones de cada clase, con respecto a las otras clases del modelo.

Para aquellos sistemas que utilicen para almacenar información un SGDBR se obtendrá el Modelo Conceptual de Datos: Diagrama entidad/relación.

Tarea		Productos	Técnicas y Prácticas	Participantes
DSI 4.1	Identificación de Clases Adicionales	- Modelo de Clases de Diseño	- Diagrama de Clases	- Equipo del Proyecto
DSI 4.2	Diseño de Asociaciones y Agregaciones	- Modelo de Clases de Diseño	- Diagrama de Clases	- Equipo del Proyecto
DSI 4.3	Identificación de Atributos de las Clases	- Modelo de Clases de Diseño	- Diagrama de Clases	- Equipo del Proyecto
DSI 4.4	Identificación de Operaciones de las Clases	- Modelo de Clases de Diseño - Comportamiento de Clases de Diseño	- Diagrama de Clases - Diagrama de Transición de Estados	- Equipo del Proyecto
DSI 4.5	Diseño de la Jerarquía	- Modelo de Clases de Diseño	- Diagrama de Clases	- Equipo del Proyecto
DSI 4.6	Descripción de Métodos de las Operaciones	- Modelo de Clases de Diseño	- Diagrama de Clases	- Equipo del Proyecto
DSI 4.7	Especificación de Necesidades de Migración y Carga Inicial de Datos	- Plan de Migración y Carga Inicial de Datos	- Sesiones de Trabajo	- Analistas - Usuarios Expertos

Ilustración 6: Tareas recomendadas por Metrica V3.

Diseño físico de datos

En esta actividad se define la estructura física de datos que utilizará el sistema, a partir del modelo lógico de datos normalizado o modelo de clases, de manera que teniendo presentes las características específicas del sistema

de gestión de datos concreto a utilizar, los requisitos establecidos para el sistema de información, y las particularidades del entorno tecnológico, se consiga una mayor eficiencia en el tratamiento de los datos.

También se analizan los caminos de acceso a los datos utilizados por cada módulo/clase del sistema en consultas y actualizaciones, con el fin de mejorar los tiempos de respuesta y optimizar los recursos de máquina.

Modelo de datos

Con los casos de uso ya resueltos a partir de las clases que se van a implementar, necesitaremos por último realizar el modelo de datos que define los datos y su almacenamiento. Este paso suele ser inicial en otras metodologías. Se obtendrá como salida dos modelos:

- **Modelo Lógico de datos:** Diagrama relacional.
- **Modelo Físico de datos:** Diagrama de tablas.

En los sistemas que utilicen para almacenar su información un SGBDR los modelos anteriores se obtendrán como reducción del modelo Entidad/Relación. En este caso se recomienda que la estructura física de los datos obtenidos cumplan, al menos, la tercera forma normal (3FN) como garantía de normalización de datos.

3.4.- Fase de Construcción del Sistema

En este proceso se genera el código de los componentes del Sistema de Información, se desarrollan todos los procedimientos de operación y seguridad y se elaboran todos los manuales de usuario final y de explotación con el objetivo de asegurar el correcto funcionamiento del Sistema para su posterior implantación.

Para conseguir dicho objetivo, en este proceso se realizan las pruebas unitarias, las pruebas de integración de los subsistemas y componentes y las pruebas del sistema, de acuerdo al plan de pruebas establecido.

Asimismo, se define la formación de usuario final y, si procede, se construyen los procedimientos de migración y carga inicial de datos.

Generación del código de los componentes

El objetivo de esta actividad es la codificación de los componentes del

sistema del información, a partir de las especificaciones de construcción obtenidas en el proceso Diseño del Sistema de Información (DSI), así como la construcción de los procedimientos de operación y seguridad establecidos para el mismo.

En paralelo a esta actividad, se desarrollan las actividades relacionadas con las pruebas unitarias y de integración del sistema de información. Esto permite una construcción incremental, en el caso de que así se haya especificado en el plan de pruebas y en el plan de integración del sistema de información.

Tarea		Productos	Técnicas y Prácticas	Participantes
CSI 2.1	Generación del Código de Componentes	<ul style="list-style-type: none">- Producto Software:<ul style="list-style-type: none">o Código Fuente de los Componentes		<ul style="list-style-type: none">- Programadores
CSI 2.2	Generación del Código de los Procedimientos de Operación y Seguridad	<ul style="list-style-type: none">- Producto Software:<ul style="list-style-type: none">o Procedimientos de Operación y Administración del Sistemao Procedimientos de Seguridad y Control de Acceso		<ul style="list-style-type: none">- Técnicos de Sistemas- Equipo de Operación- Administrador de la Base de Datos- Programadores

Ilustración 7: Tareas recomendadas por Metrica V3.

Ejecución de pruebas unitarias

En esta actividad se realizan las pruebas unitarias de cada uno de los componentes del sistema de información, una vez codificados, con el objeto de comprobar que su estructura es correcta y que se ajustan a la funcionalidad establecida.

En el plan de pruebas se ha definido el entorno necesario para la realización de cada nivel de prueba, así como las verificaciones asociadas a las pruebas unitarias, la coordinación y secuencia a seguir en la ejecución de las mismas y los criterios de registro y aceptación de los resultados.

Tarea	Productos	Técnicas y Prácticas	Participantes
CSI 3.1	Preparación del Entorno de Pruebas Unitarias	- Entorno de pruebas unitarias	- Técnicos de Sistemas - Programadores
CSI 3.2	Realización y evaluación de las Pruebas Unitarias	- Resultado de las pruebas unitarias	- Programadores

Ilustración 8: Tareas recomendadas por Metrica V3.

Ejecución de pruebas de integración

El objetivo de las pruebas de integración es verificar si los componentes o subsistemas interactúan correctamente a través de sus interfaces, tanto internas como externas, cubren la funcionalidad establecida, y se ajustan a los requisitos especificados en las verificaciones correspondientes.

La estrategia a seguir en las pruebas de integración se establece en el plan de pruebas, dónde se habrá tenido en cuenta el plan de integración del sistema de información.

Esta actividad se realiza en paralelo a las actividades Generación del Código de los Componentes y Procedimientos y Ejecución de las Pruebas Unitarias. Sin embargo, es necesario que los componentes objeto de las pruebas de integración se hayan verificado de manera unitaria.

Tarea	Productos	Técnicas y Prácticas	Participantes
CSI 4.1	Preparación del Entorno de las Pruebas de Integración	- Entorno de Pruebas de Integración	- Técnicos de Sistemas - Técnicos de Comunicaciones - Equipo de Arquitectura - Equipo del Proyecto
CSI 4.2	Realización de las Pruebas de Integración	- Resultado de las Pruebas de Integración	- Equipo del Proyecto
CSI 4.3	Evaluación del Resultado de las Pruebas de Integración	- Evaluación del Resultado de las Pruebas de Integración	- Analistas

Ilustración 9: Tareas recomendadas por Metrica V3.

Ejecución de pruebas de sistemas

El objetivo de las pruebas del sistema es comprobar la integración del sistema de información globalmente, verificando el funcionamiento correcto de las interfaces entre los distintos subsistemas que lo componen y con el resto de sistemas de información con los que se comunica.

En la realización de estas pruebas es importante comprobar la cobertura de los requisitos, dado que su incumplimiento puede comprometer la aceptación del sistema por el equipo de operación responsable de realizar las pruebas de implantación del sistema, que se llevarán a cabo en el proceso **Implantación y Aceptación del Sistema**.

Tarea		Productos	Técnicas y Prácticas	Participantes
CSI 5.1	Preparación del Entorno de las Pruebas del Sistema	- Entorno de Pruebas del Sistema		- Técnicos de Sistemas - Técnicos de Comunicaciones - Equipo de Arquitectura - Equipo del Proyecto
CSI 5.2	Realización de las Pruebas del Sistema	- Resultado de las Pruebas del Sistema	- Pruebas del Sistema	- Equipo del Proyecto
CSI 5.3	Evaluación del Resultado de las Pruebas del Sistema	- Evaluación del Resultado de las Pruebas del Sistema		- Analistas - Jefe de Proyecto

Ilustración 10: Tareas recomendadas por Metrica V3.

Manual de usuario

Tarea		Productos	Técnicas y Prácticas	Participantes
CSI 6.1	Elaboración de los Manuales de Usuario	- Producto Software: o Manuales de Usuario		- Equipo del Proyecto

Ilustración 11: Tareas recomendadas por Metrica V3.

3.5.- Fase de implantación y aceptación del sistema

Esta fase final materializa sobre los sistemas responsables de soportar la aplicación desarrollada todos los componentes que permiten que aquella opere en la forma en que ha sido diseñada:

Pruebas de implantación

La finalidad de las pruebas de implantación es doble:

- Comprobar el funcionamiento correcto del mismo en el entorno de operación.
- Permitir que el usuario determine, desde el punto de vista de operación, la aceptación del sistema instalado en su entorno real, según el cumplimiento de los requisitos especificados.

Para ello, el responsable de implantación revisa el plan de pruebas de implantación y los criterios de aceptación del sistema, previamente elaborados. Las pruebas las realizan los técnicos de sistemas y de operación, que forman parte del grupo de usuarios técnicos que ha recibido la formación necesaria para llevarlas a cabo.

Una vez ejecutadas estas pruebas, el equipo de usuarios técnicos informa de las incidencias detectadas al responsable de implantación, el cual analiza la información y toma las medidas correctoras que considere necesarias para que el sistema dé respuesta a las especificaciones previstas, momento en el que el equipo de operación lo da por probado.

Tarea	Productos	Técnicas y Prácticas	Participantes
IAS 5.1	- Preparación de las Pruebas de Implantación	- Plan de pruebas	- Jefe de Proyecto - Responsable de Implantación
IAS 5.2	- Realización de las Pruebas de Implantación	- Resultado de las pruebas de implantación	- Equipo de Implantación
IAS 5.3	- Evaluación del resultado de las Pruebas de Implantación	- Evaluación del resultado de las pruebas de implantación	- Jefe de Proyecto - Responsable de Implantación

Ilustración 12: Tareas recomendadas por Metrica V3.

Pruebas de aceptación

Será necesario disponer de un entorno de pre-producción perfectamente instalado en cuanto a hardware y software de base, así como los componentes del nuevo sistema, para realizar estas pruebas.

Las pruebas de aceptación tienen como fin validar que el sistema cumple los requisitos básicos de funcionamiento esperado y permitir que el usuario determine la aceptación del sistema.

Por este motivo, estas pruebas son realizadas por el usuario final que, durante este periodo de tiempo, debe plantear todas las deficiencias o errores que encuentre antes de dar por aprobado el sistema definitivamente.

Los Directores de los Usuarios revisan los criterios de aceptación, especificados previamente en el plan de pruebas del sistema, y dirigen las pruebas de aceptación final que llevan a cabo los usuarios expertos. A su vez, éstos últimos deben elaborar un informe que los Directores de los Usuarios analizan y evalúan para determinar la aceptación o rechazo del sistema.

Tarea	Productos	Técnicas y Prácticas	Participantes
IAS 6.1 Preparación de las Pruebas de Aceptación	- Plan de Pruebas		- Jefe de Proyecto - Directores de los Usuarios
IAS 6.2 Realización de las Pruebas de Aceptación	- Resultado de las Pruebas de Aceptación	- Pruebas de Aceptación	- Usuarios Expertos
IAS 6.3 Evaluación del resultado de las Pruebas de Aceptación	- Evaluación del Resultado de las Pruebas de Aceptación		- Jefe de Proyecto - Directores de los Usuarios

Ilustración 13: Tareas recomendadas por Metrica V3.

Paso a producción

Esta actividad tiene como objetivo establecer el punto de inicio en que el sistema pasa a producción, se traspasa la responsabilidad al equipo de mantenimiento y se empiezan a dar los servicios establecidos en el acuerdo de nivel de servicio, una vez que el Comité de Dirección ha aprobado el sistema.

Para ello es necesario que, después de haber realizado las pruebas de implantación y de aceptación del sistema, se disponga del entorno de producción perfectamente instalado en cuanto a hardware y software de base, componentes del nuevo sistema y procedimientos manuales y automáticos. En función del entorno en el que se hayan llevado a cabo las pruebas de implantación y aceptación del sistema, habrá que instalar los componentes del sistema total o parcialmente. También se tendrá en cuenta la necesidad de migrar todos los datos o una parte de ellos. Una vez que el sistema ya está en

producción, se le notifica al responsable de mantenimiento, al responsable de operación y al Comité de Dirección.

Tarea		Productos	Técnicas y Prácticas	Participantes
IAS 6.1	Preparación de las Pruebas de Aceptación	- Plan de Pruebas		- Jefe de Proyecto - Directores de los Usuarios
IAS 6.2	Realización de las Pruebas de Aceptación	- Resultado de las Pruebas de Aceptación	- Pruebas de Aceptación	- Usuarios Expertos
IAS 6.3	Evaluación del resultado de las Pruebas de Aceptación	- Evaluación del Resultado de las Pruebas de Aceptación		- Jefe de Proyecto - Directores de los Usuarios

Ilustración 14: Tareas recomendadas por Metraca V3.

Se utilizará como técnica el **diagrama de despliegue**.

4.- Arquitectura

La arquitectura propuesta en esta metodología está basada en la arquitectura clásica de tres capas: Presentación, Negocio o Control y Datos.

Capa de presentación

La capa de presentación ofrece la interfaz de la aplicación con la que interactúa el usuario. En esta capa se programará todo el código, todas las clases en programación orientada a objetos, cuya finalidad sea validar datos de entrada, controlar la navegación y acceso a los distintos servicios de las aplicaciones. En programación WEB constituiría la parte del código destinada a generar el código HTML que conformarían las páginas WEB. No debe haber ningún código de lógica de negocio o de consulta de datos en la capa de presentación. El acceso a código de negocio desde las clases de la capa de presentación se hará a través de mensajes a las clases de la capa de negocio. El objetivo principal de esta capa es hacer independiente la aplicación del medio de visualización, de tal forma que se pueda programar una capa de presentación para salida hacia un navegador, consola, PDA, etc. sin necesidad de hacer ningún cambio en las capas de negocio o de datos.

En particular, la programación WEB se hará a través de un sistema o framework de plantillas: un refinamiento más de la capa de presentación que

independiza al diseñador gráfico del programador dedicado a controlar el flujo de navegación.

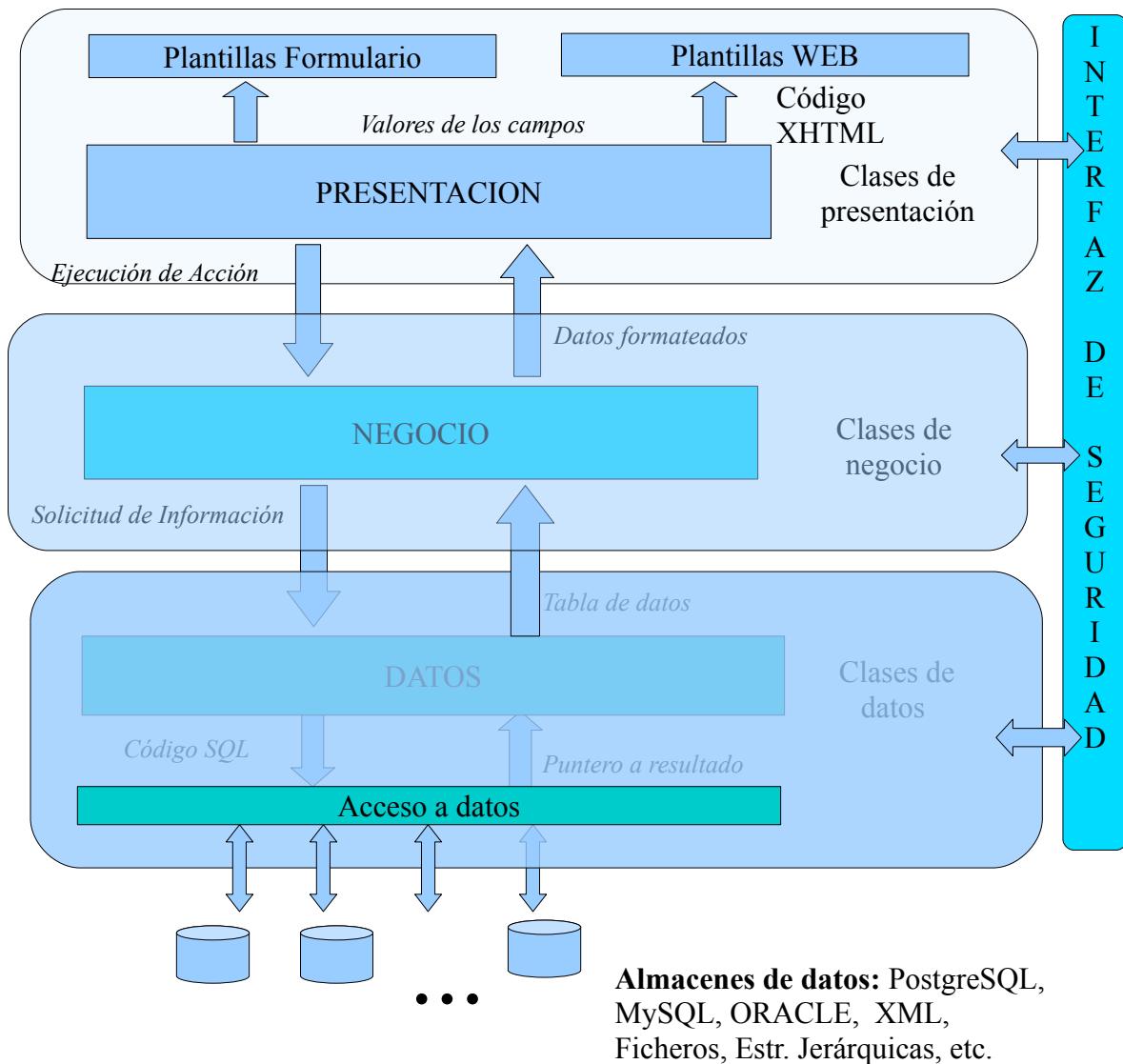
Capa de negocio

En esta capa se ubicará el código específico de control de la aplicación. Esta capa sustenta la lógica de negocio. Sus clases tienen dos objetivos fundamentales interrelacionados: por un lado realizar los cálculos, transformaciones y controles inherentes a la cobertura de los requisitos exigidos a la aplicación y por otro lado servir las necesidades básicas de datos de la capa de presentación a través de mensajes a las clases de la capa de datos. No debe haber ningún código de visualización de información o de consulta de datos en la capa de negocio.

Capa de datos

La capa de datos es la que encuentra en el nivel más bajo de la arquitectura y su único fin es proporcionar el acceso a los datos para servir las peticiones de la capa de negocio. Su objetivo fundamental es hacer independiente la aplicación de los sistemas de almacenamiento de datos, bien sean SGBD, ficheros planos, sistemas jerárquicos, etc.

Es absolutamente necesario contar aquí con un framework de acceso a datos: un refinamiento de la capa de datos que independiza las consultas de datos del sistema de almacenamiento. Por ejemplo, en la capa de datos podríamos tener consultas SQL que se ejecuten contra cualquier SGBDR sin cambiar nada en el código de las clases de la capa de datos.



5.- Lenguajes de programación

Dada la gran diversidad de sistemas de información que componen el catálogo de activos del Gobierno de Extremadura, y la particularidad de las necesidades demandadas por los servicios gestores, en cada proyecto el Director Técnico del mismo deberá especificar los criterios a seguir en este sentido. Se dispondrá para ello de documentos técnicos específicos que determinen las especificaciones técnicas a seguir en el uso de los distintos lenguajes de programación, gestores de contenidos, etc. que deberán ser utilizadas en cada proyecto.

Sin menos cabio de lo anterior cabe mencionar que se deberá optar de manera preferente por el uso de Java y su plataforma Java EE, por criterios como el posicionamiento global de las plataformas sobre la que se integra, el compromiso adquirido por la Junta de Extremadura con su apoyo al software libre, la amplitud de la comunidad que aporta herramientas complementarias y frameworks de agilización de tareas, el soporte para diversidad de fabricantes, portabilidad, etc. que hacen que esta opción sea la más adecuada según los criterios técnicos de la Junta de Extremadura.

6.- Integración con e-GobEx

La plataforma de administración electrónica e-GobEx proporciona al resto de sistemas de información una serie de servicios corporativos que aportan funcionalidades comunes. Con este objetivo también se pretende ordenar, coordinar y reducir los esfuerzos necesarios en el desarrollo y mantenimiento de proyectos, evitando que aplicativos diferentes implementen funcionalidades similares que pueden considerarse como corporativas.

Este conjunto de servicios corporativos son de uso **OBLIGATORIO** por el resto de aplicativos informáticos de la Junta de Extremadura, y no se admitirán aplicaciones que implementen funcionalidades similares a las descritas por los mismos.

Entre los servicios que ofrece la plataforma e-GobEx podemos destacar los siguientes:

- Autenticación Corporativa: sistema de autenticación que incluye todos los usuarios que necesitan acceder a aplicaciones de la Junta de Extremadura.
- Sistema de Registro Único: este sistema implementa las funcionalidades de registro de documentos tanto de entrada como de salida. Incluye los servicios necesarios para registrar un documento de entrada/salida, devolviendo el número de registro asignado y la fecha y hora de realización del asiento registral.
- Nere@: en caso de que el aplicativo informático a desarrollar incluya la gestión de procedimientos administrativos de la Junta de Extremadura.
- Notific@: para cualquier comunicación o notificación que el nuevo aplicativo deba enviar se utilizará este sistema de notificaciones corporativo.
- Pasarela de Pagos: este módulo permite realizar cargos en cuentas bancarias de administrados, asociadas a trámites o gestiones administrativas.
- Portafirmas: este sistema permite realizar la firma electrónica de documentos por parte del personal de la Junta de Extremadura con competencias para ello.
- Validación de certificados: la Plataforma también provee de una serie de servicios de validación de certificados electrónicos utilizados dentro de la organización.

Para el uso e integración de estos servicios, el Servicio de Desarrollo de Proyectos proporcionará la información detallada sobre los servicios web a utilizar en cada caso.

El Servicio de Desarrollo de Proyectos, a través del Director Técnico nombrado en cada caso, revisará con las empresas adjudicatarias de servicios cuantos términos están contenidos en el presente documento, así como los contenidos en los documentos técnicos específicos proporcionados en cada caso, pudiendo acordar sobre los productos contenidos en ellos el uso de versiones distintas a las contenidas en los documentos.

Lectura 11. El software de base sistemas operativos y lenguajes

UNIDAD TEMÁTICA 3

El software de base. Sistemas operativos y lenguajes.

Software de base: concepto.

Sistema operativo: concepto y funciones. Sistemas operativos para PC.

Programas utilitarios: concepto, clasificación.

Lenguajes de programación: niveles, paradigmas, orientaciones, traductores.

3.1) SOFTWARE

Todo computador trabaja sobre la base de un programa (conjunto de instrucciones ordenadas en una secuencia predeterminada, siendo cada instrucción una orden que se imparte al computador indicándole lo que debe hacer y usar para llevar a cabo una tarea).

Se denomina software al conjunto de programas y se lo clasifica en:

- a) Software de Base.
- b) Software de Aplicación o Aplicativo.

Software de Base: Es toda aquella parte lógica realizada generalmente por cada fabricante de computadores o por casas especializadas en el desarrollo de software de base.

Este conjunto de programas tiene por función coordinar las diversas partes del sistema computacional para hacerlo funcionar rápida y eficazmente, actuando como mediadores entre los programas de aplicaciones y el hardware del sistema, interpretando los requerimientos de cada programa que ingresa al sistema, poniendo a su disposición cualquiera de los recursos que necesite (ya sean de hardware, software o datos) para producir los resultados deseados.

Debido a la estrecha relación que existe entre las características de un computador y su software de base, no se puede concebir el uno sin el otro, a tal punto que pareciera que el software es parte integrante del hardware.

El software de base se suele clasificar en:

- a) Sistema Operativo
- b) Utilitarios
- c) Traductores de lenguajes

3.2.1) Sistema Operativo: Concepto y Funciones

Concepto: Es un conjunto de programas concebidos para efectuar la administración de los recursos del computador.

Del conjunto de instrucciones que maneja el computador, algunas residen permanentemente en la memoria central (luego de la ejecución del programa de carga inicial -IPL-) durante todo el procesamiento, mientras que otras residen solo cuando se las necesita, encontrándose almacenadas en periféricos cuando no están en la memoria central. A las primeras se las conoce como residentes, supervisor, monitor y ejecutivo. A las segundas como transientes.

Funciones:

El Sistema Operativo tiene dos funciones básicas, globalmente consideradas:

1) Proveer servicios para la ejecución de programas de aplicación y para el desarrollo de los mismos, es decir administrar los recursos en proceso, obtener automáticamente la rutina apropiada y mantener el computador sin necesidad de operación manual.

2) Actuar como entorno de la aplicación en la cual el programa es ejecutado, planeando los recursos y trabajos, puesto que ayuda a decir no solo qué recursos utilizar (asignación), sino también cuándo utilizarlos (planificación). Debido a que los dispositivos de Entrada-Salida trabajan mucho más lentamente que la C.P.U., pueden realizarse millones de instrucciones de cálculo para varios programas, mientras que los resultados se imprimen o muestran por pantalla.

Utilizando varias técnicas el Sistema Operativo combina los diversos trabajos que deben realizarse de modo que los dispositivos del sistema se empleen lo más eficientemente posible.

Las instrucciones que conforman un Sistema Operativo, materializan entre otras, las siguientes tareas en la ejecución de una aplicación:

Las instrucciones que conforman un Sistema Operativo, realizan entre otras, las siguientes tareas en la ejecución de una aplicación:

- Carga de programas.
- Gestión del tiempo de procesamiento.
- Gestión de la memoria principal.
- Gestión de la memoria secundaria (ficheros y directorios).
- Gestión del subsistema de e/s (drivers).
- Seguridad y protección del sistema.
- Interfaz de llamadas al sistema.
- Interfaz de usuario y utilidades del sistema.
- Tareas de comunicación de datos (teleprocesamiento).

Generalmente en algunos equipos el Sistema Operativo cumple funciones de monitoreo: lleva registro de las actividades del computador mientras se realiza el procesamiento. El Sistema Operativo detiene los programas que contienen errores o exceden, ya sea su tiempo máximo de ejecución o sus asignaciones de almacenamiento. Mediante el envío de mensajes informa las anomalías en los dispositivos de Entrada-Salida o en otra parte del sistema. Son también parte del Sistema Operativo la contabilización o registro de hora de ingreso y egreso, y el tiempo de duración de los programas, lo que hace posible elaborar facturas por concepto de utilización del sistema por parte de los usuarios.

Posee además mecanismos de seguridad para proteger contra el acceso no autorizado a través de la verificación de identificación ("claves" o "passwords").

3.2.2 Sistemas Operativos para PC.

La primera IBM-PC aparece con un sistema operativo desarrollado por Microsoft, denominado DOS 1.0. El mismo administraba la PC con tan solo disketteras. Aparece luego la versión 2 que controlaba también discos rígidos. La 3 agrega posibilidades de compartir dispositivos (en una red). Luego vinieron las versiones 4, 5 y 6, que fueron incorporando más utilidades (compresión de discos, resguardo, verificadores, ayudas, interfase gráfica elemental, etc..). También han sido desarrollados otros sistemas por otras compañías, como por ejemplo Digital Research que desarrolló el DR DOS cuya primera versión fue la 5.0, le siguieron la 6 y la 7, también agregando más utilidades y mejor integración a ambientes de red.

Asimismo y en forma paralela, Microsoft comienza el desarrollo de un entorno operativo que permitía una interfase gráfica mas sencilla e intuitiva para el usuario, tomando como ideas los desarrollos realizados por un área de la empresa Xerox, los que también han sido llevados a otras arquitecturas de equipos como las Apple Lisa y

Macintosh.

Este entorno se ejecutaba por encima del DOS, y no tuvo una significativa aceptación en sus versiones 1 y 2. No obstante las mejoras introducidas a la versión 3 y la mayor potencia que se tenía en el hardware, posibilitaron su gradual utilización. En realidad constituyó todo un suceso, que se afirmó con las siguientes versiones (3.1, 3.11 y 3.11 para Grupos de Trabajo). Paralelamente Microsoft estaba trabajando con IBM en el desarrollo de un nuevo sistema operativo gráfico, denominado OS/2. Por distintas motivaciones, IBM se desvincula de Microsoft en este proyecto y lo continúa sola, ofreciendo luego comercialmente a este producto. Por su parte Microsoft desarrolla una nueva versión de Windows que a diferencia de las anteriores es un sistema operativo y no solo un entorno. Guarda compatibilidad con las anteriores pero tiene significativas mejoras, tanto en lo estético como en lo funcional. La denominó Windows 95. Lanza luego otra versión destinada a un segmento de equipos de mayores requerimientos y/o administración de recursos en red de área local, que denominó Windows NT. La versión menor de Windows, la 95, tuvo una actualización denominada Windows 98, que continuó con la orientación de la 95, mejorando performance, y agregando utilidades. Por su parte la mayor, NT, también ha ido evolucionando, apareciendo las versiones 3.5, y 4. A partir de estos últimos, Microsoft ofrece una familia de productos basada en su Sistema Operativo (dependiendo de los requerimientos del usuario NT Workstation para equipos autónomo con un único usuario con altos requerimientos o NT Server para servidores de red, administrando redes de área local, preferentemente en modo dedicado).

En el segmento menor, Microsoft desarrolla para el año 2000 una versión que denominó Millenium, y para el segmento mayor actualiza el NT con una versión que denominó 2000.

3.3.) Programas utilitarios: concepto, clasificación.

Llamamos **utilitarios** a aquellos programas entregados por el fabricante, comprados a terceros o desarrollados en la propia instalación, de uso general en todo equipo, escritos con el objeto de realizar tareas repetitivas de procesamiento de datos.

Estas tareas se realizan con tanta frecuencia en el curso del procesamiento, que sería extremadamente ineficiente el que cada usuario tuviera que codificarlas en forma de programas una y otra vez.

Desde el punto de vista de las funciones que cumplen, los podemos agrupar en:

- Utilitarios de apoyo a los sistemas de aplicación: Estos programas se integran al sistema de aplicación, es decir, que su función formar parte de la secuencia de procesamiento necesaria para operar el sistema de aplicación; por ejemplo: generador de copias de archivo, generador de listados, clasificador e intercalador de archivos, etc.
- Utilitarios de Servicios: Por un lado se incluyen en este grupo un conjunto de utilitarios que ayudarán a manejar ciertos recursos del computador, y por otro a los utilitarios para el manejo de programas y sus bibliotecas; por ejemplo: listador del directorio de un disco, inicializador de discos, diskette, cinta, cassette, el que elimina o renombra archivos, el reorganizador de espacios en discos, los compiladores y compaginadores, etc.

3.4. Lenguajes de programación: niveles, paradigmas, orientaciones, traductores

Un lenguaje es el conjunto finito de símbolos básicos permitidos, combinados de acuerdo con ciertas reglas del lenguaje a las que se denominan reglas de sintáctica.

En los primeros días de la computadora, a fines de la década de 1940, cada programa (o sea la serie de instrucciones que indica a la computadora el trabajo que se va a hacer) tenía que estar escrito en lenguaje de máquina. El único que una computadora puede entender directamente y que consta de combinaciones de ceros y unos.

Todos los usuarios tenían que escribir programas compuestos de largas cadenas de ceros y unos para especificar numéricamente la dirección de los datos y los códigos de operaciones que se debían ejecutar en la máquina.

Varios años mas tarde, se desarrollaron programas llamados traductores, los cuales aceptaban como entrada cierto lenguaje simbólico o mnemotécnico para luego convertirlo automáticamente en lenguaje de máquina.

Estos traductores se conocen como ensambladores, que, aunque ahorraban al usuario mucho trabajo, no eran lo suficientemente atractivos para ellos puesto que resultaba molesto tener que especificar, aunque simbólicamente, direcciones y códigos de operaciones.

Para resolver problemas, uno tenía que programar todavía en un lenguaje parecido al de máquina.

Estos lenguajes reciben el nombre de lenguajes de bajo nivel, debido a que, como dijimos anteriormente, los programadores debían escribir instrucciones con el mas fino nivel de detalle dado que la traducción que se realiza es uno-a-uno (cada línea de código corresponde a una sola acción del sistema computacional).

Los siguientes lenguajes que aparecieron fueron los lenguajes de alto nivel en los que se introduce el concepto de macroinstrucción (la traducción es una instrucción de alto nivel a muchas de bajo nivel, una-a-muchas).

Dentro de esta categoría se encuentran lenguajes tales como BASIC, COBOL, FORTRAN, PASCAL, PL/1, APL, C, etc.

Los lenguajes de alto nivel difieren de sus antecesores de bajo nivel en que requieren menos detalle de codificación. Los traductores que convierten el programa escrito en lenguaje de alto nivel al lenguaje de máquina proporcionan el detalle.

Como resultado los programas escritos en lenguaje de alto nivel son menos extensos y mas fáciles de escribir que aquellos escritos en lenguaje de bajo nivel.

Los lenguajes de muy alto nivel, que aparecieron por primera vez en la década de 1960, se crearon para cubrir necesidades especializadas del usuario y son relativamente fáciles de aprender y de utilizar por lo que se los denominan "amigables" para el usuario. Con los lenguajes de muy alto nivel solo se necesita prescribir lo que la computadora hará en vez de como hacerlo.

Existen muchos lenguajes de muy alto nivel en el mercado y por lo general hay más de uno por cada tarea de aplicaciones:

- Generadores de informes (DMS, RPG).
- Generadores de programas (se los conoce como 4to.nivel).
- Software para procesamiento de palabras.
- Hojas o planillas electrónicas.
- Paquetes de graficación.
- etc..

La tendencia es acortar la brecha de comunicación entre hombre y máquina permitiendo que los no especialistas usen la computadora en un amplio número de disciplinas y prueben sus beneficios.

Con la venida de la nueva tecnología y de la nueva generación de computadoras, los lenguajes y sistemas en línea han sido y están siendo desarrollados para interactuar más como le gusta al hombre: rápidamente y de un modo conversacional.

LENGUAJES ORIENTADOS AL PROBLEMA Y AL PROCEDIMIENTO.

Los lenguajes de bajo y alto nivel se conocen como lenguajes de procedimientos, debido a que requieren que las personas escriban procedimientos detallados que indiquen a la computadora como realizar tareas individuales.

Los lenguajes de muy alto nivel, en contraste, reciben el nombre de lenguajes orientados al problema puesto que cada uno fue creado para resolver un problema en especial.

Además, en un amplio rango de aplicaciones, es fácil distinguir si la misma tiene características 'administrativo-contables' o 'científico-técnicas'.

Las primeras se caracterizan por requerir el manejo de un número elevado de datos, normalmente organizados en archivos, y realizar pocas operaciones sencillas con ellos. Por el contrario, las aplicaciones científico-técnicas utilizan comparativamente menor número de datos pero realizan un mayor y más complejo cálculo con ellos.

Muchos lenguajes de alto nivel o evolucionados tuvieron en cuenta estos aspectos y por lo tanto se encontraban orientados para cumplir más eficientemente alguno de los dos tipos de procesamiento tipificados anteriormente. La evolución que luego han sufrido estos lenguajes ha hecho que paulatinamente se tornen más aptos para cualquier tipo de procesos, aunque mantienen su mejor predisposición para el cual fueron diseñados.

Así por ejemplo el COBOL surge como un lenguaje para resolver los problemas del área administrativa y el FORTRAN lo hace para el área científica.

BASIC: Características del lenguaje. Estructura del programa. Definición de datos. Enunciados.

El BASIC, cuyo nombre proviene de las siglas **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode (código de instrucción simbólica de uso general para principiantes), es un lenguaje fácil de aprender y que, al paso de los años, se ha convertido en uno de los lenguajes de programación más populares y de más fácil adquisición en los proveedores especializados.

Debido a que las necesidades de almacenamiento de su traductor de lenguaje son pequeñas, trabaja con eficiencia en casi todas las computadoras personales.

Existen muchas versiones del lenguaje BASIC, desde las simplificadas, que se utilizan en computadoras de bolsillo, hasta las poderosas versiones para computadoras a gran escala que compiten con el poder de procesamiento del COBOL.

En BASIC cada instrucción se suele identificar con un número de línea: por ejemplo 10, 20, 30, etc.

La computadora siempre ejecutará las instrucciones en la secuencia especificada por los números de líneas a menos que se ordene lo contrario mediante las instrucciones de ruptura de secuencia (IF, GOTO, GOSUB, etc.).

Cada instrucción comienza con una palabra clave, la cual indica a la computadora que tipo de operación debe realizar: por ejemplo REM, READ, LET, PRINT, DATA, etc. Estas palabras claves pueden considerarse el vocabulario del sistema computacional cuando se escriben programas en lenguaje BASIC.

Uno debe siempre apegarse estrictamente a este vocabulario. Si, por ejemplo, se sustituye DATA por DATE, la computadora no sabrá que es lo que uno quiere que haga.

A pesar de sus muchas ventajas, una debilidad importante que presentan muchas versiones de este lenguaje es que no están diseñadas para facilitar la programación estructurada.

Un programa largo y no estructurado escrito en BASIC puede resultar difícil de seguir. Asimismo, ya que existen tantas versiones del lenguaje BASIC, un programa desarrollado en una computadora puede requerir modificaciones sustanciales para ejecutarse en otra máquina o en otra versión de traductor o intérprete del lenguaje.

COBOL. Características del lenguaje. Estructura del programa. Divisiones.

El COBOL, cuyo nombre proviene de las siglas **CO**mmon **B**usiness **O**riented **L**enguaje (lenguaje común orientado a los negocios), fue introducido por primera vez en los inicios de la década de 1960.

Casi todas las características principales del COBOL se relacionan con su orientación al procesamiento de datos de negocios, incluso la independencia de la máquina, la autodocumentación, y la orientación a la entrada y salida.

Independencia de la máquina: es un aspecto importante ya que los programas para el procesamiento de datos de negocios generalmente tienen que durar mucho tiempo (10 o incluso 20 años).

Durante este período, una organización puede comprar nuevo hardware o cambiar completamente de un sistema computacional a otro. De este modo, los programas escritos para un sistema deben poder ejecutarse en otros con pequeñas modificaciones.

Autodocumentación: debido a que los programas de procesamiento de datos de negocios deben durar un largo tiempo, necesitan mantenimiento continuo. Por ello, es extremadamente importante que la lógica del programa sea fácil de seguir por otros programadores o aún por el mismo que lo codificó después de transcurrido un período de tiempo.

El lenguaje COBOL se presta para un buen diseño de programas en tres formas: legibilidad, modularidad, y uso adecuado de las tres estructuras básicas de control de la diagramación estructurada: (Secuencia, bifurcación e iteración).

El lenguaje COBOL también utiliza verbos del idioma inglés (como SUBTRACT, MOVE, ADD, etc.) y conectivos (como FROM, GIVING, etc.).

Orientación a la entrada y salida: el procesamiento de datos de negocios, en contraste a las aplicaciones científicas y de ingeniería, implica la manipulación de grandes archivos con muchos registros.

Así, gran parte del trabajo en aplicaciones del procesamiento de datos de negocios se relaciona con la lectura y escritura de registros, y el lenguaje COBOL se ha diseñado para ser particularmente efectivo en esta tarea.

Contiene estipulaciones para definir de manera explícita y fácil el formato de los registros de entrada y salida. Por ejemplo, es un proceso muy sencillo el de editar cantidades monetarias , la salida con signos, puntos decimales, comas, y también redondear las cantidades.

Estructura del programa: todo programa escrito en lenguaje COBOL se agrupa en cuatro divisiones:

- División de identificación: en la que se identifica el nombre del programa, el autor, fecha de escritura y otros detalles. Esta división existe principalmente con fines de documentación.
- División de ambiente: en la que los nombres de archivos creados por el programador se vinculan a un equipo específico de entrada/salida. Aquí, por ejemplo, el programador especificaría que un archivo de entrada en particular, digamos ARCHIVO-DISCO, se localiza en disco y que un archivo de salida en particular, como ARCHIVO-IMPRESION, se dirigirá a la impresora.
- División de datos: en la que el programador nombra y define todas las variables del programa e indica su relación mutua.
- División de procedimientos: en la cual se especifican los procedimientos reales que la computadora debe seguir para crear la salida deseada.

Las tres primeras divisiones aseguran que todas las especificaciones importantes se establezcan en forma explícita en el programa.

Desventajas: Los programas escritos en lenguaje COBOL tienden a ser extensos y además se necesita un traductor de lenguaje grande y complejo para convertir los programas en el lenguaje de máquina, lo cual hace al COBOL difícil de implantar en computadoras pequeñas.

Por lo general no resulta adecuado para aplicaciones científicas y de ingeniería, las cuales utilizan demasiadas fórmulas complicadas.

Otros lenguajes de programación. Características generales.

FORTRAN cuyo nombre proviene de **FOR**mula **TRAN**slator (traductor de fórmulas), data del año 1954 y es el lenguaje comercial de alto nivel superviviente más antiguo.

Fue diseñado por científicos y está orientado hacia la resolución de problemas científicos y de ingeniería. La principal característica del FORTRAN es su capacidad para expresar con facilidad fórmulas complicadas. Aunque el BASIC es competitivo en esta tarea, el FORTRAN es generalmente superior para muchas aplicaciones debido a que hace posible una ejecución más rápida del programa y una mayor precisión, aunque hay versiones recientes del BASIC que alcanzan y aún superan sus prestaciones.

El FORTRAN por lo general utiliza un compilador como traductor del lenguaje.

Los compiladores ejecutan los programas más rápido que los intérpretes, que se utilizan en muchas de las versiones BASIC.

La lógica de los programas escritos en FORTRAN es más difícil de seguir que la lógica de algunos otros lenguajes, y es claramente inferior al COBOL para aplicaciones de procesamiento de datos de negocios.

PASCAL es un lenguaje relativamente nuevo, creado hacia 1970 para cubrir la necesidad de contar con una herramienta para la enseñanza de la programación estructurada.

Los compiladores del lenguaje PASCAL son extremadamente pequeños, lo que facilita la implementación de este lenguaje en la mayoría de las computadoras personales.

No obstante este lenguaje no resulta tan adecuado como el COBOL para las aplicaciones de procesamiento de datos de negocios y para complicadas operaciones aritméticas es superado por el FORTRAN y el BASIC.

CONCEPTO DE PROGRAMA. PROGRAMA FUENTE Y PROGRAMA OBJETO. COMPILADORES: CONCEPTO Y FUNCIONES.

Como ya se mencionó, las computadoras pueden ejecutar programas solo después de que estos han sido traducidos al lenguaje de máquina.

Hay dos motivos por los cuales las personas generalmente no escriben programas en este lenguaje:

Primero, las instrucciones del lenguaje de máquina constan de cadenas de apariencia compleja de ceros y unos. Por ejemplo:

010011110101010101010000111

Segundo, las instrucciones en el lenguaje de máquina deben ser escritas en el nivel de exposición más detallado. Por ejemplo, la computadora no puede sumar directamente A y B, colocando el resultado en C, con una sola instrucción como

C = A + B

Aún una simple tarea como ésta requiere tres o más instrucciones en lenguaje de máquina, como:

1. Cargar el valor representado por A de la memoria principal en un registro.
2. Sumar el valor representado por B de la memoria principal en el mismo registro.
3. Colocar la suma obtenida en otra zona de almacenamiento.

Estas instrucciones detalladas, a veces se denominan microinstrucciones, ya que no pueden subdividirse en comandos más pequeños. Una instrucción como C = A + B, por otro lado, es un ejemplo de macroinstrucción.

Las macroinstrucciones deben ser divididas en microinstrucciones por el sistema computacional antes de ser procesadas.

Todos los lenguajes de alto nivel (como BASIC, FORTRAN y COBOL) utilizan este tipo de instrucciones para ahorrar al operador la tediosa tarea de explicar en detalle a la computadora como hacer el trabajo.

Un traductor de lenguaje es simplemente un programa de sistemas que convierte un programa con macroinstrucciones en uno con microinstrucciones en base binaria.

Los tipos de traductores de lenguajes son: ensamblador, compiladores e intérpretes.

ENSAMBLADORES:

El ensamblador, se utiliza exclusivamente con los lenguajes ensambladores. Trabaja como un compilador, produciendo un módulo objeto que puede almacenarse.

Cada sistema computacional tiene comúnmente solo un lenguaje ensamblador a su disposición; así, solo necesita adquirirse un ensamblador.

COMPILADORES:

Un compilador traduce un programa escrito en lenguaje de alto nivel a lenguaje de máquina completamente de una sola vez. Todo lenguaje orientado a los compiladores requiere su propio compilador. Así un programa escrito en lenguaje COBOL necesita un compilador COBOL, no puede traducirse con un compilador FORTRAN. Además, un compilador que funcione con determinada computadora casi seguramente no podrá utilizarse en otra distinta, a menos que exista una cierta compatibilidad entre ellas y el resultado de la compilación también está sometido a consideraciones similares, excepto en los casos de compilación cruzada (se compila en un equipo para que se ejecute en otro específico).

El programa que se escribe en un lenguaje de alto nivel y que se introduce en la computadora se conoce como módulo fuente (o programa fuente).

El programa escrito en lenguaje de máquina que el compilador produce a partir de él es un módulo objeto (o programa objeto).

Antes de que el módulo objeto esté en condiciones de ser ejecutable, por lo común se une a otros módulos objeto que la CPU puede necesitar a fin de procesar el programa. Por ejemplo, la mayoría de las computadoras no pueden calcular directamente raíces cuadradas.

Para hacerlo, se apoyan en pequeños subprogramas, los cuales están almacenados en memoria secundaria en forma de módulos objetos. De este modo, si un programa pide el cálculo de una raíz cuadrada, el sistema operativo unirá la versión del módulo objeto del programa con esta rutina de raíz cuadrada a fin de formar un “paquete ejecutable” para la computadora.

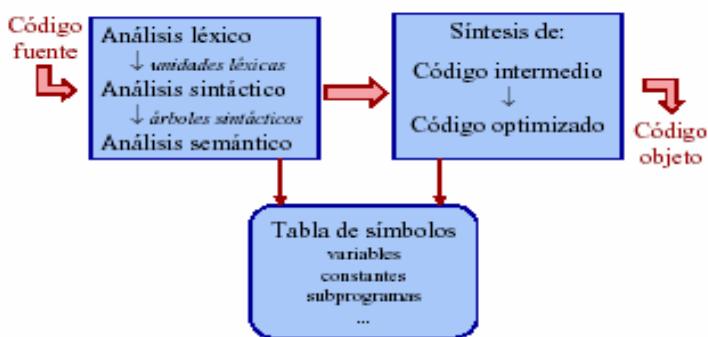
El proceso de unión se conoce como edición de enlace (o etapa de edición de enlace), y el paquete ejecutable que se forma se denomina módulo de carga (o también módulo ejecutable o programa ejecutable).

Los sistemas de computación cuentan con un programa de sistemas especial, denominado editor de enlace, para realizar el enlace de manera automática. Efectivamente, la mayoría de las personas que escriben sus propios programas ni siquiera se dan cuenta de que ocurre la edición de enlace, el sistema operativo se encarga automáticamente de esta operación.

Es el módulo de carga el que la computadora ejecuta en realidad.

Tanto los módulos objeto como los de carga pueden almacenarse en disco para su uso posterior, de modo que la compilación y la edición de enlace no necesitan realizarse cada vez que se ejecute el programa.

- Estructura de un compilador:



INTÉRPRETES: Un intérprete, a diferencia de un compilador, no crea un módulo objeto. Los intérpretes leen, traducen y ejecutan programas fuentes una línea a la vez. De este modo, la traducción al lenguaje de máquina se realiza mientras el programa está siendo ejecutado.

Los intérpretes tienen ventajas y desventajas en relación con los compiladores.

La ventaja principal es que un intérprete requiere mucho menos espacio de almacenamiento. Asimismo, el intérprete no genera un módulo objeto que tenga que ser almacenado.

Muchas versiones del lenguaje BASIC utilizan intérpretes en vez de compiladores, y por esta razón requieren menos almacenamiento que los lenguajes orientados al compilador, como es el caso de COBOL y FORTRAN. Esta es una razón principal por la que el lenguaje BASIC es tan popular en las microcomputadoras, las cuales tienen capacidad limitada de almacenamiento.

La desventaja principal de los intérpretes es que son más lentos y menos eficientes que los compiladores. El programa objeto producido por un compilador se encuentra completamente en lenguaje de máquina, de modo que puede ejecutarse rápidamente.

Los intérpretes, en contraste, traducen cada instrucción inmediatamente antes de ejecutarla, lo cual lleva más tiempo debido a que debe reiterarse este proceso cada vez que se ejecute una instrucción.

Además, el módulo objeto de un programa compilado puede almacenarse en disco, de modo que el programa fuente no tiene que volver a traducirse cada vez que se ejecute el programa; con un intérprete el programa debe ser traducido cada vez que se ejecute.

Lectura 11. El software de base sistemas operativos y lenguajes

UNIDAD TEMÁTICA 3

El software de base. Sistemas operativos y lenguajes.

Software de base: concepto.

Sistema operativo: concepto y funciones. Sistemas operativos para PC.

Programas utilitarios: concepto, clasificación.

Lenguajes de programación: niveles, paradigmas, orientaciones, traductores.

3.1) SOFTWARE

Todo computador trabaja sobre la base de un programa (conjunto de instrucciones ordenadas en una secuencia predeterminada, siendo cada instrucción una orden que se imparte al computador indicándole lo que debe hacer y usar para llevar a cabo una tarea).

Se denomina software al conjunto de programas y se lo clasifica en:

- a) Software de Base.
- b) Software de Aplicación o Aplicativo.

Software de Base: Es toda aquella parte lógica realizada generalmente por cada fabricante de computadores o por casas especializadas en el desarrollo de software de base.

Este conjunto de programas tiene por función coordinar las diversas partes del sistema computacional para hacerlo funcionar rápida y eficazmente, actuando como mediadores entre los programas de aplicaciones y el hardware del sistema, interpretando los requerimientos de cada programa que ingresa al sistema, poniendo a su disposición cualquiera de los recursos que necesite (ya sean de hardware, software o datos) para producir los resultados deseados.

Debido a la estrecha relación que existe entre las características de un computador y su software de base, no se puede concebir el uno sin el otro, a tal punto que pareciera que el software es parte integrante del hardware.

El software de base se suele clasificar en:

- a) Sistema Operativo
- b) Utilitarios
- c) Traductores de lenguajes

3.2.1) Sistema Operativo: Concepto y Funciones

Concepto: Es un conjunto de programas concebidos para efectuar la administración de los recursos del computador.

Del conjunto de instrucciones que maneja el computador, algunas residen permanentemente en la memoria central (luego de la ejecución del programa de carga inicial -IPL-) durante todo el procesamiento, mientras que otras residen solo cuando se las necesita, encontrándose almacenadas en periféricos cuando no están en la memoria central. A las primeras se las conoce como residentes, supervisor, monitor y ejecutivo. A las segundas como transientes.

Funciones:

El Sistema Operativo tiene dos funciones básicas, globalmente consideradas:

1) Proveer servicios para la ejecución de programas de aplicación y para el desarrollo de los mismos, es decir administrar los recursos en proceso, obtener automáticamente la rutina apropiada y mantener el computador sin necesidad de operación manual.

2) Actuar como entorno de la aplicación en la cual el programa es ejecutado, planeando los recursos y trabajos, puesto que ayuda a decir no solo qué recursos utilizar (asignación), sino también cuándo utilizarlos (planificación). Debido a que los dispositivos de Entrada-Salida trabajan mucho más lentamente que la C.P.U., pueden realizarse millones de instrucciones de cálculo para varios programas, mientras que los resultados se imprimen o muestran por pantalla.

Utilizando varias técnicas el Sistema Operativo combina los diversos trabajos que deben realizarse de modo que los dispositivos del sistema se empleen lo más eficientemente posible.

Las instrucciones que conforman un Sistema Operativo, materializan entre otras, las siguientes tareas en la ejecución de una aplicación:

Las instrucciones que conforman un Sistema Operativo, realizan entre otras, las siguientes tareas en la ejecución de una aplicación:

- Carga de programas.
- Gestión del tiempo de procesamiento.
- Gestión de la memoria principal.
- Gestión de la memoria secundaria (ficheros y directorios).
- Gestión del subsistema de e/s (drivers).
- Seguridad y protección del sistema.
- Interfaz de llamadas al sistema.
- Interfaz de usuario y utilidades del sistema.
- Tareas de comunicación de datos (teleprocesamiento).

Generalmente en algunos equipos el Sistema Operativo cumple funciones de monitoreo: lleva registro de las actividades del computador mientras se realiza el procesamiento. El Sistema Operativo detiene los programas que contienen errores o exceden, ya sea su tiempo máximo de ejecución o sus asignaciones de almacenamiento. Mediante el envío de mensajes informa las anomalías en los dispositivos de Entrada-Salida o en otra parte del sistema. Son también parte del Sistema Operativo la contabilización o registro de hora de ingreso y egreso, y el tiempo de duración de los programas, lo que hace posible elaborar facturas por concepto de utilización del sistema por parte de los usuarios.

Posee además mecanismos de seguridad para proteger contra el acceso no autorizado a través de la verificación de identificación ("claves" o "passwords").

3.2.2 Sistemas Operativos para PC.

La primera IBM-PC aparece con un sistema operativo desarrollado por Microsoft, denominado DOS 1.0. El mismo administraba la PC con tan solo disketteras. Aparece luego la versión 2 que controlaba también discos rígidos. La 3 agrega posibilidades de compartir dispositivos (en una red). Luego vinieron las versiones 4, 5 y 6, que fueron incorporando más utilidades (compresión de discos, resguardo, verificadores, ayudas, interfase gráfica elemental, etc..). También han sido desarrollados otros sistemas por otras compañías, como por ejemplo Digital Research que desarrolló el DR DOS cuya primera versión fue la 5.0, le siguieron la 6 y la 7, también agregando más utilidades y mejor integración a ambientes de red.

Asimismo y en forma paralela, Microsoft comienza el desarrollo de un entorno operativo que permitía una interfase gráfica mas sencilla e intuitiva para el usuario, tomando como ideas los desarrollos realizados por un área de la empresa Xerox, los que también han sido llevados a otras arquitecturas de equipos como las Apple Lisa y

Macintosh.

Este entorno se ejecutaba por encima del DOS, y no tuvo una significativa aceptación en sus versiones 1 y 2. No obstante las mejoras introducidas a la versión 3 y la mayor potencia que se tenía en el hardware, posibilitaron su gradual utilización. En realidad constituyó todo un suceso, que se afirmó con las siguientes versiones (3.1, 3.11 y 3.11 para Grupos de Trabajo). Paralelamente Microsoft estaba trabajando con IBM en el desarrollo de un nuevo sistema operativo gráfico, denominado OS/2. Por distintas motivaciones, IBM se desvincula de Microsoft en este proyecto y lo continúa sola, ofreciendo luego comercialmente a este producto. Por su parte Microsoft desarrolla una nueva versión de Windows que a diferencia de las anteriores es un sistema operativo y no solo un entorno. Guarda compatibilidad con las anteriores pero tiene significativas mejoras, tanto en lo estético como en lo funcional. La denominó Windows 95. Lanza luego otra versión destinada a un segmento de equipos de mayores requerimientos y/o administración de recursos en red de área local, que denominó Windows NT. La versión menor de Windows, la 95, tuvo una actualización denominada Windows 98, que continuó con la orientación de la 95, mejorando performance, y agregando utilidades. Por su parte la mayor, NT, también ha ido evolucionando, apareciendo las versiones 3.5, y 4. A partir de estos últimos, Microsoft ofrece una familia de productos basada en su Sistema Operativo (dependiendo de los requerimientos del usuario NT Workstation para equipos autónomo con un único usuario con altos requerimientos o NT Server para servidores de red, administrando redes de área local, preferentemente en modo dedicado).

En el segmento menor, Microsoft desarrolla para el año 2000 una versión que denominó Millenium, y para el segmento mayor actualiza el NT con una versión que denominó 2000.

3.3.) Programas utilitarios: concepto, clasificación.

Llamamos **utilitarios** a aquellos programas entregados por el fabricante, comprados a terceros o desarrollados en la propia instalación, de uso general en todo equipo, escritos con el objeto de realizar tareas repetitivas de procesamiento de datos.

Estas tareas se realizan con tanta frecuencia en el curso del procesamiento, que sería extremadamente ineficiente el que cada usuario tuviera que codificarlas en forma de programas una y otra vez.

Desde el punto de vista de las funciones que cumplen, los podemos agrupar en:

- Utilitarios de apoyo a los sistemas de aplicación: Estos programas se integran al sistema de aplicación, es decir, que su función formar parte de la secuencia de procesamiento necesaria para operar el sistema de aplicación; por ejemplo: generador de copias de archivo, generador de listados, clasificador e intercalador de archivos, etc.
- Utilitarios de Servicios: Por un lado se incluyen en este grupo un conjunto de utilitarios que ayudarán a manejar ciertos recursos del computador, y por otro a los utilitarios para el manejo de programas y sus bibliotecas; por ejemplo: listador del directorio de un disco, inicializador de discos, diskette, cinta, cassette, el que elimina o renombra archivos, el reorganizador de espacios en discos, los compiladores y compaginadores, etc.

3.4. Lenguajes de programación: niveles, paradigmas, orientaciones, traductores

Un lenguaje es el conjunto finito de símbolos básicos permitidos, combinados de acuerdo con ciertas reglas del lenguaje a las que se denominan reglas de sintáctica.

En los primeros días de la computadora, a fines de la década de 1940, cada programa (o sea la serie de instrucciones que indica a la computadora el trabajo que se va a hacer) tenía que estar escrito en lenguaje de máquina. El único que una computadora puede entender directamente y que consta de combinaciones de ceros y unos.

Todos los usuarios tenían que escribir programas compuestos de largas cadenas de ceros y unos para especificar numéricamente la dirección de los datos y los códigos de operaciones que se debían ejecutar en la máquina.

Varios años mas tarde, se desarrollaron programas llamados traductores, los cuales aceptaban como entrada cierto lenguaje simbólico o mnemotécnico para luego convertirlo automáticamente en lenguaje de máquina.

Estos traductores se conocen como ensambladores, que, aunque ahorraban al usuario mucho trabajo, no eran lo suficientemente atractivos para ellos puesto que resultaba molesto tener que especificar, aunque simbólicamente, direcciones y códigos de operaciones.

Para resolver problemas, uno tenía que programar todavía en un lenguaje parecido al de máquina.

Estos lenguajes reciben el nombre de lenguajes de bajo nivel, debido a que, como dijimos anteriormente, los programadores debían escribir instrucciones con el mas fino nivel de detalle dado que la traducción que se realiza es uno-a-uno (cada línea de código corresponde a una sola acción del sistema computacional).

Los siguientes lenguajes que aparecieron fueron los lenguajes de alto nivel en los que se introduce el concepto de macroinstrucción (la traducción es una instrucción de alto nivel a muchas de bajo nivel, una-a-muchas).

Dentro de esta categoría se encuentran lenguajes tales como BASIC, COBOL, FORTRAN, PASCAL, PL/1, APL, C, etc.

Los lenguajes de alto nivel difieren de sus antecesores de bajo nivel en que requieren menos detalle de codificación. Los traductores que convierten el programa escrito en lenguaje de alto nivel al lenguaje de máquina proporcionan el detalle.

Como resultado los programas escritos en lenguaje de alto nivel son menos extensos y mas fáciles de escribir que aquellos escritos en lenguaje de bajo nivel.

Los lenguajes de muy alto nivel, que aparecieron por primera vez en la década de 1960, se crearon para cubrir necesidades especializadas del usuario y son relativamente fáciles de aprender y de utilizar por lo que se los denominan "amigables" para el usuario. Con los lenguajes de muy alto nivel solo se necesita prescribir lo que la computadora hará en vez de como hacerlo.

Existen muchos lenguajes de muy alto nivel en el mercado y por lo general hay más de uno por cada tarea de aplicaciones:

- Generadores de informes (DMS, RPG).
- Generadores de programas (se los conoce como 4to.nivel).
- Software para procesamiento de palabras.
- Hojas o planillas electrónicas.
- Paquetes de graficación.
- etc..

La tendencia es acortar la brecha de comunicación entre hombre y máquina permitiendo que los no especialistas usen la computadora en un amplio número de disciplinas y prueben sus beneficios.

Con la venida de la nueva tecnología y de la nueva generación de computadoras, los lenguajes y sistemas en línea han sido y están siendo desarrollados para interactuar más como le gusta al hombre: rápidamente y de un modo conversacional.

LENGUAJES ORIENTADOS AL PROBLEMA Y AL PROCEDIMIENTO.

Los lenguajes de bajo y alto nivel se conocen como lenguajes de procedimientos, debido a que requieren que las personas escriban procedimientos detallados que indiquen a la computadora como realizar tareas individuales.

Los lenguajes de muy alto nivel, en contraste, reciben el nombre de lenguajes orientados al problema puesto que cada uno fue creado para resolver un problema en especial.

Además, en un amplio rango de aplicaciones, es fácil distinguir si la misma tiene características 'administrativo-contables' o 'científico-técnicas'.

Las primeras se caracterizan por requerir el manejo de un número elevado de datos, normalmente organizados en archivos, y realizar pocas operaciones sencillas con ellos. Por el contrario, las aplicaciones científico-técnicas utilizan comparativamente menor número de datos pero realizan un mayor y más complejo cálculo con ellos.

Muchos lenguajes de alto nivel o evolucionados tuvieron en cuenta estos aspectos y por lo tanto se encontraban orientados para cumplir más eficientemente alguno de los dos tipos de procesamiento tipificados anteriormente. La evolución que luego han sufrido estos lenguajes ha hecho que paulatinamente se tornen más aptos para cualquier tipo de procesos, aunque mantienen su mejor predisposición para el cual fueron diseñados.

Así por ejemplo el COBOL surge como un lenguaje para resolver los problemas del área administrativa y el FORTRAN lo hace para el área científica.

BASIC: Características del lenguaje. Estructura del programa. Definición de datos. Enunciados.

El BASIC, cuyo nombre proviene de las siglas **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode (código de instrucción simbólica de uso general para principiantes), es un lenguaje fácil de aprender y que, al paso de los años, se ha convertido en uno de los lenguajes de programación más populares y de más fácil adquisición en los proveedores especializados.

Debido a que las necesidades de almacenamiento de su traductor de lenguaje son pequeñas, trabaja con eficiencia en casi todas las computadoras personales.

Existen muchas versiones del lenguaje BASIC, desde las simplificadas, que se utilizan en computadoras de bolsillo, hasta las poderosas versiones para computadoras a gran escala que compiten con el poder de procesamiento del COBOL.

En BASIC cada instrucción se suele identificar con un número de línea: por ejemplo 10, 20, 30, etc.

La computadora siempre ejecutará las instrucciones en la secuencia especificada por los números de líneas a menos que se ordene lo contrario mediante las instrucciones de ruptura de secuencia (IF, GOTO, GOSUB, etc.).

Cada instrucción comienza con una palabra clave, la cual indica a la computadora que tipo de operación debe realizar: por ejemplo REM, READ, LET, PRINT, DATA, etc. Estas palabras claves pueden considerarse el vocabulario del sistema computacional cuando se escriben programas en lenguaje BASIC.

Uno debe siempre apegarse estrictamente a este vocabulario. Si, por ejemplo, se sustituye DATA por DATE, la computadora no sabrá que es lo que uno quiere que haga.

A pesar de sus muchas ventajas, una debilidad importante que presentan muchas versiones de este lenguaje es que no están diseñadas para facilitar la programación estructurada.

Un programa largo y no estructurado escrito en BASIC puede resultar difícil de seguir. Asimismo, ya que existen tantas versiones del lenguaje BASIC, un programa desarrollado en una computadora puede requerir modificaciones sustanciales para ejecutarse en otra máquina o en otra versión de traductor o intérprete del lenguaje.

COBOL. Características del lenguaje. Estructura del programa. Divisiones.

El COBOL, cuyo nombre proviene de las siglas **CO**mmon **B**usiness **O**riented **L**enguaje (lenguaje común orientado a los negocios), fue introducido por primera vez en los inicios de la década de 1960.

Casi todas las características principales del COBOL se relacionan con su orientación al procesamiento de datos de negocios, incluso la independencia de la máquina, la autodocumentación, y la orientación a la entrada y salida.

Independencia de la máquina: es un aspecto importante ya que los programas para el procesamiento de datos de negocios generalmente tienen que durar mucho tiempo (10 o incluso 20 años).

Durante este período, una organización puede comprar nuevo hardware o cambiar completamente de un sistema computacional a otro. De este modo, los programas escritos para un sistema deben poder ejecutarse en otros con pequeñas modificaciones.

Autodocumentación: debido a que los programas de procesamiento de datos de negocios deben durar un largo tiempo, necesitan mantenimiento continuo. Por ello, es extremadamente importante que la lógica del programa sea fácil de seguir por otros programadores o aún por el mismo que lo codificó después de transcurrido un período de tiempo.

El lenguaje COBOL se presta para un buen diseño de programas en tres formas: legibilidad, modularidad, y uso adecuado de las tres estructuras básicas de control de la diagramación estructurada: (Secuencia, bifurcación e iteración).

El lenguaje COBOL también utiliza verbos del idioma inglés (como SUBTRACT, MOVE, ADD, etc.) y conectivos (como FROM, GIVING, etc.).

Orientación a la entrada y salida: el procesamiento de datos de negocios, en contraste a las aplicaciones científicas y de ingeniería, implica la manipulación de grandes archivos con muchos registros.

Así, gran parte del trabajo en aplicaciones del procesamiento de datos de negocios se relaciona con la lectura y escritura de registros, y el lenguaje COBOL se ha diseñado para ser particularmente efectivo en esta tarea.

Contiene estipulaciones para definir de manera explícita y fácil el formato de los registros de entrada y salida. Por ejemplo, es un proceso muy sencillo el de editar cantidades monetarias , la salida con signos, puntos decimales, comas, y también redondear las cantidades.

Estructura del programa: todo programa escrito en lenguaje COBOL se agrupa en cuatro divisiones:

- División de identificación: en la que se identifica el nombre del programa, el autor, fecha de escritura y otros detalles. Esta división existe principalmente con fines de documentación.
- División de ambiente: en la que los nombres de archivos creados por el programador se vinculan a un equipo específico de entrada/salida. Aquí, por ejemplo, el programador especificaría que un archivo de entrada en particular, digamos ARCHIVO-DISCO, se localiza en disco y que un archivo de salida en particular, como ARCHIVO-IMPRESION, se dirigirá a la impresora.
- División de datos: en la que el programador nombra y define todas las variables del programa e indica su relación mutua.
- División de procedimientos: en la cual se especifican los procedimientos reales que la computadora debe seguir para crear la salida deseada.

Las tres primeras divisiones aseguran que todas las especificaciones importantes se establezcan en forma explícita en el programa.

Desventajas: Los programas escritos en lenguaje COBOL tienden a ser extensos y además se necesita un traductor de lenguaje grande y complejo para convertir los programas en el lenguaje de máquina, lo cual hace al COBOL difícil de implantar en computadoras pequeñas.

Por lo general no resulta adecuado para aplicaciones científicas y de ingeniería, las cuales utilizan demasiadas fórmulas complicadas.

Otros lenguajes de programación. Características generales.

FORTRAN cuyo nombre proviene de **FOR**mula **TRAN**slator (traductor de fórmulas), data del año 1954 y es el lenguaje comercial de alto nivel superviviente más antiguo.

Fue diseñado por científicos y está orientado hacia la resolución de problemas científicos y de ingeniería. La principal característica del FORTRAN es su capacidad para expresar con facilidad fórmulas complicadas. Aunque el BASIC es competitivo en esta tarea, el FORTRAN es generalmente superior para muchas aplicaciones debido a que hace posible una ejecución más rápida del programa y una mayor precisión, aunque hay versiones recientes del BASIC que alcanzan y aún superan sus prestaciones.

El FORTRAN por lo general utiliza un compilador como traductor del lenguaje.

Los compiladores ejecutan los programas más rápido que los intérpretes, que se utilizan en muchas de las versiones BASIC.

La lógica de los programas escritos en FORTRAN es más difícil de seguir que la lógica de algunos otros lenguajes, y es claramente inferior al COBOL para aplicaciones de procesamiento de datos de negocios.

PASCAL es un lenguaje relativamente nuevo, creado hacia 1970 para cubrir la necesidad de contar con una herramienta para la enseñanza de la programación estructurada.

Los compiladores del lenguaje PASCAL son extremadamente pequeños, lo que facilita la implementación de este lenguaje en la mayoría de las computadoras personales.

No obstante este lenguaje no resulta tan adecuado como el COBOL para las aplicaciones de procesamiento de datos de negocios y para complicadas operaciones aritméticas es superado por el FORTRAN y el BASIC.

CONCEPTO DE PROGRAMA. PROGRAMA FUENTE Y PROGRAMA OBJETO. COMPILADORES: CONCEPTO Y FUNCIONES.

Como ya se mencionó, las computadoras pueden ejecutar programas solo después de que estos han sido traducidos al lenguaje de máquina.

Hay dos motivos por los cuales las personas generalmente no escriben programas en este lenguaje:

Primero, las instrucciones del lenguaje de máquina constan de cadenas de apariencia compleja de ceros y unos. Por ejemplo:

010011110101010101010000111

Segundo, las instrucciones en el lenguaje de máquina deben ser escritas en el nivel de exposición más detallado. Por ejemplo, la computadora no puede sumar directamente A y B, colocando el resultado en C, con una sola instrucción como

C = A + B

Aún una simple tarea como ésta requiere tres o más instrucciones en lenguaje de máquina, como:

1. Cargar el valor representado por A de la memoria principal en un registro.
2. Sumar el valor representado por B de la memoria principal en el mismo registro.
3. Colocar la suma obtenida en otra zona de almacenamiento.

Estas instrucciones detalladas, a veces se denominan microinstrucciones, ya que no pueden subdividirse en comandos más pequeños. Una instrucción como C = A + B, por otro lado, es un ejemplo de macroinstrucción.

Las macroinstrucciones deben ser divididas en microinstrucciones por el sistema computacional antes de ser procesadas.

Todos los lenguajes de alto nivel (como BASIC, FORTRAN y COBOL) utilizan este tipo de instrucciones para ahorrar al operador la tediosa tarea de explicar en detalle a la computadora como hacer el trabajo.

Un traductor de lenguaje es simplemente un programa de sistemas que convierte un programa con macroinstrucciones en uno con microinstrucciones en base binaria.

Los tipos de traductores de lenguajes son: ensamblador, compiladores e intérpretes.

ENSAMBLADORES:

El ensamblador, se utiliza exclusivamente con los lenguajes ensambladores. Trabaja como un compilador, produciendo un módulo objeto que puede almacenarse.

Cada sistema computacional tiene comúnmente solo un lenguaje ensamblador a su disposición; así, solo necesita adquirirse un ensamblador.

COMPILADORES:

Un compilador traduce un programa escrito en lenguaje de alto nivel a lenguaje de máquina completamente de una sola vez. Todo lenguaje orientado a los compiladores requiere su propio compilador. Así un programa escrito en lenguaje COBOL necesita un compilador COBOL, no puede traducirse con un compilador FORTRAN. Además, un compilador que funcione con determinada computadora casi seguramente no podrá utilizarse en otra distinta, a menos que exista una cierta compatibilidad entre ellas y el resultado de la compilación también está sometido a consideraciones similares, excepto en los casos de compilación cruzada (se compila en un equipo para que se ejecute en otro específico).

El programa que se escribe en un lenguaje de alto nivel y que se introduce en la computadora se conoce como módulo fuente (o programa fuente).

El programa escrito en lenguaje de máquina que el compilador produce a partir de él es un módulo objeto (o programa objeto).

Antes de que el módulo objeto esté en condiciones de ser ejecutable, por lo común se une a otros módulos objeto que la CPU puede necesitar a fin de procesar el programa. Por ejemplo, la mayoría de las computadoras no pueden calcular directamente raíces cuadradas.

Para hacerlo, se apoyan en pequeños subprogramas, los cuales están almacenados en memoria secundaria en forma de módulos objetos. De este modo, si un programa pide el cálculo de una raíz cuadrada, el sistema operativo unirá la versión del módulo objeto del programa con esta rutina de raíz cuadrada a fin de formar un “paquete ejecutable” para la computadora.

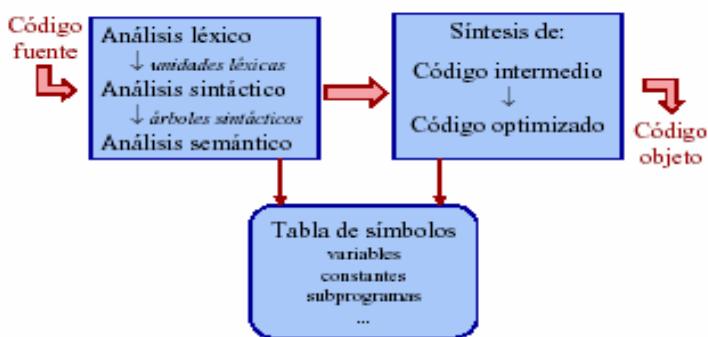
El proceso de unión se conoce como edición de enlace (o etapa de edición de enlace), y el paquete ejecutable que se forma se denomina módulo de carga (o también módulo ejecutable o programa ejecutable).

Los sistemas de computación cuentan con un programa de sistemas especial, denominado editor de enlace, para realizar el enlace de manera automática. Efectivamente, la mayoría de las personas que escriben sus propios programas ni siquiera se dan cuenta de que ocurre la edición de enlace, el sistema operativo se encarga automáticamente de esta operación.

Es el módulo de carga el que la computadora ejecuta en realidad.

Tanto los módulos objeto como los de carga pueden almacenarse en disco para su uso posterior, de modo que la compilación y la edición de enlace no necesitan realizarse cada vez que se ejecute el programa.

- Estructura de un compilador:



INTÉRPRETES: Un intérprete, a diferencia de un compilador, no crea un módulo objeto. Los intérpretes leen, traducen y ejecutan programas fuentes una línea a la vez. De este modo, la traducción al lenguaje de máquina se realiza mientras el programa está siendo ejecutado.

Los intérpretes tienen ventajas y desventajas en relación con los compiladores.

La ventaja principal es que un intérprete requiere mucho menos espacio de almacenamiento. Asimismo, el intérprete no genera un módulo objeto que tenga que ser almacenado.

Muchas versiones del lenguaje BASIC utilizan intérpretes en vez de compiladores, y por esta razón requieren menos almacenamiento que los lenguajes orientados al compilador, como es el caso de COBOL y FORTRAN. Esta es una razón principal por la que el lenguaje BASIC es tan popular en las microcomputadoras, las cuales tienen capacidad limitada de almacenamiento.

La desventaja principal de los intérpretes es que son más lentos y menos eficientes que los compiladores. El programa objeto producido por un compilador se encuentra completamente en lenguaje de máquina, de modo que puede ejecutarse rápidamente.

Los intérpretes, en contraste, traducen cada instrucción inmediatamente antes de ejecutarla, lo cual lleva más tiempo debido a que debe reiterarse este proceso cada vez que se ejecute una instrucción.

Además, el módulo objeto de un programa compilado puede almacenarse en disco, de modo que el programa fuente no tiene que volver a traducirse cada vez que se ejecute el programa; con un intérprete el programa debe ser traducido cada vez que se ejecute.

Lectura 12. Algoritmos y estructuras de datos

ALGORITMOS

Y ESTRUCTURAS DE DATOS

APUNTE DE TEORIA

AUTORA: Ing. ESTELA M. SORRIBAS

ALGORITMOS Y ESTRUCTURAS DE DATOS

► INTRODUCCIÓN:

El desarrollo de la tecnología de la información y de las comunicaciones, ha sido responsable de una buena parte de los cambios sociales y productivos en el mundo de las últimas décadas.

Las sociedades se distinguen entre sí por la complejidad de los problemas que puedan resolver, para lo cual deben acceder al conocimiento. Este acceso al conocimiento depende de cómo se procesa, almacena y transmite la información en un país. Para brindar respuesta a esta necesidad social, la educación juega un papel muy importante.

Una de las prioridades de los sistemas educativos de los países que pretendan un crecimiento económico y un desarrollo social sustentable, es la alfabetización en tecnología.

El área Programación tiene como objetivo formar e informar acerca de metodologías, técnicas y lenguajes de programación, como herramientas básicas para el desarrollo de software y el estudio de disciplinas que permitan crear nuevas tecnologías.

► OBJETIVOS:

Esta asignatura, tiene como primer objetivo presentar a la programación como el arte o la técnica de construir y formular algoritmos en forma sistemática. Se debe aprender a proceder metódica y sistemáticamente en el diseño de algoritmos mediante la demostración de problemas y técnicas que son típicas de la programación, pero independientes del área de aplicación en particular. Por esta razón, ningún área de aplicación específica se enfatiza como un fin. Con el mismo espíritu se resta importancia a la notación y al lenguaje de programación, el lenguaje es nuestra herramienta, no un fin en sí mismo.

El fin primario de los cursos de programación no debe ser enseñar la perfección en el conocimiento de todas las características e idiosincrasias de un lenguaje en particular, sino entender el problema que se nos pide resolver y resolverlo de una manera comprensible y sobre todo confiable.

• Objetivos Generales:

- Mejorar la capacidad de razonamiento.
- Adquirir habilidad y seguridad en la resolución de problemas.
- Desarrollar aptitudes que les permitan seguir aprendiendo por sí mismos.

- Desarrollar hábitos de observación, razonamiento, orden, autocritica y trabajo metódico.
- Fomentar la participación activa creando un ambiente de aprendizaje creativo.

● Objetivos Específicos:

- Formular un problema en forma correcta, completa y sin ambigüedades.
- Utilizar los conocimientos adquiridos para elegir un método para hallar la solución de los problemas.
- Expresar el método elegido de forma tal que pueda ser interpretado por el procesador a utilizarse.
- Reconocer datos e incógnitas
- Elegir correctamente la estructura de datos.
- Ejecutar el procedimiento elegido para obtener la solución del problema.
- Expresar el algoritmo en lenguaje de programación.

► CONTENIDOS:

La secuencia de contenidos conceptuales se organizó en siete unidades didácticas:

◊ Unidad 1:

Algoritmo, Programa, Lenguaje de programación, Lenguaje de máquina, Compilador – Definiciones

Representación de algoritmos – Diagramación – Diagramas de Nassi- Schneiderman o de Chapin.

Diseño general de un algoritmo: Partes básicas.

Constantes y variables – Identificadores – Asignación – Operadores Matemáticos
Operadores relacionales – Definición.

◊ Unidad 2:

Introducción al Pascal – Programa Pascal – Encabezamiento – Bloque – Cuerpo
Declaraciones y Definiciones – Tipos de Datos standard: enteros, reales,
caracteres y lógicos

Cuerpo – Sentencia de asignación – Sentencia de entrada – Sentencia de Salida – Salidas formateadas

◊ Unidad 3:

Estructuras de Control

Bifurcación ó Selección Simple – Diagrama y programa -

Repetición ó Iteración: con cantidad conocida de veces y con cantidad desconocida de veces – Diagrama y programa

Selección múltiple – Diagrama y programa

◊ Unidad 4:

Tipos de Datos no standard o definidos por el programador

Tipo Enumerado ó Escalar

Tipo Subrango ó Intervalo

Tipo Estructurado – Arreglos unidimensionales y multidimensionales

Ordenamiento de un arreglo unidimensional – Búsqueda de un valor en un arreglo unidimensional: Búsqueda Secuencial y Búsqueda Dicotómica

Intercalación de arreglos unidimensionales ordenados

Ordenamiento y Búsqueda en un arreglo bidimensional

◊ Unidad 5:

Subprogramas – Definición

Funciones y Procedimientos – Definiciones – Diferencias

Variables Locales – Variables Globales

Correspondencia Argumento-Parámetro

Parámetro por valor – Parámetro por Referencia ó por Variable

◊ Unidad 6:

Otras Estructuras de Datos

Registros – Registros Jerárquicos

Arreglos de Registros

◊ Unidad 7:

Archivos – Introducción - Buffers

Operaciones básicas sobre archivos – Otras operaciones sobre archivos

Organización y acceso a un archivo

► BIBLIOGRAFIA:

- ***INTRODUCCIÓN A LA PROGRAMACIÓN Y A LAS ESTRUCTURAS DE DATOS***
Silvia Braunstein ; Alicia Gioia - EUDEBA
- ***INTRODUCCIÓN AL PASCAL***
Nell Dale ; Orshalick – McGraw Hill
- ***PASCAL MAS ESTRUCTURAS DE DATOS***
Nell Dale ; Susan Lilly – McGraw Hill
- ***ALGORITMOS, DATOS Y PROGRAMAS CON APLICACIONES EN PASCAL, DELPHI Y VISUAL DA VINCI***
Armando E. de Giusti – Prentice Hall
- ***ALGORITMOS + ESTRUCTURAS DE DATOS = PROGRAMAS***
Niklaus Wirth – El Ateneo

UNIDAD N° 1

1-1. INTRODUCCIÓN

Dentro de los objetivos planteados para esta asignatura aparecen palabras tales como: algoritmos, programas, lenguaje de programación, etc., con las cuales no estamos familiarizados; para entenderlos mejor veamos algunas definiciones:

1-1-1. ALGORITMOS:

Secuencia de acciones o pasos que permite resolver un problema. Un mismo problema puede ser resuelto con distintos algoritmos.-

1-1-2. PROGRAMA:

Traducción o codificación de un algoritmo a un lenguaje de programación; o sea, una secuencia de instrucciones que indica las acciones que han de ejecutarse.-

1-1-3. LENGUAJE DE PROGRAMACION:

Conjunto de reglas, símbolos y palabras especiales utilizadas para construir un programa.-

1-1-4. LENGUAJE DE MAQUINA:

Lenguaje usado directamente por la computadora y compuesto de instrucciones codificadas en sistema binario.-

1-1-5. COMPILADOR:

Programa que traduce un programa escrito en lenguaje de alto nivel a lenguaje de máquina.-

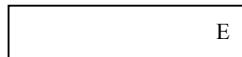
1-2. DIAGRAMACION:

Una vez comprendido el problema, se hace una representación gráfica de los pasos a seguir para resolverlo. Esta representación se llama diagramación.-

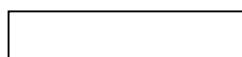
Los diagramas son un conjunto de símbolos que han convenido de distintas maneras distintos autores. En este curso adoptaremos los diagramas de NASSI-

SCHNEIDERMAN, más conocidos como diagramas de CHAPIN; que permiten representar adecuadamente las estructuras de control de los lenguajes estructurados como PASCAL.-

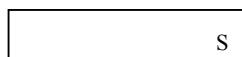
Veamos algunos símbolos:



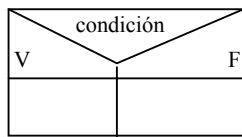
Lectura o entrada de datos



Ordenes u operaciones



Salida de datos y/o resultados

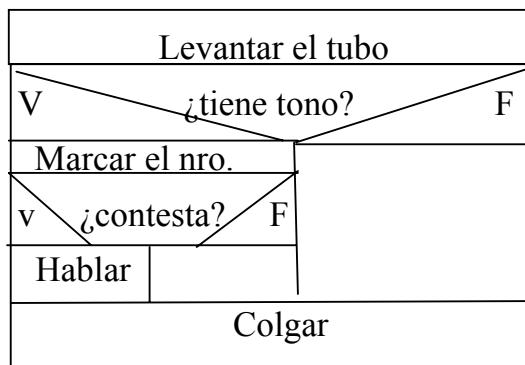


Pregunta o comparación

Decisión simple

EJEMPLO:

Describir mediante un diagrama de Chapin, el procedimiento lógico que debe ser seguido para hablar por teléfono.-



1-3. CONSTANTES Y VARIABLES:

1-3-1. IDENTIFICADOR:

Como en el álgebra, a cada dato o elemento, ya sea constante o variable, se le bautiza con un nombre o identificador; el cual si se ha elegido adecuadamente, ayuda mucho a la persona que lea el programa.-

1-3-2. CONSTANTES:

Como su nombre lo indica, son datos que no varían durante la ejecución de un programa.-

1-3-3. VARIABLES:

Son datos que cambian o evolucionan durante la vida o ejecución de un programa.-

1-4. SENTENCIA DE ASIGNACION:

Asigna el valor de la expresión que está a la derecha del signo := (signo de asignación en lenguaje Pascal), a la variable que está a la izquierda. En el diagrama este signo puede ser reemplazado por ←

1-5. OPERADORES:

+	Suma
-	Resta
*	Multiplicación
/	División real
DIV	División entera
MOD	Resto de la división entera

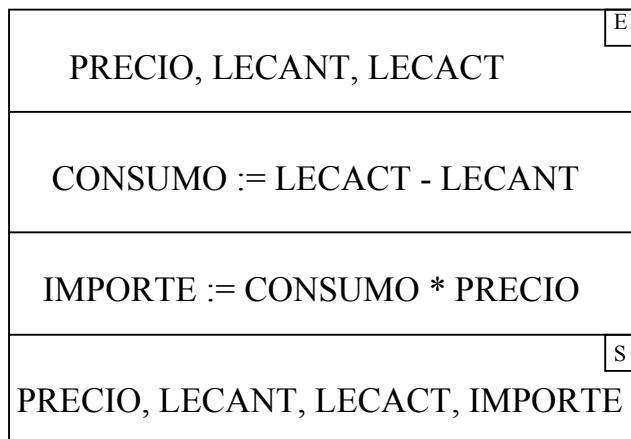
1-6. OPERADORES RELACIONALES:

=	Igual
<>	Distinto
<	Menor
>	Mayor
<= ó =<	Menor o igual
>= ó =>	Mayor o igual
OR	ó lógico
AND	y lógico
NOT	no

1-7. EJEMPLO:

Realizar un algoritmo en diagrama de Chapin para resolver el siguiente problema:

Dados como datos: el precio del kilowatt-hora, la lectura actual y la lectura anterior de un medidor; calcular el importe que una persona deberá abonar a la E.P.E. Mostrar los datos y el resultado obtenido.-



UNIDAD N° 2

INTRODUCCION AL PASCAL

2-1. PROGRAMA EN PASCAL:

Un programa en lenguaje PASCAL está dividido en dos partes:

a)- *ENCABEZAMIENTO:* Este se compone de la palabra reservada PROGRAM seguida de un identificador (nombre del programa definido por el programador), y una lista de parámetros encerrados entre paréntesis que son los nombres de los ficheros a través de los cuales el programa se comunica con y desde el medio exterior. Nosotros utilizaremos INPUT (o sea entrada de datos por teclado), y OUTPUT (o sea salida de datos y/o resultados por pantalla).-

b)- *BLOQUE:* Este consta de dos partes:

i) DECLARACIONES Y DEFINICIONES:

- *Declaración de etiquetas.*
- *Definición de constantes.*
- *Definición de tipos.*
- *Declaración de variables.*
- *Declaración de procedimientos y funciones.*

Todas o algunas de ellas pueden no estar. No nos ocuparemos en este curso de la declaración de etiquetas. En este capítulo veremos definición de constantes y declaración de variables; y más adelante veremos las restantes.-

ii) CUERPO: Está compuesto por las sentencias ejecutables del programa encerradas entre BEGIN y END. -

Los comentarios del programa deben ir entre (* *) o entre { }

2-2. TIPOS DE DATOS STANDARD:

2-2-1. INTEGER:

Son todos los datos que contienen números positivos o negativos sin parte decimal (o sea números enteros). Ellos constan de un signo y dígitos sin comas:

Ejemplos: -109 0 12 +35 -1

Cuando se omite el signo se asume que el número es positivo.-

Teóricamente, no hay límite sobre el tamaño de los enteros, pero las limitaciones del hardware de la computadora y las consideraciones prácticas dan lugar a que haya que limitar el tamaño de un entero. Puesto que este límite varía de unas máquinas a otras, Pascal tiene un identificador predefinido, MAXINT, cuyo valor es el mayor valor entero que puede representarse en la computadora; en general MAXINT es 32767 (aunque puede ser diferente para su máquina, por lo que puede imprimirlo para ver cuál es su valor), entonces el rango de los enteros permitido sería:

desde: -MAXINT hasta: MAXINT

ó

desde: -32767 hasta: 32767

2-2-2. REAL:

Son todos los datos que contienen números positivos o negativos que tienen una parte entera y una parte decimal.-

Cuando en un programa, a un dato se le asigne o se le introduzca un valor real, siempre debe usarse un punto decimal con al menos un dígito a cada lado; y si usamos la notación científica, o sea potencias de 10, debe colocarse una "E" antes del exponente.-

Ejemplos:

Reales válidos	Reales no válidos
12.34	12. (ningún dígito después del ".")
63E4	12.E3 (ningún dígito después del ".")
34.2E5	.46E-6 (ningún dígito antes del ".")
100E-9	245 (ni "E" ni ".")
-50E-4	56.5E (ningún dígito después de "E")

2-2-3. CHAR:

Son todos los datos que contienen un carácter alfanumérico (sólo uno). Los caracteres alfanuméricos incluyen dígitos, letras y símbolos especiales.-

Ejemplos: 'A'; 'a'; '4'; '+'; '?' ; '\$'; ''

El compilador Pascal necesita las comillas simples o apóstrofos para diferenciar entre el dato carácter '4' ó '+' del entero 4 o el signo suma. Observe que el blanco (' ') es un carácter.-

No se puede sumar '4' y '9', pero puede comparar valores de tipo CHAR. El conjunto de caracteres de una máquina está ordenado, 'A' es siempre menor que 'B', 'B' es menor que 'C', y así sucesivamente. También '1' es menor que '2', '3' es menor que '4',etc.-

2-2-4. BOOLEAN:

Son los datos que contienen alguno de los dos valores:

TRUE o FALSE (verdadero o falso).-

Los datos Booleanos no pueden leerse como datos, pero pueden imprimirse.-

Cuando se aplican operadores lógicos (and(\wedge); or(\vee); y not(\neg)) a operandos Booleanos, producen un valor Boolean, que representaremos en el siguiente cuadro, siendo p y q dos operandos Boolean:

p	q	$p \vee q$	$p \wedge q$	$\neg p$
.T.	.T.	.T.	.T.	.F.
.T.	.F.	.T.	.F.	.F.
.F.	.T.	.T.	.F.	.T.
.F.	.F.	.F.	.F.	.T.

2-2-5. STRING:

Turbo Pascal proporciona el tipo string para el procesamiento de cadenas (secuencias de caracteres).

La definición de un tipo string debe especificar el número máximo de caracteres que puede contener, esto es, la máxima longitud para las cadenas de ese tipo. La longitud se especifica por una constante entera en el rango de 1 a 255.

El formato para definir un tipo string es : <identificador> = string [*limite_superior*];

Ejemplo:

Var

```
nombre : string[30];
domicilio : string[30];
ciudad : string[40];
```

Una Vez declaradas las variables se pueden realizar asignaciones:

```
nombre := 'Juan José Perez';
domicilio := 'Buenos Aires 2416 – 1ºpiso';
ciudad := 'Rosario';
```

U operaciones de lectura/escritura:

```
ReadLn (nombre);
WriteLn('Hola ',nombre);
```

Es posible acceder a posiciones individuales dentro de una variable cadena, mediante la utilización de corchetes que dentro de ellos se especifica el número indice dentro de la cadena a utilizar así para el ejemplo anterior se tiene :

```
nombre[1] ==> 'J'  
nombre[2] ==> 'u'  
nombre[3] ==> 'a'  
nombre[4] ==> 'n'
```

Operaciones entre cadenas

Las operaciones básicas entre cadenas son : *asignación, comparación y concatenación*. Es posible *asignar* una cadena a otra cadena, incluso aunque sea de longitud física más pequeña en cuyo caso ocurriría un truncamiento de la cadena.

Si la cantidad de caracteres asignados a una variable string es menor que la definición realizada, Pascal completa con blancos a la derecha de la cadena; si se le asigna una cantidad mayor, trunca

Ejemplo:

Var

```
Nombre1 : String[9];  
Nombre2 : String[20];
```

.

.

.

```
Nombre1 := 'Instituto Tecnológico';  
Nombre2 := 'Universidad';
```

El resultado de la asignación en la variable Nombre1 será la cadena 'Instituto' y en la variable Nombre2 será 'Universidad'

2-3. SENTENCIAS DE ENTRADA Y SALIDA:

En un equipo de cálculo, la entrada y salida representa la interfase final entre el usuario de un programa y el programa que ejecuta la computadora.-

Las sentencias de entrada de un programa hacen que la computadora lea los datos durante la ejecución del programa, lo que permite al programador escribir programas generales que pueden ser utilizados repetidamente para conjuntos de datos diferentes, cuyos valores no tienen porqué ser conocidos exactamente por el programador en el momento de escribir el programa.-

Sin las sentencias de salida, muchos programas serían inútiles ya que el usuario no tendría ninguna forma de conocer el resultado obtenido.-

2-3-1. SENTENCIA READ:

Es la sentencia de entrada en Pascal y permite la lectura de uno o varios datos y su asignación a una o varias variables del programa.-

Ejemplo:

```
READ (NUM);
```

Significa que le asigna a la variable NUM el valor que ingresamos como dato.-

Si queremos ingresar más de un dato, cada variable va separada de otra por coma, y los datos respectivos deben ir separados por uno o más blancos.-

Ejemplo:

```
READ (PRECIO, CANTIDAD);
```

2-3-2. SENTENCIA WRITE:

Es la sentencia de salida en Pascal y permite que el contenido de una variable o el valor de una expresión sea conocida por nosotros mediante una impresión.-

Ejemplo:

```
WRITE (IMPORTE);
```

Si la sentencia de lectura anterior representa el ingreso del precio de un determinado artículo y la cantidad vendida del mismo y deseáramos conocer cuál es el importe que debe pagar el cliente, el segmento de programa sería:

```
READ (PRECIO, CANTIDAD);
IMPORTE := PRECIO * CANTIDAD;
WRITE (IMPORTE);
```

Si quisiéramos imprimir también los datos leídos, la sentencia write quedaría:

```
WRITE (PRECIO, CANTIDAD, IMPORTE);
```

Pero dijimos que la sentencia write también permite conocer el valor de una expresión; así este mismo ejemplo podría haber sido resuelto de la siguiente manera:

```
READ (PRECIO, CANTIDAD);  
WRITE (PRECIO * CANTIDAD);
```

Si deseáramos que los valores numéricos impresos estuvieran acompañados por un texto, es necesario que dicho texto aparezca en la sentencia write encerrado entre apóstrofos.-

Ejemplo:

```
WRITE ('EL IMPORTE DE LA FACTURA ES: ', IMPORTE);
```

Si quisiéramos que los datos se exhiban en un renglón y el resultado en otro, necesitaríamos utilizar la sentencia WRITELN. Esta sentencia provoca lo siguiente: una vez que la computadora exhibió lo indicado en la lista de parámetros encerrados entre paréntesis el cursor queda al principio del renglón siguiente.-

Ejemplo:

```
WRITELN (PRECIO, CANTIDAD);  
WRITE (IMPORTE);
```

en este caso exhibiría los contenidos de las variables precio y cantidad en un renglón, y el de la variable importe en otro.-

Si utilizamos una sentencia WRITELN sin parámetros, obtenemos una línea en blanco.-

Algo similar ocurre con la sentencia READLN. Después de una sentencia READLN, la parte no leída de la línea actual de entrada de datos es saltada y la siguiente entrada será tomada de la siguiente línea.-

2-3-3. SALIDAS FORMATEADAS:

Se puede controlar la apariencia de una salida indicando cuántas columnas se quiere

que ocupe una variable, una constante o un mensaje.-

Si el contenido de una variable o constante es un valor entero o carácter, se pone un ':' seguido de un valor entero después de la variable o constante en la lista de parámetros de la sentencia WRITE o WRITELN. El entero que sigue a ':' indica cuántas columnas o posiciones ocupará sobre la línea, la variable o constante que se va a imprimir a partir del lugar donde está el cursor en ese momento.-

El valor de la variable o constante se imprimirá justificada a la derecha con blancos a la izquierda para ocupar el número correcto de columnas indicado después del ':'.-

EJEMPLO:

Si NUM = 25 (entero); CONT = 54 (entero) y LET = 'B'(carácter)

Sentencia	Salida ('-' representa blanco)
WRITE (NUM:4, CONT:5, LET:3)	--25---54--B
WRITE (NUM:3, CONT:2, LET:1)	-2554B
WRITE (NUM:6, LET:1)	----25B
WRITE (CONT:6, LET:4)	----54---B

El mismo criterio se aplica a los literales. Por ejemplo:

Sentencia	Salida
WRITE ('EL NUMERO ES ':16)	---EL-NUMERO-ES-
WRITE ('EL NUMERO ES ':15, NUM:3)	--EL-NUMERO-ES--25

Si queremos exhibir el valor de una variable o constante con contenido real y no le damos un formato, el valor se imprimirá en la notación E con un dígito antes del punto decimal, lo cual no es muy claro. Por lo tanto, si ese valor lo queremos imprimir en notación decimal, Pascal nos proporciona un formato para ello:

Si se coloca un ':' seguido de un valor entero y luego otro ':' seguido de otro valor entero; el primer número sigue especificando la cantidad total de columnas que se usarán (incluido el punto decimal), y el segundo número especifica la cantidad de

dígitos que se imprimirán después del punto decimal (si el número tiene más decimales que lo especificado en el formato, la computadora redondea el siguiente a la cantidad determinada).-

EJEMPLO: Si PROM = 23.346

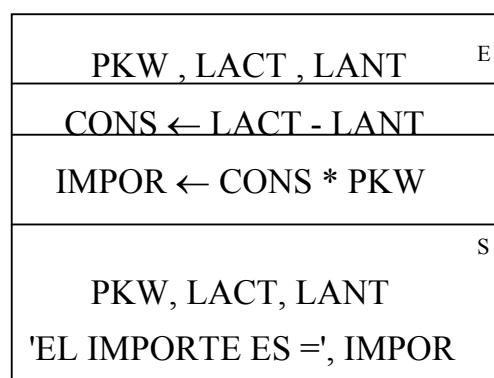
Sentencia	Salida
WRITE (PROM:9:3);	---23.346
WRITE (PROM:10:2);	----23.35
WRITE (PROM:10:4);	---23.3460
WRITE (PROM:6:0);	---23.

2-4. RESOLUCION DE UN PROBLEMA:

Dados como datos el precio del kilowatt-hora y las lecturas actual y anterior de un medidor, calcular el importe que una persona deberá abonar a la E.P.E.

Exhibir los datos en un renglón y el resultado en otro.-

2-4-1. DIAGRAMA DE CHAPIN:



2-4-2. PROGRAMA EN PASCAL:

```
PROGRAM CONSLUZ (INPUT, OUTPUT);
{Calcula el consumo de energía y el importe a abonar}
VAR
  PKW, LACT, LANT, CONS, IMPOR : REAL;
BEGIN
  WRITE ('INGRESE PRECIO Y LECTURAS ACTUAL Y ANTERIOR');
  READLN (PKW, LACT, LANT);
  CONS := LACT - LANT;
  IMPOR := CONS * PKW;
  WRITELN (PKW:6:2, LACT:12:0, LANT:12:0);
  WRITELN;
  WRITE ('EL IMPORTE ES =':23, IMPOR:7:2)
END.
```

UNIDAD N° 3

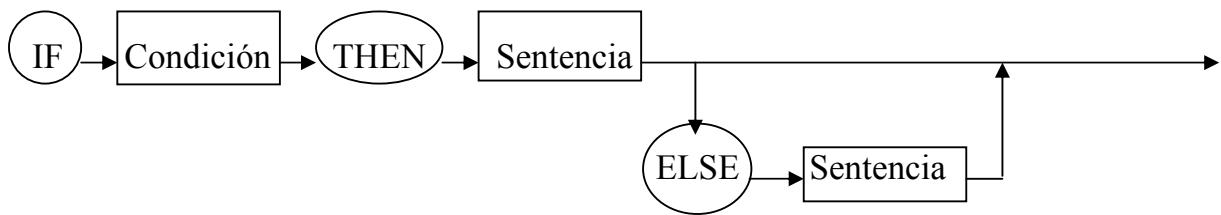
ESTRUCTURAS DE CONTROL

Hay tres estructuras básicas de control:

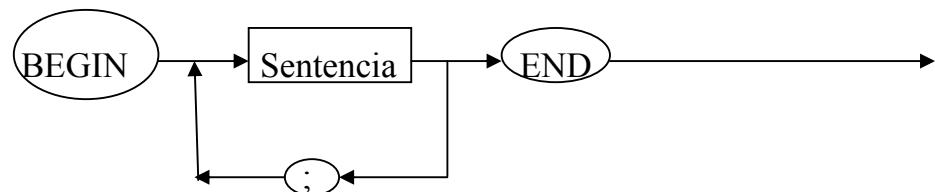
- LA SECUENCIAL (vista en las unidades anteriores)
- LA BIFURCACION O SELECCION
- LA REPETICION O ITERACION

3-1. LA BIFURCACION O SELECCION:

Hemos visto la selección entre dos caminos dependiendo de una condición. Esta sentencia es la sentencia IF , y su sintaxis es la siguiente:



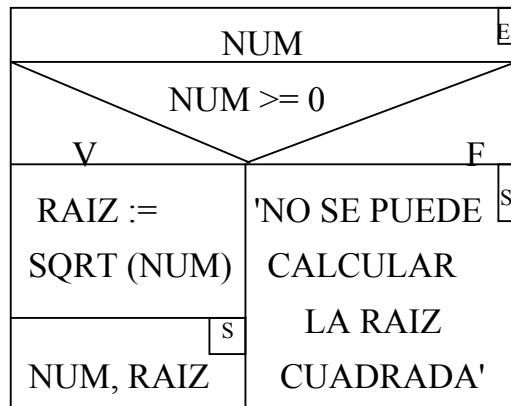
Donde dice Sentencia, puede tratarse de un grupo de sentencias, a este grupo se lo denomina: SENTENCIA COMPUESTA y va encerrado entre BEGIN y END ; la sintaxis de una sentencia compuesta es la siguiente:



3-1-1- EJEMPLOS:

1)- Dado un número real hallar, si es posible, su raíz cuadrada

a) DIAGRAMA DE CHAPIN:

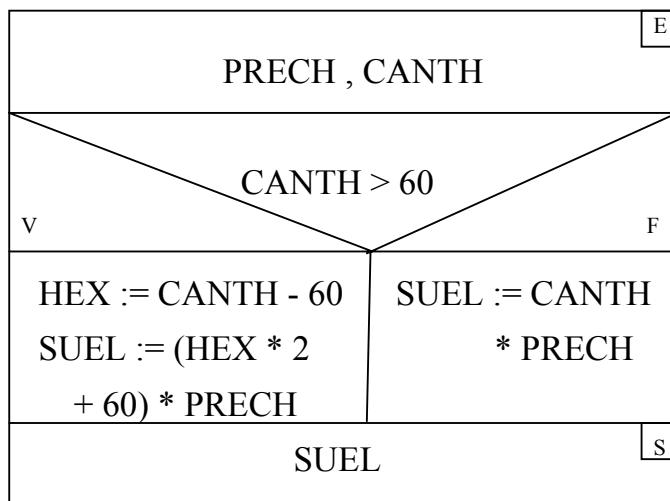


b) PROGRAMA EN PASCAL:

```
PROGRAM RAIZCUAD (Input, Output);
VAR
    NUM, RAIZ : REAL;
BEGIN
    WRITE ('INGRESE NUMERO');
    READLN (NUM);
    IF NUM >= 0
        THEN BEGIN
            RAIZ := SQRT (NUM);
            WRITE (NUM:10:2, RAIZ:10:4)
        END
        ELSE WRITE ('NO SE PUEDE CALCULAR LA RAIZ CUADRADA')
    END.
```

2)- A partir de los datos de: Pago por hora y Cantidad de horas trabajadas calcular el Sueldo de un operario, sabiendo que si las horas trabajadas superan 60, las excedentes se pagan el doble.-

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA EN PASCAL:

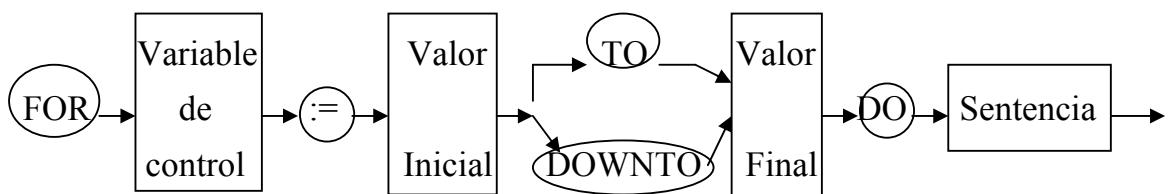
```
PROGRAM SUELDO (INPUT, OUTPUT);
VAR
    PRECH, SUEL : REAL;
    CANTH, HEX : INTEGER;
BEGIN
    WRITE ('INGRESE PRECIO-HORA Y CANT-HORAS TRABAJADAS');
    READLN ( PRECH, CANTH);
    IF CANTH > 60
    THEN BEGIN
        HEX := CANTH - 60;
        SUEL := (HEX * 2 + 60) * PRECH
    END
    ELSE SUEL := CANTH * PRECH;
```

WRITE (' EL SUELDO DEL OPERARIO ES DE \$', SUEL:7:2)
END.

3-2. REPETICION O ITERACION:

3-2-1. CON CANTIDAD CONOCIDA DE VECES

Cuando una sentencia o un grupo de sentencias deben ejecutarse más de una vez utilizamos una estructura de repetición. Si sabemos qué cantidad de veces se van repetir, utilizamos la sentencia PARA; esta sentencia en PASCAL es la sentencia FOR, y su sintaxis es la siguiente:



La variable de control, el valor inicial y el valor final deben ser del mismo tipo. La variable de control debe declararse como cualquier otra variable.-

Donde dice Sentencia puede tratarse de una Sentencia Compuesta (cuya sintaxis ya la hemos visto).

TO y DOWNTO son palabras especiales. Se usa TO cuando la variable de control toma el valor inicial y llega al valor final incrementándose (sentencia FOR ascendente); de lo contrario se usa DOWNTO (sentencia FOR descendente).-

La ejecución de la sentencia FOR tiene lugar de la siguiente forma: primero se evalúan las expresiones (si así fuera) que dan el valor inicial y valor el final y se almacenan en memoria , y después se asigna a la variable de control el valor inicial.-

Se realiza entonces una comparación entre el valor de la variable de control y el final. Si el valor de la variable de control es mayor que el valor final (suponiendo que estamos usando TO y no DOWNTO), entonces ya no se ejecutan las sentencias del ciclo; en cuyo caso el control pasa a la sentencia siguiente de la última que constituye el rango del ciclo.-

Si por el contrario el valor de la variable de control es menor ó igual que el valor final entonces se ejecutan una vez más las sentencias que constituyen el ciclo, se incrementa la variable de control y se vuelve a comparar.-

El proceso continúa hasta que la variable de control toma un valor mayor que el valor final, en cuyo caso termina el ciclo.-

Es importante destacar que:

*** *TODA SENTENCIA DENTRO DEL CICLO QUE INTENTE CAMBIAR EL VALOR DE LA VARIABLE DE CONTROL, DARÁ ERROR.-***

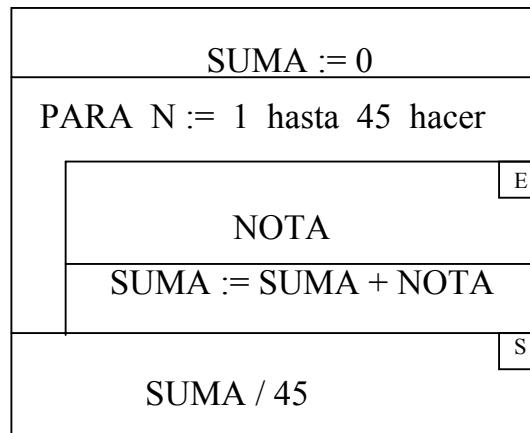
*** *AL SALIR DEL CICLO LA VARIABLE DE CONTROL QUEDA CON VALOR INDEFINIDO.***

(No así dentro del ciclo, donde se puede utilizar su contenido).-

EJEMPLOS:

1)-Dadas las notas de un parcial de los 45 alumnos de un curso, se desea obtener la nota promedio del curso.-

a) **DIAGRAMA DE CHAPIN:**

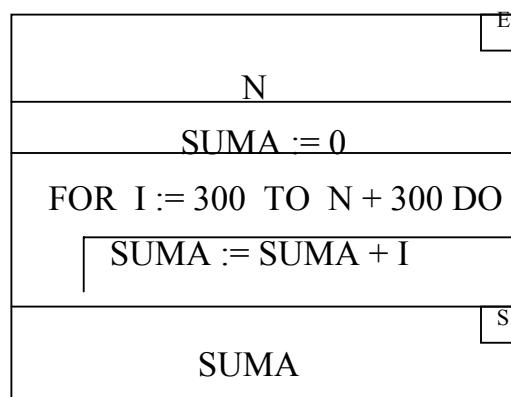


b) PROGRAMA EN PASCAL:

```
PROGRAM PROMEDIO (INPUT, OUTPUT);
VAR
    N : INTEGER;
    SUMA , NOTA : REAL;
BEGIN
    SUMA := 0;
    FOR N := 1 TO 45 DO
        BEGIN
            WRITE (' INGRESE NOTA ');
            READLN ( NOTA );
            SUMA := SUMA + NOTA
        END;
    WRITE (' EL PROMEDIO DEL CURSO ES = ', SUMA/45 :4:2)
END.
```

2)- Se desea obtener la suma de los N números naturales posteriores al número 300 inclusive.-

a) DIAGRAMA DE CHAPIN:



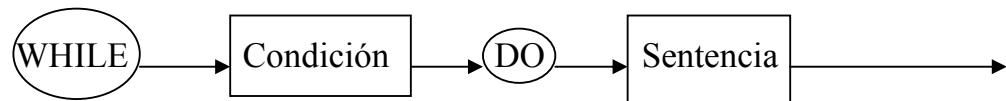
b) PROGRAMA EN PASCAL:

```
PROGRAM SUMANAT ( INPUT, OUTPUT);
VAR
  N , I : INTEGER;
  SUMA : REAL;
BEGIN
  WRITE ('INGRESE CANTIDAD DE NROS.');
  READLN (N);
  SUMA := 0;
  FOR I := 300 TO N + 300 DO
    SUMA := SUMA + I;
  WRITE ('LA SUMA DE LOS',N:4,'NROS NATURALES >= 300 ES',
        SUMA:10:0)
END.
```

3-2-2. CON CANTIDAD DESCONOCIDA DE VECES

Cuando una sentencia o un grupo de sentencias deben repetirse más de una vez, dependiendo de una condición, utilizamos la estructura MIENTRAS o REPETIR.-

3-2-2-1. La sentencia Mientras es la sentencia WHILE en PASCAL, y su sintaxis es la siguiente:



Donde dice Sentencia puede tratarse de una SENTENCIA COMPUESTA, como la que ya vimos.-

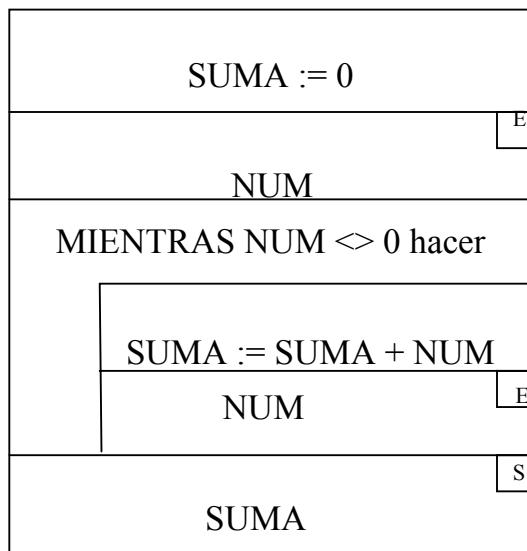
Mientras la Condición sea verdadera se ejecutan la ó las sentencias del ciclo. Cuando la condición sea falsa, el control pasa a la sentencia siguiente de la última que compone el ciclo.-

Por lo tanto si la condición es falsa la primera vez, pueden no ejecutarse nunca las sentencias del ciclo de repetición.-

EJEMPLOS:

1)-Se van ingresando números distintos de cero, salvo el último valor. Determinar su suma.-

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA EN PASCAL:

```
PROGRAM SUMANDO (INPUT, OUTPUT);
VAR
    NUM , SUMA : REAL;
BEGIN
    SUMA := 0;
    WRITE ('INGRESE NUMERO');
    READLN (NUM);
    WHILE NUM <> 0 DO
        BEGIN
            SUMA := SUMA + NUM;
```

```
WRITE ('INGRESE NUMERO');
READLN (NUM)
END;
WRITE ('LA SUMA ES = ', SUMA:9:2)
END.
```

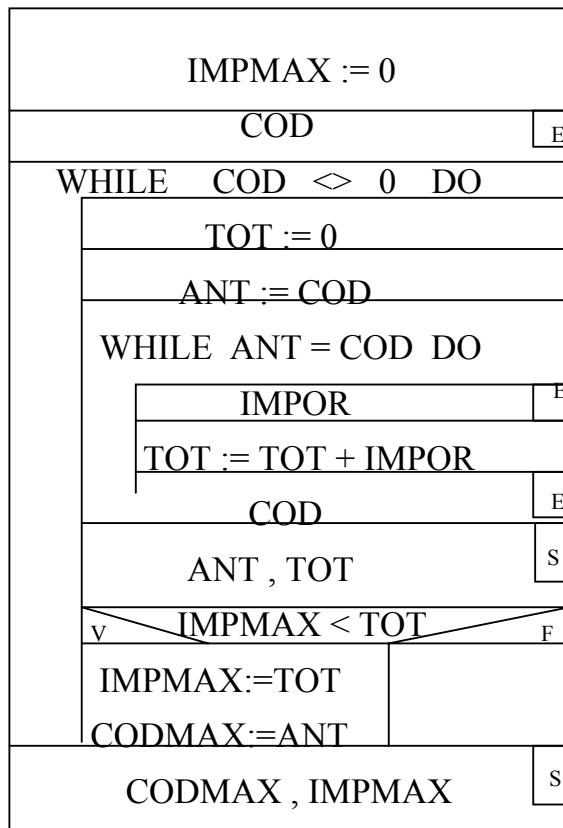
2)- Se desea saber el total de ventas de cada uno de los vendedores de una empresa. A tal fin se tienen como datos: el código de vendedor y el importe de cada una de las ventas; un vendedor puede haber realizado más de una venta. No se sabe la cantidad de vendedores que tiene la empresa ni la cantidad de ventas hechas por cada vendedor (un código de vendedor igual a cero es fin de datos).-

ESTOS DATOS ESTAN ORDENADOS POR CODIGO DE VENDEDOR

Exhibir cada código de vendedor y su total correspondiente y al final, el código de vendedor con mayor importe vendido y dicho importe.-

Resolverlo usando CORTE DE CONTROL.-

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA EN PASCAL:

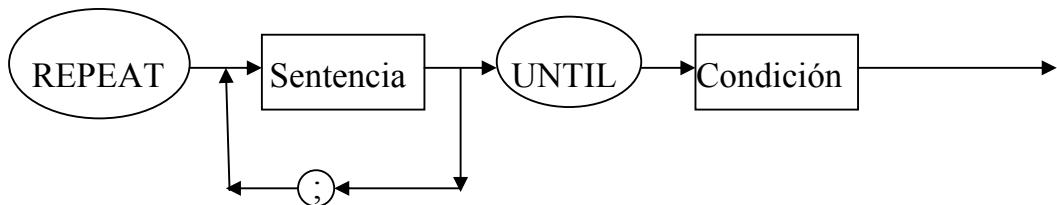
```
PROGRAM VENDEDOR (INPUT, OUTPUT);
{EJEMPLO DE CORTE DE CONTROL}

VAR
  COD , ANT , CODMAX : INTEGER;
  IMPOR , TOT , IMPMAX : REAL;

BEGIN
  IMPMAX := 0;
  WRITE ('INGRESE CODIGO');
  READLN (COD);
  WHILE COD <> 0 DO
    BEGIN
      TOT := 0;
      ANT := COD;
      WHILE ANT = COD DO
        BEGIN
          WRITE ('INGRESE IMPORTE');
          READLN (IMPOR);
          TOT := TOT + IMPOR;
          WRITE ('INGRESE CODIGO');
          READLN (COD)
        END;
      WRITE ('EL VENDEDOR',ANT:4,' VENDIO $ ',TOT:15:2);
      IF TOT > IMPMAX
        THEN
          BEGIN
            IMPMAX := TOT;
            CODMAX := ANT
          END;
      END;
      WRITE ('EL VENDEDOR :,CODMAX:4,'TUVO MAYOR IMPORTE: $,',
```

IMPMAX:15:2)
END.

3-2-2-2. La sentencia REPETIR es la sentencia REPEAT en PASCAL, y su sintaxis es la siguiente:

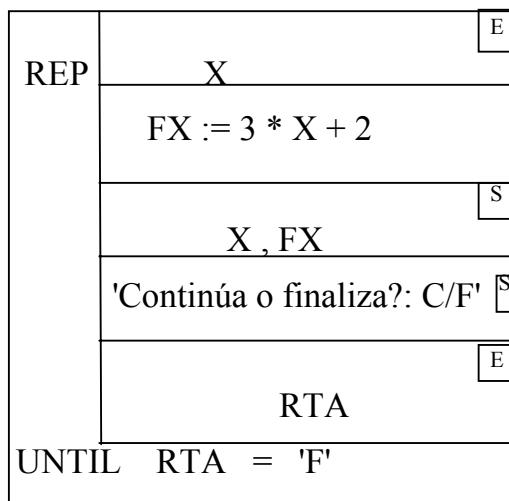


Repite la ejecución de la ó las sentencias del ciclo hasta que la condición sea verdadera. Por lo tanto el ciclo se ejecuta al menos una vez, pues compara al final del mismo. Esta es la gran diferencia que tiene con la sentencia WHILE.-

EJEMPLOS:

1)- Evaluar y tabular la función $f(X) = 3X + 2$ para diferentes valores de X .-

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA EN PASCAL:

PROGRAM FUNCION (INPUT, OUTPUT);

VAR

X , FX : INTEGER;

RTA : CHAR;

BEGIN

REPEAT

 WRITE ('INGRESE VALOR ');

 READLN (X);

 FX := 3 * X + 2;

 WRITE ('CONTINUA O FINALIZA INGRESANDO? C/F ');

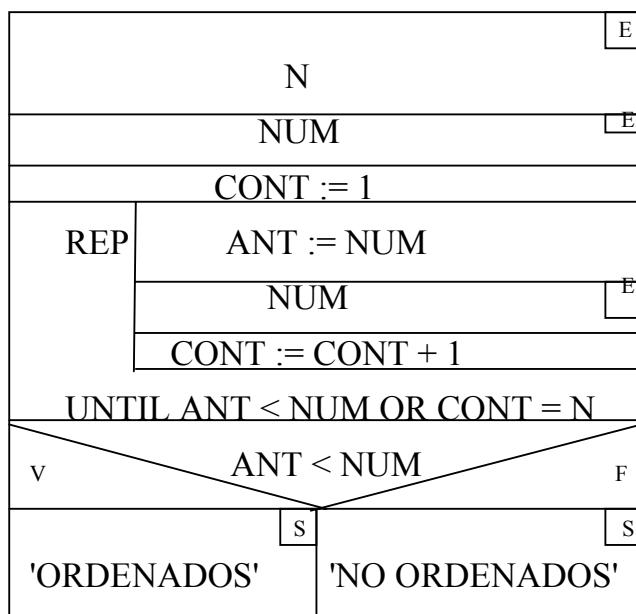
 READLN (RTA)

 UNTIL RTA = 'F'

END.

2)- Determinar si una lista de N números está ordenada de mayor a menor.-

a) DIAGRAMA DE CHAPIN:



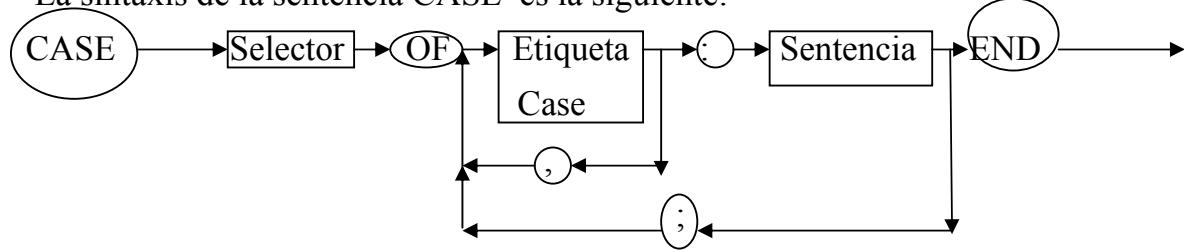
b) PROGRAMA EN PASCAL:

```
PROGRAM ORDEN (INPUT, OUTPUT);
VAR
  N , CONT , ANT , NUM : INTEGER;
BEGIN
  WRITE ('INGRESE LA CANTIDAD DE NROS.');
  READLN (N);
  WRITE ('INGRESE UN NRO.');
  READLN (NUM);
  CONT := 1;
  REPEAT
    ANT := NUM;
    WRITE ('INGRESE NRO.');
    READLN (NUM);
    CONT := CONT + 1
  UNTIL ANT < NUM OR CONT = N;
  IF ANT < NUM
    THEN WRITE ('NO ESTAN ORDENADOS')
    ELSE WRITE ('ESTAN ORDENADOS')
END.
```

3-3. SELECCION MULTIPLE:

Hemos visto también que si para optar por un camino a seguir no se depende de una condición, sino del valor que contenga un dato o expresión (el selector) podíamos utilizar la sentencia CASOS; esta sentencia es la sentencia CASE en PASCAL.-

La sintaxis de la sentencia CASE es la siguiente:



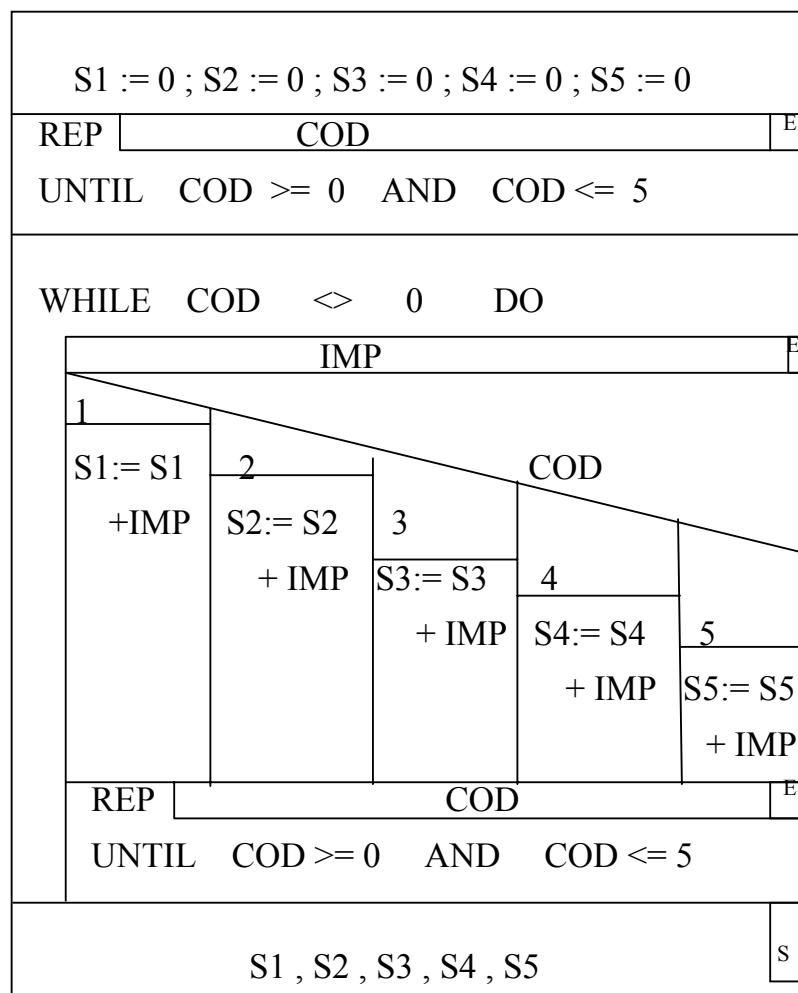
La sentencia CASE selecciona para su ejecución aquella sentencia cuya etiqueta sea igual al valor actual del selector. Si no existe la etiqueta el efecto es indefinido (NO TIENE SALIDA POR ERROR).-

Donde dice Sentencia puede tratarse de una SENTENCIA COMPUESTA.

EJEMPLO:

Se tienen como datos los importes de las ventas de cada una de las sucursales de una empresa, junto con el código de sucursal (1, 2, 3, 4 ó 5).- Cada sucursal puede tener varias ventas. Los datos no están ordenados por código de sucursal. Un código igual a cero indica fin de datos.- Obtener el total de ventas para cada sucursal.-

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA EN PASCAL:

```
PROGRAM VENTAS (INPUT, OUTPUT);
VAR
  COD : INTEGER;
  S1 , S2 , S3 , S4 , S5 , IMP : REAL;
BEGIN
  S1:= 0; S2:= 0; S3:= 0; S4:= 0; S5:= 0;
  REPEAT
    WRITE ('INGRESE CODIGO'); READLN (COD)
    UNTIL ( COD >= 0 ) AND ( COD <= 5 );
  WHILE COD <> 0 DO
    BEGIN
      WRITE ('INGRESE IMPORTE');
      READLN (IMP);
      CASE COD OF
        1 : S1 := S1 + IMP;
        2 : S2 := S2 + IMP;
        3 : S3 := S3 + IMP;
        4 : S4 := S4 + IMP;
        5 : S5 := S5 + IMP
      END;
      REPEAT
        WRITE ('INGRESE CODIGO'); READLN (COD)
        UNTIL ( COD >= 0 ) AND ( COD <= 5 )
      END;
      WRITELN ('TOTAL SUCURSAL 1 ':30, S1:12:2);
      WRITELN ('TOTAL SUCURSAL 2 ':30, S2:12:2);
      WRITELN ('TOTAL SUCURSAL 3 ':30, S3:12:2);
      WRITELN ('TOTAL SUCURSAL 4 ':30, S4:12:2);
      WRITE ('TOTAL SUCURSAL 5 ':30, S5:12:2)
    END.
```

UNIDAD N° 4

TIPOS DE DATOS

4-1. TIPOS ENUMERADOS O ESCALARES:

PASCAL nos proporciona la manera de especificar nuestros propios tipos de datos.-

Los enumerados o escalares son aquellos en los que mencionamos cada uno de los valores que puede contener, mediante la definición:

TYPE T = (C1, C2,, Cn);

EJEMPLO:

Si declaramos:

TYPE

COLOR = (BLANCO, VERDE, ROJO);

DIAS = (LUNES, MARTES MIERCOLES, JUEVES, VIERNES);

VAR

D: DIAS;

BAN: COLOR;

podemos escribir las siguientes sentencias:

D := MARTES;

BAN := BLANCO;

Si por error escribíramos esta sentencia de asignación:

D:= SABADO;

nos marcaría el error.-

PASCAL también nos permite declarar específicamente las variables de esta manera:

VAR

BAN1, BAN2: (BLANCO, VERDE, ROJO);

4-2. TIPOS SUBRANGO O INTERVALO:

Son los tipos de datos cuyos valores estén dentro de ciertos límites: LIMITE INFERIOR y LIMITE SUPERIOR.-

La forma general de definición es:

TYPE T = mín .. máx ;

siendo mín y máx, constantes.-

EJEMPLO:

Si declaramos:

TYPE

NUM = 20 .. 100;

VAR

EDAD: NUM;

podemos escribir la sentencia:

EDAD := 50;

Si pusiéramos por descuido o error:

EDAD := 150;

nos daría error.-

PASCAL también nos permite declarar las variables de esta manera:

VAR

EDAD: 20 .. 100;

4-3. VARIABLES ESTRUCTURADAS: (ARREGLOS)

Hasta ahora hemos visto tipos de datos simples. Algunas veces es necesario almacenar y referenciar variables como un grupo. Estas estructuras de datos nos permiten escribir programas para manipular datos más fácilmente.-

Un ARRAY es un grupo de elementos a los que se les da un nombre común.-

Se accede a cada elemento por su posición dentro del grupo.-

Todos los elementos de un ARRAY deben ser del mismo tipo.-

4-3-1. ARRAYS UNIDIMENSIONALES (Vectores):

Un ARRAY unidimensional es un tipo de datos estructurados compuesto de un número fijo de componentes del mismo tipo, con cada componente directamente accesible mediante el índice.-

La forma general de declaración es:

VAR

nombre : ARRAY [índice inferior .. índice superior] OF tipo;

Donde nombre es el nombre de la variable (cualquier identificador válido en PASCAL); índice inferior e índice superior indican el rango del ARRAY; y tipo es el tipo de datos de todos los elementos de la variable.-

Un elemento particular del ARRAY se representa por el nombre de la variable seguido de un índice encerrado entre corchetes, que indica la posición que ocupa ese elemento:

nombre [índice]

EJEMPLOS:

a) VAR

NOM: ARRAY [1 .. 20] OF CHAR;
CONT: ARRAY ['A' .. 'P'] OF INTEGER;

b) TYPE

COD = 20..60;
VAR
IMP: ARRAY [COD] OF REAL;

c) TYPE

COLOR = (ROJO, VERDE, BLANCO, NEGRO, AZUL);
VAR
BANDERA; ARRAY [1..30] OF COLOR;

Eventualmente puede hacer falta durante la ejecución de un programa, realizar asignaciones de todos los elementos de un array a otro; es decir, asignar a cada elemento de un array los elementos correspondientes de otro array ; para poder hacer esto, debemos declarar los dos arrays de "igual" manera, y escribir la sentencia de asignación con los nombres de los arrays solamente sin índice:

Por ejemplo si declaramos:

VAR SUM, NUM1, NUM2: ARRAY [1..20] OF INTEGER;

Podemos escribir la siguiente sentencia de asignación:

NUM1 := NUM2;

PASCAL NO PERMITE:

a) REALIZAR OPERACIONES CON ARRAYS COMPLETOS TAL COMO:

SUM := NUM1 + NUM2;

b) ASIGNARLE UNA CONSTANTE A TODO UN ARRAY:

NUM2 := 0;

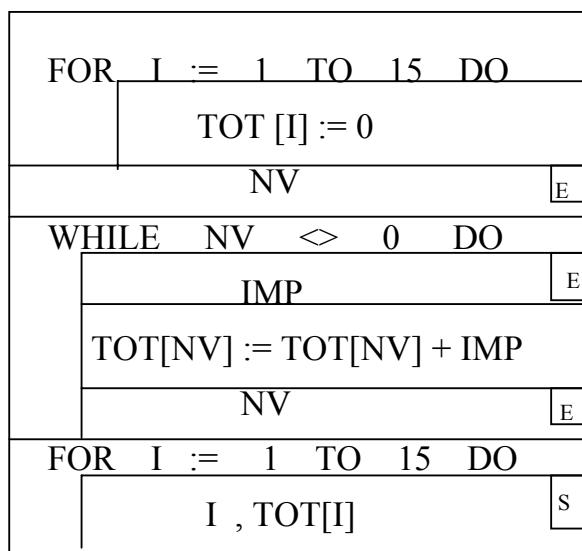
EJEMPLO:

Se tienen como datos: NRO. DE VENDEDOR e IMPORTE de cada una de las ventas realizadas por cada uno de los vendedores de una empresa.-

No se sabe cuántas ventas se han realizado por lo que un NRO. DE VENDEDOR igual a cero indica fin de datos.- Los vendedores están numerados del 1 al 15.

Se desea obtener el total vendido por cada vendedor.

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA PASCAL:

PROGRAM VENTAS (INPUT, OUTPUT);

VAR

TOT : ARRAY [1..15] OF REAL;

NV, I : INTEGER;

IMP : REAL;

BEGIN

```
FOR I := 1 TO 15 DO
    TOT[I] := 0;
    WRITE ('INGRESE NRO VENDEDOR');
    READLN (NV);
    WHILE NV <> 0 DO
        BEGIN
            WRITE ('INGRESE IMPORTE');
            READLN (IMP);
            TOT[NV] := TOT[NV] + IMP;
            WRITE ('INGRESE NRO. VENDEDOR');
            READLN (NV)
        END;
    FOR I := 1 TO 15 DO
        WRITELN ('EL VENDEDOR NRO.':25, I:3, 'VENDIO $ ', TOT[I]:5:2)
    END.
```

4-3-1-1. ORDENAMIENTO DE UN ARREGLO UNIDIMENSIONAL:

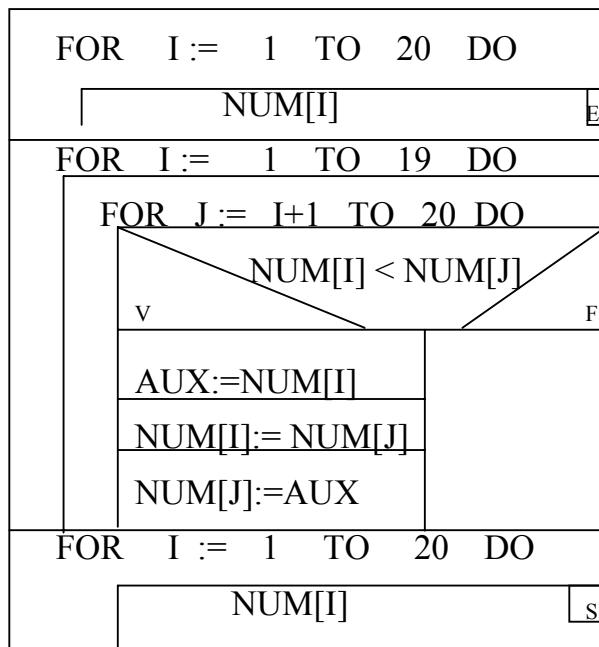
Para ordenar los elementos de un arreglo, ya sea en forma creciente o decreciente, existen varios métodos; nosotros veremos el método conocido por Falso Burbuja, que sin ser el más rápido en algunos casos, es uno de los más fáciles de entender.-

Este método consiste en comparar cada uno de los elementos del arreglo con todos los restantes y cambiarlos o no entre sí de acuerdo a la forma en que estemos ordenando el arreglo.-

EJEMPLO:

Ingresar 20 números enteros en un arreglo y luego mostrarlos ordenados en forma decreciente.-

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA PASCAL:

```
PROGRAM ORDEN (INPUT, OUTPUT);
VAR
  I, J, AUX : INTEGER;
  NUM : ARRAY [1..20] OF INTEGER;
BEGIN
  FOR I := 1 TO 20 DO
    BEGIN
      WRITE ('INGRESE NRO.');
      READLN (NUM[I])
    END;
  FOR I := 1 TO 19 DO
    FOR J := I+1 TO 20 DO
      IF NUM[I] < NUM[J]
      THEN
        BEGIN
```

```
AUX := NUM[I];  
NUM[I] := NUM[J];  
NUM[J] := AUX  
END;  
FOR I := 1 TO 20 DO  
    WRITE (NUM[I],')')  
END.
```

4-3-1-2. BUSQUEDA DE UN VALOR EN UN ARREGLO ORDENADO:

Si tenemos un arreglo ordenado, ya sea en forma creciente o decreciente y tenemos que buscar un valor dentro del mismo, la forma más rápida para hacerlo es por medio de la Búsqueda Dicotómica.-

Para explicar este método, supongamos que los elementos del arreglo están ordenados en forma creciente procediéndose entonces, de la siguiente manera:

Primero se fijan los extremos del intervalo de búsqueda, que serán las posiciones que ocupan el primero y el último elemento del arreglo respectivamente.-

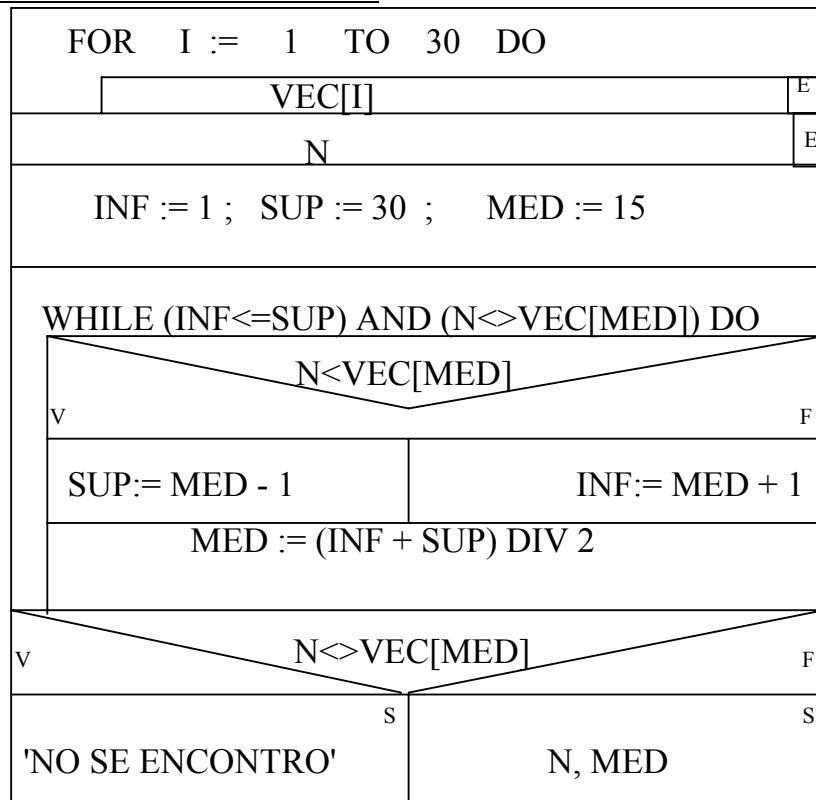
Luego se obtiene la posición media de dicho intervalo y se compara el valor a buscar con el contenido del arreglo en esa posición media; si el valor es mayor que dicho elemento se continúa la búsqueda en la segunda mitad del arreglo; si por el contrario el valor es menor, se continúa en la primera mitad y se halla nuevamente la posición media del nuevo intervalo de búsqueda.-

El proceso se repite hasta que se encuentra el valor a buscar, o bien hasta que no haya más intervalo de búsqueda, lo que significará que el valor buscado no se encuentra en el arreglo.-

EJEMPLO:

Se ingresan 30 números enteros ordenados en forma creciente y un valor N. Se desea saber si el valor N coincide con algún elemento del arreglo; si es así, indicar la posición en que fue encontrado, sino exhibir cartel aclaratorio.-

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA PASCAL:

```

PROGRAM BUSQUEDA (INPUT, OUTPUT);
VAR
  I, INF, SUP, MED, N : INTEGER;
  VEC: ARRAY [1..30] OF INTEGER;
BEGIN
  FOR I := 1 TO 30 DO
    BEGIN
      WRITE ('INGRESE NRO.');
      READLN (VEC[I])
    END;
  WRITE ('INGRESE NRO. A BUSCAR');
  READLN (N);
  INF := 1;
  SUP := 30;

```

```
MED := 15;  
WHILE (INF<=SUP) AND (N<>VEC[MED]) DO  
    BEGIN  
        IF N<VEC[MED]  THEN SUP:= MED - 1  
        ELSE INF:= MED + 1;  
        MED := (INF + SUP) DIV 2  
    END;  
    IF N <> VEC[MED]  
        THEN WRITE ('EL VALOR', N, 'NO FUE ENCONTRADO')  
        ELSE WRITE ('EL VALOR', N, 'SE ENCUENTRA EN LA POSICION',  
MED)  
    END.
```

4-3-1-3. INTERCALACION DE ARRAYS ORDENADOS:

La intercalación de arrays es un proceso que consiste en tomar dos arrays ordenados, ya sea ambos en forma creciente o en forma decreciente, y obtener un array ordenado de igual manera que los datos.-

Para explicar este método, supongamos tener dos arrays ordenados en forma creciente cuyas dimensiones son n y m respectivamente, se obtendrá un nuevo array de dimensión (n + m), también ordenado en forma creciente, procediéndose de la siguiente manera:

Se comparan los primeros elementos de cada array, se selecciona el menor y se coloca en la primera posición del nuevo array.-

Luego se compara el elemento que quedó sin colocar, con el elemento que sigue del array cuyo elemento fue utilizado poniéndose el menor en la siguiente posición del nuevo array.-

Si dos elementos comparados son iguales, se coloca uno y luego el otro en el nuevo array y se pasan a comparar los elementos siguientes de cada uno de los dos arrays dados.-

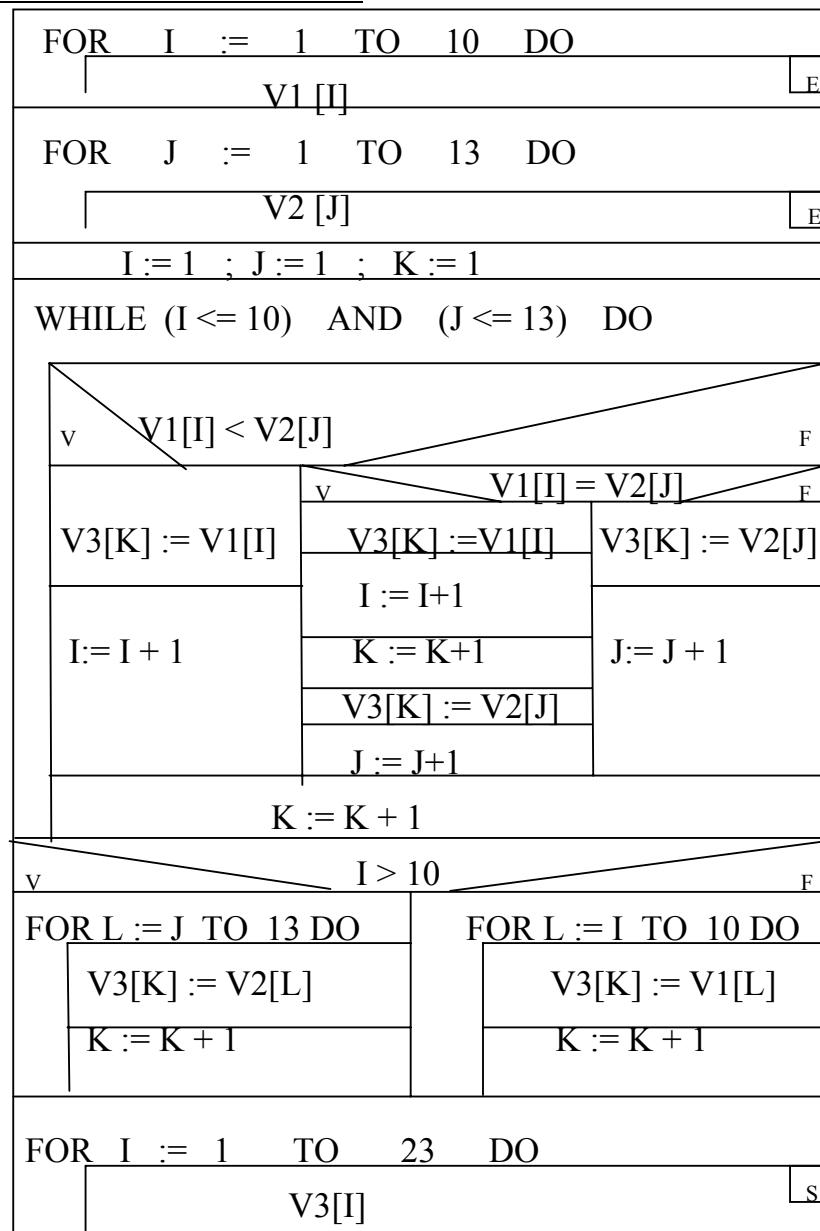
El proceso continúa hasta que todos los elementos del primero o segundo array hayan sido utilizados, en cuyo caso los elementos restantes se agregan como están en el nuevo array.-

EJEMPLO:

Ingresar 10 números enteros ordenados en forma creciente en un array; luego 13 números enteros, también ordenados en forma creciente en otro array.-

Obtener por intercalación, un tercer array ordenado en forma creciente, y luego mostrarlo.-

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA PASCAL:

```
PROGRAM MERGE (INPUT, OUTPUT);
VAR
  I, J, K, L : INTEGER;
  V1 : ARRAY [1..10] OF INTEGER;
  V2 : ARRAY [1..13] OF INTEGER;
  V3 : ARRAY [1..23] OF INTEGER;
BEGIN
  FOR I := 1 TO 10 DO
    BEGIN
      WRITE ('INGRESE NRO.');
      READLN (V1[I])
    END;
  FOR J := 1 TO 13 DO
    BEGIN
      WRITE ('INGRESE NRO.');
      READLN (V2[J])
    END;
  I := 1; J := 1; K := 1;
  WHILE ( I<= 10 ) AND ( J <= 13 ) DO
    BEGIN
      IF V1[I] < V2[J]
        THEN BEGIN
          V3[K] := V1[I];
          I := I + 1
        END
      ELSE IF V1[I] = V2[J]
        THEN BEGIN
          V3[K] := V1[I];
          I := I + 1;
          K := K + 1;
        END
    END
  
```

```
V3[K] := V2[J];
J := J + 1
END
ELSE BEGIN
    V3[K] := V2[J];
    J := J + 1
END;
K := K + 1
END;
IF I > 10
THEN FOR L := J TO 13 DO
BEGIN
    V3[K] := V2[L];
    K := K + 1
END
ELSE FOR L := I TO 10 DO
BEGIN
    V3[K] := V1[L];
    K := K + 1
END;
WRITELN (' ARREGLO ORDENADO ');
FOR I := 1 TO 23 DO
    WRITE (V3[I], ' ')
END.
```

4-3-2. ARRAYS BIDIMENSIONALES (matrices):

Un array bidimensional es también un tipo de datos estructurados compuesto de un número fijo de componentes del mismo tipo, accediéndose a cada una de ellas mediante dos subíndices, el primero indica el número de fila y el segundo el número de columna en donde está dicho elemento.-

La forma general de declaración es :

VAR

nombre: ARRAY [índ inf .. índ sup , índ inf .. índ sup] OF tipo;

Un elemento particular del ARRAY se representa por:

nombre [índice fila , índice columna]

EJEMPLO:

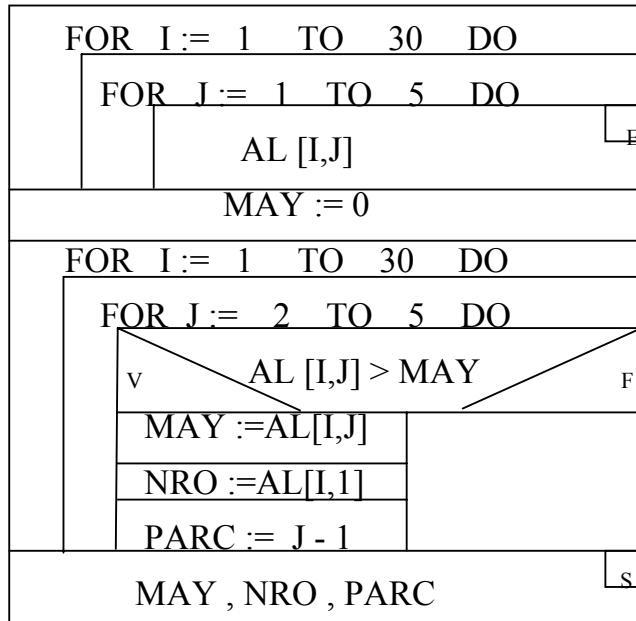
Se tienen los siguientes datos de 30 alumnos de primer año:

NRO. ALUMNO (entero)

NOTA 1, NOTA 2, NOTA 3, NOTA 4 (enteras)

Se desea cargar los datos en un arreglo bidimensional y luego exhibir el nro. de alumno que tuvo la mejor nota de todas y en qué parcial la obtuvo.-

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA PASCAL:

```
PROGRAM PARCIAL (INPUT, OUTPUT);
VAR
  I, J, MAY, NRO, PARC : INTEGER;
  AL : ARRAY [1..30, 1..5] OF INTEGER;
BEGIN
  FOR I := 1 TO 30 DO
    BEGIN
      WRITE ('INGRESE NRO. DEL ALUMNO Y LAS 4 NOTAS');
      FOR J := 1 TO 5 DO
        READ (AL [I,J]);
      WRITELN
    END;
  MAY := 0;
  FOR I := 1 TO 30 DO
    FOR J := 2 TO 5 DO
      IF AL [I,J] > MAY
      THEN BEGIN
        MAY := AL [I,J];
        NRO := AL [I,1];
        PARC := J - 1
      END;
    WRITE ('LA MAYOR NOTA: ',MAY,' CORRESPONDE AL ALUMNO: ',
          NRO, 'EN EL PARCIAL NUMERO: ',PARC)
  END.
```

4-3-2-1. ORDENAMIENTO DE UN ARRAY BIDIMENSIONAL:

Tenemos que tener en cuenta que los datos que se cargan en un arreglo bidimensional responden, por fila o por columna, a una persona, a una agencia, a una comisión, etc.; por lo tanto, lo mismo que en una matriz matemática, no podemos

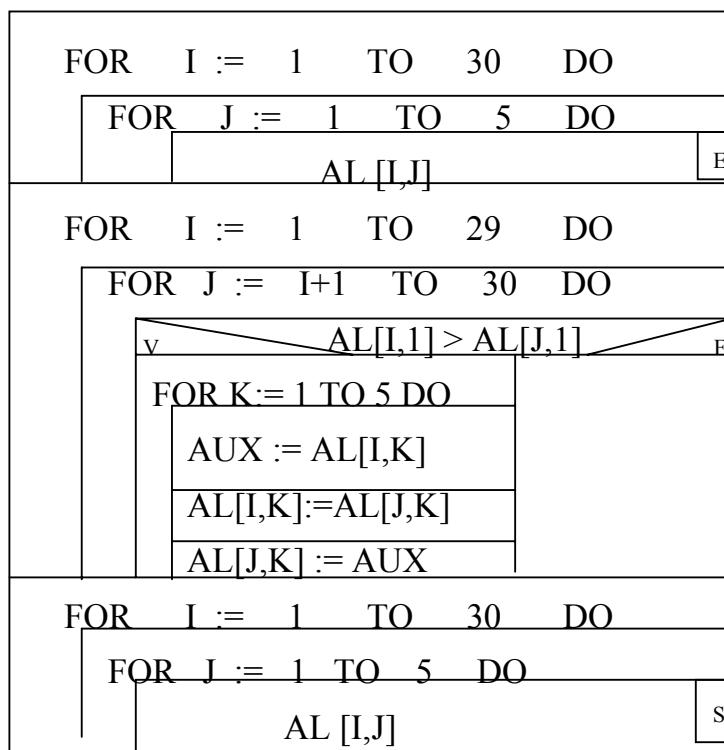
cambiar un elemento por otro de lugar; lo que sí podemos hacer es cambiar una fila o una columna por otra. Luego, un ordenamiento en un arreglo bidimensional tiene sentido, si se pide que se ordene el mismo de manera tal que los elementos de una fila o columna queden ordenados en forma creciente o decreciente.-

EJEMPLO:

Se tienen los mismos datos del ejercicio anterior.

Se desea un listado de los mismos ordenados en forma creciente por NRO. DE ALUMNO.-

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA PASCAL:

PROGRAM LISTADO (INPUT, OUTPUT);

VAR

I, J, K, AUX : INTEGER;

```
AL : ARRAY [1..30,1..5] OF INTEGER;

BEGIN
  WRITELN ('INGRESE EL NRO Y 4 NOTAS DE LOS 30 ALUMNOS');
  FOR I := 1 TO 30 DO
    BEGIN
      FOR J := 1 TO 5 DO
        READ ( AL[I,J]);
      WRITELN
      END;
  FOR I := 1 TO 29 DO
    FOR J := I+1 TO 30 DO
      IF AL[I,1] > AL[J,1]
      THEN
        FOR K := 1 TO 5 DO
          BEGIN
            AUX := AL[I,K];
            AL[I,K] := AL[J,K];
            AL[J,K] := AUX
          END;
      WRITELN ('  ALUMNO NOTA 1 NOTA 2 NOTA 3 NOTA 4');
      FOR I := 1 TO 30 DO
        BEGIN
          FOR J := 1 TO 5 DO
            WRITE ( AL[I,J] : 8);
        WRITELN
        END
      END.
```

4-3-2-2. BUSQUEDA EN UN ARREGLO BIDIMENSIONAL:

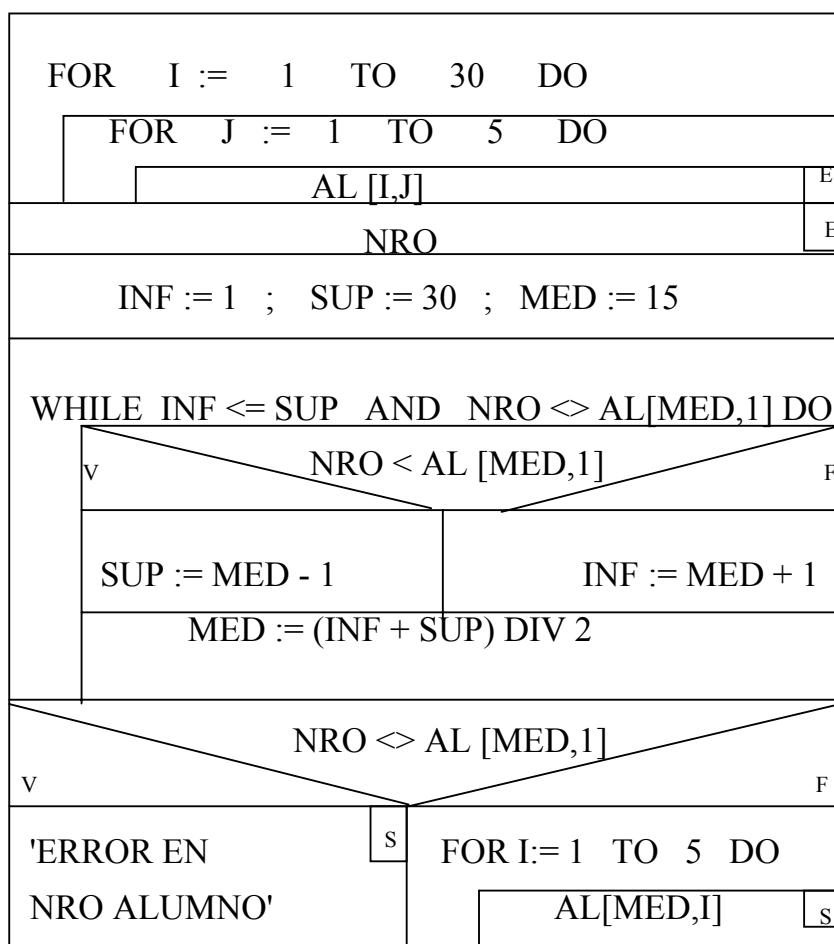
Si los elementos de un arreglo bidimensional están ordenados por una columna o una fila, ya sea en forma creciente o decreciente, se puede buscar un valor dentro de dicha fila o columna por medio de la búsqueda dicotómica.-

EJEMPLO:

Se tiene los datos de los 30 alumnos del ejercicio anterior ya ordenados en forma creciente por NRO DE ALUMNO.

Se desea ingresar un NRO DE ALUMNO y buscarlo por medio de la búsqueda dicotómica dentro del arreglo. Si se encuentra, dar el nro de alumno junto con las notas de los 4 parciales; sino exhibir cartel aclaratorio..-

a) DIAGRAMA DE CHAPIN:



b) PROGRAMA PASCAL:

```
PROGRAM BUSQUEDA (INPUT, OUTPUT);
VAR
  I, J, INF, SUP, MED, NRO : INTEGER;
  AL : ARRAY [1..30, 1..5] OF INTEGER;
BEGIN
  WRITE ('INGRESE NRO Y 4 NOTAS DE LOS 30 ALUMNOS');
  FOR I := 1 TO 30 DO
    BEGIN
      FOR J := 1 TO 5 DO
        READ (AL[I,J]); WRITELN
    END;
  WRITE ('INGRESE NRO. DEL ALUMNO A BUSCAR');
  READLN (NRO);
  INF := 1; SUP := 30; MED := 15;
  WHILE INF <= SUP AND NRO <> AL[MED,1] DO
    BEGIN
      IF NRO < AL[MED,1]
      THEN SUP := MED - 1
      ELSE INF := MED + 1;
      MED := (INF + SUP) DIV 2
    END;
  IF NRO <> AL [MED,1]
  THEN WRITE (' ERROR EN EL NRO. DE ALUMNO')
  ELSE
    BEGIN
      WRITELN (' ALUMNO NOTA 1 NOTA 2 NOTA 3 NOTA 4');
      FOR I := 1 TO 5 DO
        WRITE (AL[MED,I])
    END
  END.
```

UNIDAD N°5

SUBPROGRAMAS

5 -1. INTRODUCCION

Es frecuente en programación que un grupo de sentencias deba repetirse varias veces con distintos datos, o sea que debamos escribirlas varias veces. PASCAL permite escribirlas una sola vez bajo la forma de subprogramas y usarlas las veces que sea necesario.-

Además, si un grupo de sentencias realiza una tarea específica, puede estar justificado el aislarlas formando un subprograma, aunque se las use una sola vez.-

Hay dos tipos de subprogramas: ***FUNCIONES*** y ***PROCEDIMIENTOS***.-

5-1-1. FUNCIONES:

Una función PASCAL es un grupo de sentencias dentro de un programa que forman un bloque (un subprograma) que realiza un número determinado de operaciones sobre un conjunto de argumentos dado y devuelve un "**solo valor**".-

Cada vez que se llama a la función, se transfiere el control al bloque de sentencias definidas por esa función.

Después que las sentencias han sido ejecutadas, el control vuelve a la sentencia en que fue llamada la función.-

La invocación de una función es de la forma:

nombre (argumento1, argumento2,...)

donde nombre, que es el nombre de la función, es un identificador válido en PASCAL y es donde vuelve el resultado; y cada argumento puede ser cualquier variable válida, constante o expresión.-

Esta invocación debe ser asignada a una variable, formar parte de una expresión asignada a una variable, puede estar en un write o en un if.-

Una definición de función tiene la forma:

FUNCTION *nombre (declaración de parámetros): tipo;*
declaración de identificadores locales

Begin

sentencias ejecutables

End;

donde nombre es el nombre de la función; declaración de parámetros, contiene los parámetros (cada uno de los cuales debe ser un identificador válido en PASCAL) de la función y los tipos de datos que se asocian con cada uno de ellos; y tipo es el tipo de resultado que devuelve la función.-

El orden de la lista de parámetros es el orden de correspondencia de dichos parámetros con la lista de argumentos de la llamada. Por lo tanto en número de parámetros y de argumentos debe ser el mismo, y el tipo de los que se correspondan debe coincidir.-

Si una función no necesita pasar parámetros, entonces se omiten las listas de parámetros y de argumentos.-

En la lista de parámetros sólo se permiten identificadores de tipo standard o de un tipo definido por el programador que debe declararse en el programa principal (o en el que hace la llamada).-

En la declaración de identificadores locales van las declaraciones de las variables que no son parámetros, usadas por la función.-

Las sentencias propiamente dichas de la función (cuerpo de la función) van entre Begin y End; , como ocurre con el programa principal.-

Las definiciones de función siempre tienen lugar después de la sección de declaración de variables, pero antes del cuerpo del programa en el cual es invocada dicha función.-

EJEMPLO:

Escribir un programa que calcule la expresión : $\sum_{i=0}^n x^i$

para cualquier par de valores n y x.-

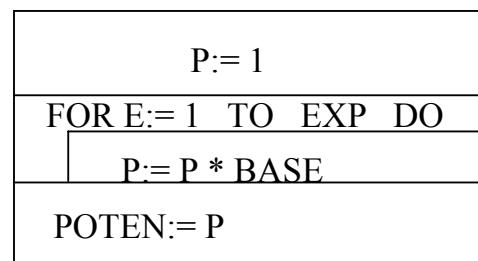
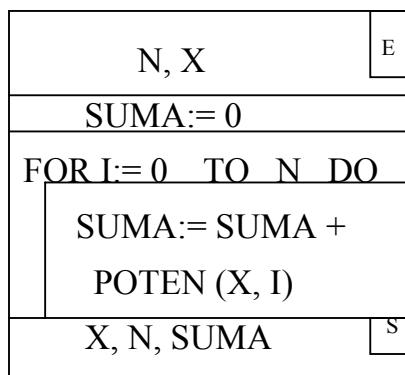
Para evaluar cada uno de los términos de la sumatoria, crear y utilizar una función llamada POTEN, que tenga como parámetros la base x y el exponente i.-

Exhibir: x, n y el resultado de la sumatoria.-

a) DIAGRAMA DE CHAPIN:

PROG. PPAL.

FUNCTION POTEN (BASE: real; EXP: integer): real;



b) PROGRAMA PASCAL:

PROGRAM SUMATORI (INPUT, OUTPUT);

VAR

 N, I: INTEGER;

 X, SUMA: REAL;

FUNCTION POTEN (BASE: REAL; EXP: INTEGER) : REAL;

```
VAR
P: REAL;
E: INTEGER;
BEGIN
P:= 1;
FOR E:= 1 TO EXP DO
P:= P * BASE;
POTEN:= P
END;
BEGIN
WRITE ('INGRESE EXTREMO SUMATORIA Y NRO.');
READLN (N, X);
SUMA:= 0;
FOR I:= 0 TO N DO
SUMA:= SUMA + POTEN(X, I);
WRITELN ('LA SUMATORIA DE LOS TERMINOS DE BASE',X:3:2);
WRITELN ('DESDE POTENCIA 0 A POTENCIA', N);
WRITE ('ES IGUAL A', SUMA:10:2)
END.
```

5-1-2- PROCEDIMIENTOS:

Los procedimientos son subprogramas similares a las funciones pero con dos diferencias importantes.

La llamada a un procedimiento es similar a la de la función:

nombre (argumento1, argumento2,...)

pero esta debe estar sola, es decir no puede estar formando parte de expresiones, ni asignada a una variable, ni en un write, ni en un if. (primer diferencia)

La segunda diferencia es que con el procedimiento no se devuelve un valor al punto de llamada.

Un procedimiento está compuesto de un grupo de sentencias a las que se asigna un nombre (*identificador*) y constituye una unidad del programa. La tarea asignada al procedimiento se ejecuta siempre que Pascal encuentra el nombre del procedimiento.-

La definición de un procedimiento en PASCAL es muy similar a la de una función, salvo algunas diferencias. Primero, la palabra clave es **PROCEDURE** en lugar de **FUNCTION** en la cabecera, y segundo que no contiene ningún atributo detrás de las declaraciones de los parámetros puesto que no vuelve ningún valor en el nombre del procedimiento.-

Una definición de procedimiento tiene la forma:

PROCEDURE *nombre* (*declaración de parámetros*);
declaración de identificadores locales

Begin

sentencias ejecutables

End;

Si un parámetro tiene que modificar el valor de su argumento correspondiente, dicho parámetro debe ir precedido por la palabra **VAR**. En ausencia de **VAR**, cualquier cambio hecho al valor del parámetro no será reflejado en su argumento.-

Tanto en procedimientos como en funciones, como ya dijimos, las variables que son utilizadas por ellos van declaradas en la parte de declaración de **variables locales** y se las denomina de esa manera.-

Las variables declaradas en un programa que contenga funciones o procedimientos, son válidas (pueden utilizarse) en cualquier parte del programa, también dentro de las funciones y los procedimientos; a estas variables se las denomina **variables globales**.

Si durante la ejecución de un subprograma se usa y se altera el valor de una variable global, al terminar el subprograma, el valor de la variable corresponde al último valor y no al original que tenía antes de que se ejecutara el subprograma.-

5 -2. CORRESPONDENCIA ARGUMENTO-PARAMETRO:

Como ya hemos visto, cada vez que se llama a una función o a un procedimiento, se establece una correspondencia entre cada argumento y su parámetro.-

Esta llamada puede ser:

por valor o por referencia.-

5-2-1. LLAMADA POR VALOR:

Es la asignación del valor del argumento a su parámetro. El parámetro es entonces una variable independiente, de nueva creación que recibe el valor del argumento al comienzo de la ejecución del subprograma.

Puesto que parámetro y argumento son variables independientes, cualquier cambio que se produzca en el parámetro durante la ejecución del subprograma no tiene efecto en el valor del argumento.

Por lo tanto cuando se utiliza la llamada por valor, es imposible la devolución de valores al punto de llamada por medio de los parámetros; pues al terminar la ejecución del subprograma, la variable parámetro se destruye y el valor que contenía se pierde.-

5-2-2. LA LLAMADA POR REFERENCIA:

En el caso de que se requiera que el valor de una variable sea modificado por el subprograma invocado, debe hacerse el paso de parámetro por referencia, por medio del cual el subprograma invocado tiene acceso a la dirección en que se guarda el valor a modificar.

No implica la creación de una posición aparte de memoria para el parámetro, sino

mas bien produce el paso de la dirección de memoria donde se almacena el valor del argumento.-

El parámetro, en efecto, viene a ser simplemente otro nombre (o el mismo) para la misma dirección ya creada para el valor del argumento.

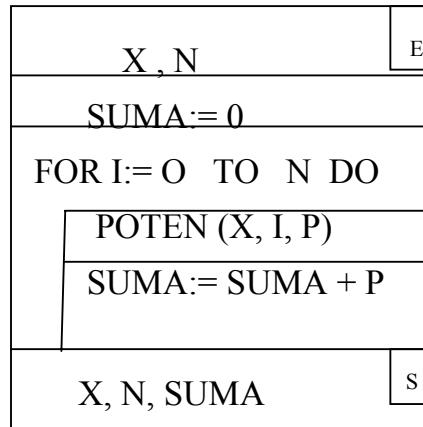
En este tipo de llamada, los parámetros van precedidos por la palabra **VAR**.-

EJEMPLO:

Realicemos el ejemplo anterior, pero en lugar de utilizar una función, utilicemos un Procedimiento llamado POTEN.-

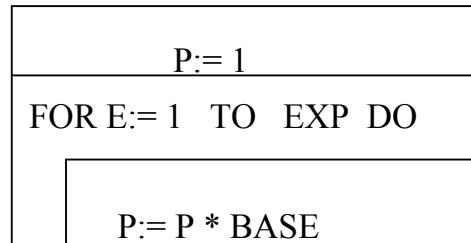
a) DIAGRAMA DE CHAPIN:

PROG. PPAL.



PROCEDURE POTEN (BASE:real;

EXP:integer; var P:real)



b) PROGRAMA PASCAL:

PROGRAM SUMATORI (INPUT, OUTPUT);

VAR

N, I: INTEGER;

X, SUMA, P: REAL;

```
PROCEDURE POTEN (BASE: REAL; EXP: INTEGER; VAR P: REAL);
VAR
  E: INTEGER;
BEGIN
  P:= 1;
  FOR E:= 1 TO EXP DO
    P:= P * BASE
  END;
  BEGIN
    WRITE ('INGRESE EXTREMO SUMATORIA Y NRO.');
    READLN (N, X);
    SUMA:= 0;
    FOR I:= 0 TO N DO
      BEGIN
        POTEN (X, I, P);
        SUMA:= SUMA + P
      END;
    WRITELN ('LA SUMATORIA DE LOS TERMINOS DE BASE',X:3:2);
    WRITELN ('DESDE POTENCIA 0 HASTA POTENCIA', N);
    WRITE ('ES IGUAL A', SUMA:10:2)
  END.
```

5-3- RECUSIVIDAD

En Pacal, a un procedimiento o función le es permitido no sólo invocar a otro procedimiento o función, sino también invocarse a sí mismo. Una invocación de éste tipo se dice que es *recursiva*.

La *función recursiva* más utilizada como ejemplo es la que calcula el factorial de un número entero no negativo, partiendo de las siguientes definiciones :

$$\text{factorial (0)} = 1$$

$$\text{factorial (n)} = n * \text{factorial}(n-1), \text{ para } n > 0$$

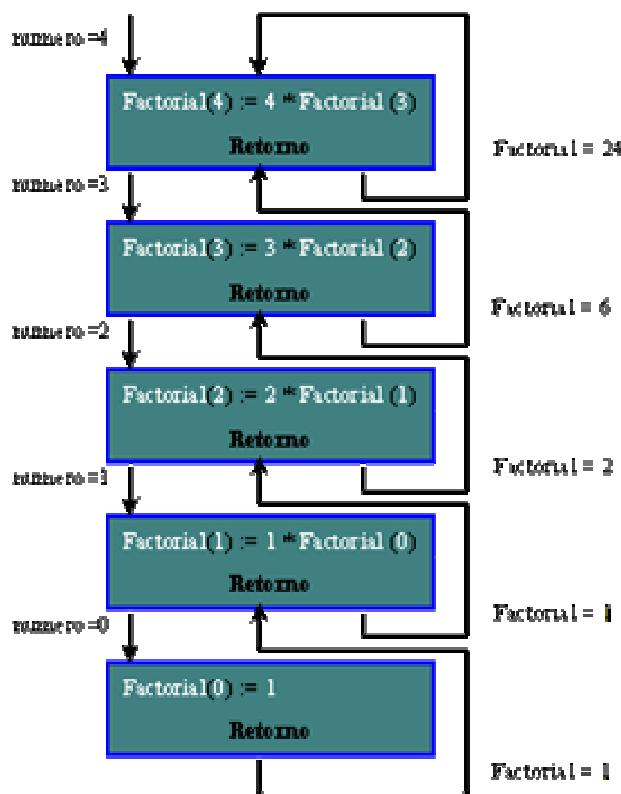
La función, escrita en Pascal, queda de la siguiente manera :

```

function factorial(numero:integer):integer;
begin
  if numero = 0 then
    factorial := 1
  else
    factorial := numero * factorial(numero-1)
end;

```

Si numero = 4, la función realiza los siguientes pasos :



1. $\text{factorial}(4) := 4 * \text{factorial}(3)$ Se invoca a si misma y crea una segunda variable cuyo nombre es numero y su valor es igual a 3.
2. $\text{factorial}(3) := 3 * \text{factorial}(2)$ Se invoca a si misma y crea una tercera variable cuyo nombre es numero y su valor es igual a 2.
3. $\text{factorial}(2) := 2 * \text{factorial}(1)$ Se invoca a si misma y crea una cuarta variable cuyo nombre es numero y su valor es igual a 1.
4. $\text{factorial}(1) := 1 * \text{factorial}(0)$ Se invoca a si misma y crea una quinta variable cuyo nombre es numero y su valor es igual a 0.
5. Como $\text{factorial}(0) := 1$, con este valor se regresa a completar la invocación: $\text{factorial}(1) := 1 * 1$, por lo que $\text{factorial}(1) := 1$

Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

6. Con factorial(1) := 1, se regresa a completar: factorial(2) := 2 * 1, por lo que factorial(2) := 2
7. Con factorial(2) := 2, se regresa a completar : factorial(3) := 3 * 2, por lo que factorial(3) := 6
8. Con factorial(3) := 6, se regresa a completar : factorial(4) := 4 * 6, por lo que factorial(4) := 24 y éste será el valor que la función factorial devolverá al módulo que la haya invocado con un valor de parámetro local igual a 4 .

Otro ejemplo de procedimiento recursivo es el siguiente:

Supóngase que una persona se mete a una piscina cuya profundidad es de 5 metros. Su intención es tocar el fondo de la piscina y después salir a la superficie. Tanto en el descenso como en el ascenso se le va indicando la distancia desde la superficie (a cada metro).

Program Piscina;

Const

prof_max = 5;

Var

profundidad:integer;

procedure zambullida(Var profun :integer);

begin

 WriteLn('BAJA 1 PROFUNDIDAD = ',profun);

 profun := profun + 1;

 if profun <= prof_max then

 zambullida(profun)

 else

 WriteLn;

 profun := profun - 1;

 WriteLn('SUBE 1 PROFUNDIDAD = ', profun-1)

end;

begin

 profundidad := 1;

 zambullida(profundidad)

end.

UNIDAD N° 6

OTRAS ESTRUCTURAS DE DATOS

6-1. REGISTROS:

Hasta ahora, la única estructura de datos que hemos visto son los arrays. Los registros son similares a los arrays pues también representan un grupo de elementos con un nombre común. Sin embargo, mientras que los elementos de un array deben ser todos del mismo tipo, los elementos de un registro pueden ser de distintos tipos de datos.

O sea, los registros son un tipo de datos estructurado (ó variable compuesta), con un número fijo de componentes (no necesariamente del mismo tipo) a las que se accede por el nombre, no por un subíndice.-

La declaración en Pascal de este tipo de datos es:

```
TYPE nombre = RECORD  
    identificador del campo: tipo;
```

```
    .  
    .  
    .  
    identificador del campo: tipo  
END;
```

donde:

nombre es un identificador válido en Pascal

identificador del campo, es el nombre de una componente del registro

tipo, es el tipo de elemento de cada campo

Ejemplo:

```
TYPE ALUMNO = RECORD
    NOMBRE: STRING [20];
    LEGAJO: REAL;
    DNI: REAL;
    NOTAS: ARRAY (1..6) OF INTEGER;
    ACURSA: INTEGER
END;

VAR
    ESTUD: ALUMNO;
```

Para referirnos a un campo del registro especificamos el nombre de la variable y el nombre del campo separados por un punto.

Ejemplo: ESTUD.DNI

Si el campo es una variable estructurada, como en el ejemplo anterior que NOTAS es un array, se debe además especificar el subíndice para acceder a un elemento particular de dicho campo.

Por ejemplo, si quisiéramos mostrar el campo NOTAS:

```
FOR I:= 1 TO 6 DO
    WRITE (ESTUD.NOTAS[I]);
```

Como ya hemos dicho, las componentes de un registro pueden ser de cualquier tipo, o sea una componente de un registro puede ser también otro registro. Los registros que tienen este tipo de componentes se llaman registros jerárquicos.

Ejemplo:

```
TYPE NOMB = ARRAY [1..20] OF CHAR;
    EMPLE = RECORD
```

```
NOMBRE : NOMB;  
DIRECC : RECORD  
    CALLE : NOMB;  
    NUM : INTEGER;  
    PISO : INTEGER;  
    DPTO: CHAR  
END;  
SUELDO : REAL  
END;  
VAR  
    EMPLEADO : EMPLE;
```

Un elemento particular de esta variable sería:

```
EMPLEADO.DIRECC.PISO
```

6-2. ARREGLOS DE REGISTROS:

Un arreglo de este tipo, es simplemente un arreglo cuyos elementos son registros.

Ejemplo: si tuviéramos el tipo de datos definido anteriormente, podríamos definir :

```
TYPE EMPRESA = ARRAY [1..200] OF EMPLE;  
VAR PERSONAL : EMPRESA;
```

Así:

```
WRITE (PERSONAL(4).SUELDO);
```

Nos mostraría el sueldo del empleado que está en la posición 4

EJERCICIO:

Un negocio de ventas al por mayor y al por menor, comercializa 100 productos distintos.-

El comerciante desea actualizar la lista de precios al público y de stock de los productos; para ello se ingresan, para cada uno de ellos, los siguientes datos:

- **CODIGO** (entero)
- **DESCRIPCION** (hasta 20 caracteres)
- **CANTIDAD EN STOCK**
- **CANTIDAD MINIMA REQUERIDA EN STOCK**
- **PRECIO POR UNIDAD** (para ventas al por menor)
- **PRECIO POR UNIDAD PARA MAS DE 20 UNIDADES** (para ventas al por mayor)
- **PRECIO POR UNIDAD PARA MAS DE 50 UNIDADES** (" " " ")

Estos datos, que están *ordenados en forma creciente* por código de producto, se ingresarán por medio de un procedimiento.-

Luego se van ingresando los datos de los productos que tienen modificaciones (*no necesariamente los 100*):

- **CODIGO DEL ARTICULO** (entero)
- **CODIGO DE OPERACION** ("C": compra; "V": venta)
- **CANTIDAD** (cantidad comprada o vendida)
- **COEFICIENTE** (coeficiente de ajuste del precio unitario)

Con la cantidad comprada o vendida se actualizará el stock.-

El coeficiente de ajuste se deberá multiplicar al precio unitario para obtener el nuevo precio (ventas por menor); y para obtener los precios por unidad para más de 20 o más de 50 unidades, se le aplicará *al nuevo precio unitario obtenido* un descuento del 10% y el 15% respectivamente.-

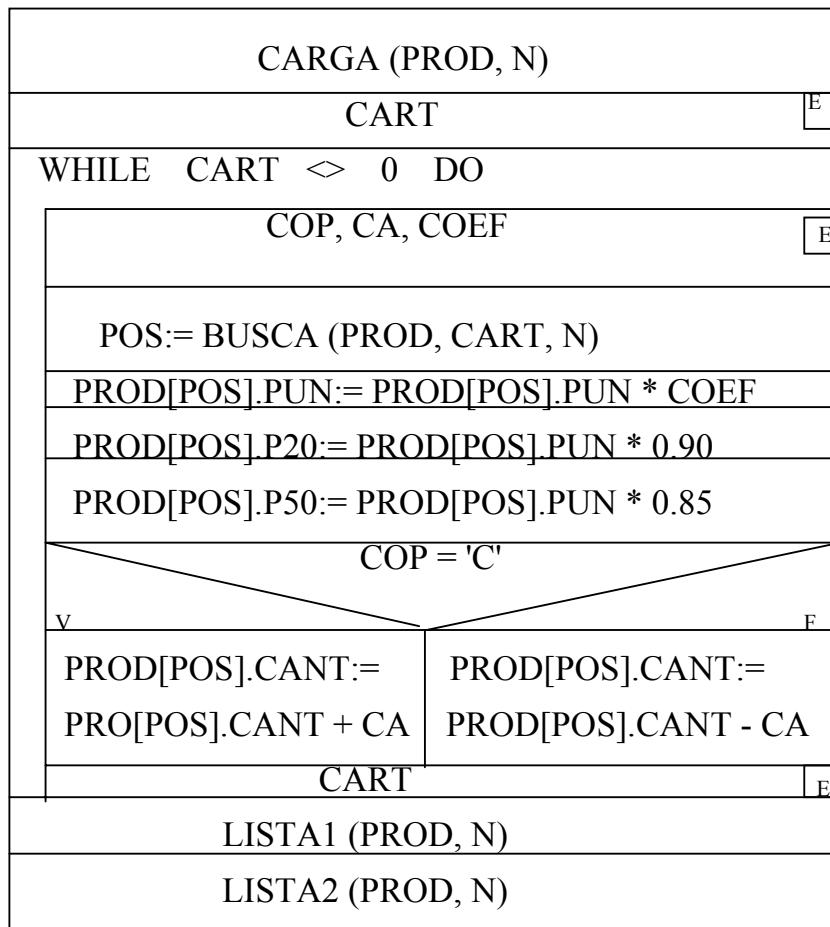
Se pide:

- * Un listado actualizado de los 100 productos.-
- * Un listado con los códigos y la descripción de los productos que tienen faltantes en stock con dicho faltante.-

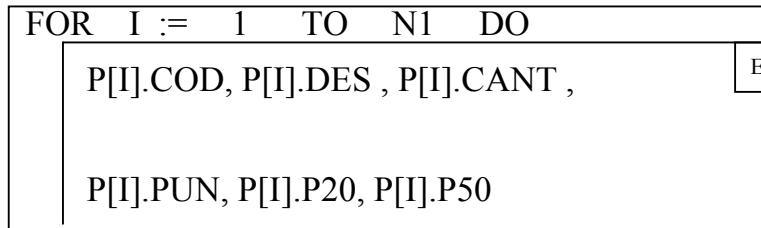
NOTA: La búsqueda del código a actualizar se hará por medio de una *FUNCION* que devuelva la posición en donde se encuentra dicho código (suponer que siempre se encontrará el mismo).-

a) Diagrama de Chapin:

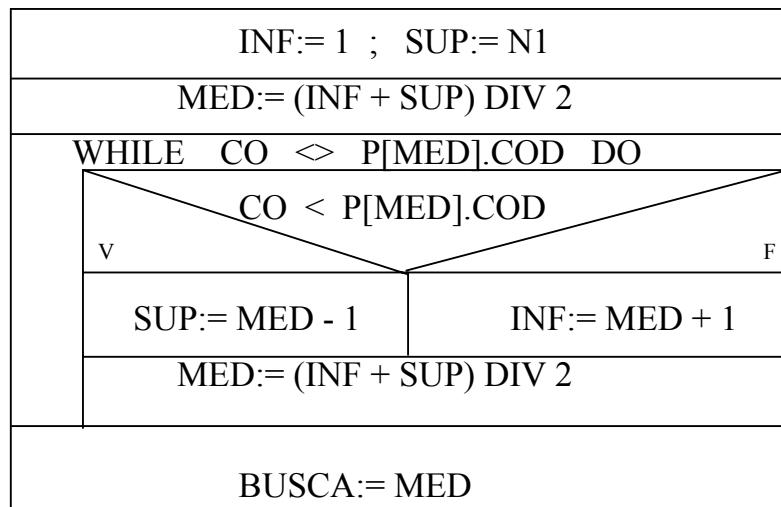
PROGRAMA PRINCIPAL



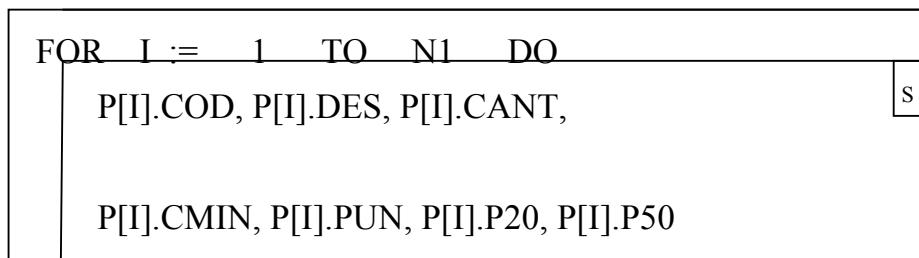
PROCEDURE CARGA (VAR P: PRODUCTO; N1: INTEGER);



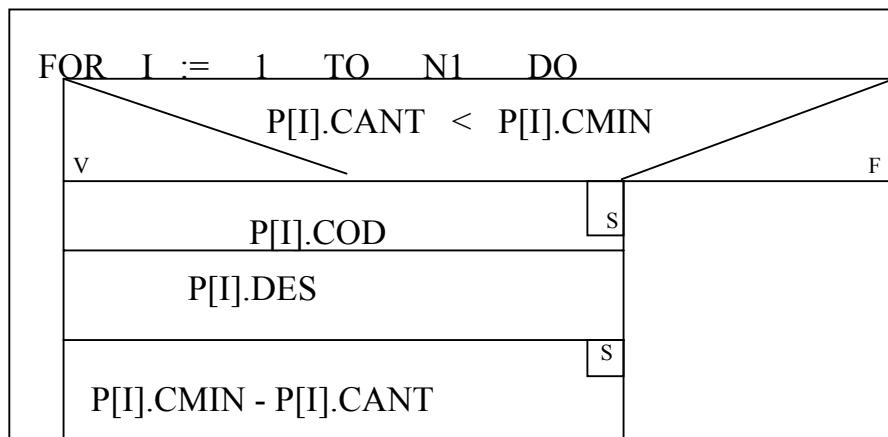
FUNCTION BUSCA (VAR P: PRODUCTO; CO, N1: INTEGER) : INTEGER



PROCEDURE LISTA1 (VAR P: PRODUCTO; N1: INTEGER)



PROCEDURE LISTA2 (VAR P: PRODUCTO; N1: INTEGER)



Programa PASCAL:

```
PROGRAM STOCK (input, output);
CONST
  N = 100;
TYPE
  ARTI = RECORD
    COD, CANT, CMIN: INTEGER;
    DES: STRING(20);
    PUN, P20, P50: REAL
  END;
  PRODUCTO = ARRAY [1..N] OF ARTI;
VAR
  PROD: PRODUCTO;
  CART, CA, POS: INTEGER;
  COP: CHAR;
  COEF: REAL;

PROCEDURE CARGA (VAR P: PRODUCTO; N1: INTEGER);
VAR
  I, C: INTEGER;
BEGIN
  FOR I := 1 TO N1 DO
    BEGIN
      WRITE ('INGRESE CODIGO');
      READLN (P[I].COD);
      WRITE ('INGRESE DESCRIPCION DEL ARTICULO');
      READLN (P[I].DES);
      WRITE ('INGRESE CANTIDAD');
      READLN (P[I].CANT);
      WRITE ('INGRESE CANT. MINIMA');
      READLN (P[I].CMIN);
    END;
END;
```

```
WRITE ('INGRESE PRECIO UNITARIO');
READLN (P[I].PUN);
WRITE ('INGRESE PRECIO UNIT. P/20 Y P/50');
READLN (P[I].P20, P[I].P50)
END
END;
```

```
FUNCTION BUSCA (VAR P: PRODUCTO; CO, N1: INTEGER) :INTEGER;
VAR
  INF, SUP, MED: INTEGER;
BEGIN
  INF:= 1 ; SUP:= N1 ;
  MED:=(INF + SUP) DIV 2;
  WHILE (CO <> P[MED].COD) DO
    BEGIN
      IF CO < P[MED].COD
        THEN SUP := MED - 1
        ELSE INF := MED + 1;
      MED := (INF + SUP) DIV 2
    END;
  BUSCA:= MED
END;
```

```
PROCEDURE LISTA1 (VAR P: PRODUCTO; N1: INTEGER);
VAR
  I, C : INTEGER;
BEGIN
  FOR I := 1 TO N1 DO
    BEGIN
      WRITE (P[I].COD);
      WRITE (P[I].DES);
    END;
END;
```

```
    WRITE (P[I].CANT);
    WRITELN (P[I].PUN, P[I].P20, P[I].P50)
END
END;
```

```
PROCEDURE LISTA2 (VAR P: PRODUCTO; N1: INTEGER);
```

```
VAR
```

```
    I, C: INTEGER;
```

```
BEGIN
```

```
    FOR I:= 1 TO N1 DO
```

```
        IF P[I].CANT < P[I].CMIN
```

```
        THEN
```

```
            BEGIN
```

```
                WRITE (P[I].COD);
```

```
                WRITE (P[I].DES[C]);
```

```
                WRITELN (P[I].CMIN - P[I].CANT)
```

```
            END
```

```
        END;
```

```
BEGIN
```

```
    CARGA (PROD, N);
```

```
    WRITE ('INGRESE CODIGO ARTICULO');
```

```
    READLN (CART);
```

```
    WHILE CART <> 0 DO
```

```
        BEGIN
```

```
            WRITE ('INGRESE COD. OPERACION, CANT. Y COEF.');
```

```
            READLN (COP, CA, COEF);
```

```
            POS := BUSCA (PROD, CART, N);
```

```
            PROD[POS].PUN := PROD[POS].PUN * COEF;
```

```
            PROD[POS].P20 := PROD[POS].PUN * 0.9;
```

```
            PROD[POS].P50 := PROD[POS].PUN * 0.85;
```

```
IF COP = 'C'  
    THEN PROD[POS].CANT := PROD[POS].CANT + CA  
    ELSE PROD[POS].CANT := PROD[POS].CANT - CA;  
    WRITE ('INGRESE CODIGO ARTICULO');  
    READLN (CART)  
END;  
LISTA1 (PROD, N);  
LISTA2 (PROD, N)  
END.
```

UNIDAD N° 7

ARCHIVOS

7-1- INTRODUCCION

Hasta ahora hemos visto diferentes estructuras de datos que son definidas en un algoritmo y ocupan memoria RAM, se utilizan en la ejecución y todos los valores contenidos en ellas se pierden, a lo sumo, cuando el algoritmo finaliza.

Para determinados problemas es necesario disponer de un tipo de estructuras de datos que permita guardar su información en soporte no volátil (disco, diskette, cinta, zip), y de esta forma preservarlas aunque el programa finalice. Estas estructuras de datos son conocidas además como estructuras de almacenamiento, y deben asociarse con un dispositivo de memoria auxiliar permanente donde archivar la información. Dichas estructuras se denominan archivos o ficheros.

7-2- ARCHIVOS

Un archivo es una estructura de datos que guarda en un dispositivo de almacenamiento secundario de una computadora, a una colección de elementos del mismo tipo.

Cuando se trabaja con un archivo en disco rígido se hace referencia a un conjunto de datos almacenados en memoria secundaria. El disco rígido puede almacenar un gran número de archivos.

Sin embargo, desde un programa la noción de archivo no es exactamente la misma. Un programa no sabe con exactitud el lugar donde residirán los datos, o sea , el lugar dentro del disco rígido donde estará el archivo físico; el programa envía y recibe información, desde y hacia el archivo. Para poder efectuar esto, el programa de aplicación se apoya en el sistema operativo de la computadora, que es el encargado de los detalles de almacenamiento, tales como lugar dentro del dispositivo físico, cantidad de espacio que va a utilizar, tipos de acceso permitido (si se puede leer, leer y escribir, si el acceso está vedado, etc.), usuarios que tienen acceso a los datos del

archivo, etc. El programa referencia directamente al archivo físico utilizando para ello un nombre lógico, que se denomina archivo lógico.

Antes de que el programa pueda operar sobre archivos, el sistema operativo debe recibir instrucciones para hacer un enlace entre el nombre lógico que utilizará el algoritmo y el archivo físico. Cuyo formato será, en líneas generales:

Asignar_correspondencia (nombre_físico, nombre_lógico)

En Pascal:

Assign (nombre_físico, nombre_lógico)

El nombre físico define exactamente el nombre con el que el sistema operativo encontrará al archivo dentro del almacenamiento secundario. En tanto, el nombre lógico se corresponde con una variable definida en el algoritmo. Dicha variable debe ser de tipo archivo.

La declaración de un archivo en Pascal se hará de la siguiente forma:

Var archivo: file of tipo_de_datos;

ó:

Type archivo = file of tipo_de_datos;

Var archi: archivo;

7-3- BUFFERS

Se denomina buffer a una memoria intermedia entre un archivo y un programa, donde los datos residen provisoriamente hasta ser almacenados definitivamente en memoria secundaria o donde los datos residen una vez recuperados de dicha memoria secundaria. Los buffers ocupan una zona de la memoria RAM de la computadora.

Manejar buffers implica trabajar con grupos de datos en memoria RAM para que el número de accesos al almacenamiento secundario se reduzca. Básicamente las operaciones de lectura y escritura no se realizan directamente sobre la memoria secundaria. Si esto fuera así, se necesitaría, ante cada uno de estas operaciones una determinada cantidad de milisegundos para realizarla. Por lo tanto, estas operaciones que se definen en los algoritmos, interactúan con un buffer, el cual al encontrarse en memoria RAM agiliza el proceso.

El sistema operativo de la computadora es el encargado de manipular los buffers. Cuando un programa realiza una operación de lectura y en el buffer no hay información para satisfacerla, se lee de memoria secundaria los datos para completar nuevamente dicho buffer y así poder satisfacer el requerimiento. Algo similar ocurre con la escritura, cuando se intenta escribir en un buffer y el mismo no tiene capacidad, la información contenida es bajada o guardada en memoria secundaria, una vez vacío el buffer puede tomar los datos definidos en la orden de escritura.

7-4- OPERACIONES BASICAS SOBRE ARCHIVOS

7-4-1- APERTURA Y CREACIÓN

- Para abrir un archivo existente en memoria secundaria como lectura y escritura:

Reset (nombre_lógico);

- Para crear un archivo, o sea para abrir un archivo de sólo escritura:

Rewrite (nombre_lógico);

Donde nombre_lógico es la variable de tipo archivo sobre la que se realizó la asignación correspondiente.

Téngase en cuenta que si se realiza la apertura con la instrucción Rewrite y el archivo contiene información, la misma se perderá completamente.

7-4-2- CIERRE

Para efectuar el cierre explícito de un archivo y colocar la marca de fin de archivo:

Close (nombre_lógico);

7-4-3- LECTURA Y ESCRITURA

Para leer datos de un archivo:

Read (nombre_lógico, variable);

Para escribir datos en un archivo:

Write (nombre_lógico, variable);

Donde variable es una variable cuyo tipo de dato debe corresponderse con la definición del archivo.

Cada una de estas instrucciones opera sobre la posición actual del archivo, y luego avanza a la posición siguiente.

7-5- OTRAS OPERACIONES SOBRE ARCHIVOS

7-5-1- FIN DE ARCHIVO

Para recorrer un archivo desde el primer elemento hasta el final, es necesario contar con operación que detecte el fin de dicho archivo:

Eof (nombre_lógico);

Esta función devuelve un valor booleano que será True si la posición corriente dentro del archivo referencia a la marca de fin, y False, en caso contrario.

7-5-2- NÚMERO DE ELEMENTOS DEL ARCHIVO

Si necesitamos saber la cantidad de elementos o registros que posee un archivo:

Filesize (nombre_lógico);

Esta función devuelve un número entero, que corresponde a la cantidad de registros del archivo.

7-5-3- POSICIÓN ACTUAL

A veces necesitamos operar en otros procesos con la posición actual del archivo, para lo cual debemos poder determinarla:

Filepos (nombre_lógico);

Esta función devuelve un número entero que corresponde a la posición actual del apuntador del archivo.

7-5-4- POSICIONARSE EN UN ELEMENTO

Para modificar el contenido de un elemento particular de un archivo necesitamos posicionarnos en esa dirección:

Seek (nombre_lógico, posición);

Esta función permite llegar a un elemento particular del archivo. Donde posición es un número entero menor a la cantidad de elementos del archivo.

7-6- ORGANIZACIÓN Y ACCESO A UN ARCHIVO

Según el modo en que se organizan los registros dentro de un archivo, se consideran dos tipos de acceso:

- **Acceso secuencial**
- **Acceso directo**

El acceso secuencial permite acceder a los elementos o registros uno tras otro y en el orden físico en que están guardados. El acceso directo, en cambio, permite obtener un registro determinado sin necesidad de haber accedido a los anteriores.

De acuerdo a su organización, un archivo puede clasificarse como:

- **Directo**
- **Secuencial**
- **Secuencial Indizado**

Esta organización define la manera en que los registros se distribuyen sobre el almacenamiento secundario.

Un archivo secuencial consiste de un conjunto de registros almacenados consecutivamente de manera que para acceder al registro n-ésimo se debe, previamente, acceder a los n-1 registros anteriores. Los registros se graban en forma consecutiva, a medida que se ingresan, y se recuperan en el mismo orden.

Un archivo directo consiste de un conjunto de registros donde el ordenamiento físico no necesariamente corresponde con el ordenamiento lógico. Los registros se recuperan accediendo por su posición dentro del archivo. Por lo tanto es posible acceder al n-ésimo lugar sin haber accedido a los n-1 registros anteriores. Esta organización presenta la ventaja que se puede obtener cualquier elemento del archivo en cualquier orden, siendo muy eficientes en cuanto a tiempo de acceso necesario para recuperar la información; pero presenta el inconveniente de tener que determinar la posición donde se encuentra cada elemento.

Un archivo secuencial indizado utiliza estructuras de datos auxiliares para permitir un acceso pseudo directo a los registros del archivo. Un ejemplo de esta organización es la guía telefónica, en la que se puede acceder por letra, y dentro de cada página existe una indicación de apellido de comienzo y apellido de fin dentro de la hoja; de esta forma se puede acotar el espacio de búsqueda de un determinado teléfono dentro de la guía, haciendo referencia mucho más rápidamente a la hoja donde se encuentra el dato. Los archivos organizados con esta técnica tienen la ventaja de tener un acceso mucho más rápido que los secuenciales, pero necesitan más espacio para mantener las estructuras de los índices. Estas estructuras se denominan directorios del archivo.

7-7- ARCHIVOS DE TEXTO

Pascal estándar define dos archivos de texto (TEXT) por defecto que son los archivos **INPUT** y **OUTPUT**, que se corresponden con los periféricos estándar de entrada y salida en la instalación, normalmente el teclado y la pantalla o impresora.

Los procedimientos **READ** y **WRITE** cuando omiten el archivo empleado, se sobreentiende que se emplea estos dos archivos por defecto, por ello es obligatorio su utilización como parámetros en el encabezamiento del programa, no así en Turbo Pascal, donde se pueden omitir.

Los archivos de texto se dividen en líneas formadas por conjuntos de caracteres, separadas unas de otras por caracteres de control especiales. En el código ASCII, la marca separadora de líneas está constituida por la combinación de caracteres *CR/LF* (retorno de carro/avance de línea).

En este tipo de archivos también se utiliza la variable booleana **Eoln** (f) para indicar si el buffer se encuentra o no sobre la marca separadora de líneas. En caso afirmativo, toma el valor true. En algunas implementaciones de Pascal, el contenido del buffer cuando se encuentra en la marca de fin de línea es un carácter en blanco (#32), por lo que debe tenerse esto en cuenta, por ejemplo, si hacemos una estadística de caracteres de un archivo de texto.

Las funciones **Eoln** y **Eof** se refieren al fichero **Input** a menos que se especifique un fichero o archivo diferente, en ese caso sería **Eoln(nombre_lógico)** y **Eof(nombre_lógico)**.

7-8- EJERCICIO

Escribir un procedimiento que actualice los salarios de los empleados de una empresa, aumentándolos un 10%, considerando que el archivo ya existe, que la asignación nombre físico, nombre lógico está realizada en el programa principal y que el archivo no está abierto.

El archivo debe quedar actualizado y cerrado.

Supongamos que en el programa principal está hecha esta declaración:

Type registro = Record

 Nombre: string(20);

 Dirección: string (20);

Salario: real

End;

Empleados = file of registro;

Procedure Actualizar (var Emp: Empleados); *{se recibe el archivo como parámetro por referencia}*

Var E: registro;

Begin

Reset (Emp);

{el archivo contiene datos, se abre de E/S}

While not eof (Emp) do

{se evalúa si no se llegó a la marca de fin de archivo}

Begin

Read (Emp, E); *{se obtiene el elemento del archivo}*

E.salario := E.salario * 1.1; *{se incrementa el salario}*

Seek (Emp, filepos (Emp) - 1); *{luego de la lectura la posición corriente del archivo avanza una posición, para hacer la escritura se debe retroceder en uno dicha posición}*

Write (Emp, E);

End;

Close (Emp)

End;

Lectura 13. Modelos de Computación I

Modelos de Computación I

Serafín Moral

Departamento de Ciencias de la Computación e I.A.

ETSI Informática

Universidad de Granada

Índice general

1. Introducción	5
1.1. Introducción Histórica	7
1.2. Diferentes Modelos de Computación	11
1.2.1. Autómatas y Lenguajes	15
1.3. Lenguajes y Gramáticas. Aspectos de su traducción	17
1.3.1. Alfabetos y Palabras	17
1.3.2. Lenguajes	19
1.3.3. Gramáticas Generativas	22
1.3.4. Jerarquía de Chomsky	27
2. Autómatas Finitos, Expresiones Regulares y Gramáticas de tipo 3	33
2.1. Autómatas Finitos Determinísticos	34
2.1.1. Proceso de cálculo asociado a un Autómata de Estado Finito	37
2.1.2. Lenguaje aceptado por un Autómata de Estado Finito	39
2.2. Autómatas Finitos No-Determinísticos (AFND)	42
2.2.1. Diagramas de Transición	44
2.2.2. Equivalencia de Autómatas Determinísticos y No-Determinísticos	45
2.3. Autómatas Finitos No Determinísticos con transiciones nulas	48
2.4. Expresiones Regulares	52
2.4.1. Propiedades de las Expresiones Regulares	53
2.4.2. Expresiones Regulares y Autómatas Finitos	54
2.5. Gramáticas Regulares	61
2.6. Máquinas de Estado Finito	65
2.6.1. Máquinas de Moore	65
2.6.2. Máquinas de Mealy	67
2.6.3. Equivalencia de Máquinas de Mealy y Máquinas de Moore	68
3. Propiedades de los Conjuntos Regulares	71
3.1. Lema de Bombeo	72

3.2. Operaciones con Conjuntos Regulares	76
3.3. Algoritmos de Decision para Autómatas Finitos	78
3.4. Teorema de Myhill-Nerode. Minimización de Autómatas	79
3.4.1. Minimización de Autómatas	83
4. Gramáticas Libres de Contexto	89
4.1. Arbol de Derivación y Ambigüedad	90
4.2. Simplificación De Las Gramáticas Libres De Contexto	93
4.2.1. Eliminación de Símbolos y Producciones Inútiles	94
4.2.2. Producciones Nulas	97
4.2.3. Producciones Unitarias	99
4.3. Formas Normales	101
4.3.1. Forma Normal de Chomsky	101
4.3.2. Forma Normal de Greibach	102
5. Autómatas con Pila	107
5.1. Definición de Autómata con Pila	108
5.1.1. Lenguaje aceptado por un autómata con pila	109
5.2. Autómatas con Pila y Lenguajes Libres de Contexto	111
5.3. Lenguajes Independientes del Contexto Deterministas	113
6. Propiedades de los Lenguajes Libres del Contexto	117
6.1. Lema de Bombeo	118
6.2. Propiedades de Clausura de los Lenguajes Libres de Contexto	120
6.3. Algoritmos de Decisión para los Lenguajes Libres de Contexto	121
6.3.1. Algoritmos de Pertenencia	122
6.3.2. Problemas Indecidibles para Lenguajes Libres de Contexto	127

Capítulo 1

Introducción

Hoy en día parece que no existe ningún límite a lo que un ordenador puede llegar a hacer, y da la impresión de que cada vez se pueden resolver nuevos y más difíciles problemas.

Casi desde que aparece sobre La Tierra, el hombre ha tratado de buscar procedimientos y máquinas que le facilitasen la realización de cálculos (aritméticos primero, y otros más complejos posteriormente).

El avance tecnológico para representar datos y/o información por un lado, y el diseño de nuevas formas de manejarlos, propicia el desarrollo de dispositivos y máquinas de calcular.

Un aspecto importante en el desarrollo de los ordenadores, es sin duda, su aplicación para resolver problemas científicos y empresariales. Esta aplicación hubiese resultado muy difícil sin la utilización de procedimientos que permiten resolver estos problemas mediante una sucesión de pasos claros, concretos y sencillos, es decir algoritmos. El avance de las matemáticas permite la utilización de nuevas metodologías para la representación y manejo de la información.

Por otro lado, aparece el intento de los matemáticos y científicos para obtener un procedimiento general para poder resolver cualquier problema (matemático) claramente formulado. Es lo que podríamos llamar **El problema de la computación teórica**. El avance de la tecnología y de las matemáticas, y más en concreto de la teoría de conjuntos y de la lógica, permiten plantearse aspectos de la computación en 3 caminos.

- a) Computación teórica. Autómatas, Funciones Recursivas, ...
- b) Ordenadores digitales. Nuevas tecnologías, nuevos lenguajes,
- c) Intentos de modelizar el cerebro biológico
 - 1. Redes Neuronales (intentan modelizar el "procesador")
 - 2. Conjuntos y Lógica Difusa (representar y manejar la información)

En este texto veremos aspectos relativos a a) y c.1).

Consideremos el problema general, bajo un planteamiento de Programación.

Desde un punto de vista global, un programa (traducido a lenguaje máquina) se puede ver como una sucesión de ceros y unos, cuya ejecución produce una salida, que es otra serie de ceros y unos. Si añadimos un 1 al inicio de cada cadena binaria (programa y salida), podemos entender los programas como aplicaciones concretas de una función entre números naturales en binario. El argumento (variable independiente) sería el programa y la función (var. dependiente) la salida del programa.

Obviamente, el número de funciones posibles de N en N es **nonumerable**, mientras que el número de posibles programas que podemos escribir con un Lenguaje de Programación, que tiene un número finito de símbolos, es **numerable**.

Esto significa (cada programa puede calcular una sola función como las indicadas antes) que existen muchas funciones para las que no podemos escribir un programa en nuestro L. de Alto Nivel, aunque seguramente estamos interesados en resolver el problema asociado a la función.

Entonces nos preguntamos cosas como:

¿Para qué problemas no podemos escribir un programa ?

¿Podremos resolver algunos de estos problemas con otro lenguaje de programación y/o con otros computadores ?.

Además, para los problemas que si tienen un programa asociado,

¿Se podrá ejecutar el programa en un ordenador actual en un tiempo razonable ? (p.e., antes de que llegue nuestra jubilación).

¿Los ordenadores futuros podrán hacerlo ?

Trataremos de dar respuestas a algunas de estas cuestiones, a lo largo del desarrollo de la asignatura.

1.1. Introducción Histórica

Uno de los principales factores determinantes de la profunda revolución experimentada en el ámbito de la ciencia, la técnica y la cultura de nuestros días es el desarrollo de la informática. La palabra ‘informática’ (**Información automática**), es un nombre colectivo que designa un vasto conjunto de teorías y técnicas científicas -desde la matemática abstracta hasta la ingeniería y la gestión administrativa- cuyo objeto es el diseño y el uso de los ordenadores. Pero el núcleo teórico más sólido y fundamental de todo ese conjunto de doctrinas y prácticas es la llamada ‘Teoría de la Computabilidad’, formalmente elaborada en los años 30 y 40 gracias a los descubrimientos de lógicos matemáticos como Gödel, Turing, Post, Church, y Kleene, aunque sus orígenes más remotos datan de antiguo, con el planteamiento de la cuestión de saber si, al cabo de cierto esfuerzo, el hombre podría llegar a un extremo en la investigación en que, eventualmente, toda clase de problemas pudiera ser atacado por un procedimiento general de forma que no requiriera el más leve esfuerzo de imaginación creadora para llevarlo a cabo. Si todo queda determinado así en detalle, entonces sería obviamente posible abandonar la ejecución del método a una máquina, máxime si la máquina en cuestión es totalmente automática. Esta idea, ambiciosa sin duda, ha influido poderosamente en diferentes épocas el desarrollo de la ciencia.

El propósito inicial es hacer precisa la noción intuitiva de función calculable; esto es, una función cuyos valores pueden ser calculados de forma automática o efectiva mediante un algoritmo, y construir modelos teóricos para ello (de computación). Así podemos obtener una comprensión más clara de esta idea intuitiva; y solo de esta forma podemos explorar matemáticamente el concepto de computabilidad y los conceptos relacionadas con ella, tales como decibilidad, etc...

La teoría de la computabilidad puede caracterizarse, desde el punto de vista de las C.C., como la búsqueda de respuestas para las siguientes preguntas:

1) ¿Qué pueden hacer los ordenadores (sin restricciones de ningún tipo)?

2) ¿Cuales son las limitaciones inherentes a los métodos automáticos de cálculo?.

El primer paso en la búsqueda de las respuestas a estas preguntas está en el estudio de los modelos de computación.

Los comienzos de la Teoría. La Tesis de Church-Turing

Los modelos abstractos de computación tienen su origen en los años 30, bastante antes de que existieran los ordenadores modernos, en el trabajo de los lógicos Church, Gödel, Kleene, Post, y Turing. Estos primeros trabajos han tenido una profunda influencia no solo en el desarrollo teórico de las C.C., sino que muchos aspectos de la práctica de la computación que son ahora lugar común de los informáticos, fueron presagiados por ellos; incluyendo la existencia de ordenadores de propósito general, la posibilidad de interpretar programas, la dualidad entre software y hardware, y la representación de lenguajes por estructuras formales basados en reglas de producción.

El punto de partida de estos primeros trabajos fueron las cuestiones fundamentales que D. Hilbert formuló en 1928, durante el transcurso de un congreso internacional:

1.- ¿Son completas las matemáticas, en el sentido de que pueda probarse o no cada aseveración matemática?

2.- ¿Son las matemáticas consistentes, en el sentido de que no pueda probarse simultáneamente una aseveración y su negación ?

3.- ¿Son las matemáticas decidibles, en el sentido de que exista un método definido que se pueda aplicar a cualquier aseveración matemática, y que determine si dicha aseveración es cierta?.

La meta de Hilbert era crear un sistema matemático formal "completo y consistente", en el que todas las aseveraciones pudieran plantearse con precisión. Su idea era encontrar un algoritmo que determinara la verdad o falsedad de cualquier proposición en el sistema formal. A este problema le llamó el 'Entscheidungsproblem'.

Por desgracia para Hilbert, en la década de 1930 se produjeron una serie de investigaciones que mostraron que esto no era posible. Las primeras noticias en contra surgen en 1931 con K. Gödel y su Teorema de Incompletitud: "Todo sistema de primer orden consistente que contenga los teoremas de la aritmética y cuyo conjunto de (números de Gödel de) axiomas sea recursivo no es completo."

Como consecuencia no será posible encontrar el sistema formal deseado por Hilbert en el marco de la lógica de primer orden, a no ser que se tome un conjunto no recursivo de axiomas, hecho que escapaba a la mente de los matemáticos. Una versión posterior y más general del teorema de Gödel elimina la posibilidad de considerar sistemas deductivos más potentes que los sistemas de primer orden, demostrando que no pueden ser consistentes y completos a la vez.

Un aspecto a destacar dentro del teorema de incompletitud de Gödel, fué la idea de codificación. Se indica un método (numeración de Gödel) mediante el cual se asigna un número de código (entero positivo) a cada fórmula bien formada del sistema (fbf) y a cada sucesión finita de fórmulas bien formadas, de tal modo que la fbf o sucesión finita de fbf se recupera fácilmente a partir de su número de código. A través de este código, los enunciados referentes a enteros positivos, pueden considerarse como enunciados referentes a números de código de expresiones, o

incluso referentes a las propias expresiones. Esta misma idea fué posteriormente utilizada para codificar algoritmos como enteros positivos, y así poder considerar un algoritmo, cuyas entradas fuesen enteros positivos, como un algoritmo cuyas entradas fuesen algoritmos.

El siguiente paso importante lo constituye la aparición casi simultanea en 1936 de varias caracterizaciones independientes de la noción de calculabilidad efectiva, en los trabajos de Church, Kleene, Turing y Post. Los tres primeros mostraban problemas que eran efectivamente indecidibles; Church y Turing probaron además que el Entscheidungsproblem era un problema indecidible.

Church propuso la noción de función λ -definible como función efectivamente calculable. La demostración de teoremas se convierte en una transformación de una cadena de símbolos en otra, en cálculo lambda, según un conjunto de reglas formales. Este sistema resultó ser inconsistente, pero la capacidad para expresar-calcular funciones numéricas como términos del sistema llamó pronto la atención de él y sus colaboradores.

Gödel había recogido la idea de Herbrand de que una función f podría definirse por un conjunto de ecuaciones entre términos que incluían a la función f y a símbolos para funciones previamente definidas, y precisó esta idea requiriendo que cada valor de f se obtenga de las ecuaciones por sustitución de las variables por números y los términos libres de variables por los valores que ya se habían probado que designaban. Esto define la clase de ‘las funciones recursivas de Herbrand-Gödel’.

En 1936, Church hace un esquema de la demostración de la equivalencia entre las funciones λ -definibles y las funciones recursivas de Herbrand-Gödel (esta equivalencia también había sido probada por Kleene); y aventura que estas iban a ser las únicas funciones calculables por medio de un algoritmo a través de la tesis que lleva su nombre, y utilizando la noción de función λ -definible, dió ejemplos de problemas de decisión irresolubles, y demostró que el Entscheidungsproblem era uno de esos problemas.

Por otra parte Kleene, pocos meses después, demuestra formalmente la equivalencia entre funciones λ -definible y funciones recursivas de Herbrand-Gödel, y dá ejemplos de problemas irresolubles utilizando la noción de función recursiva.

La tercera noción de función calculable proviene del matemático inglés A. Turing, quién argumentó que la tercera cuestión de Hilbert (el Entscheidungsproblem) podía atacarse con la ayuda de una máquina, al menos con el concepto abstracto de máquina.

Turing señaló que había tenido éxito en caracterizar de un modo matemáticamente preciso, por medio de sus máquinas, la clase de las funciones calculables mediante un algoritmo, lo que se conoce hoy como *Tesis de Turing*.

Aunque no se puede dar ninguna prueba formal de que una máquina pueda tener esa propiedad, Turing dió un elevado número de argumentos a su favor, en base a lo cual presentó la tesis como un teorema demostrado. Además, utilizó su concepto de máquina para demostrar que existen funciones que no son calculables por un método definido y en particular, que el Entscheidungsproblem era uno de esos problemas.

Cuando Turing conoció los trabajos de Church-Kleene, demostró que los conceptos de función λ -definible y función calculable por medio de una máquina de Turing coinciden. Naturalmente a la luz de esto la Tesis de Turing resulta ser equivalente a la de Church.

Finalmente, cabe reseñar el trabajo de E. Post. Este estaba interesado en marcar la frontera entre lo que se puede hacer en matemáticas simplemente por procedimientos formales y lo que depende de la comprensión y el entendimiento. De esta forma, Post formula un modelo de procedimiento efectivo a través de los llamados sistemas deductivos normales. Estos son sistemas puramente formales en los que puede ‘deducirse’ sucesiones finitas de símbolos como consecuencia de otras sucesiones finitas de símbolos por medio de un tipo normalizado de reglas y a partir de un conjunto de axiomas. Así pues, dada una sucesión finita de símbolos como entrada, las reglas permiten convertirla en una sucesión finita de salida. En su artículo, Post demostró resultados de incompletitud e indecibilidad en estos sistemas.

Los resultados hasta ahora citados, se refieren a funciones totales. La existencia de algoritmos que con determinadas entradas nunca terminan, condujo de forma natural a considerar funciones parciales. Kleene fué el primero en hacer tal consideración en 1938. El estudio de estas funciones ha mostrado la posibilidad de generalizar todos los resultados anteriores a funciones parciales. Por otro lado, el estudio de las funciones parciales calculables ha resultado esencial para el posterior desarrollo de la materia.

Posteriormente, se demostró la equivalencia entre lo que se podía calcular mediante una máquina de Turing y lo que se podía calcular mediante un sistema formal en general.

A la vista de estos resultados, la Tesis de Church-Turing es aceptada como un axioma en la teoría de la computación, y ha servido como punto de partida en la investigación de los problemas que se pueden resolver mediante un algoritmo.

Problemas no computables

Usando la codificación de Gödel, se demostró que era posible construir una máquina de propósito general, es decir, capaz de resolver cualquier problema que se pudiese resolver mediante un algoritmo. Dicha máquina tendría como entrada el entero que codificaría el algoritmo solución del problema y la propia entrada del problema, de tal forma, que la máquina aplicaría el algoritmo codificado a la entrada del problema. Esta hipotética máquina puede considerarse como el padre de los actuales ordenadores de propósito general.

Una de las cuestiones más estudiadas en la teoría de la computabilidad ha sido la posibilidad de construir algoritmos que nos determinen si un determinado algoritmo posee o no una determinada propiedad. Así, sería interesante responder de forma automática a cuestiones como:

- ¿Calculan los algoritmos A y B la misma función? (Problema de la equivalencia)
 - ¿Parará el algoritmo A para una de sus entradas? (Problema de la parada)
 - ¿Parará el algoritmo A para todas sus entradas? (Problema de la totalidad)
 - ¿Calcula el algoritmo A la función f? (Problema de la verificación?)
- etc ...

En un principio se fueron obteniendo demostraciones individuales de la no computabilidad

de cada una de estas cuestiones, de forma que se tenía la sensación de que casi cualquier pregunta interesante acerca de algoritmos era no computable.

A pesar de esto, y como consecuencia de la existencia de un programa universal hay otras muchas cuestiones interesantes que se han demostrado computables.

El identificar los problemas que son computables y los que no lo son tiene un considerable interés, pues indica el alcance y los límites de la computabilidad, y así demuestra los límites teóricos de los ordenadores. Además de las cuestiones sobre algoritmos, se han encontrado numerosos problemas menos "generales" que han resultado ser no computables. Como ejemplo citamos:

- Décimo problema de Hilbert. Una ecuación diofántica es la ecuación de los ceros enteros de un polinomio con coeficientes enteros. Se pregunta si hay un procedimiento efectivo que determine si una ecuación diofántica tiene o no solución.

Por otro lado, son muchos los problemas interesantes que se han demostrado computables. Todas las funciones construidas por recursividad primitiva o minimalización a partir de funciones calculables resultan ser calculables como consecuencia de los trabajos de Church y Turing. Pero además, otras funciones más complejamente definidas también son computables. Como ejemplo más interesante de aplicación de este tipo de recursión tenemos la función de Ackermann ψ :

$$\begin{aligned}\psi(0, y) &= y + 1; \\ \psi(x + 1, 0) &= \psi(x, 1); \\ \psi(x + 1, y + 1) &= \psi(x, \psi(x + 1, y)).\end{aligned}$$

1.2. Diferentes Modelos de Computación

Consideraremos las Ciencias de la Computación como un cuerpo de conocimiento cuyo principal objetivo es la resolución de problemas por medio de un ordenador. Podemos citar las siguientes definiciones:

a) La ACM (Asociation Computing Machinering):

'la disciplina Ciencias de la Computación es el estudio sistemático de los procesos algorítmicos que describen y transforman información: teoría, análisis, diseño, eficiencia, implementación, y aplicación.'

b) Norman E. Gibbs y Allen B. Tucker (1986) indican que: 'no debemos entender que el objetivo de las Ciencias de la Computación sea la construcción de programas sino el estudio sistemático de los algoritmos y estructura de datos, específicamente de sus propiedades formales'.

Para ser más concretos (A. Berztiss 1987), vamos a considerar a las C.C. como un cuerpo de conocimiento cuyo objetivo es obtener respuestas para las siguientes cuestiones:

- A) ¿Qué problemas se pueden resolver mediante un ordenador?.
- B) ¿Cómo puede construirse un programa para resolver un problema?.
- C) ¿Resuelve realmente nuestro programa el problema?.
- D) ¿Cuanto tiempo y espacio consume nuestro problema?.

Analizando en profundidad los 4 puntos anteriores llegaremos a descubrir explícitamente los diferentes contenidos abarcados por las C.C.

El planteamiento de la primera cuestión nos conduce a precisar el concepto de problema y de lo que un ordenador es capaz de realizar.

Durante muchos años se creyó que si un problema podía enunciarse de manera precisa, entonces con suficiente esfuerzo y tiempo sería posible encontrar un ‘algoritmo’ o método para encontrar una solución (o tal vez podría proporcionarse una prueba de que tal solución no existe). En otras palabras, se creía que no había problema que fuera tan intrínsecamente difícil que en principio nunca pudiera resolverse.

Uno de los grandes promotores de esta creencia fué el matemático David Hilbert (1862-1943), quien en un congreso mundial afirmó:

"Todo problema matemático bien definido debe ser necesariamente susceptible de un planteamiento exacto, ya sea en forma de una respuesta real a la pregunta planteada o debido a la constatación de la imposibilidad de resolverlo, a lo que se debería el necesario fallo de todos los intentos... "

El principal obstáculo que los matemáticos de principios de siglo encontraban al plantearse estas cuestiones era concretar con exactitud lo que significa la palabra algoritmo como sinónimo de método para encontrar una solución. La noción de algoritmo era intuitiva y no matemáticamente precisa.

Las descripciones dadas por los primeros investigadores tomaron diferentes formas, que pueden clasificarse ampliamente del siguiente modo:

- (a) máquinas computadoras abstractas (definidas de modo preciso),
- (b) construcciones formales de procedimientos de cómputo, y
- (c) construcciones formales productoras de clases de funciones.

Las dos primeras caracterizaciones se refieren a la propia noción de algoritmo (en principio no hay gran diferencia entre ambas). La última da descripciones de la clase de funciones computables mediante un algoritmo.

Ejemplos de (a) son los Autómatas y las *máquinas de Turing*, (diseñadas por Turing en los años 30). Un ejemplo de (b) son los *sistemas de Thue*. Por último, las *funciones recursivas* constituyen el ejemplo clásico de (c).

El resultado crucial es que las diversas caracterizaciones de las funciones (parciales) computables mediante un algoritmo condujeron todas a una misma clase, a saber, la clase de las funciones parciales recursivas. Esto es algo susceptible de demostración, y que ha sido demostrado. Lo que no es susceptible de demostración es que la clase de las funciones parciales recursivas coincida con la clase de las funciones computables mediante un algoritmo. No obstante, a la luz

de las evidencias a favor y de la falta de evidencias en contra, aceptamos la *Tesis de Church* que afirma la equivalencia de ambas clases.

Se clasifican los problemas según que siempre sea posible encontrar la solución por medio de un algoritmo (problemas computables) ó que no existan algoritmos que siempre produzcan una solución (problemas no computables).

Surge de modo inmediato la cuestión B) de como diseñar un programa (algoritmo especificado para poder ser ejecutado por un ordenador) que resuelva un problema dado. En la primera época del desarrollo informático los programas dependían intrínsecamente del ordenador utilizado, pues se expresaban en lenguaje máquina, directamente interpretable por el ordenador.

Surgió entonces la necesidad de idear otros mecanismos para construir y expresar los programas. El hilo conductor de tales mecanismos fué la abstracción: separar el programa del ordenador y acercarlo cada vez más al problema.

Los subprogramas empezaron ya a usarse a principios de los 50, dando lugar posteriormente al primer tipo de abstracción, la procedimental. A principios de los 60, se empezaron a entender los conceptos abstractos asociados a estructuras de datos básicas pero aún no se separaban los conceptos de las implementaciones. Con el nacimiento en esta época de los primeros lenguajes de alto nivel, Fortran p.ej., se llegó a la abstracción sintáctica, al abstraerse la semántica de las expresiones matemáticas y encapsular el acceso a ellas a través de la sintaxis propia del lenguaje. En cualquier caso con el desarrollo de estos lenguajes de alto nivel se solventaron los problemas de flexibilidad en la comunicación con el ordenador, y se empezaron a estudiar los algoritmos de forma independiente del ordenador concreto en que se probaban y del lenguaje concreto en que se expresaran.

Aparece la necesidad de traducir los programas escritos en lenguajes de alto nivel al lenguaje máquina, de forma automática, y se buscan máquinas o procedimientos que puedan reconocer el léxico y la sintaxis de dichos lenguajes.

Hay que comentar que no hay un algoritmo para enseñar a diseñar algoritmos, y que muchas veces el proceso de construcción puede llegar a ser muy poco disciplinado. No obstante, existen técnicas de diseño de algoritmos, que vienen a ser modelos abstractos de los mismos aplicables a gran variedad de problemas reales.

Una vez construido un programa para un problema, surge la cuestión C) de si lo resuelve realmente. Normalmente los programadores prueban sus programas sobre una gran cantidad de datos de entrada para descubrir la mayoría de los errores lógicos presentes, aunque con este método (al que suele denominarse de prueba y depuración) no se puede estar completamente seguro de que el programa no contiene errores. Necesitaríamos para realizar la verificación formal, reglas que describan de forma precisa el efecto que cada instrucción tiene en el estado actual del programa, para, aplicando dichas reglas demostrar rigurosamente que lo que hace el programa coincide con sus especificaciones. En cualquier caso y cuando la prueba formal resulte muy complicada, podemos aumentar la confianza en nuestro programa realizando en el mismo los "testcuidadosos de que hablábamos al principio.

Alcanzado este punto, ya tenemos un programa que en principio es solución de un problema. Se plantea entonces la duda de que hacer en caso de que para el mismo problema seamos capaces de construir otro programa que también lo resuelva. ¿Cómo decidirnos por una u otra solución? o más aún, ¿qué ocurre si el programa aún siendo correcto consume demasiados recursos y es inaceptable?. La respuesta viene dada a través del punto D) en nuestro recorrido: el análisis del tiempo y espacio que necesita una solución concreta; en definitiva, el estudio de la eficiencia de los programas, midiendo la complejidad en espacio, por el número de variables y el número y tamaño de las estructuras de datos que se usan, y la complejidad en tiempo por el número de acciones elementales llevadas a cabo en la ejecución del programa.

Los problemas computables fueron entonces clasificados en dos tipos: problemas eficientemente computables, para los que existía un algoritmo eficiente; y problemas intratables, para los que no existen algoritmos eficientes. La existencia de problemas intratables no ha sido probada, si bien se han encontrado muchas evidencias a su favor.

Otra clase de problemas a considerar es la clase NP de los problemas para los que existía un algoritmo no determinístico en tiempo polinomial, y dentro de ella, los problemas NP-completos.

Los intentos (desde los años 40) de construir máquinas para modelizar algunas de las funciones del cerebro biológico, ha permitido desarrollar máquinas capaces de 'aprender' (y reproducir) funciones (o sistemas) cuya forma (o comportamiento) se desconoce, pero sí conocemos una serie de ejemplos que reflejan esta forma (o comportamiento). Estas máquinas llamadas Redes Neuronales Artificiales también aportan su granito de arena al desarrollo de la computación.

A menudo se utiliza la técnica de reducir un problema a otro para comprobar si tiene o no solución efectiva. La estrategia en el caso de la respuesta negativa es la siguiente, si se reduce de forma efectiva un problema sin solución efectiva a otro problema (mediante una función calculable), entonces este nuevo problema tampoco tendrá solución efectiva. La razón es muy simple, si tuviese solución efectiva, componiendo el algoritmo solución con el algoritmo de transformación obtendríamos una solución para el problema efectivamente irresoluble. En sentido inverso, si se reduce un problema a otro para el que se conoce una solución efectiva, entonces componiendo se obtiene una solución para el primer problema. Esta técnica es muy útil y se utiliza a menudo. Por otro lado, esta misma técnica es muy empleada en el campo de la complejidad algorítmica.

La Complejidad Algorítmica trata de estudiar la relativa dificultad computacional de las funciones computables. Rabin (1960) fué de los primeros en plantear la cuestión ¿Qué quiere decir que f sea más difícil de computar que g ?

J. Hartmanis and R.E. Stearns, en *On the computational complexity of algorithms* (1965) introducen la noción fundamental de medida de complejidad definida como el tiempo de computación sobre una máquina de Turing multicinta.

Después surge la definición de funciones computables en tiempo polinomial, y se establece una jerarquía de complejidad, los problemas NP, NP-duros y NP-completos. De todas formas, el

perfil de la plaza no se ocupa directamente de este problema que además se estudia ampliamente en otras materias y asignaturas del currículum de Informática.

1.2.1. Autómatas y Lenguajes

El desarrollo de los ordenadores en la década de los 40, con la introducción de los programas en la memoria principal, y posteriormente con los lenguajes de programación de alto nivel, propician la distinción entre lenguajes formales, con reglas sintácticas y semánticas rígidas, concretas y bien definidas, de los lenguajes naturales como el inglés, donde la sintaxis y la semántica no se pueden controlar fácilmente. Los intentos de formalizar los lenguajes naturales, lleva a la construcción de gramáticas, como una forma de describir estos lenguajes, utilizando para ello reglas de producción para construir las frases del lenguaje. Se puede entonces caracterizar un Lenguaje, mediante las reglas de una gramática adecuada.

Los trabajos de McCulloch y Pitts (1943) describen los cálculos lógicos inmersos en un dispositivo (neurona artificial) que habían diseñado para simular la actividad de una neurona biológica. El dispositivo recibía o no, una serie de impulsos eléctricos por sus entradas que se ponderaban, y producía una salida binaria (existe pulso eléctrico o no). Las entradas y salidas se podían considerar como cadenas de 0 y 1, indicando entonces la forma de combinar la cadena de entrada para producir la salida. La notación utilizada es la base para el desarrollo de expresiones regulares en la descripción de conjuntos de cadenas de caracteres.

C. Shannon (1948) define los fundamentos de la teoría de la información, y utiliza esquemas para poder definir sistemas discretos, parecidos a los autómatas finitos, relacionándolos con cadenas de Markov, para realizar aproximaciones a los lenguajes naturales.

J. Von Neumann (1948) introduce el término de teoría de autómatas, y dice sobre los trabajos de McCulloch-Pitts: “*.. el resultado más importante de McCulloch-Pitts, es que cualquier funcionamiento en este sentido, que pueda ser definido en todo, lógicamente, estrictamente y sin ambigüedad, en un número finito de palabras, puede ser realizado también por una tal red neuronal formal*”

La necesidad de traducir los algoritmos escritos en lenguajes de alto nivel al lenguaje máquina, propicia la utilización de máquinas como los autómatas de estados finitos, para reconocer si una cadena determinada pertenece (es una frase de) a un lenguaje concreto, usando para ello la función de transición de estados, mediante un diagrama de transición o una tabla adecuada. Tenemos así otra forma de caracterizar los lenguajes, de acuerdo con máquinas automáticas que permitan reconocer sus frases.

S.C. Kleene, en 1951, realiza un informe (solicitado por la RAND Corporation) sobre los trabajos de McCulloch-Pitts, que se publica en 1956. En este informe, Kleene demuestra la equivalencia entre lo que él llama "dos formas de definir una misma cosa", que son los *sucesos regulares* (que se pueden describir a partir de sucesos bases y los operadores unión, concatenación e iteración (*)), es decir, expresiones regulares, y sucesos especificados por un autómata

finito.

Rabin y Scott (1960) obtienen un modelo de computador con una cantidad finita de memoria, al que llamaron autómata de estados finitos. Demostraron que su comportamiento posible, era básicamente el mismo que el descrito mediante expresiones regulares, desarrolladas a partir de los trabajos de McCulloch y Pitts.

No obstante lo dicho, para un alfabeto concreto, no todos los lenguajes que se pueden construir son regulares. Ni siquiera todos los interesantes desde el punto de vista de la construcción de algoritmos para resolver problemas. Hay entonces muchos problemas que no son calculables con estos lenguajes. Esto pone de manifiesto las limitaciones de los autómatas finitos y las gramáticas regulares, y propicia el desarrollo de máquinas reconocedoras de otros tipos de lenguajes y de las gramáticas correspondientes, asociadas a los mismos.

En 1956, la Princeton Univ. Press publica el libro *Automata Studies*, editado por C. Shannon y J. McCarthy, donde se recogen una serie de trabajos sobre autómatas y lenguajes formales. D. A. Huffman (1954) ya utiliza conceptos como *estado de un autómata* y *tabla de transiciones*.

N. Chomsky (1956) propone tres modelos para la descripción de lenguajes, que son la base de su futura jerarquía de los tipos de lenguajes, que ayudó también en el desarrollo de los lenguajes de programación. Para ello intentó utilizar autómatas para extraer estructuras sintácticas ("... *el inglés no es un lenguaje de estados finitos*") y dirige sus estudios a las gramáticas, indicando que la diferencia esencial entre autómatas y gramáticas es que la lógica asociada a los autómatas (p.e., para ver la equivalencia entre dos de ellos) es Decidible, mientras que la asociada a las gramáticas no lo es.

Desarrolla el concepto de gramática libre del contexto, en el transcurso de sus investigaciones sobre la sintaxis de los lenguajes naturales.

Backus y Naur desarrollaron una notación formal para describir la sintaxis de algunos lenguajes de programación, que básicamente se sigue utilizando todavía, y que podía considerarse equivalente a las gramáticas libres del contexto.

Consideramos entonces los lenguajes libres (independientes) del contexto, y las gramáticas libres del contexto y los autómatas con pila, como forma de caracterizarlos y manejarlos. Los distintos lenguajes formales que se pueden construir sobre un alfabeto concreto pueden clasificarse en clases cada vez más amplias que incluyen como subconjunto a las anteriores, de acuerdo con la jerarquía establecida por Chomsky en los años 50.

Se puede llegar así, de una forma casi natural a considerar las máquinas de Turing, establecidas casi 20 años antes, como máquinas reconocedoras de los lenguajes formales dependientes del contexto o estructurados por frases, e incluso a interpretar la Tesis de Turing como que un sistema computacional nunca podrá efectuar un análisis sintáctico de aquellos lenguajes que están por encima de los lenguajes estructurados por frases, según la jerarquía de Chomsky".

En consecuencia, podemos utilizar la teoría de autómatas y los conceptos relativos a gramáticas sobre distintos tipos de lenguajes, para decidir (si se puede) si una función (o problema) es calculable, en base a que podamos construir un algoritmo solución mediante un lenguaje que

puede ser analizado mediante alguna máquina de las citadas anteriormente.

Los temas sobre autómatas, computabilidad, e incluso la complejidad algorítmica fueron incorporándose a los currículum de ciencias de la computación de diferentes universidades, mediada la década de los 60. Esta incorporación puso de manifiesto que las ciencias de la computación habían usado gran cantidad de ideas de muy diferentes campos para su desarrollo, y que la investigación sobre aspectos básicos podía cooperar y aumentar los avances de la computación.

1.3. Lenguajes y Gramáticas. Aspectos de su traducción

Una idea básica en programación es la enorme diferencia entre los lenguajes naturales (LN) y los lenguajes de programación (LP), fundamentalmente porque los LP tienen unas reglas de sintaxis y de semántica mucho más rígidas, lo que les hace manejables en los computadores.

En los LN, las reglas gramaticales se desarrollan para reglamentar "de alguna forma la propia evolución del lenguaje. Se trata pues de "explicar" la estructura del lenguaje, y no delimitarla. Esto obliga a reglas muy complejas y que se quedan obsoletas rápidamente.

Los lenguajes más formalizados (LF) como los lenguajes de programación o el lenguaje matemático, tienen unas estructuras claramente definidas y determinadas por sus reglas gramaticales (sintácticas y semánticas). Esto ha posibilitado y propiciado la construcción de traductores automáticos para estos lenguajes.

En el proceso de traducción que realizan los compiladores, la primera tarea que se realiza, es la identificación de *tokens* como bloques u objetos individuales contenidos en el "diccionario" del lenguaje (p.e., identificadores, palabras clave, operadores,). Esto lo realiza un módulo del compilador que es el **analizador de léxico**, que transforma el programa fuente, en una secuencia de tokens representando cada uno un solo objeto.

El siguiente paso, realizado por el **analizador sintáctico**, es identificar los tokens que forman parte de cada instrucción y ver que estas estén correctamente escritas. El tercer paso es generar el código con el **generador de código**.

Los dos primeros pasos requieren imperiosamente unas reglas gramaticales claramente definidas, en las que apoyarse para la automatización del proceso de traducción. Generalmente se utilizan en estas tareas, máquinas (o algoritmos) como los autómatas, que estudiaremos más adelante.

Vamos a precisar previamente, los conceptos básicos relativos a Lenguajes y Gramáticas formales.

1.3.1. Alfabetos y Palabras

Los dispositivos que vamos a estudiar trabajan con símbolos y cadenas (o palabras) fundamentalmente. En este apartado vamos a definir de forma precisa estos conceptos.

Definición 1 Un alfabeto es un conjunto finito A . Sus elementos se llamarán símbolos o letras.

Para notar los alfabetos, en general, usaremos siempre que sea posible las primeras letras en mayúsculas: A, B, C, \dots . Para los símbolos trataremos de emplear las primeras letras en minúsculas: a, b, c, \dots o números.

Las siguientes son algunas normas de notación que respetaremos en lo posible a lo largo del curso. En casos particulares en los que sea imposible seguirlas lo indicaremos explicitamente.

Ejemplo 1 $A = \{0, 1\}$ es un alfabeto con símbolos 0 y 1.

También es un alfabeto $B = \{<0, 0>, <0, 1>, <1, 0>, <1, 1>\}$ con símbolos $<0, 0>$, $<0, 1>$, $<1, 0>$ y $<1, 1>$. En este caso no hay que confundir los símbolos del alfabeto B con los símbolos del lenguaje (o más precisamente meta-lenguaje) que usamos para expresarnos todos los días. Son dos cosas totalmente distintas. Nosotros para comunicarnos usamos un lenguaje que tiene unos símbolos que son las letras del alfabeto latino, los números, las letras del alfabeto griego, y una serie de símbolos especiales propios del lenguaje matemático (el meta-lenguaje). Con este lenguaje también definimos los conceptos de alfabeto y símbolo. Ahora bien, los símbolos de un lenguaje no tienen que ser algunos de los usados en el metalenguaje, sino que cada uno puede estar formado por 0, 1 o más símbolos del metalenguaje. De hecho como en este último caso (lenguaje B), un símbolo del alfabeto definido está formado por varios símbolos del metalenguaje. Para que esto no de lugar a confusión, siempre que ocurra una situación similar, encerraremos los símbolos entre ángulos $< \dots >$. Esto indicará que todo lo que aparece es un único símbolo.

Definición 2 Una palabra sobre el alfabeto A es una sucesión finita de elementos de A . Es decir u es una palabra sobre A , si y solo si $u = a_1 \dots a_n$ donde $a_i \in A$, $\forall i = 1, \dots, n$.

Por ejemplo, si $A = \{0, 1\}$ entonces 0111 es una palabra sobre este alfabeto.

El conjunto de todas las palabras sobre un alfabeto A se nota como A^* .

Para las palabras usaremos, en lo posible, las últimas letras del alfabeto latino en minúsculas: u, v, x, y, z, \dots

Definición 3 Si $u \in A^*$, entonces la longitud de la palabra u es el número de símbolos de A que contiene. La longitud de u se nota como $|u|$. Es decir si $u = a_1 \dots a_n$, entonces $|u| = n$.

Definición 4 La palabra vacía es la palabra de longitud cero. Es la misma para todos los alfabetos, y se nota como ϵ .

El conjunto de cadenas sobre un alfabeto A excluyendo la cadena vacía se nota como A^+ . Usaremos indistintamente palabra o cadena. Si no hay confusión, la palabra formada por un solo símbolo se representa por el propio símbolo.

La operación fundamental en el conjunto de las cadenas A^* es la concatenación.

Definición 5 Si $u, v \in A^*$, $u = a_1 \dots a_n$, $v = b_1 \dots b_m$, se llama concatenación de u y v a la cadena $u.v$ (o simplemente uv) dada por $a_1 \dots a_n b_1 \dots b_m$.

La concatenación tiene las siguientes propiedades:

1. $|u.v| = |u| + |v|, \forall u, v \in A^*$
2. Asociativa.- $u.(v.w) = (u.v).w, \forall u, v, w \in A^*$
3. Elemento Neutro.- $u.\epsilon = \epsilon.u = u, \forall u \in A^*$

Los propiedades asociativa y elemento neutro dotan al conjunto de las cadenas con la operación de concatenación de la estructura de monoide. La propiedad conmutativa no se verifica.

Consideramos la iteración n-sima de una cadena como la concatenación con ella misma n veces, y se define de forma recursiva:

Definición 6 Si $u \in A^*$ entonces

- $u^0 = \epsilon$
- $u^{i+1} = u^i.u, \forall i \geq 0$

Definición 7 Si $u = a_1 \dots a_n \in A^*$, entonces la cadena inversa de u es la cadena $u^{-1} = a_n \dots a_1 \in A^*$.

1.3.2. Lenguajes

Definición 8 Un lenguaje sobre el alfabeto A es un subconjunto del conjunto de las cadenas sobre A : $L \subseteq A^*$.

Los lenguajes los notaremos con las letras intermedias del alfabeto latino en mayúsculas.

Ejemplo 2 Los siguientes son lenguajes sobre un alfabeto A , cuyo contenido asumimos conocer claramente:

- $L_1 = \{a, b, \epsilon\}$, símbolos a , b y la cadena vacía
- $L_2 = \{a^i b^i \mid i = 0, 1, 2, \dots\}$, palabras formadas de una sucesión de símbolos a , seguida de la misma cantidad de símbolos b .
- $L_3 = \{uu^{-1} \mid u \in A^*\}$, palabras formadas con símbolos del alfabeto A y que consisten de una palabra, seguida de la misma palabra escrita en orden inverso.

- $L_4 = \{a^{n^2} \mid n = 1, 2, 3, \dots\}$, palabras que tienen un número de símbolos a que sea cuadrado perfecto, pero nunca nulo.

A parte de las operaciones de unión e intersección de lenguajes, dada su condición de conjuntos existe la operación de concatenación.

Definición 9 If L_1, L_2 son dos lenguajes sobre el alfabeto A, la concatenación de estos dos lenguajes es el que se obtiene de acuerdo con la siguiente expresión,

$$L_1 L_2 = \{u_1 u_2 \mid u_1 \in L_1, u_2 \in L_2\}$$

Ejemplo 3 Si $L_1 = \{0^i 1^i : i \geq 0\}$, $L_2 = \{1^i 0^i : i \geq 0\}$ entonces,

$$L_1 L_2 = \{0^i 1^i 1^j 0^j : i, j \geq 0\}$$

Propiedades:

- $L\emptyset = \emptyset L = \emptyset$ (\emptyset es el Lenguaje que contiene 0 palabras)
- Elemento Neutro.- $\{\epsilon\}L = L\{\epsilon\} = L$
- Asociativa.- $L_1(L_2L_3) = (L_1L_2)L_3$

La iteración de lenguajes se define como en las palabras, de forma recursiva:

Definición 10 Si L es un lenguaje sobre el alfabeto A, entonces la iteración de este lenguaje se define de acuerdo con las siguientes expresiones recursivas,

$$L^0 = \{\epsilon\}$$

$$L^{i+1} = L^i L$$

Definición 11 Si L es un lenguaje sobre el alfabeto A, la clausura de Kleene de L es el lenguaje obtenido de acuerdo con la siguiente expresión:

$$L^* = \bigcup_{i \geq 0} L^i$$

Definición 12 Si L es un lenguaje sobre el alfabeto A, entonces L^+ es el lenguaje dado por:

$$L^+ = \bigcup_{i \geq 1} L^i$$

Propiedades:

- $L^+ = L^* \text{ si } \epsilon \in L$
- $L^+ = L^* - \{\epsilon\} \text{ si } \epsilon \notin L$

Ejemplo 4 Si $L = \{0, 01\}$, entonces, $L^* = \text{Conjunto de palabras sobre } \{0, 1\} \text{ en las que un uno va siempre precedido de un cero.}$

Por otra parte, $L^+ = \text{Conjunto de palabras sobre } \{0, 1\} \text{ en las que un uno va siempre precedido de un cero y distintas de la palabra vacía.}$

Definición 13 Si L es un lenguaje, el lenguaje inverso de L es el lenguaje dado por:

$$L^{-1} = \{u \mid u^{-1} \in L\}$$

Definición 14 Si L es un lenguaje sobre el alfabeto A , entonces la cabecera de L es el lenguaje dado por

$$CAB(L) = \{u \mid u \in A^* \text{ y } \exists v \in A^* \text{ tal que } uv \in L\}$$

Es decir, la $CAB(L)$ contiene todas las palabras del lenguaje y aquellas otras que tengan como primeros caracteres, alguna palabra del mismo.

Ejemplo 5 Si $L = \{0^i 1^i : i \geq 0\}$, entonces $CAB(L) = \{0^i 1^j : i \geq j \geq 0\}$.

Definición 15 Si A_1 y A_2 son dos alfabetos, una aplicación

$$h : A_1^* \rightarrow A_2^*$$

se dice que es un homomorfismo si y solo si

$$h(uv) = h(u)h(v)$$

Consecuencias:

- $h(\epsilon) = \epsilon$
- $h(a_1 \dots a_n) = h(a_1) \dots h(a_n)$

Ejemplo 6 Si $A_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $A_2 = \{0, 1\}$
la siguiente aplicación es un homomorfismo

$$\begin{array}{llll} h(0) = 0000, & h(1) = 0001, & h(2) = 0010, & h(3) = 0011 \\ h(4) = 0100, & h(5) = 0101, & h(6) = 0110, & h(7) = 0111 \\ h(8) = 1000 & h(9) = 1001 \end{array}$$

- Ejemplo 7**
- Si A es un alfabeto, la aplicación que transforma cada palabra $u \in A^*$ en su inversa no es un homomorfismo de A^* en A^* , ya que la transformación no se hace símbolo a símbolo.
 - La transformación que a cada palabra sobre $\{0,1\}^*$ le añade 00 al principio y 11 al final no es un homomorfismo. Si 0 se transforma en 00011, entonces 00 se debería de transformar en 0001100011 si fuese un homomorfismo, y en realidad se transforma en 000011.

1.3.3. Gramáticas Generativas

Desde un punto de vista matemático una gramática se define de la siguiente forma:

Definición 16 Una gramática generativa es un cuadrupla (V, T, P, S) en la que

- V es un alfabeto, llamado de variables o símbolos no terminales. Sus elementos se suelen representar con letras mayúsculas.
- T es un alfabeto, llamado de símbolos terminales. Sus elementos se suelen representar con letras minúsculas.
- P es un conjunto de pares (α, β) , llamados reglas de producción, donde $\alpha, \beta \in (V \cup T)^*$ y α contiene, al menos un símbolo de V .

El par (α, β) se suele representar como $\alpha \rightarrow \beta$.

- S es un elemento de V , llamado símbolo de partida.

La razón de notar los elementos del alfabeto V con letras mayúsculas es para no confundirlos con los símbolos terminales. Las cadenas del alfabeto $(V \cup T)$ se notan con letras griegas para no confundirlas con las cadenas del alfabeto T , que seguirán notándose como de costumbre: u, v, x, \dots

Ejemplo 8 Sea la gramática (V, T, P, S) dada por los siguientes elementos,

- $V = \{E\}$
- $T = \{+, *, (,), a, b, c\}$

- P está compuesto por las siguientes reglas de producción

$$\begin{aligned} E \rightarrow E + E, \quad E \rightarrow E * E, \quad E \rightarrow (E), \\ E \rightarrow a, \quad E \rightarrow b, \quad E \rightarrow c \end{aligned}$$

- $S = E$

Una gramática se usa para generar las distintas palabras de un determinado lenguaje. Esta generación se hace mediante una aplicación sucesiva de reglas de producción comenzando por el símbolo de partida S . Las siguientes definiciones expresan esta idea de forma más rigurosa.

Definición 17 Dada una gramática $G = (V, T, P, S)$ y dos palabras $\alpha, \beta \in (V \cup T)^*$, decimos que β es derivable a partir de α en un paso ($\alpha \Rightarrow \beta$) si y solo si existe una producción $\gamma \rightarrow \varphi$ tal que

- γ es una subcadena de α .
- β se puede obtener a partir de α , cambiando la subcadena γ por φ .

Ejemplo 9 Haciendo referencia a la gramática del ejemplo anterior, tenemos las siguientes derivaciones,

$$E \Rightarrow E + E \Rightarrow (E) + E \Rightarrow (E) + (E) \Rightarrow (E * E) + (E) \Rightarrow (E * E) + (E * E)$$

Definición 18 Dada una gramática $G = (V, T, P, S)$ y dos palabras $\alpha, \beta \in (V \cup T)^*$, decimos que β es derivable de α ($\alpha \stackrel{*}{\Rightarrow} \beta$), si y solo si existe una sucesión de palabras $\gamma_1, \dots, \gamma_n$ ($n \geq 1$) tales que

$$\alpha = \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_n = \beta$$

Ejemplo 10 En el caso anterior podemos decir que $(E * E) + (E * E)$ es derivable a partir de E :

$$E \stackrel{*}{\Rightarrow} (E * E) + (E * E)$$

Definición 19 Se llama lenguaje generado por una gramática $G = (V, T, P, S)$ al conjunto de cadenas formadas por símbolos terminales y que son derivables a partir del símbolo de partida. Es decir,

$$L(G) = \{u \in T^* \mid S \stackrel{*}{\Rightarrow} u\}$$

Ejemplo 11 En el caso de la gramática de los ejemplos anteriores $(E * E) + (E * E)$ no pertenece al lenguaje generado por G , ya que hay símbolos que no son terminales. Sin embargo, $(a + c) * (a + b)$ si pertenece a $L(G)$, ya que se puede comprobar que es derivable a partir de E (símbolo de partida) y solo tiene símbolos terminales.

Si en una gramática comenzamos a hacer derivaciones a partir del símbolo original S , dicha derivación acabará cuando solo queden símbolos terminales, en cuyo caso la palabra resultante pertenece a $L(G)$, o cuando queden variables pero no se pueda aplicar ninguna regla de producción, en cuyo caso dicha derivación no puede llevar a ninguna palabra de $L(G)$. En general, cuando estemos haciendo una derivación puede haber más de una regla de producción aplicable en cada momento.

Cuando no sea importante distinguir si la derivación de una palabra en una gramática se haya realizado en uno o varios pasos, entonces eliminaremos la $*$ del símbolo de derivación. Así, escribiremos $\alpha \Rightarrow \beta$, en lugar de $\alpha \xrightarrow{*} \beta$.

Ejemplo 12 Sea $G = (V, T, P, S)$ una gramática, donde $V = \{S, A, B\}$, $T = \{a, b\}$, las reglas de producción son

$$\begin{array}{llll} S \rightarrow aB, & S \rightarrow bA, & A \rightarrow a, & A \rightarrow aS, \\ A \rightarrow bAA, & B \rightarrow b, & B \rightarrow bS, & B \rightarrow aBB \end{array}$$

y el símbolo de partida es S .

Esta gramática genera el lenguaje

$$L(G) = \{u \mid u \in \{a, b\}^+ \text{ y } N_a(u) = N_b(u)\}$$

donde $N_a(u)$ y $N_b(u)$ son el número de apariciones de símbolos a y b , en u , respectivamente.

Esto es fácil de ver interpretando que,

- S genera (o produce) palabras con igual número de a que de b .
- A genera palabras con una a de más.
- B produce palabras con una b de más.
- S genera palabras con igual número de a que de b .

Hay que demostrar que todas las palabras del lenguaje tienen el mismo número de a que de b , hay que probar que todas las palabras generadas cumplen esta condición y que todas las palabras que cumplen esta condición son generadas.

Para lo primero basta con considerar el siguiente razonamiento. Supongamos $N_{a+A}(\alpha)$ y $N_{b+B}(\alpha)$ que son el número de a + el número de A en α y el número de b + el número de B en α , respectivamente. Entonces,

- Cuando se empieza a generar una palabra, comenzamos con S y tenemos la igualdad $N_{a+A}(S) = N_{b+B}(S) = 0$.
- También se puede comprobar que si α' se obtiene de α en un paso de derivación y $N_{a+A}(\alpha) = N_{b+B}(\alpha)$, entonces $N_{a+A}(\alpha') = N_{b+B}(\alpha')$
- Si la condición de igualdad de N_{a+A} y N_{b+B} se verifica al principio y, si se verifica antes de un paso, entonces se verifica después de aplicarlo, necesariamente se verifica al final de la derivación. Si hemos derivado u , entonces $N_{a+A}(u) = N_{b+B}(u)$.
- Como u no tiene variables, entonces $N_{a+A}(u) = N_a(u)$ y $N_{b+B}(u) = N_b(u)$, por lo tanto, $N_a(u) = N_b(u)$, es decir si u es generada contiene el mismo número de a que de b .

Para demostrar que todas las palabras del lenguaje son generadas por la gramática, damos el siguiente algoritmo que en n pasos es capaz de generar una palabra de n símbolos. El algoritmo genera las palabras por la izquierda obteniendo, en cada paso, un nuevo símbolo de la palabra a generar.

- Para generar una a
 - Si a último símbolo de la palabra, aplicar $A \rightarrow a$
 - Si no es el último símbolo
 - Si la primera variable es S aplicar $S \rightarrow aB$
 - Si la primera variable es B aplicar $B \rightarrow aBB$
 - Si la primera variable es A
 - ◊ Si haya más variables aplicar $A \rightarrow a$
 - ◊ Si no hay más, aplicar $A \rightarrow aS$
- Para generar una b
 - Si b último símbolo de la palabra, aplicar $B \rightarrow b$
 - Si no es el último símbolo
 - Si la primera variable es S aplicar $S \rightarrow bA$
 - Si la primera variable es A aplicar $A \rightarrow bAA$
 - Si la primera variable es B
 - ◊ Si haya más variables aplicar $B \rightarrow b$
 - ◊ Si no hay más, aplicar $B \rightarrow bS$

Las condiciones que garantizan que todas las palabras son generadas mediante este algoritmo son las siguientes:

- Las palabras generadas tienen primero símbolos terminales y después variables.
- Se genera un símbolo de la palabra en cada paso de derivación
- Las variables que aparecen en la palabra pueden ser:
 - Una cadena de A (si hemos generado más b que a)
 - Una cadena de B (si hemos generado más a que b)
 - Una S si hemos generado las mismas a que b
- Antes de generar el último símbolo tendremos como variables:
 - Una A si tenemos que generar a
 - Una B si tenemos que generar b
- Entonces aplicamos la primera opción para generar los símbolos y la palabra queda generada.

Ejemplo 13 Sea $G = (\{S, X, Y\}, \{a, b, c\}, P, S)$ donde P tiene las reglas,

$$\begin{array}{llll} S \rightarrow abc & S \rightarrow aXbc & Xb \rightarrow bX & Xc \rightarrow Ybcc \\ bY \rightarrow Yb & aY \rightarrow aaX & aY \rightarrow aa & \end{array}$$

Esta gramática genera el lenguaje: $\{a^n b^n c^n \mid n = 1, 2, \dots\}$.

Para ver esto observemos que S en un paso, puede generar abc ó aXbc. Así que $abc \in L(G)$. A partir de aXbc solo se puede relizar la siguiente sucesión de derivaciones,

$$aXbc \implies abXc \implies abYbcc \implies aYbbcc$$

En este momento podemos aplicar dos reglas:

- $aY \rightarrow aa$, en cuyo caso producimos $aabbcc = a^2b^2c^2 \in L(G)$
- $aY \rightarrow aaX$, en cuyo caso producimos $aaXbbcc$

A partir de $aaXbbcc$, se puede comprobar que necesariamente llegamos a $a^2Yb^3c^3$. Aquí podemos aplicar otra vez las dos reglas de antes, produciendo $a^3b^3c^3$ ó $a^3Xb^3c^3$. Así, mediante un proceso de inducción, se puede llegar a demostrar que las únicas palabras de símbolos terminales que se pueden llegar a demostrar son $a^n b^n c^n, n \geq 1$.

1.3.4. Jerarquía de Chomsky

De acuerdo con lo que hemos visto, toda gramática genera un único lenguaje, pero distintas gramáticas pueden generar el mismo lenguaje. Podríamos pensar en clasificar las gramáticas por el lenguaje que generan, por este motivo hacemos la siguiente definición.

Definición 20 *Dos gramáticas se dicen debilmente equivalentes si generan el mismo lenguaje.*

Sin embargo, al hacer esta clasificación nos encontramos con que el problema de saber si dos gramáticas generan el mismo lenguaje es indecidible. No existe ningún algoritmo que acepte como entrada dos gramáticas y nos diga (la salida del algoritmo) si generan o no el mismo lenguaje.

De esta forma, tenemos que pensar en clasificaciones basadas en la forma de la gramática, más que en la naturaleza del lenguaje que generan. La siguiente clasificación se conoce como jerarquía de Chomsky y sigue esta dirección.

Definición 21 *Una gramática se dice que es de*

- **Tipo 0** *Cualquier gramática. Sin restricciones.*
- **Tipo 1** *Si todas las producciones tienen la forma*

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$$

donde $\alpha_1, \alpha_2, \beta \in (V \cup T)^$, $A \in V$, y $\beta \neq \epsilon$, excepto posiblemente la regla $S \rightarrow \epsilon$, en cuyo caso S no aparece a la derecha de las reglas.*

- **Tipo 2** *Si cualquier producción tiene la forma*

$$A \rightarrow \alpha$$

dónde $A \in V, \alpha \in (V \cup T)^$.*

- **Tipo 3** *Si toda regla tiene la forma*

$$A \rightarrow uB \text{ ó } A \rightarrow u$$

donde $u \in T^$ y $A, B \in V$*

Definición 22 *Un lenguaje se dice que es de tipo i ($i = 0, 1, 2, 3$) si y solo si es generado por una gramática de tipo i . La clase o familia de lenguajes de tipo i se denota por L_i .*

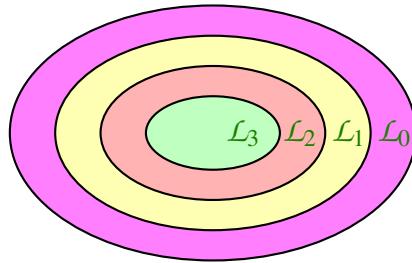


Figura 1.1: Estructura de las clases de lenguajes

Se puede demostrar que $L_3 \subseteq L_2 \subseteq L_1 \subseteq L_0$.

Las gramáticas de tipo 0 se llaman también gramáticas con estructura de frase, por su origen lingüístico. Los lenguajes aceptados por las gramáticas de tipo 0 son los *recursivamente enumerables*.

Las gramáticas de tipo 1 se denominan dependientes del contexto. Los lenguajes aceptados por estas gramáticas son los lenguajes *dependientes del contexto*.

Las gramáticas de tipo 2, así como los lenguajes generados, se llaman independientes del contexto.

Las gramáticas de tipo 3 se denominan regulares o de estado finito. Los lenguajes aceptados por estas gramáticas se denominan *conjuntos regulares*.

Los homorfismos son útiles para demostrar teoremas.

Teorema 1 *Para toda gramática $G = (V, T, P, S)$ podemos dar otra gramática $G' = (V', T, P', S)$ que genere el mismo lenguaje y tal que en la parte izquierda de las reglas solo aparezcan variables.*

Demostración

Si la gramática es de tipo 3 ó 2 no hay nada que demostrar.

Si la gramática es de tipo 0 ó 1, entonces para cada $a_i \in T$ introducimos una variable $A_i \notin V$. Entonces hacemos $V' = V \cup \{A_1, \dots, A_k\}$, donde k es el número de símbolos terminales.

Ahora P' estará formado por las reglas de P donde, en todas ellas, se cambia a_i por A_i . Aparte de ello añadimos una regla $A_i \rightarrow a_i$ para cada $a_i \in T$.

Podemos ver que $L(G) \subseteq L(G')$. En efecto, si derivamos $u = a_{i_1} \dots a_{i_n} \in L(G)$, entonces usando las reglas correspondientes, podemos derivar $A_{i_1} \dots A_{i_n}$ en G' . Como en G' tenemos las reglas $A_i \rightarrow a_i$, entonces podemos derivar $u \in L(G')$.

Para demostrar la inclusión inversa: $L(G') \subseteq L(G)$, definimos un homomorfismo h de $(V' \cup T)^*$ en $(V \cup T)^*$ de la siguiente forma,

1. $h(A_i) = a_i, \quad i = 1, \dots, k$
2. $h(x) = x, \quad \forall x \in V \cup T$

Ahora se puede demostrar que este homorfismo transforma las reglas: si $\alpha \rightarrow \beta \in P'$, entonces $h(\alpha) \rightarrow h(\beta)$ es una producción de P ó $h(\alpha) = h(\beta)$.

Partiendo de esto se puede demostrar que si $\alpha \Rightarrow \beta$ entonces $h(\alpha) \Rightarrow h(\beta)$. En particular, como consecuencia, si $S \Rightarrow u, \quad u \in T^*$, entonces $h(S) = S \Rightarrow h(u) = u$. Es decir, si $u \in L(G')$ entonces $u \in L(G)$. Con lo que definitivamente $L(G) = L(G')$. ■

Ejercicios

1. *Demostrar que la gramática*

$$G = (\{S\}, \{a, b\}, \{S \rightarrow \epsilon, S \rightarrow aSb\}, S)$$

genera el lenguaje

$$L = \{a^i b^i \mid i = 0, 1, 2\}$$

Solución:

Si seguimos este procedimiento, nos encontramos que podemos ir generando todas las palabras de la forma $a^i b^i$, y siempre nos queda la palabra $a^i S b^i$ para seguir generando las palabras de mayor longitud.

Por otra parte, estas son las únicas palabras que se pueden generar.

2. *Encontrar el lenguaje generado por la gramática $G = (\{A, B, S\}, \{a, b\}, P, S)$ donde P contiene las siguientes producciones*

$$\begin{array}{lll} S \rightarrow aAB & bB \rightarrow a & Ab \rightarrow SBb \\ Aa \rightarrow SaB & B \rightarrow SA & B \rightarrow ab \end{array}$$

Solución:

El resultado es el Lenguaje vacío: nunca se puede llegar a generar una palabra con símbolos terminales. Siempre que se sustituye S aparece A , y siempre que se sustituye A aparece S .

3. Encontrar una gramática libre del contexto para generar cada uno de los siguientes lenguajes

a) $L = \{a^i b^j \mid i, j \in N, i \leq j\}$

Solución:

$$S \rightarrow aSb$$

$$S \rightarrow \epsilon$$

$$S \rightarrow Sb$$

b) $L = \{a^i b^j a^j b^i \mid i, j \in N\}$

Solución:

$$S \rightarrow aSb$$

$$S \rightarrow B, \quad B \rightarrow bBa, \quad B \rightarrow \epsilon$$

c) $L = \{a^i b^i a^j b^j \mid i, j \in N\}$

Solución:

Podemos generar $\{a^i b^i \mid i \in N\}$ con:

$$S_1 \rightarrow aS_1b, \quad S_1 \rightarrow \epsilon$$

El lenguaje L se puede generar añadiendo:

$$S \rightarrow S_1 S_1$$

siendo S el símbolo inicial.

d) $L = \{a^i b^i \mid i \in N\} \cup \{b^i a^i \mid i \in N\}$

Solución:

Podemos generar $\{a^i b^i \mid i \in N\}$ con:

$$S_1 \rightarrow aS_1b, \quad S_1 \rightarrow \epsilon$$

y $\{b^i a^i \mid i \in N\}$ con

$$S_2 \rightarrow bS_2a, \quad S_2 \rightarrow \epsilon$$

El lenguaje L se puede generar añadiendo:

$$S \rightarrow S_1, \quad S \rightarrow S_2$$

siendo S el símbolo inicial.

e) $L = \{uu^{-1} \mid u \in \{a,b\}^*\}$

Solución:

$$S \rightarrow aSa, \quad S \rightarrow bSb, \quad S \rightarrow \epsilon$$

f) $L = \{a^i b^j c^{i+j} \mid i, j \in N\}$

Solución:

$$S \rightarrow aSc, \quad S \rightarrow B,$$

$$B \rightarrow bBc, \quad B \rightarrow \epsilon$$

donde \mathcal{N} es el conjunto de los números naturales incluyendo el 0.

4. Determinar si la gramática $G = (\{S, A, B\}, \{a, b, c, d\}, P, S)$ donde P es el conjunto de reglas de producción

$$\begin{array}{lll} S \rightarrow AB & A \rightarrow Ab & A \rightarrow a \\ B \rightarrow cB & B \rightarrow d & \end{array}$$

genera un lenguaje de tipo 3.

Solución:

Esta gramática genera el lenguaje: $\{ab^i c^j d : i, j \in N\}$, y este lenguaje se puede generar mediante la gramática:

$$S \rightarrow aB, \quad B \rightarrow bB, \quad B \rightarrow C, \quad C \rightarrow cC, \quad C \rightarrow d$$

Como esta gramática es de tipo 3, el lenguaje lo es.

Capítulo 2

Autómatas Finitos, Expresiones Regulares y Gramáticas de tipo 3

Los autómatas finitos son capaces de reconocer solamente, un determinado tipo de lenguajes, llamados Lenguajes Regulares, que pueden ser caracterizados también, mediante un tipo de gramáticas llamadas también regulares. Una forma adicional de caracterizar los lenguajes regulares, es mediante las llamadas expresiones regulares, que son las frases del lenguaje, construidas mediante operadores sobre el alfabeto del mismo y otras expresiones regulares, incluyendo el lenguaje vacío.

Estas caracterizaciones de los lenguajes regulares se utilizan en la práctica, según que la situación concreta esté favorecida por la forma de describir el lenguaje de cada una de ellas. Los autómatas finitos se utilizan generalmente para verificar que las cadenas pertenecen al lenguaje, y como un analizador en la traducción de algoritmos al ordenador.

Las gramáticas y sus reglas de producción se usan frecuentemente en la descripción de la sintaxis de los lenguajes de programación que se suele incluir en los manuales correspondientes. Por otro lado, las expresiones regulares proporcionan una forma concisa y relativamente sencilla (aunque menos intuitiva) para describir los lenguajes regulares, poniendo de manifiesto algunos detalles de su estructura que no quedan tan claros en las otras caracterizaciones. Su uso es habitual en editores de texto, para búsqueda y sustitución de cadenas.

En definitiva, las caracterizaciones señaladas de los lenguajes (formales) regulares, y por tanto ellos mismos, tienen un uso habitual en la computación práctica actual. Esto por sí mismo justificaría su inclusión en un currículum de computación teórica.

2.1. Autómatas Finitos Determinísticos

Antes de dar una definición formal de lo que es un autómata vamos a dar una descripción intuitiva mediante un ejemplo.

Ejemplo 14 Vamos a diseñar un autómata que reconozca el paso de un alumno por una asignatura, por ejemplo, *Modelos de Computación I*. Representará las distintas decisiones que se realizan y si se aprueba o suspende la asignatura. Se controla que no haya más de dos convocatorias por año y se termina cuando se aprueba la asignatura.

Habrá un alfabeto de entrada contendrá los siguientes elementos:

- **P**: El alumno se presenta a un examen.
- **N**: El alumno no se presenta a un examen.
- **A**: El alumno aprueba un examen.
- **S**: El alumno suspende un examen.

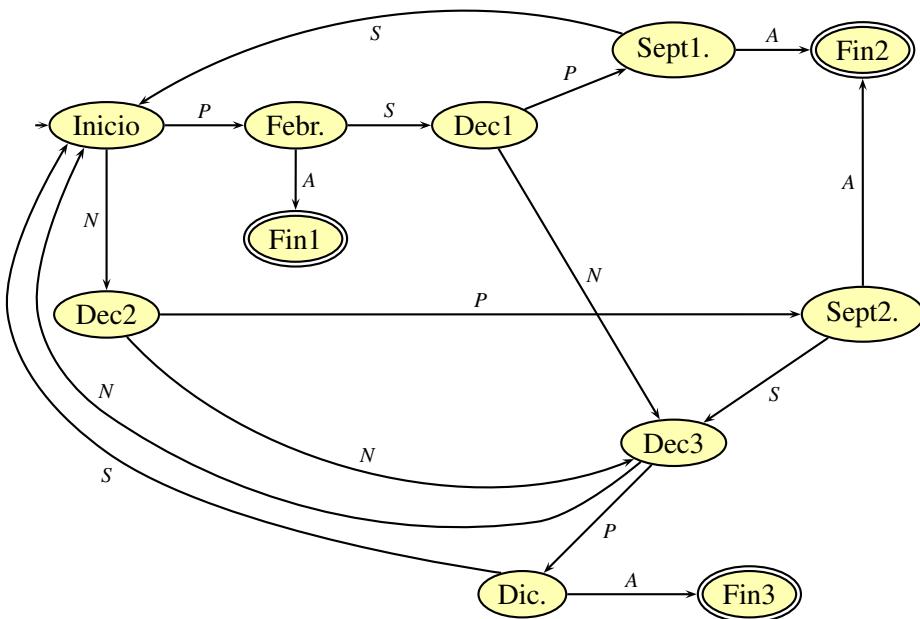


Figura 2.1: Paso de un alumno por una asignatura

La secuencia se ilustra en la figura 2.1. Comenzamos en un estado, Inicio. A continuación decidimos si presentarnos en Febrero o no. Si nos presentamos y aprobamos, terminamos. Si no nos presentamos o suspendemos, tenemos que decidir si nos presentamos en septiembre, pero como hay que controlar que un estudiante no se presente a tres convocatorias en un año, los estados son distintos en ambos casos. Si en septiembre aprobamos, terminamos. Si suspendemos y ya nos habíamos presentado en febrero, comenzamos de nuevo. En otro caso, podemos decidir si presentarnos en diciembre. Si aprobamos, terminamos y si suspendemos, empezamos de nuevo.

Este esquema corresponde a un autómata finito. Se caracteriza por una estructura de control que depende de un conjunto finito de estados. Se pasa de unos a otros leyendo símbolos del alfabeto de entrada. Este autómata representa una versión simplificada del problema real, ya que no controla el número total de convocatorias. Un autómata para todas las asignaturas se podría construir uniendo autómatas para cada una de las asignaturas, pero teniendo en cuenta relaciones como requisitos entre las mismas.

Definición 23 Un autómata finito es una quintupla $M = (Q, A, \delta, q_0, F)$ en que

- Q es un conjunto finito llamado conjunto de estados

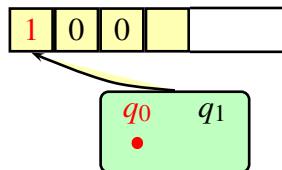


Figura 2.2: Autómata de Estado Finito

- A es un alfabeto llamado alfabeto de entrada
- δ es una aplicación llamada función de transición

$$\delta : Q \times A \rightarrow Q$$

- q_0 es un elemento de Q , llamado estado inicial
- F es un subconjunto de Q , llamado conjunto de estados finales.

Desde el punto de vista intuitivo, podemos ver un autómata finito como una caja negra de control (ver Figura 2.2), que va leyendo símbolos de una cadena escrita en una cinta, que se puede considerar ilimitada por la derecha. Existe una cabeza de lectura que en cada momento está situada en una casilla de la cinta. Inicialmente, esta se sitúa en la casilla de más a la izquierda. El autómata en cada momento está en uno de los estados de Q . Inicialmente se encuentra en q_0 .

En cada paso, el autómata lee un símbolo y según el estado en que se encuentre, cambia de estado y pasa a leer el siguiente símbolo. Así sucesivamente hasta que termine de leer todos los símbolos de la cadena. Si en ese momento la máquina está en un estado final, se dice que el autómata acepta la cadena. Si no está en un estado final, la rechaza.

Definición 24 *El diagrama de transición de un Autómata de Estado Finito es un grafo en el que los vértices representan los distintos estados y los arcos las transiciones entre los estados. Cada arco va etiquetado con el símbolo que corresponde a dicha transición. El estado inicial y los finales vienen señalados de forma especial (por ejemplo, con un ángulo el estado inicial y con un doble círculo los finales).*

Ejemplo 15 Supongamos el autómata $M = (Q, A, q_0, \delta, F)$ donde

- $Q = \{q_0, q_1, q_2\}$
- $A = \{a, b\}$

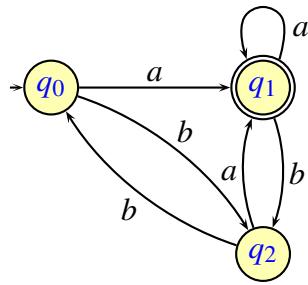


Figura 2.3: Diagrama de Transición Asociado a un Autómata de Estado Finito

- La función de transición δ está definida por las siguientes igualdades:

$$\begin{array}{ll} \delta(q_0, a) = q_1 & \delta(q_0, b) = q_2 \\ \delta(q_1, a) = q_1 & \delta(q_1, b) = q_2 \\ \delta(q_2, a) = q_1 & \delta(q_2, b) = q_0 \end{array}$$

- $F = \{q_1\}$

El diagrama de transición viene expresado en la Figura 2.3.

2.1.1. Proceso de cálculo asociado a un Autómata de Estado Finito

Para describir formalmente el comportamiento de un autómata de estado finito, vamos a introducir el concepto de configuración y paso de cálculo.

Definición 25 Si $M = (Q, A, \delta, q_0, F)$ es un autómata de estado finito una configuración es un elemento del producto cartesiano $Q \times A^*$.

Una configuración es un par (q, u) donde q es un estado y u una palabra. Intuitivamente una configuración contiene los elementos que determinan la evolución futura del autómata. En este sentido, q será el estado en el que se encuentra el autómata y u lo que queda por leer en un momento dado.

Inicialmente, autómata está en el estado inicial q_0 y nos queda por leer toda la palabra de entrada. El siguiente concepto recoge esta idea.

Definición 26 Si $M = (Q, A, \delta, q_0, F)$ es una autómata finito determinista y $u \in A^*$ es una palabra, entonces se llama configuración inicial asociada a esa palabra a la configuración (q_0, u) .

Intuitivamente, la configuración inicial indica que estamos en el estado inicial y nos queda por leer toda la palabra u .

A continuación definimos la relación paso de cálculo entre dos configuraciones.

Definición 27 Si $M = (Q, A, \delta, q_0, F)$ es una autómata finito determinista y $(p, u), (q, v)$ son dos configuraciones, decimos que se puede pasar de (p, u) a (q, v) en un paso de cálculo, lo que se nota como $(p, u) \vdash (q, v)$ si y sólo si $u = av$, donde $a \in A$ y $\delta(p, a) = q$.

Finalmente, definimos la relación de cálculo entre dos configuraciones.

Definición 28 Si $M = (Q, A, \delta, q_0, F)$ es una autómata finito determinista y $(p, u), (q, v)$ son dos configuraciones, decimos que puede pasar de (p, u) a (q, v) en una secuencia de cálculo, lo que se nota como $(p, u) \vdash^* (q, v)$ si y solo si existe una sucesión de configuraciones: $(p_0, u_0), \dots, (p_n, u_n)$ donde $n \geq 1$, de tal forma que $(p_0, u_0) = (p, u)$, $(p_n, u_n) = (q, v)$ y $(p_i, u_i) \vdash (p_{i+1}, u_{i+1}), \forall i < n$.

Existe otro enfoque alternativo para definir formalmente el cálculo asociado a un autómata de estado finito y consiste en definir la función de transición de un autómata aplicada a una palabra. Esta función que llamaremos δ' se define de forma recursiva

Definición 29 Si $M = (Q, A, \delta, q_0, F)$ es un autómata de estado finito se define la función de transición asociada a palabras, como la función

$$\delta' : Q \times A^* \rightarrow Q$$

dada por

$$\begin{aligned}\delta'(q, \epsilon) &= q \\ \delta'(q, aw) &= \delta'(\delta(q, a), w), \quad w \in A^*, a \in A\end{aligned}$$

Ejemplo 16 En el caso del autómata del ejemplo 15, tenemos que

$$\begin{aligned}\delta'(q_0, aba) &= \delta'(\delta(q_0, a), ba) = \delta'(q_1, ba) = \delta'(\delta(q_1, b), a) = \\ \delta'(q_2, b) &= \delta'(\delta(q_2, b), \epsilon) = \delta'(q_1, \epsilon) = q_1\end{aligned}$$

Desde el punto de vista intuitivo, δ representa el comportamiento del autómata en un paso de cálculo, ante la lectura de un carácter. Esto viene dado por el estado al que evoluciona el autómata. La cabeza de lectura siempre se mueve a la derecha. δ' representa una sucesión de cálculos del autómata: a qué estado evoluciona después de haber leído una cadena de caracteres.

La relación entre δ' y \vdash^* es la siguiente: $\delta'(p, u) = q$ si y sólo si $(p, u) \vdash^* (q, \epsilon)$.

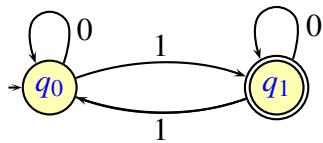


Figura 2.4: Autómata que acepta el lenguaje de palabras con un número impar de unos

2.1.2. Lenguaje aceptado por un Autómata de Estado Finito

Definición 30 Una palabra $u \in A^*$ se dice *aceptada* por un autómata $M = (Q, A, \delta, q_0, F)$ si y solo si existe un $q \in F$ tal que $(q_0, u) \xrightarrow{*} (q, \epsilon)$. En caso contrario, se dice que la palabra es *rechazada* por el autómata.

Es decir, una palabra es aceptada por un autómata si comenzando a calcular en el estado q_0 y leyendo los distintos símbolos de la palabra llega a un estado final.

En el caso del autómata del ejemplo 15, la palabra *aba* es aceptada por el autómata. La palabra *aab* es rechazada.

Definición 31 Dado un autómata $M = (Q, A, \delta, q_0, F)$ se llama *lenguaje aceptado o reconocido* por dicho autómata al conjunto de las palabras de A^* que acepta:

$$L(M) = \{u \in A^* : (q_0, u) \xrightarrow{*} (q, \epsilon), q \in F\}$$

Ejemplo 17 El autómata dado por el diagrama de transición de la Figura 2.4 acepta el lenguaje formado por aquellas cadenas con un número impar de unos.

Ejemplo 18 Los autómatas suelen servir para reconocer constantes o identificadores de un lenguaje de programación. Supongamos, por ejemplo, que los reales vienen definidos por la gramática $G = (V, T, P, S)$ en la que

- $T = \{+, -, E, 0, 1, \dots, 9, .\}$
- $V = \{\langle \text{Signo} \rangle, \langle \text{Digito} \rangle, \langle \text{Natural} \rangle, \langle \text{Entero} \rangle, \langle \text{Real} \rangle\}$
- $S = \langle \text{Real} \rangle$
- P contiene las siguientes producciones

$$\begin{aligned} \langle \text{Signo} \rangle &\rightarrow + | - \\ \langle \text{Digito} \rangle &\rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \end{aligned}$$

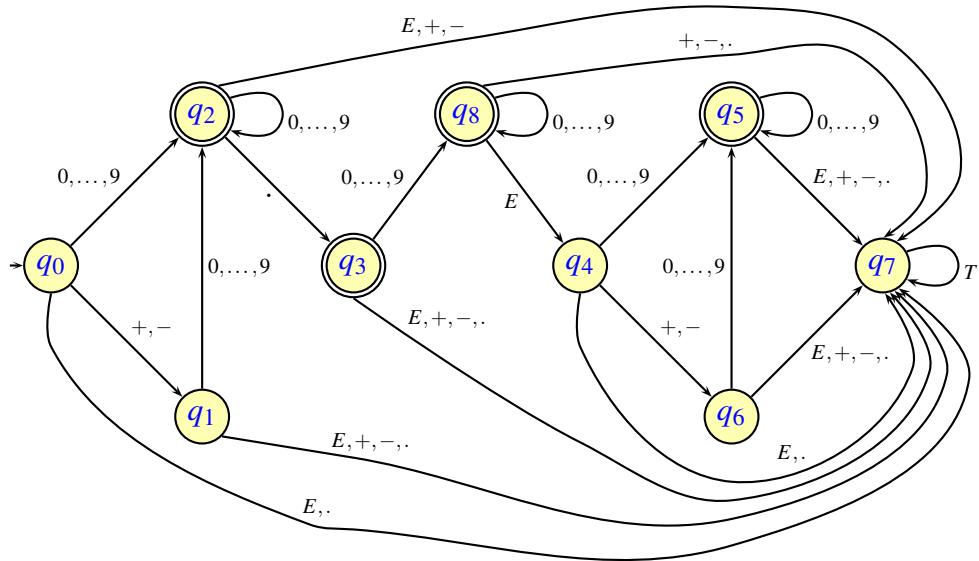


Figura 2.5: Autómata Finito Determinista que acepta constantes reales

$\langle \text{Natural} \rangle \rightarrow \langle \text{Digito} \rangle \mid \langle \text{Digito} \rangle \langle \text{Natural} \rangle$
 $\langle \text{Entero} \rangle \rightarrow \langle \text{Natural} \rangle$
 $\langle \text{Entero} \rangle \rightarrow \langle \text{Signo} \rangle \langle \text{Natural} \rangle$
 $\langle \text{Real} \rangle \rightarrow \langle \text{Entero} \rangle$
 $\langle \text{Real} \rangle \rightarrow \langle \text{Entero} \rangle .$
 $\langle \text{Real} \rangle \rightarrow \langle \text{Entero} \rangle . \langle \text{Natural} \rangle$
 $\langle \text{Real} \rangle \rightarrow \langle \text{Entero} \rangle . \langle \text{Natural} \rangle E \langle \text{Entero} \rangle$

Este lenguaje puede ser descrito también por un autómata de estado finito, como el de la Figura 18, en el que el alfabeto es $A = \{0, 1, 2, \dots, 9, +, -, E\}$.

Si un lenguaje es aceptado por un autómata de estado finito, tenemos un procedimiento algorítmico sencillo que nos permite decir qué palabras pertenecen al lenguaje y qué palabras no pertenecen. Es mucho más cómodo que la gramática inicial, aunque la gramática es más expresiva. No todos los lenguajes pueden describirse mediante autómatas de estado finito. Los más importantes son los identificadores y constantes de los lenguajes. Sin embargo, es importante señalar que a partir de una gramática de tipo 3 podemos construir de forma algorítmica un autómata que acepta el mismo lenguaje que el generado por la gramática.

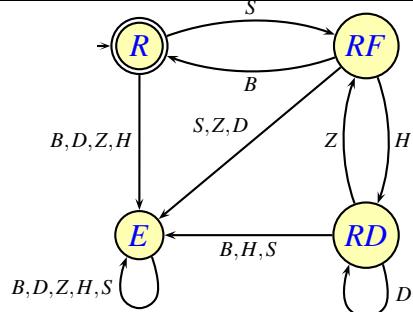


Figura 2.6: Recepción de Ficheros en el protocolo Kermit

Ejemplo 19 Comunicaciones. Protocolo Kermit.- *Los autómatas finitos proporcionan modelos de programación para numerosas aplicaciones. El protocolo Kermit se puede implementar como un control de estado finito. Vamos a fijarnos en una pequeña parte del mismo: la transferencia de ficheros entre micros y mainframes y, más concretamente en la recepción de estos ficheros.*

El esquema del programa de recepción de datos corresponde al de una máquina de estado finito (ver Figura 2.6). Inicialmente, la máquina está en un estado de espera, R. En este estado está preparado para recibir una cabecera de transmisión (símbolo de entrada S), en cuyo caso pasa al estado RF, en el que espera la recepción de una cabecera de fichero (símbolo de entrada H). En ese momento pasa al estado RD, en el que procesa una serie de datos correspondientes al fichero (símbolos D). Si, en un momento dado, recibe un fin de fichero (símbolo Z) pasa al estado RF, donde puede recibir otra cabecera de fichero. Si estando en el estado RF se recibe un código de fin de la transmisión el autómata pasa al estado inicial R. El estado final es R: Una transmisión es aceptada si, al final de cada fichero se produce un fin de fichero y al final de todos los ficheros un fin de transmisión que le permite volver a R. Si, en un momento dado se produce una entrada inesperada (por ejemplo, un fin de la transmisión cuando se están recibiendo datos de un fichero en particular) se pasa a un estado de error, E. Este estado es absorbente: cualquier otra entrada le hace quedar en el mismo estado.

El esquema del emisor es mas complicado. También ésto es solo un esquema del comportamiento del receptor. En realidad, éste no solo determina si la transmisión se ha realizado con éxito, sino que ejecuta acciones cada vez que recibe datos. Esto no lo puede hacer un autómata de estado finito, pero sí otro tipo de máquinas un poco más sofisticadas que estudiaremos más adelante. De todas formas, lo importante es advertir como el modelo de computación de los autómatas finitos sirve para expresar de forma elegante el mecanismo de control de muchos programas.

2.2. Autómatas Finitos No-Determinísticos (AFND)

En este apartado vamos a introducir el concepto de autómatas finitos no determinísticos (AFND) y veremos que aceptan exactamente los mismos lenguajes que los autómatas determinísticos. Sin embargo, serán importantes para demostrar teoremas y por su más alto poder expresivo.

En la definición de autómata no-determinístico lo único que cambia, respecto a la definición de autómata determinístico es la función de transición. Antes estaba definida

$$\delta : Q \times A \rightarrow Q$$

y ahora será una aplicación de $Q \times A$ en $\wp(Q)$ (subconjuntos de Q):

$$\delta : Q \times A \rightarrow \wp(Q)$$

La definición completa quedaría como sigue.

Definición 32 *Un autómata finito no-determinístico es una quintupla $M = (Q, A, \delta, q_0, F)$ en que*

- Q es un conjunto finito llamado conjunto de estados
- A es un alfabeto llamado alfabeto de entrada
- δ es una aplicación llamada función de transición

$$\delta : Q \times A \rightarrow \wp(Q)$$

- q_0 es un elemento de Q , llamado estado inicial
- F es un subconjunto de Q , llamado conjunto de estados finales.

La interpretación intuitiva es que ahora el autómata, ante una entrada y un estado dado, puede evolucionar a varios estados posibles (incluyendo un solo estado o ninguno si $\delta(q, a) = \emptyset$). Es decir es como un algoritmo que en un momento dado nos deja varias opciones posibles o incluso puede no dejarnos ninguna.

Una palabra se dice aceptada por un AFND si, siguiendo en cada momento alguna de las opciones posibles, llegamos a un estado final.

A continuación, vamos a extender δ definida en $Q \times A$ a una función δ^* definida en $\wp(Q) \times A$, es decir, para que se pueda aplicar a conjuntos de estados, no sólo a un estado dado. δ^* se define de la siguiente forma,

$$\text{Si } P \subseteq Q \text{ y } u \in A, \delta^*(P, a) = \bigcup_{q \in P} \delta(q, a)$$

Esta función se puede extender para aplicarse también a palabras de la siguiente forma:

Si $B \subseteq Q$,

- $\delta^*(B, \epsilon) = B$
- $\delta^*(B, au) = \delta^*(\delta^*(B, a), u)$

δ^* se puede aplicar también a un sólo estado de acuerdo con la siguiente expresión: $\delta^*(q, u) = \delta^*(\{q\}, u)$

Con estos elementos podemos definir cuando una palabra es aceptada por un autómata no-determinístico.

Definición 33 Sea $M = (Q, A, \delta, q_0, F)$ un autómata finito no-determinístico y $u \in A^*$. Se dice que la palabra u es aceptada por M si y solo si $\delta'(q_0, u) \cap F \neq \emptyset$.

Definición 34 Sea $M = (Q, A, \delta, q_0, F)$ un AFND, se llama lenguaje aceptado por el autómata al conjunto de palabras de A^* que acepta, es decir

$$L(M) = \{u \in A^* \mid \delta'(q_0, u) \cap F \neq \emptyset\}$$

Cuando no haya lugar a confusión, δ^* será representadas como δ , simplemente.

En clase, el lenguaje aceptado por un autómata no determinista se ha definido de forma distinta, pero ambas definiciones son totalmente equivalentes. Se define en función de la definición de la relación de proceso de cálculo entre dos configuraciones que, análogamente como se hizo con los autómatas deterministas, se basa en definir los siguientes elementos:

■ **Descripción Instantánea o Configuración:**

Un elemento de $Q \times A^*$: (q, u) .

■ **Configuración Inicial para $u \in A^*$:** (q_0, u)

■ **Relación paso de cálculo entre dos configuraciones:**

$$((q, au) \vdash (p, v)) \Leftrightarrow p \in \delta(q, a)$$

Aquí, al contrario de lo que ocurría en los autómatas deterministas, desde una configuración se puede pasar a varias configuraciones distintas en un paso de cálculo, e incluso a ninguna.

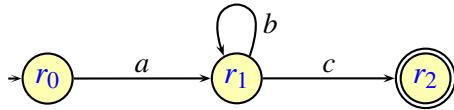


Figura 2.7: Autómata Finito No-Determinístico

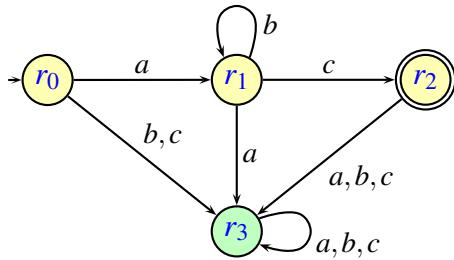


Figura 2.8: Autómata Finito Determinista

■ **Relación de cálculo entre dos configuraciones:**

$((q, u) \xrightarrow{*} (p, v))$ si y solo si existe una sucesión de configuraciones C_0, \dots, C_n tales que $C_0 = (q, u), C_n = (p, v)$ y $\forall i \leq n - 1, C_i \vdash C_{i+1}$.

Finalmente, el lenguaje aceptado por un AF no-determinista es

$$L(M) = \{u \in A^* : \exists q \in F, (q_0, u) \xrightarrow{*} (q, \varepsilon)\}$$

2.2.1. Diagramas de Transición

Los diagramas de transición de los AFND son totalmente análogos a los de los autómatas determinísticos. Solo que ahora no tiene que salir de cada vértice un y solo un arco para cada símbolo del alfabeto de entrada. En un autómata no determinístico, de un vértice pueden salir ninguna, una o varias flechas con la misma etiqueta.

Ejemplo 20 Un AFND que acepta el lenguaje

$$L = \{u \in \{a, b, c\}^* \mid u = ab^i c, i \geq 0\}$$

es el de la Figura 2.7.

Este es un autómata no-determinista ya que hay transiciones no definidas.

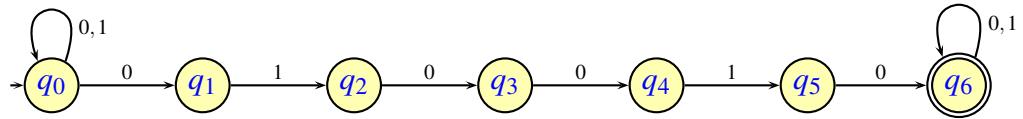


Figura 2.9: Autómata No-Determinista que reconoce la cadena 010010.

En general, los autómatas no-determinísticos son más simples que los determinísticos. Por ejemplo, un autómata determinístico que acepta el mismo lenguaje que el anterior es el que viene dado por la Figura 2.8

Ejemplo 21 (Reconocimiento de Patrones).- *Supongamos un ejemplo de transmisión de datos entre barcos. El receptor de un barco debe de estar siempre esperando la transmisión de datos que puede llegar en cualquier momento. Cuando no hay transmisión de datos hay un ruido de fondo (sucesión aleatoria de 0, 1). Para comenzar la transmisión se manda una cadena de aviso, p.e. 010010. Si esa cadena se reconoce hay que registrar los datos que siguen.*

El programa que reconoce esta cadena puede estar basado en un autómata finito. La idea es que este no pueda llegar a un estado no final mientras no se reciba la cadena inicial. En ese momento el autómata pasa a un estado final. A partir de ahí todo lo que llegue se registra. Nuestro propósito es hacer un autómata que llegue a un estado final tan pronto como se reconozca 010010. Intentar hacer un autómata finito determinístico directamente puede ser complicado, pero es muy fácil el hacer un AFND, como el de la Figura 2.9.

Hay que señalar que esto sería solamente el esquema de una sola parte de la transmisión. Se podría complicar incluyendo también una cadena para el fin de la transmisión.

Ejemplo 22 Autómata no-determinístico que acepta constantes reales *Ver Figura 2.10*

2.2.2. Equivalencia de Autómatas Determinísticos y No-Determinísticos

La siguiente definición muestra como construir a partir de un autómata finito no-determinístico otro determinístico que acepte el mismo lenguaje.

Definición 35 *Dado un AFND $M = (Q, A, \delta, q_0, F)$ se llama autómata determinístico asociado a M , al autómata $\bar{M} = (\bar{Q}, A, \bar{\delta}, \bar{q}_0, \bar{F})$ dado por*

- $\bar{Q} = \mathcal{P}(Q)$
- $\bar{q}_0 = \{q_0\}$
- $\bar{\delta}(A, a) = \delta^*(A, a)$

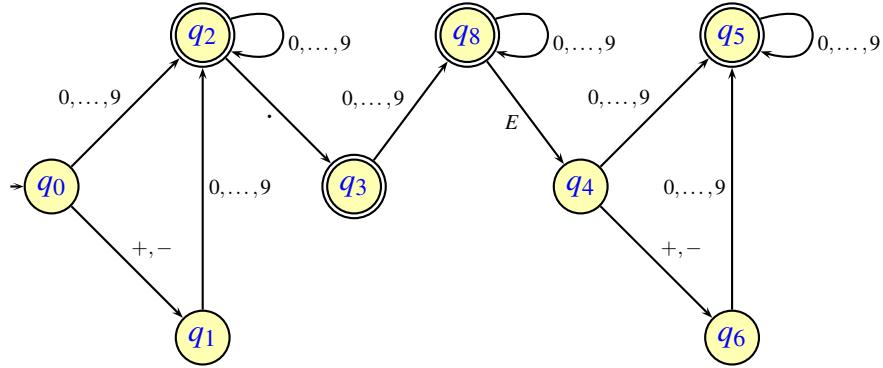


Figura 2.10: Autómata No-Determinístico que reconoce constantes reales.

$$-\bar{F} = \{A \in \mathcal{P}(Q) \mid A \cap F \neq \emptyset\}$$

Dado un autómata no determinístico se le hace corresponder uno determinístico que recorre todos los caminos al mismo tiempo.

Ejemplo 23 En la Figura 2.11 podemos ver un autómata finito no-determinista del ejemplo 21 (figura 2.9) y su autómata determinístico asociado.

Teorema 2 Un AFND M y su correspondiente Autómata determinístico \bar{M} aceptan el mismo lenguaje.

Demostración

La demostración se basa en probar que

$$\bar{\delta}^*(\bar{q}_0, u) = \delta^*(q_0, u)$$

■

Es decir, que el conjunto de estados en los que puede estar el autómata no determinista coincide con el estado en el que está el autómata determinista que hemos construido. Ambos aceptan una palabra si al final de su lectura uno de estos estados es final.

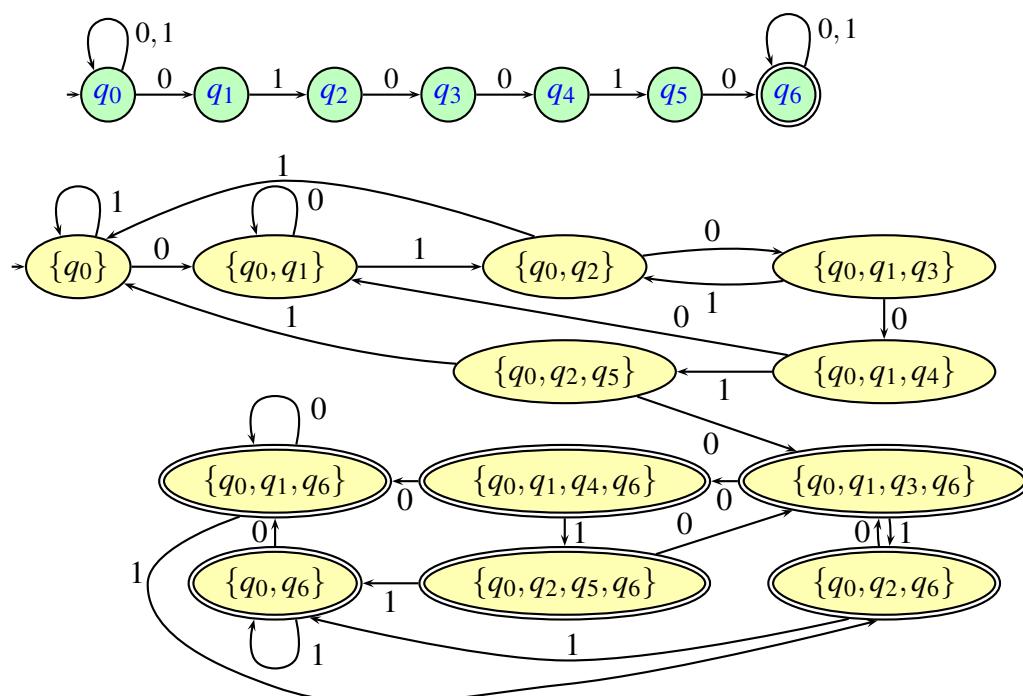


Figura 2.11: Automata Finito No-Determinístico y Autómata Finito Determinístico asociado

El teorema inverso es evidente: dado un autómata determinístico $M = (Q, A, \delta, q_0, F)$ podemos construir uno no-determinista $\bar{M} = (Q, A, \bar{\delta}, q_0, F)$ que acepta el mismo lenguaje. Solo hay que considerar que $\bar{\delta}(q_0, a) = \{\delta(q_0, a)\}$: En cada situación solo hay un camino posible, el dado por δ .

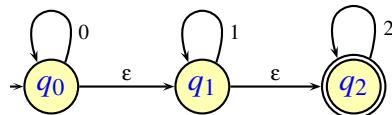
2.3. Autómatas Finitos No Deterministicos con transiciones nulas

Son autómatas que pueden realizar una transición sin consumir entrada. Estas transiciones se etiquetan con ϵ en el diagrama asociado.

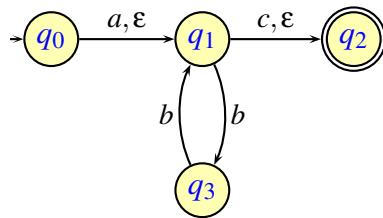
En la definición, lo único que hay que cambiar es la función de transición que ahora está definida de $Q \times (A \cup \{\epsilon\})$ en $\mathcal{P}(Q)$.

Las transiciones nulas dan una nueva capacidad al autómata. Si se tiene una transición nula, el autómata puede quedarse donde está o cambiar de estado sin consumir ningún símbolo de la palabra de entrada. Como en los AFND, una palabra será aceptada si se llega a un estado final con alguna de las elecciones posibles.

Ejemplo 24 El autómata siguiente acepta las palabras que son una sucesión de ceros, seguida de una sucesión de 1, seguida de una sucesión de 2.



Ejemplo 25 El autómata dado por el siguiente diagrama,



acepta

el lenguaje $\{b^2i : i \geq 0\} \cup \{ab^2i : i \geq 0\} \cup \{b^2ic : i \geq 0\} \cup \{ab^2ic : i \geq 0\}$

Definición 36 Se llama clausura de un estado al conjunto de estados a los que puede evolucionar sin consumir ninguna entrada,

$$CL(p) = \{q \in Q : \exists q_1, \dots, q_k \text{ tal que } p = q_1, q = q_k, q_i \in \delta(q_{i-1}, \epsilon), i \geq 2\}$$

Definición 37 Si $P \subseteq Q$ se llama clausura de P a

$$CL(P) = \bigcup_{p \in P} CL(p)$$

Ejemplo 26 En el ejemplo anterior tenemos que

$$CL(q_1) = \{q_1, q_2\}$$

$$CL(q_0, q_1) = \{q_0, q_1, q_2\}$$

La función de transición δ se extiende a conjuntos de estados de la siguiente forma:

$$\delta(R, u) = \bigcup_{q \in R} \delta(q, u)$$

A continuación vamos a determinar la función δ' que asocia a un estado y a una palabra de entrada el conjunto de los estados posibles.

$$\delta'(q, \varepsilon) = CL(q)$$

$$\delta'(q, au) = \bigcup_{r \in \delta(CL(q), a)} \delta'(r, u)$$

Es conveniente extender la función δ' a conjuntos de estados,

$$\delta'(R, u) = \bigcup_{q \in R} \delta'(q, u)$$

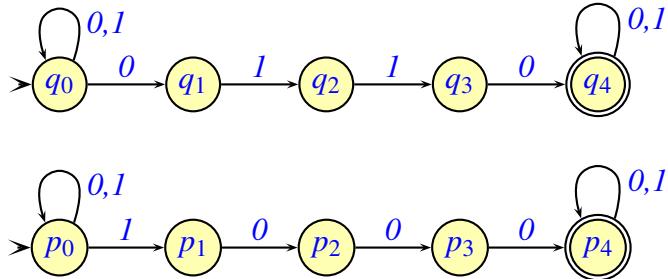
Notemos que $\delta'(q, a)$ no es lo mismo que $\delta(q, a)$ por lo que distinguiremos entre las dos funciones.

Una palabra, $u \in A^*$, se dice aceptada por un AFND con transiciones nulas $M = (Q, A, \delta, q_0, F)$ si y solo si

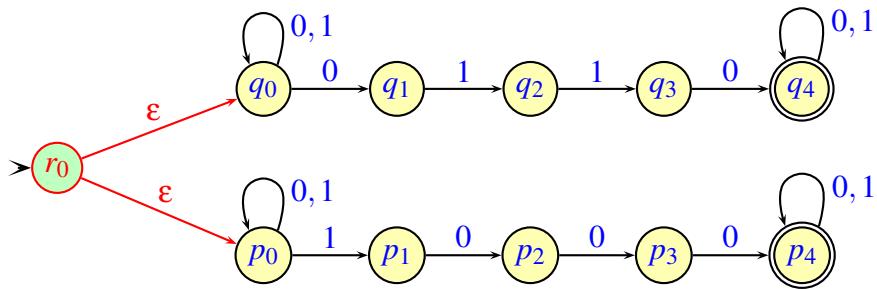
$$\delta'(q, u) \cap F \neq \emptyset$$

El lenguaje aceptado por el autómata, $L(M)$, es el conjunto de palabras de A^* que acepta. Las transiciones nulas son muy útiles para modificar autómatas o para construir autómatas más complejos partiendo de varios autómatas.

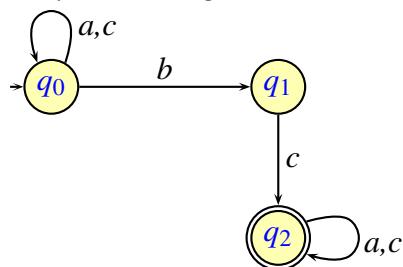
Ejemplo 27 Los siguientes dos autómatas aceptan las palabras que contienen en su interior la subcadena 0110 y 1000 respectivamente



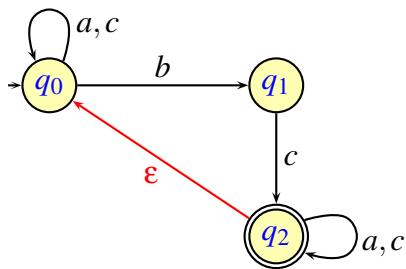
Para construir un autómata que acepte las palabras aceptadas por uno cualquiera de estos dos autómatas, es decir, las palabras con una subcadena 0110 ó una subcadena 1000 se pone un nuevo estado inicial que se une con los antiguos mediante transiciones nulas.



Ejemplo 28 Consideremos el siguiente autómata que acepta las palabras del alfabeto $\{a, b, c\}$ que tienen una sola b y ésta va seguida de una c .



Si L es el lenguaje que acepta el autómata anterior modifiquémoslo para que acepte cualquier sucesión no nula de palabras de L : cualquier palabra de L^+ . Para ello se unen los estados finales del autómata con el estado inicial mediante una transición nula:



Las transiciones nulas, como vemos dan más capacidad de expresividad a los autómatas, pero el conjunto de lenguajes aceptados es el mismo que para los autómatas finitos determinísticos, como queda expresado en el siguiente teorema.

Teorema 3 *Para todo autómata AFND con transiciones nulas existe un autómata finito determinístico que acepta el mismo lenguaje.*

Demostración.- Si $M = (Q, A, \delta, q_0, F)$ es un AFND con transiciones nulas, basta construir un AFND sin transiciones nulas $\bar{M} = (\bar{Q}, \bar{A}, \bar{\delta}, \bar{q}_0, \bar{F})$. A partir de éste se puede construir uno determinístico.

El autómata \bar{M} se define de la siguiente forma

$$\bar{F} = \begin{cases} F & \text{if } CL(q_0) \cap F = \emptyset \\ F \cup \{q_0\} & \text{en caso contrario} \end{cases} \quad (2.1)$$

$$\bar{\delta}(q, a) = \delta'(q, a) \quad (2.2)$$

El resto de los elementos se definen igual que en M .

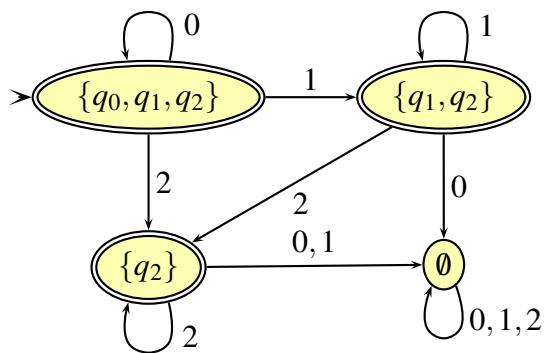
Para demostrar que ambos autómatas aceptan el mismo lenguaje basta con probar que

$$\bar{\delta}'(q, u) = \delta'(q, u) \text{ si } u \neq \epsilon \quad (2.3)$$

y que ϵ se acepta de igual forma en las dos máquinas.

■

Ejemplo 29 El AFND correspondiente al autómata con transiciones nulas del ejemplo 19, acepta el mismo lenguaje que el siguiente autómata determinista:



2.4. Expresiones Regulares

Una expresión regular es una forma de representar cierto tipo de lenguajes sobre un determinado alfabeto. Veremos que son exactamente los aceptados por los autómatas de estado finito.

Definición 38 Si A es un alfabeto, una expresión regular sobre este alfabeto se define de la siguiente forma:

- \emptyset es una expresión regular que denota el lenguaje vacío.
- ϵ es una expresión regular que denota el lenguaje $\{\epsilon\}$
- Si $a \in A$, a es una expresión regular que denota el lenguaje $\{a\}$
- Si r y s son expresiones regulares denotando los lenguajes R y S entonces definimos las siguientes operaciones:
 - Unión: $(r + s)$ es una expresión regular que denota el lenguaje $R \cup S$
 - Concatenación: (rs) es una expresión regular que denota el lenguaje RS
 - Clausura: r^* es una expresión regular que denota el lenguaje R^* .

De acuerdo con la definición anterior, se puede determinar no solo cuales son las expresiones regulares sobre un determinado alfabeto, sino también cuales son los lenguajes que denotan o lenguajes asociados. Los lenguajes que pueden representarse mediante una expresión regular se llaman lenguajes regulares. Estos coinciden con los aceptados por los autómatas finitos, como veremos mas adelante.

Los paréntesis se pueden eliminar siempre que no haya dudas. La precedencia de las operaciones es

- Clausura

- Concatenación
- Unión

Ejemplo 30 Si $A = \{a, b, c\}$

- $(a + \epsilon)b^*$ es una expresión regular que denota el lenguaje $\{a^i b^j : i = 0, 1; j \geq 0\}$

Si $A = \{0, 1\}$

- **00** es una expresión regular con lenguaje asociado $\{00\}$.
- **01^{*} + 0** es una expresión regular que denota el lenguaje $\{01^i : i \geq 0\}$.
- **(1 + 10)^{*}** representa el lenguaje de las cadenas que comienzan por 1 y no tienen dos ceros consecutivos.
- **(0 + 1)^{*}011** representa el lenguaje de las cadenas que terminan en 011.
- **0^{*}1^{*}** representa el lenguaje de las cadenas que no tienen un 1 antes de un 0
- **00^{*}11^{*}** representa un subconjunto del lenguaje anterior formado por cadenas que al menos tienen un 1 y un 0. Si la expresión regular rr^* se representa como r^+ , **00^{*}11^{*}** se puede representar como **0⁺1⁺**.

2.4.1. Propiedades de las Expresiones Regulares

Dos expresiones regulares se consideran iguales si representan el mismo lenguaje. Teniendo esto en cuenta, se puede comprobar que siempre se verifican las siguientes igualdades.

1. $r_1 + r_2 = r_2 + r_1$
2. $r_1 + (r_2 + r_3) = (r_1 + r_2) + r_3$
3. $r_1(r_2r_3) = (r_1r_2)r_3$
4. $r\epsilon = r$
5. $r\emptyset = \emptyset$
6. $r + \emptyset = r$
7. $\epsilon^* = \epsilon$
8. $r_1(r_2 + r_3) = r_1r_2 + r_1r_3$

$$9. \quad (\mathbf{r}_1 + \mathbf{r}_2)\mathbf{r}_3 = \mathbf{r}_1\mathbf{r}_3 + \mathbf{r}_2\mathbf{r}_3$$

$$10. \quad \mathbf{r}^+ + \epsilon = \mathbf{r}^*$$

$$11. \quad \mathbf{r}^* + \epsilon = \mathbf{r}^*$$

$$12. \quad (\mathbf{r} + \epsilon)^* = \mathbf{r}^*$$

$$13. \quad (\mathbf{r} + \epsilon)^+ = \mathbf{r}^*$$

$$14. \quad (\mathbf{r}_1^* + \mathbf{r}_2^*)^* = (\mathbf{r}_1 + \mathbf{r}_2)^*$$

$$15. \quad (\mathbf{r}_1^*\mathbf{r}_2^*)^* = (\mathbf{r}_1 + \mathbf{r}_2)^*$$

2.4.2. Expresiones Regulares y Autómatas Finitos

A continuación vamos a relacionar los lenguajes representados por las expresiones regulares con los aceptados por los autómatas finitos. Demostraremos dos teoremas. El primero nos dice que todo lenguaje asociado por una expresión regular puede ser aceptado por un autómata de estado finito. El segundo la relación recíproca. Dado un autómata de estado finito, veremos como se puede construir una expresión regular que denote el lenguaje aceptado por dicho autómata.

Teorema 4 *Dada una expresión regular existe un autómata finito que acepta el lenguaje asociado a esta expresión regular.*

Demostración.- Vamos a demostrar que existe un AFND con transiciones nulas. A partir de él se podría construir el autómata determinístico asociado.

La construcción del autómata va a ser recursiva. No vamos a expresar como son los estados y las transiciones matemáticamente, sino que los autómatas vendrán expresados de forma gráfica.

Para las expresiones regulares iniciales tenemos los siguiente autómatas:

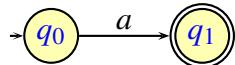
– \emptyset



– ϵ



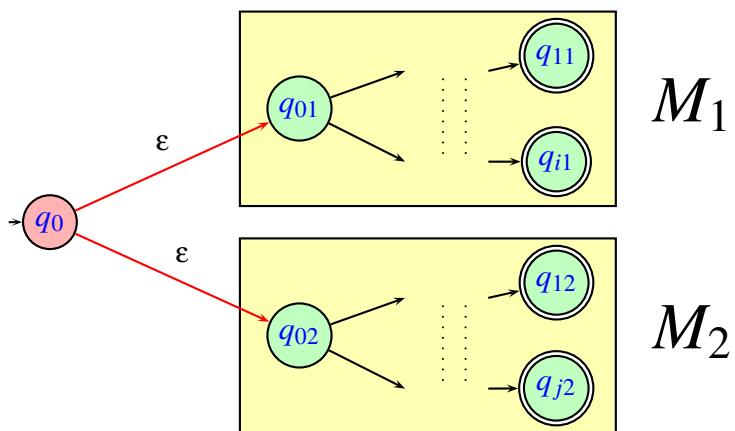
– a



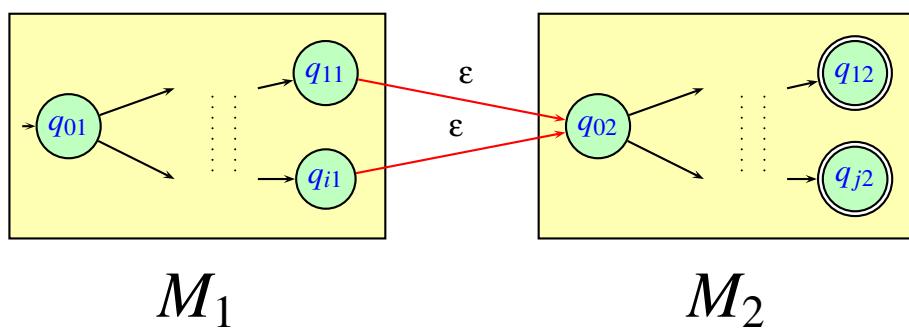
Ahora veremos como se pueden construir autómatas para las expresiones regulares compuestas a partir de los autómatas que aceptan cada una de sus componentes.

Si M_1 es el autómata que acepta el mismo lenguaje que el representado por r_1 y M_2 el que acepta el mismo lenguaje que el de r_2 , entonces

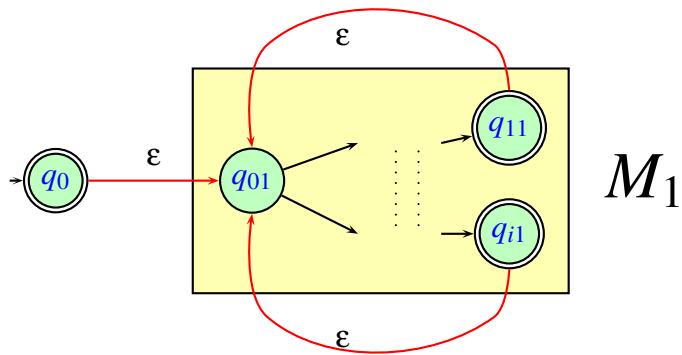
- El autómata que acepta que mismo lenguaje que el asociado a $(r_1 + r_2)$ es



- El autómata para la expresión $(r_1 r_2)$ es



- El autómata para r_1^* es

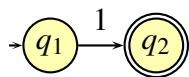


Se puede comprobar que efectivamente estos autómatas verifican las condiciones pedidas y que el procedimiento de construcción siempre acaba.

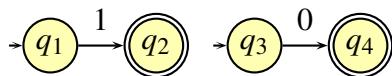
■

Ejemplo 31 Construyamos un autómata que acepta el mismo lenguaje que el asociado a la expresión regular $r = (\mathbf{0} + \mathbf{1}\mathbf{0})^*\mathbf{0}\mathbf{1}\mathbf{1}$

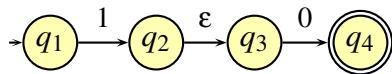
- Autómata correspondiente a **1**



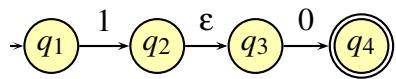
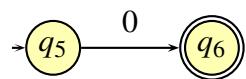
- Autómatas correspondientes a **1** y **0**



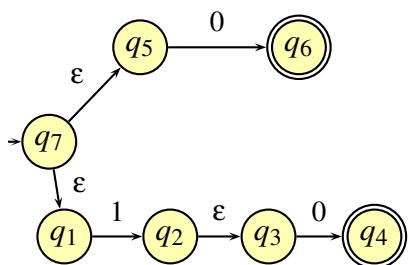
- El autómata asociado a **10** es



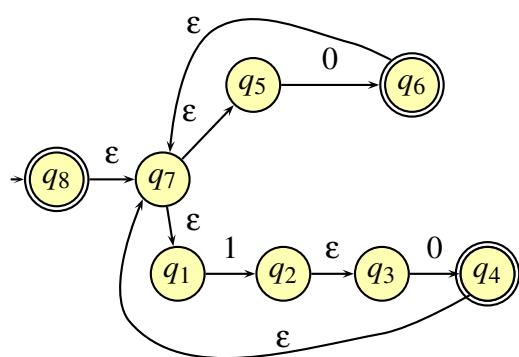
- Autómatas asociado a **10** y **0**



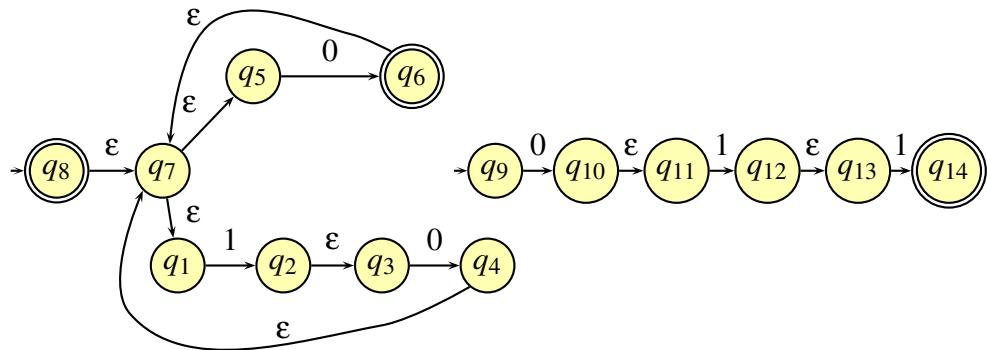
- Autómata asociado a $\mathbf{10 + 0}$



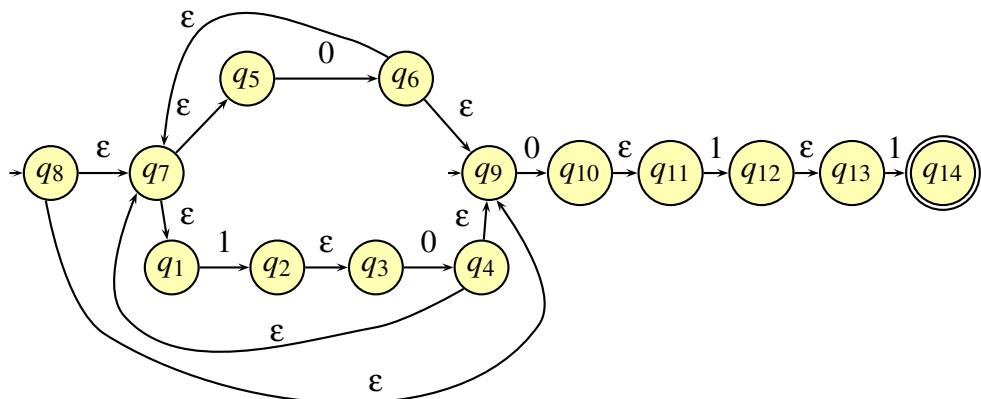
- Autómata asociado a $(\mathbf{10 + 0})^*$



- Autómatas asociado a $(\mathbf{10 + 0})^*$ y $\mathbf{011}$



- Autómatas asociado a $(\mathbf{10} + \mathbf{0})^*$ y $\mathbf{011}$



Teorema 5 Si L es aceptado por un autómata finito determinístico, entonces puede venir expresado mediante una expresión regular.

Demostración.-

Sea el autómata $M = (Q, A, \delta, q_1, F)$ donde $Q = \{q_1, \dots, q_n\}$ y q_1 es el estado inicial.

Sea R_{ij}^k el conjunto de las cadenas de A^* que premiten pasar del estado q_i al estado q_j y no pasa por ningún estado intermedio de numeración mayor que k (q_i y q_j si pueden tener numeración mayor que k).

R_{ij}^k se puede definir de forma recursiva:

$$R_{ij}^0 = \begin{cases} \{a : \delta(q_i, a) = q_j\} & \text{si } i \neq j \\ \{a : \delta(q_i, a) = q_i\} \cup \{\epsilon\} & \text{si } i = j \end{cases}$$

Para $k \geq 1$, tenemos la siguiente ecuación:

$$R_{ij}^k = R_{i(k-1)}^{k-1} (R_{(k-1)(k-1)}^{k-1})^* R_{(k-1)j}^{k-1} \cup R_{ij}^{k-1}$$

Vamos a demostrar que para todos los lenguajes R_{ij}^k existe una expresión regular r_{ij}^k que lo representa. Lo vamos a demostrar por inducción sobre k .

Para $k = 0$ es inmediato. La expresión regular r_{ij}^k puede escribirse como

$$\begin{array}{ll} \mathbf{a_1 + \dots + a_l} & \text{si } i \neq j \\ \mathbf{a_1 + \dots + a_l + \epsilon} & \text{si } i = j \end{array}$$

donde $\{a_1, \dots, a_l\}$ es el conjunto $\{a : \delta(q_i, a) = q_j\}$. Si este conjunto es vacío la expresión regular sería:

$$\begin{array}{ll} \emptyset & \text{si } i \neq j \\ \epsilon & \text{si } i = j \end{array}$$

Es claro que estas expresiones regulares representan los conjuntos R_{ij}^0 .

Supongamos ahora que es cierto para $k - 1$. Entonces sabemos que

$$R_{ij}^k = R_{i(k-1)}^{k-1} \left(R_{(k-1)(k-1)}^{k-1} \right)^* R_{(k-1)j}^{k-1} \cup R_{ij}^{k-1}$$

Por la hipótesis de inducción, existen expresiones regulares para los lenguajes R_{lm}^{k-1} , que se denotan como r_{lm}^{k-1} . De aquí se deduce que una expresión regular para R_{ij}^{k-1} viene dada por

$$r_{i(k-1)}^{k-1} (r_{(k-1)(k-1)}^{k-1})^* r_{(k-1)j}^{k-1} + r_{ij}^{k-1}$$

con lo que concluye la demostración de que existen expresiones regulares para los lenguajes R_{ij}^k . Además esta demostración nos proporciona un método recursivo para calcular estas expresiones regulares.

Finalmente, para demostrar que el lenguaje aceptado por el autómata puede venir expresado mediante una expresión regular, solo hay que observar que

$$L(M) = \bigcup_{q_j \in F} R_{1j}^n$$

Por tanto, $L(M)$ viene denotado por la expresión regular $r_{j_1} + \dots + r_{j_k}$ donde $F = \{q_{j_1}, \dots, q_{j_k}\}$. ■

Ejemplo 32 Sea el Autómata Finito Determinístico de la Figura 32. Vamos a construir la expresión regular que representa el lenguaje aceptado por este autómata.

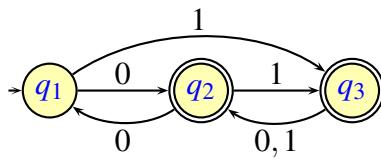


Figura 2.12: Autómata Finito Determinístico

$$r_{11}^0 = \epsilon$$

$$r_{12}^0 = 0$$

$$r_{13}^0 = 1$$

$$r_{21}^0 = 0$$

$$r_{22}^0 = \epsilon$$

$$r_{23}^0 = 1$$

$$r_{31}^0 = \emptyset$$

$$r_{32}^0 = 0 + 1$$

$$r_{33}^0 = \epsilon$$

$$r_{11}^1 = r_{11}^0 + r_{11}^0(r_{11}^0)^*r_{11}^0 = \epsilon + \epsilon(\epsilon)^*\epsilon = \epsilon$$

$$r_{12}^1 = r_{12}^0 + r_{11}^0(r_{11}^0)^*r_{12}^0 = 0 + \epsilon(\epsilon)^*0 = 0$$

$$r_{13}^1 = r_{13}^0 + r_{11}^0(r_{11}^0)^*r_{13}^0 = 1 + \epsilon(\epsilon)^*1 = 1$$

$$r_{21}^1 = r_{21}^0 + r_{21}^0(r_{11}^0)^*r_{11}^0 = 0 + 0(\epsilon)^*\epsilon = 0$$

$$r_{22}^1 = r_{22}^0 + r_{21}^0(r_{11}^0)^*r_{12}^0 = \epsilon + 0(\epsilon)^*0 = \epsilon + 00$$

$$r_{23}^1 = r_{23}^0 + r_{21}^0(r_{11}^0)^*r_{13}^0 = 1 + 0(\epsilon)^*1 = 1 + 01$$

$$r_{31}^1 = r_{31}^0 + r_{31}^0(r_{11}^0)^*r_{11}^0 = \emptyset + \emptyset(\epsilon)^*\epsilon = \emptyset$$

$$r_{32}^1 = r_{32}^0 + r_{31}^0(r_{11}^0)^*r_{12}^0 = 0 + 1 + \emptyset(\epsilon)^*0 = 0 + 1$$

$$r_{33}^1 = r_{33}^0 + r_{31}^0(r_{11}^0)^*r_{13}^0 = \epsilon + \emptyset(\epsilon)^*1 = \epsilon$$

$$r_{11}^2 = r_{11}^1 + r_{12}^1(r_{22}^1)^*r_{21}^1 = \epsilon + 0(\epsilon + 00)^*0 = (00)^*$$

$$r_{12}^2 = r_{12}^1 + r_{12}^1(r_{22}^1)^*r_{22}^1 = 0 + 0(\epsilon + 00)^*(\epsilon + 00) = 0(00)^*$$

$$r_{13}^2 = r_{13}^1 + r_{12}^1(r_{22}^1)^*r_{23}^1 = 1 + 0(\epsilon + 00)^*(1 + 01) = 0^*1$$

$$r_{21}^2 = r_{21}^1 + r_{22}^1(r_{22}^1)^*r_{21}^1 = 0 + (\epsilon + 00)(\epsilon + 00)^*0 = (00)^*0$$

$$r_{22}^2 = r_{22}^1 + r_{22}^1(r_{22}^1)^*r_{22}^1 = \epsilon + 00 + (\epsilon + 00)(\epsilon + 00)^*(\epsilon + 00) = (00)^*$$

$$r_{23}^2 = r_{23}^1 + r_{22}^1(r_{22}^1)^*r_{23}^1 = 1 + 01 + (\epsilon + 00)(\epsilon + 00)^*(1 + 01) = 0^*1$$

$$r_{31}^2 = r_{31}^1 + r_{32}^1(r_{22}^1)^*r_{21}^1 = \emptyset + (0 + 1)(\epsilon + 00)^*0 = (0 + 1)(00)^*$$

$$r_{32}^2 = r_{32}^1 + r_{32}^1(r_{22}^1)^*r_{22}^1 = 0 + 1 + (0 + 1)(\epsilon + 00)^*(\epsilon + 00) = (0 + 1)(00)^*$$

$$r_{33}^2 = r_{33}^1 + r_{32}^1(r_{22}^1)^*r_{23}^1 = \epsilon + (0 + 1)(\epsilon + 00)^*(1 + 01) = \epsilon + (0 + 1)0^*1$$

Finalmente la expresión regular para el lenguaje aceptado es:

$$r_{12}^3 + r_{13}^3 = r_{12}^2 + r_{13}^2(r_{33}^2)^*r_{32}^2 + r_{13}^2 + r_{13}^2(r_{33}^2)^*r_{33}^2 =$$

$$0(00)^* + 0^*1(\epsilon + (0+1)0^*1)^*(0+1)(00)^* + 0^*1 + 0^*1(\epsilon + (0+1)0^*1)^*(\epsilon + (0+1)0^*1) =$$

$$0(00)^* + 0^*1((0+1)0^*1)^*(0+1)(00)^* + 0^*1 + 0^*1((0+1)0^*1)^*$$

Por último señalaremos que, por abuso del lenguaje, se suele identificar una expresión regular con el lenguaje asociado. Es decir, diremos el lenguaje $(00+11)^*$, cuando, en realidad, nos estaremos refiriendo al lenguaje asociado a esta expresión regular.

2.5. Gramáticas Regulares

Estas son las gramáticas de tipo 3. Pueden ser de dos formas

- *Lineales por la derecha*.- Cuando todas las producciones tienen la forma

$$A \rightarrow uB$$

$$A \rightarrow u$$

- *Lineales por la izquierda*.- Cuando todas las producciones tienen la forma

$$A \rightarrow Bu$$

$$A \rightarrow u$$

Ejemplo 33 La gramática dada por $V = \{S, A\}$, $T = \{0, 1\}$ y las producciones

$$S \rightarrow 0A$$

$$A \rightarrow 10A$$

$$A \rightarrow \epsilon$$

es lineal por la derecha. Genera el lenguaje $0(01)^*$.

El mismo lenguaje es generado por la siguiente gramática lineal por la izquierda

$$S \rightarrow S10$$

$$S \rightarrow 0$$

Teorema 6 Si L es un lenguaje generado por una gramática regular, entonces existe un autómata finito determinístico que lo reconoce.

Demostración.- Supongamos que L es un lenguaje generado por la gramática $G = (V, T, P, S)$ que es lineal por la derecha. Vamos a construir un AFND con movimientos nulos que acepta L .

Este autómata será $M = (Q, T, \delta, q_0, F)$ donde

- $Q = \{[\alpha] : (\alpha = S) \vee (\exists A \in V, u \in T, \text{ tales que } A \rightarrow u\alpha \in P)\}$
- $q_0 = [S]$
- $F = \{[\epsilon]\}$
- δ viene definida por

- Si A es una variable

$$\delta([A], \epsilon) = \{[\alpha] : (A \rightarrow \alpha) \in P\}$$

- Si $a \in T$ y $\alpha \in (T^*V)$, entonces

$$\delta([a\alpha], a) = [\alpha]$$

La aceptación de una palabra en este autómata simula la aplicación de reglas de derivación en la gramática original. La demostración formal de que esto es así no la vamos a considerar.

En el caso de una gramática lineal por la izquierda, $G = (V, T, P, S)$, consideraremos la gramática $G' = (V, T, P', S)$ donde $P' = \{A \rightarrow \alpha : A \rightarrow \alpha \in P\}$. Es decir que invertimos la parte derecha de las producciones. La gramática resultante G' es lineal por la derecha y el lenguaje que genera es $L(G') = L(G)^{-1}$.

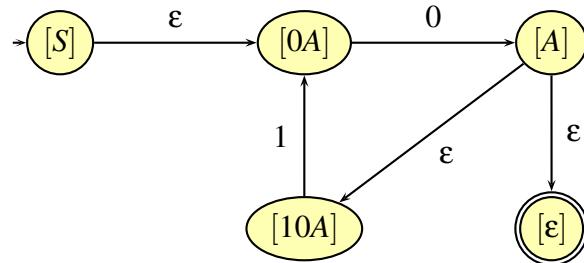
Ahora, podemos construir un autómata que acepte el lenguaje $L(G')$, siguiendo el procedimiento anterior. El autómata no determinístico correspondiente se puede transformar en uno equivalente con un solo estado final. Para ello basta con añadir un nuevo estado final, pasar a no-finales los estados finales originales y unir estos mediante una transición nula con el nuevo estado final.

El siguiente paso es invertir el autómata ya con un solo estado final, para que pase de aceptar el lenguaje $L(G') = L(G)^{-1}$ al lenguaje $L(G'^{-1}) = L(G)$. Para ello los pasos son:

- Invertir las transiciones
- Intercambiar el estado inicial y el final.

A partir de este autómata se podría construir un autómata determinístico, con lo que termina la demostración. ■

Ejemplo 34 De la gramática $S \rightarrow 0A, A \rightarrow 10A, A \rightarrow \epsilon$ se obtiene el autómata,



Ejemplo 35 La gramática lineal por la izquierda que genera el lenguaje $0(01)^*$ tiene las siguientes producciones

$$S \rightarrow S10$$

$$S \rightarrow 0$$

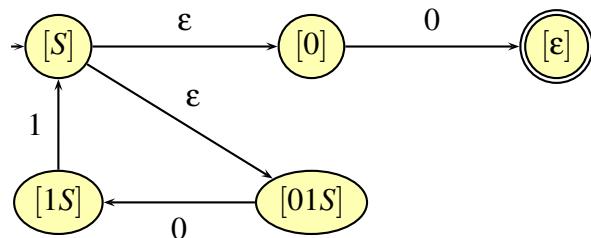
Para construir un AFND con transiciones nulas que acepte este lenguaje se dan los siguientes pasos:

1. Invertir la parte derecha de las producciones

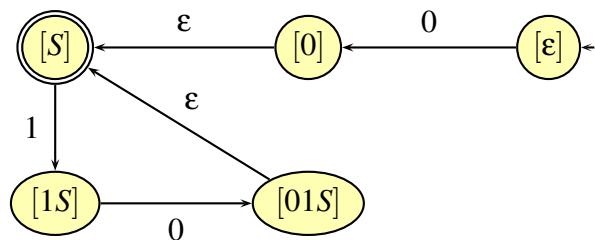
$$S \rightarrow 01S$$

$$S \rightarrow 0$$

2. Construir el AFND con transiciones nulas asociado



3. Invertimos las transiciones



Teorema 7 Si L es aceptado por un Autómata Finito Determinístico entonces L puede generarse mediante una gramática lineal por la derecha y por una lineal por la izquierda.

Demostración.- Sea $L = L(M)$ donde $M = (Q, A, \delta, q_0, F)$ es un autómata finito determinístico. Construiremos, en primer lugar, una gramática lineal por la derecha.

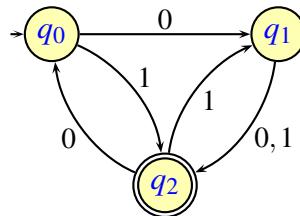
La gramática que construimos es $G = (Q, A, P, q_0)$ donde las variables son los estados, la variable inicial es q_0 y P contiene las producciones,

$$p \rightarrow aq, \quad \text{si } \delta(p, a) = q$$

$$p \rightarrow \epsilon, \quad \text{si } p \in F$$

Para el caso de una gramática lineal por la izquierda, invertimos el autómata, construimos la gramática lineal por la derecha asociada e invertimos la parte derecha de las producciones. ■

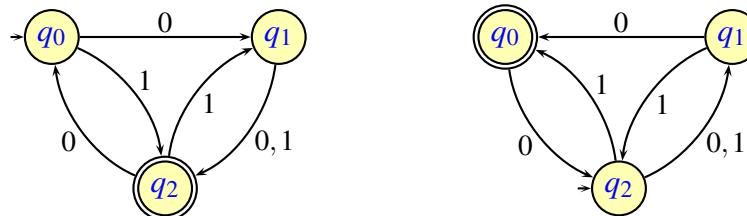
Ejemplo 36 Consideremos el autómata:



La gramática lineal por la derecha es (variable inicial q_0):

$$\begin{aligned} q_0 &\rightarrow 0q_1, & q_0 &\rightarrow 1q_2, & q_1 &\rightarrow 0q_2, & q_1 &\rightarrow 1q_2 \\ q_2 &\rightarrow 0q_0, & q_2 &\rightarrow 1q_1, & q_2 &\rightarrow \epsilon \end{aligned}$$

Si lo que queremos es una gramática lineal por la izquierda, entonces invertimos el autómata:



La gramática asociada es (variable inicial q_2):

$$\begin{aligned} q_1 &\rightarrow 0q_0, & q_2 &\rightarrow 1q_0, & q_2 &\rightarrow 0q_1, & q_2 &\rightarrow 1q_1 \\ q_0 &\rightarrow 0q_2, & q_1 &\rightarrow 1q_2, & q_0 &\rightarrow \epsilon \end{aligned}$$

Invertimos la parte derecha de las producciones, para obtener la gramática lineal por la izquierda asociada:

$$\begin{aligned} q_1 &\rightarrow q_00, & q_2 &\rightarrow q_01, & q_2 &\rightarrow q_10, & q_2 &\rightarrow q_11 \\ q_0 &\rightarrow q_20, & q_1 &\rightarrow q_21, & q_0 &\rightarrow \epsilon \end{aligned}$$

2.6. Máquinas de Estado Finito

Las máquinas de estado finito son autómatas finitos con salida. Veremos dos tipos de máquinas:

- M áquinas de Moore: con salida asociada al estado
- M áquinas de Mealy: con salida asociada a la transición

Sin embargo, ambas máquinas son equivalentes, en el sentido de que calculan las mismas funciones.

2.6.1. Máquinas de Moore

Una máquina de Moore es una sextupla

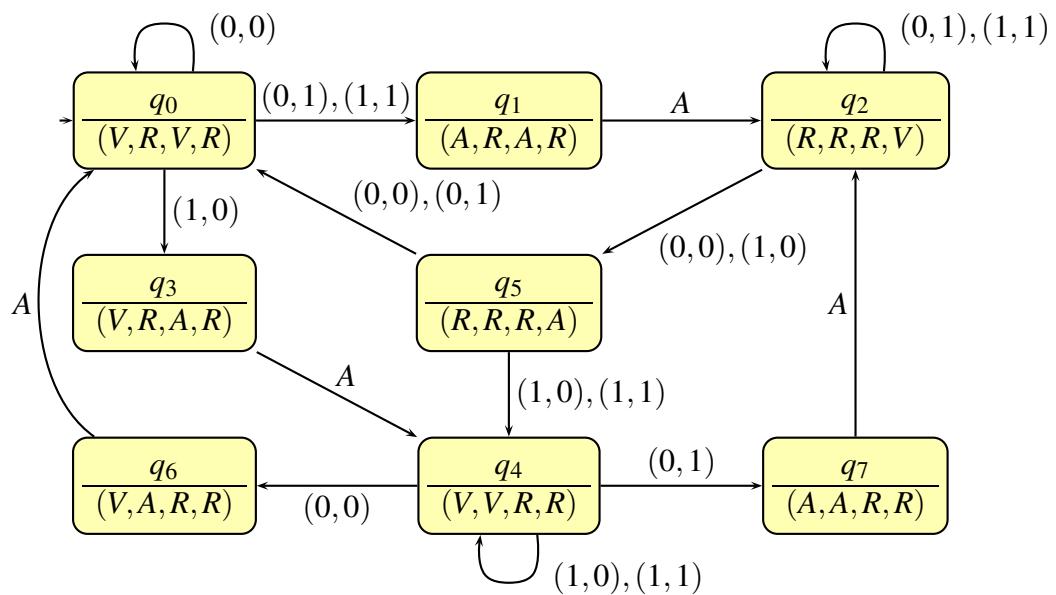
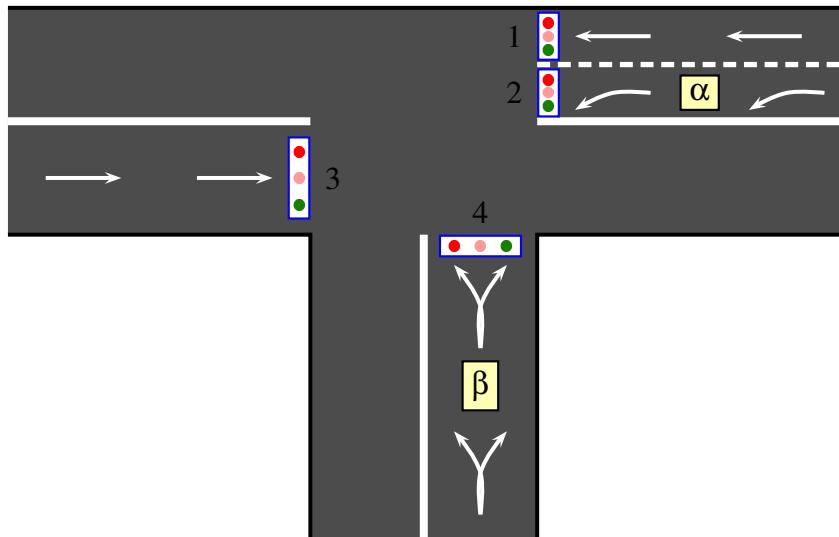
$\{(Q, A, B, \delta, \lambda, q_0)\}$ donde todos los elementos son como en los autómatas finitos determinísticos, excepto

- B alfabeto de salida
- $\lambda : Q \rightarrow B$ que es una aplicación que hace corresponder a cada estado su salida correspondiente.

No hay estados finales, porque ahora la respuesta no es 'acepta' o 'no acepta' sino que es una cadena formada por los símbolos de salida correspondientes a los estados por los que pasa el autómata.

Si el autómata lee la cadena u y pasa por los estados $q_0q_1\dots q_n$ entonces produce la salida $\lambda(q_0)\lambda(q_1)\dots\lambda(q_n)$

Es conveniente señalar que produce una salida para la cadena vacía: $\lambda(q_0)$.



Ejemplo 37 Control de semáforos en un cruce. Consideremos un cruce de carreteras, cuya estructura viene dada por la figura 37

El tráfico importante va a estar en la carretera horizontal. La otra dirección es menos densa. Por eso se han puesto dos sensores, α y β , que mandan información sobre si hay coches esperando en las colas de los semáforos 2 y 4 respectivamente: 1 si hay coches esperando y 0 si no hay coches. Estos semáforos solo se abrirán en el caso de que hay coches esperando en la cola. En caso contrario permanecerán cerrados.

Este cruce va a ser controlado por una Máquina de Moore que lee los datos de los sensores: pares (a, b) donde a es la información del sensor α y b la información del sensor β . Las salidas será n cuadruplas (a_1, a_2, a_3, a_4) donde $a_i \in \{R, A, V\}$ indicando como se colocan los semáforos.

El esquema de una máquina de Moore que controla el tráfico es el de la figura anterior. Junto a cada estado se especifica la salida correspondiente a dicho estado.

2.6.2. Máquinas de Mealy

Una Máquina de Mealy es también una sextupla $M = (Q, A, B, \delta, \lambda, q_0)$ donde todo es igual que en las máquinas de Moore, excepto que λ es una aplicación de $Q \times A$ en B ,

$$\lambda : Q \times A \rightarrow B$$

es decir, que la salida depende del estado en el que está el autómata y del símbolo leido.

Si la entrada es $a_1 \dots a_n$ y pasa por los estados q_0, q_1, \dots, q_n , la salida es

$$\lambda(q_0, a_1)\lambda(q_1, a_2) \dots \lambda(q_{n-1}, a_n)$$

Si se le suministra ϵ como entrada produce ϵ como salida.

Ejemplo 38 Supongamos una máquina codificadora que actúa de la siguiente forma:

El alfabeto de entrada es $\{0, 1\}$ y el de salida $\{0, 1\}$. La traducción viene dada por las siguientes reglas,

– Primer símbolo

$$0 \rightarrow 0$$

$$1 \rightarrow 1$$

– Siguientes símbolos

- Si el anterior es un 0

$$0 \rightarrow 0$$

$$1 \rightarrow 1$$

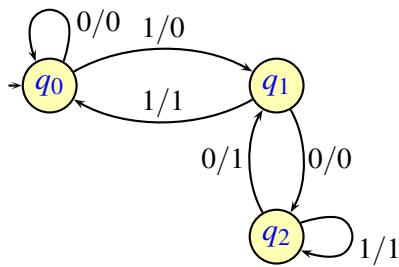


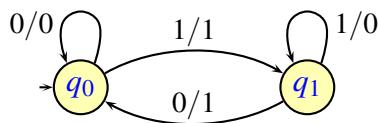
Figura 2.13: Máquina de Mealy que realiza la división entera por 3

- Si el anterior es un 1

$$0 \rightarrow 1$$

$$1 \rightarrow 0$$

La máquina de Mealy que realiza esta codificación es la siguiente



si lee 0101, la salida correspondiente es 0111.

Ejemplo 39 Una máquina de Mealy que realiza la división entera por 3 es el de la figura 2.13.

2.6.3. Equivalencia de Máquinas de Mealy y Máquinas de Moore

Una máquina de Mealy y una máquina de Moore calculan funciones análogas. Sea M una máquina de Moore y M' una máquina de Mealy. Si notamos como $T_M(u)$ y $T'_M(u)$ las salidas que producen ante una entrada u , entonces se puede comprobar que siempre, $|T_M(u)| = |T'_M(u)| + 1$. Luego, en sentido estricto, nunca pueden ser iguales las funciones que calculan una máquina de Mealy y una máquina de Moore. Sin embargo, la primera salida de una máquina de Moore es siempre la misma: la correspondiente al estado inicial. Si despreciamos esta salida, que es siempre la misma, entonces si podemos comparar las funciones calculadas por las máquinas de Mealy y de Moore.

Definición 39 Una máquina de Moore, M , y una máquina de Mealy, M' , se dicen equivalentes si para todo $u \in A^*$ $T_M(u) = bT_{M'}(u)$ donde b es la salida correspondiente al estado inicial de la máquina de Moore M .

Teorema 8 Dada una máquina de Moore, existe una máquina de Mealy equivalente.

Demostración.- Sea $M = (Q, A, B, \delta, \lambda, q_0)$ una máquina de Moore, la máquina de Mealy equivalente será $M' = (Q, A, B, \delta, \lambda', q_0)$, donde

$$\lambda'(q, a) = \lambda(\delta(q, a))$$

Es decir se le asigna a cada transición la salida del estado de llegada en la máquina de Moore. Es inmediato comprobar que ambas máquinas son equivalentes. ■

Teorema 9 Dada una máquina de Mealy, existe una máquina de Moore equivalente

Demostración.- Sea $M = (Q, A, B, \delta, \lambda, q_0)$ una máquina de Mealy.

La máquina de Moore será: $M = (Q', A, B, \delta', \lambda', q'_0)$ donde

- $Q' = Q \times B$
- $\delta'((q, b), a) = (\delta(q, a), \lambda(q, a))$
- $\lambda'(q, b) = b$
- $q'_0 = (q_0, b)$, donde $b \in B$, cualquiera.

Ejemplo 40 Dada la máquina de Mealy de la figura 2.13, aplicando el procedimiento del teorema anterior, obtenemos la máquina de Moore equivalente de la figura 2.14.

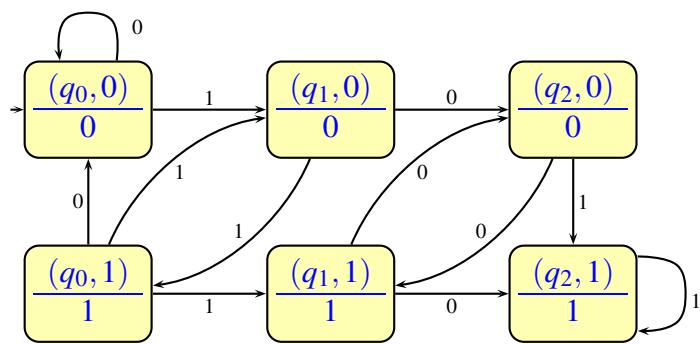


Figura 2.14: Máquina de Moore que realiza la división entera por 3

Capítulo 3

Propiedades de los Conjuntos Regulares

3.1. Lema de Bombeo

A los lenguajes aceptados por un AFD se les llama también **conjuntos regulares**. El lema de bombeo se usa para demostrar que un determinado conjunto no es regular, es decir, no puede llegar a ser aceptado por un autómata finito determinístico.

Lema 1 (Lema de Bombeo) *Sea L un conjunto regular, entonces existe un $n \in N$ tal que $\forall z \in L$, si $|z| \geq n$, entonces z se puede expresar de la forma $z = uvw$ donde*

1. $|uv| \leq n$
2. $|v| \geq 1$
3. $(\forall i \geq 0) uv^i w \in L$

además n puede ser el número de estados de cualquier autómata que acepte el lenguaje L .

Demostración.-

Sea M un autómata finito determinístico que acepta el lenguaje L .

Supongamos que el conjunto de estados es $Q = \{q_0, q_1, \dots, q_k\}$.

Hagamos $n = k + 1$.

Entonces si $z \in L$ y $|z| \geq n = k + 1$ resulta que z tiene, al menos, tantos símbolos como estados tiene el autómata.

Cuando el autómata lee la palabra z , realiza un cambio de estado por cada símbolo de dicha palabra, empezando en q_0 y terminando en un estado final q_m .

Sea z' la parte de z constituida por los n primeros símbolos de z . Consideremos el vector de los estados por los que $(q_{i_0}, q_{i_1}, \dots, q_{i_n})$ donde q_{i_0} es el estado inicial, q_0 , y $q_{i_j} = \delta(q_{i_{j-1}}, a_j)$.

Como el vector $(q_{i_0}, q_{i_1}, \dots, q_{i_n})$ tiene $n + 1$ estados y sólo hay n estados distintos, tiene que haber necesariamente un estado que se repita. Sea q_l el primer estado que se repite en este vector. Entonces consideramos:

- Sea u la parte de la palabra z' que lleva al autómata desde q_0 a la primera aparición de q_l .
- Sea v la parte de la palabra de z' , que lleva el autómata de la primera aparición de q_l a la segunda aparición de q_l .
- Sea w lo que le falta a uv para completar la palabra z .

Con esta partición se puede probar:

1. $z = uvw$

2. $|uv| \leq n$, porque uv forman parte de z' que sólo tiene n símbolos.
3. $|v| \geq 1$, porque para pasar de la primera aparición de q_l a la segunda, hay que leer al menos un símbolo.
4. $uv^i w \in L, \forall i \in N$. En efecto, esta palabra es aceptada por el autómata, porque u lleva al autómata del estado inicial q_0 al estado q_l . Cada aparición de v mantiene al autómata en el mismo estado q_l . Por último, w lleva al autómata desde q_l al estado final q_m . Como después de leer $uv^i w$ llegamos a un estado final, entonces la palabra es aceptada por el autómata y pertenece al lenguaje L .

Con esto queda demostrado el lema. ■

Para demostrar que un determinado lenguaje no es regular, basta probar que no se verifica la condición que aparece en el lema de bombeo. Para ello hay que dar los siguientes pasos:

1. Suponer un $n \in N$ arbitrario que se suponga cumpla las condiciones del lema.
2. Encontrar una palabra, z (que puede depender de n) de longitud mayor o igual que n para la que sea imposible encontrar una partición como la del lema.
3. Demostrar que una palabra z no cumple las condiciones del lema, para lo que hay que hacer:
 - a) Suponer una partición arbitraria de z , $z = uvw$, tal que $|uv| \leq n$, $|v| \geq 1$.
 - b) Encontrar un $i \in N$ tal que $uv^i w \notin L$.

Ejemplo 41 Demostrar que el lenguaje $L = \{0^k 1^k : k \geq 0\}$ no es regular.

1. Supongamos un n cualquiera
2. Sea la palabra $0^n 1^n$ que es de longitud mayor o igual que n
3. Demostremos que $0^n 1^n$ no se puede descomponer de acuerdo con las condiciones del lema:
 - a) Supongamos una partición arbitraria de $0^n 1^n = uvw$, con $|uv| \leq n$ y $|v| \geq 1$.
Como $|uv| \leq n$ resulta que v está formada de 0 solamente y como $|v| \geq 1$, v tiene, al menos un cero.

b) Basta considerar $i = 2$ y

$uv^2w = uvvw = 0^n v 1^n$ y esta no es una palabra de L , ya que si v tiene solo ceros, no hay el mismo número de ceros que de unos, y si v tiene ceros y unos, z tendrá unos antes de algún cero.

Con esto es suficiente para probar que L no es regular. Un resultado mucho mas difícil de encontrar directamente.

Ejemplo 42 Demostrar que el lenguaje $L = \{0^{j^2} : j \geq 0\}$ no es regular.

Sea un $n \in \mathbb{N}$, consideremos la palabra $z = 0^{n^2} \in L$ y de longitud mayor o igual a n .

Supongamos una descomposición cualquiera $z = uvw$. Si se verifica

- $|uv| \leq n$
- $|v| \geq 1$

entonces tenemos que $u = 0^k, v = 0^l, w = 0^{n^2-l-k}$, con $l \geq 1, l \leq n$.

Haciendo $i = 2, uv^2w = 0^k 0^{2l} 0^{n^2-l-k} = 0^{n^2+l}$.

Como $(n+1)^2 - n^2 = n^2 + 2n + 1 - n^2 = 2n + 1 > n \geq l$, tenemos que $n^2 < n^2 + l < (n+1)^2$ y $uv^2w = 0^{n^2+l} \notin L$.

Con esto el lenguaje no puede ser regular.

Ejemplo 43 El lenguaje $\{u \in \{0, 1\}^* : u = u^{-1}\}$ no es regular.

Sea un $\forall n \in \mathbb{N}$ cualquiera. Consideremos la palabra $z = 0^n 1 0^n \in L$ y que tiene longitud $2n + 1 \geq n$.

Supongamos una descomposición cualquiera: $z = uvw$.

Si se verifica

- $|uv| \leq n$
- $|v| \geq 1$

entonces tenemos que $u = 0^k, v = 0^l, w = 0^{n-k-l} 1^n 0^n$, con $l \geq 1$.

Haciendo $i = 2, uv^2w = 0^k 0^{2l} 0^{n-k-l} 1^n 0^n = 0^{n+l} 1^n 0^n \notin L$.

Como hemos encontrado un i tal que $uv^i w \notin L$, tenemos que el lenguaje no verifica la condición que aparece en el lema de bombeo y no puede ser regular.

Hay lenguajes que no son regulares y sin embargo verifican la condición que aparece en el lema de bombeo. A continuación se da un ejemplo de este caso.

Ejemplo 44 Sea $L = \{a^i b^j c^k : (i = 0) \vee (j = k)\}$ un lenguaje sobre el alfabeto $A = \{a, b, c\}$. Vamos a demostrar que se verifica la condición del lema de bombeo para $n = 2$.

En efecto si $z \in L$ y $|z| \geq 2$ entonces $z = a^i b^j c^k$ con $i = 0$ o $j = k$. Caben dos posibilidades:

- a) $i = 0$. En este caso $z = b^j c^k$ una descomposición de z se puede obtener de la siguiente forma:

$$u = \epsilon$$

v es el primer símbolo de z

w es z menos su primer símbolo

Está claro que se verifican las tres condiciones exigidas en el lema de bombeo.

1. $|uv| = 1 \leq n = 2$
2. $|v| = 1 \geq 0$
3. Si $l \geq 0$ entonces $uv^l w$ sigue siendo una sucesión de b seguida de una sucesión de c y por tanto una palabra de L .

- b) $i \geq 0$. En ese caso $z = ab^j c^j$, y una partición de esta palabra se obtiene de acuerdo con lo siguiente:

$$u = \epsilon$$

$v = a$ es el primer símbolo de z

w es z menos su primer símbolo

También aquí se verifican las tres condiciones:

1. $|uv| = 1 \leq n = 2$
2. $|v| = 1 \geq 0$
3. Si $l \geq 0$ entonces $uv^l w$ sigue siendo una sucesión de a seguida de una sucesión de b y otra de c , en la que la cantidad de b es igual que la cantidad de c , y por tanto, una palabra de L .

Es fácil intuir que este lenguaje no es regular, ya que el número de estados necesarios para tener en cuenta el número de a y de b puede hacerse ilimitado (y no sería finito).

3.2. Operaciones con Conjuntos Regulares

Ya conocemos las siguientes propiedades,

- *Unión:* Si L_1 y L_2 son conjuntos regulares, entonces $L_1 \cup L_2$ es regular.
- *Concatenación:* Si L_1 y L_2 son regulares, entonces L_1L_2 es regular.
- *Clausura de Kleene:* Si L es regular, entonces L^* es regular.

Adicionalmente, vamos a demostrar las siguientes propiedades.

Proposición 1 *Si $L \subseteq A^*$ es un lenguaje regular entonces $\overline{L} = A^* - L$ es regular.*

Demostración.- Basta con considerar que si $M = (Q, A, \delta, q_0, F)$ es un autómata finito determinístico que acepta el lenguaje L , entonces $M' = (Q, A, \delta, q_0, Q - F)$ acepta el lenguaje complementario $A^* - L$. ■

Proposición 2 *Si L_1 y L_2 son dos lenguajes regulares sobre el alfabeto A , entonces $L_1 \cap L_2$ es regular.*

Demostración.-

Es inmediato ya que $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.

Existe también una demostración constructiva. Si $M_1 = (Q_1, A, \delta_1, q_0^1, F_1)$ es un autómata finito determinístico que acepta L_1 , y $M_2 = (Q_2, A, \delta_2, q_0^2, F_2)$ es un autómata que acepta L_2 , entonces

$$M = (Q_1 \times Q_2, A, \delta, (q_0^1, q_0^2), F_1 \times F_2)$$

donde $\delta((q_i, q_j), a) = (\delta(q_i, a), \delta(q_j, a))$, acepta el lenguaje $L_1 \cap L_2$. ■

Proposición 3 *Si A y B son alfabetos y $f : A^* \longrightarrow B^*$ un homomorfismo entre ellos, entonces si $L \subseteq A^*$ es un lenguaje regular, $f(L) = \{u \in B : \exists v \in A \text{ verificando } f(v) = u\}$ es también un lenguaje regular.*

Demostración.-

Basta con comprobar que se puede conseguir una expresión regular para $f(L)$ partiendo de una expresión regular para L : Basta con substituir cada símbolo, a , de L , por la correspondiente palabra $f(a)$. ■

Ejemplo 45 Si $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ y $B = \{0, 1\}$ y f es el homomorfismo dado por

$$\begin{aligned} f(0) &= 0000, & f(1) &= 0001, & f(2) &= 0010, & f(3) &= 0011 \\ f(4) &= 0100, & f(5) &= 0101, & f(6) &= 0110, & f(7) &= 0111 \\ f(8) &= 1000, & f(9) &= 1001 \end{aligned}$$

Entonces si $L \subseteq A^*$ es el lenguaje regular dado por la expresión regular $(1+2)^*9$, entonces el lenguaje $f(L)$ también es regular y viene dado por la expresión regular $(0001+0010)^*1001$.

Proposición 4 Si A y B son alfabetos y $f : A^* \rightarrow B^*$ es un homomorfismo, entonces si $L \subseteq B^*$ es un conjunto regular, también lo es $f^{-1}(L) = \{u \in A : f(u) \in L\}$.

Demostración.-

Supongamos que $M = (Q, B, \delta, q_0, F)$ es un autómata que acepta el lenguaje L , entonces el autómata $M' = (Q, A, \bar{\delta}, q_0, F)$ donde

$$\bar{\delta}(q, a) = \delta'(q, f(a)),$$

acepta el lenguaje $f^{-1}(L)$. ■

Ejemplo 46 Si $A = B = \{0, 1\}$ y f es el homomorfismo dado por

$$f(0) = 00, \quad f(1) = 11$$

entonces el lenguaje $L = \{0^{2k}1^{2k} : k \geq 0\}$ no es regular, porque si lo fuese su imagen inversa, $f^{-1}(L) = \{0^k1^k : k \geq 0\}$ sería también regular y no lo es.

Proposición 5 Si R es un conjunto regular y L un lenguaje culaquiero, entonces el cociente de lenguajes $R/L = \{u : \exists v \in L \text{ verificando } uv \in R\}$ es un conjunto regular.

Demostración.-

Sea $M = (Q, A, \delta, q_0, F)$ un autómata finito determinístico que acepta el lenguaje R .

Entonces R/L es aceptado por el autómata

$$M' = (Q, A, \delta, q_0, F')$$

donde $F' = \{q \in Q : \exists y \in L \text{ tal que } \delta'(q, y) \in F\}$

Esta demostración no es constructiva, en el sentido de que puede que no exista un algoritmo tal que dado un estado $q \in Q$ nos diga si existe una palabra $y \in L$ tal que $\delta(q, y) \in F$. Si L es regular, se puede asegurar que dicho algoritmo existe. ■

3.3. Algoritmos de Decisión para Autómatas Finitos

El lema de bombeo sirve para encontrar algoritmos de decisión para los autómatas finitos, por ejemplo, para saber si el lenguaje que acepta un determinado autómata es vacío o no.

Teorema 10 *El conjunto de palabras aceptado por un autómata finito con n estados es*

1. *No vacío si y solo si existe una palabra de longitud menor que n que es aceptada por el autómata.*
2. *Infinito si y solo si acepta una palabra de longitud l , donde $n \leq l < 2n$.*

Demostración.-

1. La parte *si* es evidente: si acepta una palabra de longitud menor o igual que n , el lenguaje que acepta no es vacío.

Para la parte *solo si*, supongamos que el lenguaje aceptado por un autómata es no vacío. Consideremos una palabra, u , de longitud mínima aceptada por el autómata. En dicho caso, $|u| < n$, porque si $|u| \geq n$, entonces por el lema de bombeo $u = xyz$, donde $|y| \geq 1$ y xz es aceptada por el autómata, siendo $|xz| < |u|$, en contra de que u sea de longitud mínima.

2. Si $u \in L(M)$ y $n \leq |u| < 2n$, entonces por el lema de bombeo se pueden obtener infinitas palabras aceptadas por el autómata y $L(M)$ es infinito.

Recíprocamente, si $L(M)$ es infinito, habrá infinitas palabras de longitud mayor o igual que n . Sea u una palabra de longitud mínima de todas aquellas que tienen de longitud mayor o igual que n .

Entonces, u ha de tener una longitud menor de $2n$, porque si no es así, por el lema

de bombeo, u se puede descomponer $u = xyz$, donde $1 \leq |y| \leq n$, y $xz \in L(M)$. Por tanto, xz sería aceptada por el autómata, tendría una longitud mayor o igual que n , y tendría menos longitud que u . En contra de que u sea de longitud mínima.

■

Este teorema se puede aplicar a la construcción de algoritmos que determinen si el lenguaje aceptado por un autómata es vacío o no, y para ver si es o no finito. Para lo primero se tiene que comprobar si el autómata acepta alguna palabra de longitud menor o igual que n . Para lo segundo si acepta una palabra de longitud mayor o igual que n y menor que $2n$. Como en ambos

casos, el número de palabras que hay que comprobar es finito, estos algoritmos siempre paran. Sin embargo, en general será n ineficientes.

Existe otro procedimiento más eficiente para determinar si el lenguaje asociado a un autómata finito determinístico es o no finito. Lo vamos a explicar en función del diagrama de transición asociado. Consiste en eliminar previamente todos los estados que son inaccesibles desde el estado inicial (un estado es inaccesible desde el estado inicial si no existe un camino dirigido entre el estado inicial y dicho estado). Después de eliminar los estados inaccesibles se obtiene un autómata finito determinístico que acepta el mismo lenguaje que el inicial. Si en dicho autómata queda algún estado final, el lenguaje aceptado por el autómata no es vacío. Es vacío si no queda ningún estado final.

Para comprobar si es o no finito, se eliminan además los estados desde los que no se puede acceder a un estado final y las transiciones asociadas. Se obtiene así un autómata finito que puede ser no determinístico y que acepta el mismo lenguaje que el original. El lenguaje aceptado será infinito si y solo si existe un ciclo en el diagrama de transición del nuevo autómata.

Teorema 11 *Existe un algoritmo para determinar si dos autómatas finitos determinísticos aceptan el mismo lenguaje.*

Demostración.-

Sean M_1 y M_2 dos autómatas finitos determinísticos. Entonces de forma algorítmica se puede construir un autómata, M , que acepte el lenguaje

$$L(M) = (L(M_1) \cap \overline{L(M_2)}) \cup (\overline{L(M_1)} \cap L(M_2))$$

Entonces el comprobar si $L(M_1) = L(M_2)$ es equivalente a comprobar si $L(M) \neq \emptyset$, lo cual se puede hacer también de forma algorítmica. ■

3.4. Teorema de Myhill-Nerode. Minimización de Autómatas

Sea $L \subseteq A^*$ un lenguaje arbitrario. Asociado a este lenguaje L podemos definir una relación de equivalencia R_L en el conjunto A , de la siguiente forma:

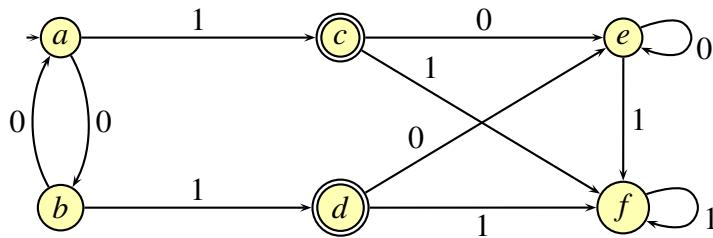
Si $x, y \in A^*$, entonces $(xR_L y)$ si y solo si $(\forall z \in A^*, (xz \in L \Leftrightarrow yz \in L))$

Esta relación de equivalencia dividirá el conjunto A^* en clases de equivalencia.

El número de clases de equivalencia se llama índice de la relación.

Ejemplo 47 *Sea $L = 0^*10^*$, entonces tenemos que*

$$00R_L000$$

Figura 3.1: Autómata que acepta el lenguaje L

$$001R_L1$$

$$01 \not R_L 0$$

$$11R_L101$$

También se puede definir una relación de equivalencia, R_M , en A^* asociada a un autómata finito determinístico $M = (Q, A, \delta, q_0, F)$ cualquiera de la siguiente forma,

Si $u, v \in A^*$, entonces $uR_M v$ si y solo si $(\delta'(q_0, u) = \delta'(q_0, v))$

Esta relación de equivalencia divide también el lenguaje A en clases de equivalencia.

Ejemplo 48 Consideremos el autómata de la Figura 3.1, que acepta el mismo lenguaje, L , del ejemplo anterior. Ahora tenemos que

$$00R_M0000$$

$$0010R_M00010$$

$$00R_M0000$$

$$0010R_M00010$$

pero R_L y R_M no son exactamente la misma relación de equivalencia.

Por ejemplo, $000R_L00$ pero $000 \not R_M 00$.

En general, esto se va a verificar siempre. Si tenemos un lenguaje regular L y un autómata finito determinístico M que acepta este lenguaje entonces,

$$uR_M v \Rightarrow uR_L v$$

pero la implicación inversa no se va a verificar siempre. Solo, como veremos más adelante, para los autómatas minimales.

El índice (n. de clases de equivalencia) de R será a lo más el número de estados del autómata finito que sean accesibles desde el estado inicial. En efecto, si $q \in Q$, y este estado es accesible, entonces definirá una clase de equivalencia:

$$[q] = \{x \in A^* : \delta'(q_0, x) = q\}$$

Esta clase es no vacía ya que $\exists x \in A$ tal que $\delta'(q_0, x) = q$ (q es accesible desde q_0).

Definición 40 Una relación de equivalencia R en A^* se dice que es invariante por la derecha para la concatenación si y solo si $((uRv) \Rightarrow (\forall z \in A^*, xzRyz))$.

Proposición 6 – Sea $L \subseteq A^*$, entonces R_L es invariante por la derecha.

– Si M es un autómata finito, entonces R_M es invariante por la derecha.

El siguiente teorema es la base para la construcción de autómatas finitos minimales.

Teorema 12 (Teorema de Myhill-Nerode) Si $L \subseteq A^*$ entonces las tres siguientes afirmaciones son equivalentes

1. L es aceptado por un autómata finito
2. L es la unión de algunas de las clases de equivalencia de una relación de equivalencia en A^* de índice finito que sea invariante por la derecha.
3. La relación de equivalencia R_L es de índice finito.

Demostración.-

Demostraremos $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$.

$1 \Rightarrow 2$ Si M es un autómata finito determinístico que acepta el lenguaje L . Entonces la relación de equivalencia R_M tiene un número finito de clases de equivalencia (tantos como estados accesibles desde el estado inicial tenga M) y además es invariante por la derecha. Solo hay que probar que siendo $[x]_M$ la clase de equivalencia asociada a x ($[x]_M = \{y : xR_M y\}$ entonces

$$L = \bigcup_{\delta(q_0, x) \in F} [x]_M$$

$2 \Rightarrow 3$ Sea R la relación de equivalencia que verifica 2. Vamos a probar que

$$xRy \Rightarrow xR_Ly$$

En efecto, si xRy , entonces si $z \in A^*$, por ser R invariante por la derecha, tenemos que $xzRyz$. Es decir, xz, yz pertenecen a la misma clase de equivalencia. Como L es la unión de ciertas clases de equivalencia de la relación R , entonces todos los elementos de una misma clase pertenecen o no pertenecen al mismo tiempo al lenguaje L . De esto se deduce que $xz \in L \Leftrightarrow yz \in L$.

En conclusión, si xRy , entonces para todo $z \in A^*$, se tiene que $xz \in L \Leftrightarrow yz \in L$. Por tanto xR_Ly .

Una vez demostrado esto, veremos que R_L tiene menos clases de equivalencia que R . Más concretamente, veremos que si $[x]_R = [y]_R$ entonces $[x]_L = [y]_L$. Pero esto es inmediato, ya que si $[x]_R = [y]_R$, entonces xRy . Y acabamos de demostrar que de aquí se deduce que xR_Ly . Y por tanto, $[x]_L = [y]_L$.

Por último, como R es de índice finito y R_L tiene menos clases que R , se deduce que R_L es de índice finito.

$3 \Rightarrow 1$ Construiremos un autómata a partir de la relación R_L .

El autómata será $M = (Q, A, \delta, q_0, F)$, donde

- $Q = \{[x] : x \in A^*\}$
- $\delta([x]_L, a) = [xa]_L$

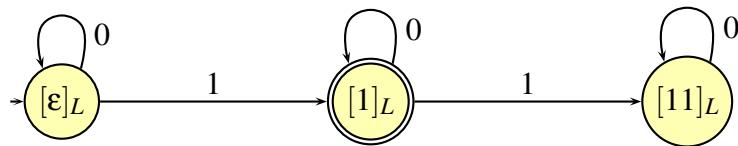
Esta definición es consistente ya que, siendo R_L invariante por la derecha, si $[x]_L = [y]_L$ entonces $[xa]_L = [ya]_L$.

- $q_0 = [\epsilon]_L$,
- $F = \{[x]_L : x \in L\}$.

F está bien definido, ya que si $[x]_L = [y]_L$, entonces xR_Ly , y por tanto, $x\epsilon \in L \Leftrightarrow y\epsilon \in L$, es decir, $x \in L \Leftrightarrow y \in L$.

Este autómata acepta el lenguaje L . En efecto, $\delta(q_0, x) = [x]_L$ y $[x]_L \in F \Leftrightarrow x \in L$. Por tanto una palabra $x \in A^*$ es aceptada cuando $x \in L$.

■

Figura 3.2: Autómata asociado a la relación R_L

Ejemplo 49 Si consideramos el lenguaje L dado por la expresión regular 0^*10^* , entonces la relación R_L divide al conjunto de las cadenas en tres clases de equivalencia:

$$[\epsilon]_L = C_1, \quad [1]_L = C_2, \quad [11]_L = C_3$$

Si consideramos el autómata asociado a este lenguaje (ver Fig. 3.1), este autómata divide A^* en seis clases de equivalencia.

$$\begin{aligned} C_a &= (00)^* & C_b &= 0(00)^* & C_c &= (00)^*1 \\ C_d &= (00)^*01 & C_e &= 0^*100^* & C_f &= 0^*10^*1(0+1)^* \end{aligned}$$

$L = C_c \cup C_d \cup C_e$, la unión de las tres clases correspondientes a los estados finales.

De acuerdo con el teorema anterior, con la relación de equivalencia R_L podemos construir un autómata finito que acepta el mismo lenguaje L . Este autómata viene dado por el diagrama de transición de la figura 3.2,

3.4.1. Minimización de Autómatas

Definición 41 Un autómata finito determinístico se dice minimal si no hay otro autómata finito determinístico que acepte el mismo lenguaje y tenga menos estados que el.

En el ejemplo que vimos después del teorema de Myhill-Nerode, a partir de la relación R_L asociada a un lenguaje L , construimos un autómata que aceptaba ese lenguaje y que tenía *pocos estados*. En particular tenía menos estados que el autómata original. El siguiente teorema demuestra que es precisamente un autómata minimal.

Teorema 13 Si L es un conjunto regular y R_L la relación de equivalencia asociada, entonces el autómata construido en el teorema anterior es minimal y único salvo isomorfismos.

Demostración.-

Comprobaremos únicamente que es minimal.

En efecto, si M' es el autómata construido según el teorema anterior y M un autómata cualquiera que acepta el lenguaje L entonces tenemos que se verifica

$$xR_My \Rightarrow xRy$$

Por tanto R_L tiene menos clases que R_M .

El teorema queda demostrado si tenemos en cuenta que:

$$\text{N. de estados de } M \geq \text{N. de clases de } R_M \geq \text{N. de clases de } R_L = \text{N. de estados de } M'$$

■

El teorema anterior nos permite encontrar el autómata minimal asociado a un lenguaje, partiendo de la relación de equivalencia de dicho lenguaje. El problema fundamental es que normalmente no tenemos dicha relación y normalmente no es fácil de calcular. Lo que solemos tener es un autómata finito determinístico que acepta el lenguaje L y lo que nos interesa es transformarlo en un autómata minimal. Esto se puede hacer de forma algorítmica. El procedimiento de minimización está basado en el hecho de que los estados del autómata minimal construido a partir de la relación R_L están formados por uniones de estados de un autómata M cualquiera que acepte el mismo lenguaje. Los estados que se pueden identificar y unir en un solo estado son los llamado estados indistinguibles, que se definen a continuación.

Definición 42 Si $M = (Q, A, \delta, q_0, F)$ es un autómata finito determinístico y q_i, q_j son dos estados de Q , se dice que q_i y q_j son indistinguibles si y solo si $\forall u \in A^*, \delta'(q_i, u) \in F \Leftrightarrow \delta'(q_j, u) \in F$.

Es decir, es indiferente estar en dos estados indistinguibles para el único objetivo que nos interesa: saber si vamos a llegar a un estado final o no.

El siguiente algoritmo identifica las parejas de estados indistinguibles. Supone que no hay estados inaccesibles.

Para el algoritmo, asociaremos a cada pareja de estados accesibles del autómata una variable booleana: *marcado*, y una lista de parejas. Al principio todas las variables booleanas están a falso y las listas vacías. Los pasos del algoritmo son como siguen,

1. Eliminar estados inaccesibles.
2. Para cada pareja de estados accesibles $\{q_i, q_j\}$
 3. Si uno de ellos es final y el otro no, hacer la variable booleana asociada igual a true.
 4. Para cada pareja de estados accesibles $\{q_i, q_j\}$
 5. Para cada símbolo a del alfabeto de entrada
 6. Calcular los estados q_k y q_l a los que evoluciona el autómata desde q_i y q_j leyendo a
 7. Si $q_k \neq q_l$ entonces
 8. Si la pareja $\{q_k, q_l\}$ está marcada entonces se marca la pareja $\{q_i, q_j\}$ y recursivamente se marcan también todas las parejas en la lista asociada.
 9. Si la pareja $\{q_k, q_l\}$ no está marcada, se añade la pareja

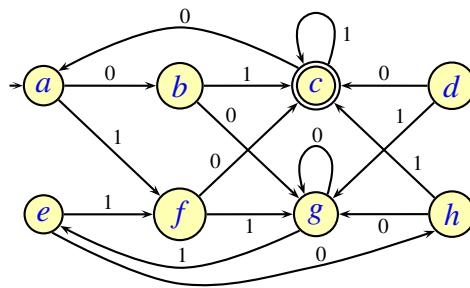


Figura 3.3: Autómata para minimizar

$\{q_i, q_j\}$ a la lista asociada a la pareja $\{q_k, q_l\}$.

Al final del algoritmo, todas las parejas de estados marcados son distinguibles y los no marcados indistinguibles.

Una vez identificados los estados indistinguibles, el autómata minimal se puede construir identificando los estados indistinguibles. Más concretamente, si el autómata original es $M = (Q, A, \delta, q_0, F)$, R es la relación de equivalencia de indistinguibilidad entre estados y $[q]$ la clase de equivalencia asociada al estado q , entonces el nuevo autómata, $M_m = (Q_m, A, \delta_m, q_0^m, F_m)$ tiene los siguientes elementos,

- $Q_m = \{[q] : q \text{ es accesible desde } q_0\}$
- $F_m = \{[q] : q \in F\}$
- $\delta_m([q], a) = [\delta(q, a)]$
- $q_0^m = [q_0]$

Se puede demostrar el siguiente teorema.

Teorema 14 Si M es un autómata finito determinístico sin estados inaccesibles, el autómata M_m construido anteriormente es minimal.

Ejemplo 50 Minimizar el autómata de la Figura 3.3.

Primero eliminamos el estado inaccesible d y obtenemos el autómata de la figura ???. A continuación organizamos los datos del algoritmo en una tabla triangular como la de la Figura 3.5. En las casillas vamos anotando la lista asociada a cada pareja y si están o no marcadas.

El resultado se puede ver en la tabla 3.6. Esta nos indica que los estados a y e son equivalentes, así como los estados b y h : $a \equiv e$, $b \equiv h$. Identificando estos estados, obtenemos el autómata de la figura 3.7.

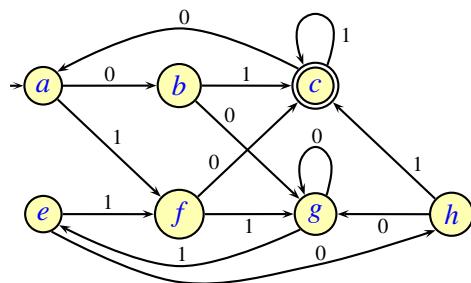


Figura 3.4: Autómata para minimizar, después de eliminar estados inaccesibles.

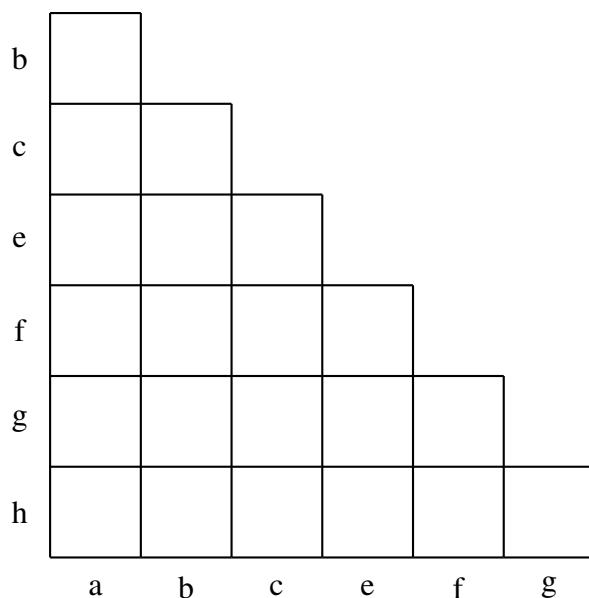


Figura 3.5: Tabla de minimización de autómata

b						
c						
e						
f						
g						
h		(e,a)				
	a	b	c	e	f	g

$\mathbf{b} \equiv \mathbf{h}, \quad \mathbf{a} \equiv \mathbf{e}$

Figura 3.6: Estado final de la tabla de minimización de autómata

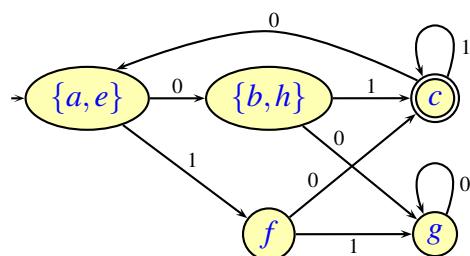


Figura 3.7: Autómata minimal

Capítulo 4

Gramáticas Libres de Contexto

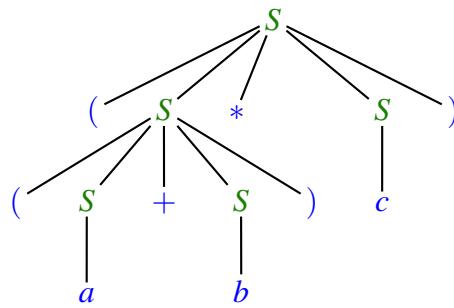


Figura 4.1: Árbol de Derivación

4.1. Árbol de Derivación y Ambigüedad

Consideremos una gramática $G = (V, T, P, S)$ donde $V = \{S\}$, $T = \{a, b, c, \emptyset, \epsilon, (,), *, +\}$ y las producciones son

$$\begin{aligned} S &\rightarrow a|b|c|\emptyset|\epsilon \\ S &\rightarrow (S + S) \\ S &\rightarrow (S * S) \end{aligned}$$

Esta es una gramática libre de contexto: En la parte izquierda de las producciones solo aparece una variable. Al lenguaje generado por esta gramática pertenece la palabra $((a + b) * c)$. Solo hay que aplicar la siguiente cadena de producciones

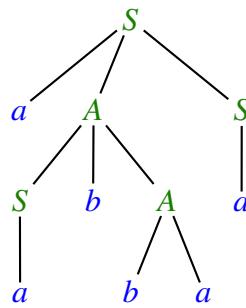
$$S \Rightarrow (S * S) \Rightarrow ((S + S) * S) \Rightarrow ((a + S) * S) \Rightarrow ((a + b) * S) \Rightarrow ((a + b) * c)$$

Una palabra nos puede ayudar a determinar si una palabra pertenece a un determinado lenguaje, pero también a algo más: a determinar la estructura sintáctica de la misma. Esta viene dada por lo que llamaremos árbol de derivación. Este se construye a partir de la cadena de derivaciones de la siguiente forma. Cada nodo del árbol va a contener un símbolo. En el nodo raíz se pone el símbolo inicial S .

Entonces, si a este nodo se le aplica una determinada regla $S \rightarrow \alpha$, entonces para cada símbolo que aparezca en α se añade un hijo con el símbolo correspondiente, situados en el orden de izquierda a derecha. Este proceso se repite para todo paso de la derivación. Si la derivación se aplica a una variable que aparece en un nodo, entonces se le añaden tantos hijos como símbolos tenga la parte derecha de la producción. Si la parte derecha es una cadena vacía, entonces se añade un solo hijo, etiquetado con ϵ . En cada momento, leyendo los nodos de izquierda a derecha se lee la palabra generada.

En nuestro caso tenemos el árbol de derivación de la Figura 4.1.

Un árbol se dice completo cuando todas las etiquetas de los nodos hojas son símbolos terminales o bien la cadena vacía.

Figura 4.2: Arbol de Derivación de *aabbaa*

Ejemplo 51 Consideremos la gramática

$$S \rightarrow aAS, \quad S \rightarrow a, \quad A \rightarrow SbA, \quad A \rightarrow SS, \quad A \rightarrow ba$$

y la palabra *aabbaa*. Esta palabra tiene una derivación que tiene asociado el árbol de la Figura 4.2

Aunque toda cadena de derivaciones lleva asociado un solo árbol, éste puede provenir de varias cadenas de derivaciones distintas. Sin embargo, siempre se pueden distinguir las siguientes:

- *Derivación por la izquierda*.- Cuando en cada paso siempre se sustituye primero la primera variable (más a la izquierda) de la palabra que tenemos.
- *Derivación por la derecha*.- Cuando en cada paso siempre se sustituye primero la última variable (más a la derecha) de la palabra que tenemos.

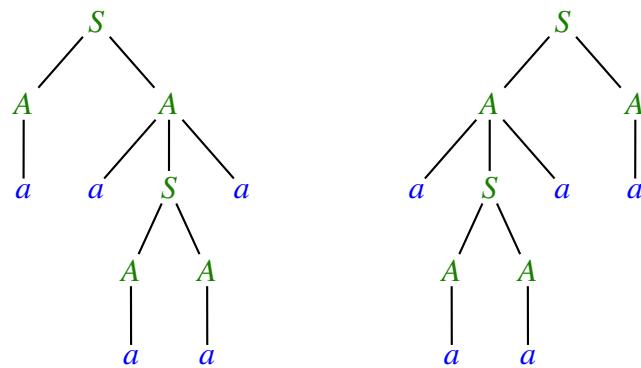
Ejemplo 52 Al árbol de derivación de la palabra *aabbaa*, le corresponden las siguientes derivaciones:

- *Derivación por la izquierda*:

$$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS \Rightarrow aabbaa$$

- *Derivación por la derecha*:

$$S \Rightarrow aAS \Rightarrow aAa \Rightarrow aSbAa \Rightarrow aSbbAa \Rightarrow aabbaa$$

Figura 4.3: Dos árboles de derivación para a^5

La existencia en una gramática de varias derivaciones para una misma palabra no produce ningún problema, siempre que den lugar al mismo árbol. Sin embargo, existe otro problema que si puede ser más grave: la ambigüedad. Una gramática se dice ambigua si existen dos árboles de derivación distintos para una misma palabra. Esto es un problema, ya que la gramática no determina la estructura sintáctica de los elementos de la palabra, no determina como se agrupan los distintos elementos para formar la palabra completa.

Ejemplo 53 *La gramática,*

$$S \rightarrow AA, \quad A \rightarrow aSa, \quad A \rightarrow a$$

es ambigua: la palabra a^5 tiene dos árboles de derivación distintos (ver Fig. 4.3).

Es suficiente que haya una palabra con dos árboles de derivación distintos para que la gramática sea ambigua.

El lenguaje generado por esta gramática no es inherentemente ambiguo. Esto quiere decir que existe otra gramática de tipo 2 no ambigua y que genera el mismo lenguaje. El lenguaje generado es $\{a^{2+3i} : i \geq 0\}$ y otra gramática no ambigua que también geneta este lenguaje es

$$S \rightarrow aa, \quad S \rightarrow aaU, \quad U \rightarrow aaaU, \quad U \rightarrow aaa$$

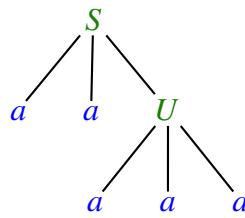
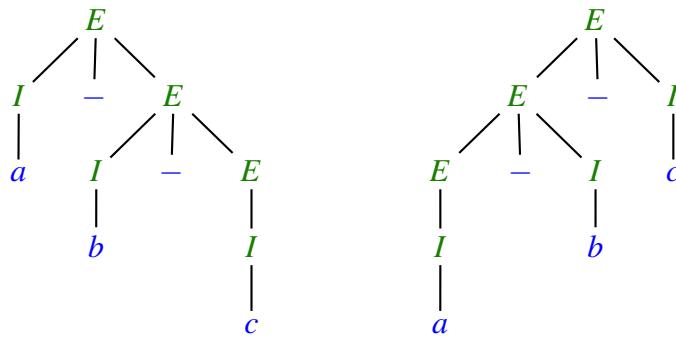
Aquí a solo tiene un árbol de derivación asociado (ver Fig. 4.4).

Ejemplo 54 *La gramática*

$$E \rightarrow I, \quad E \rightarrow I - E, \quad E \rightarrow E - I, \quad I \rightarrow a|b|c|d$$

es ambigua. La palabra $a - b - c$ admite dos árboles de derivación (ver Fig. 4.5).

Se puede eliminar la ambigüedad eliminando la producción $E \rightarrow I - E$.

Figura 4.4: Árbol de derivación para a^5 Figura 4.5: Dos árboles de derivación para $a - b - c$

Existen lenguajes inherentemente ambiguos, como es el siguiente

$$L = \{a^n b^n c^m d^m : n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n : n \geq 1, m \geq 1\}$$

Este lenguaje es de tipo 2 o libre del contexto, ya que puede generarse por la gramática

$$S \rightarrow AB, \quad A \rightarrow ab, \quad A \rightarrow aAb,$$

$$B \rightarrow cd, \quad B \rightarrow cBd,$$

$$S \rightarrow aCd, \quad C \rightarrow aCd, \quad C \rightarrow bDc,$$

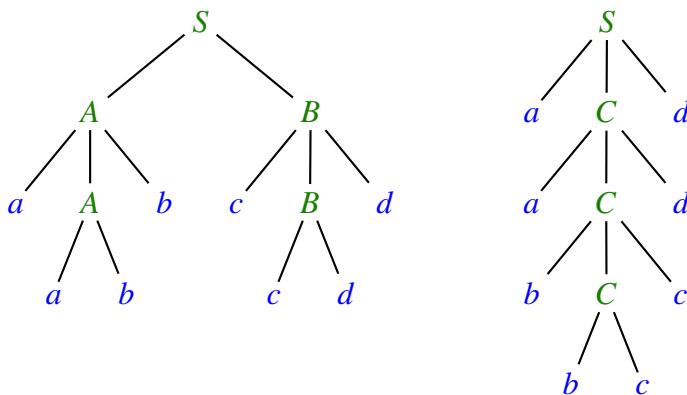
$$C \rightarrow bc, \quad D \rightarrow bDc, \quad D \rightarrow bc$$

Entonces la palabra $aabbccdd$ puede tener dos derivaciones (ver Fig. 4.6).

Aunque no lo demostraremos aquí esta ambigüedad no se puede eliminar. Pueden existir otras gramáticas de tipo 2 que generen el mismo lenguaje, pero todas serán ambiguas.

4.2. Simplificación De Las Gramáticas Libres De Contexto

Para un mismo lenguaje de tipo 2 existen muchas gramáticas libres de contexto que lo generan. Estas serán de formas muy diversas por lo que, en general se hace muy difícil trabajar

Figura 4.6: Dos árboles de derivación para $aabbccdd$

con ellas. Por este motivo interesa simplificarlas lo mas posible y definir unas formas normales para las gramáticas que las hagan mas homogéneas.

4.2.1. Eliminación de Símbolos y Producciones Inútiles

Un símbolo $X \in (V \cup T)$ se dice útil si y solo si existe una cadena de derivaciones en G tal que

$$S \Rightarrow \alpha X \beta \Rightarrow w \in T$$

es decir si interviene en la derivación de alguna palabra del lenguaje generado por la gramática. Una producción se dice útil si y solo si todos sus símbolos son útiles. Esto es equivalente a que pueda usarse en la derivación de alguna palabra del lenguaje asociado a la gramática. Esta claro que eliminando todos los símbolos y producciones inútiles el lenguaje generado por la gramática no cambia.

El algoritmo para eliminar los símbolos y producciones inútiles consta de dos pasos fundamentales:

1. Eliminar las variables desde las que no se puede llegar a una palabra de T y las producciones en las que aparezcan.
2. Eliminar aquellos símbolos que no sean alcanzables desde el estado inicial, S , y las producciones en las que estos aparezcan.

El primer paso se realiza con el siguiente algoritmo (V' es un conjunto de variables):

1. $V' = \emptyset$
2. Para cada producción de la forma $A \rightarrow w$, A se introduce en V' .
3. Mientras V' cambie
 4. Para cada producción $B \rightarrow \alpha$
 5. Si todas las variables de α pertenecen a V' ,
 B se introduce en V'
 6. Eliminar las variables que estén en V y no en V'
 7. Eliminar todas las producciones donde aparezca una variable de las eliminadas en el paso anterior

El segundo paso se realiza con el siguiente algoritmo:

- V'' y J son conjuntos de variables. J son las variables por analizar.
- T' es un conjunto de símbolos terminales

1. $J = \{S\}$
 $V'' = \{S\}$
 $T' = \emptyset$
2. Mientras $J \neq \emptyset$
 3. Extraer un elemento de $J : A$, ($J = J - \{A\}$).
 4. Para cada producción de la forma $A \rightarrow \alpha$
 5. Para cada variable B en α
 6. Si B no está en V'' añadir B a J y a V''
 7. Poner todos los símbolos terminales de α en T'
 8. Eliminar todas las variables que no estén en V'' y todos los símbolos terminales que no estén en T' .
 9. Eliminar todas las producciones donde aparezca un símbolo o variable de los eliminados

Ejemplo 55 Es importante aplicar los algoritmos anteriores en el orden especificado para que se garantice que se eliminan todos los símbolos y variables inútiles. Como ejemplo de lo anterior, supongamos que tenemos la gramática dada por

$$S \rightarrow AB, \quad S \rightarrow a, \quad A \rightarrow a$$

En el primer algoritmo se elimina B y la producción $S \rightarrow AB$.

Entonces en el segundo se elimina la variable A y la producción $A \rightarrow a$.

Sin embargo, si aplicamos primero el segundo algoritmo, entonces no se elimina nada. Al aplicar después el primero de los algoritmos se elimina B y la producción $S \rightarrow AB$. En definitiva, nos queda la gramática

$$S \rightarrow a, A \rightarrow a$$

donde todavía nos queda la variable inútil A .

Ejemplo 56 Eliminar símbolos y producciones inútiles de la gramática

$$\begin{aligned} S &\rightarrow gAe, \quad S \rightarrow aYB, \quad S \rightarrow cY, \\ A &\rightarrow bBY, \quad A \rightarrow ooC, \quad B \rightarrow dd, \\ B &\rightarrow D, \quad C \rightarrow jVB, \quad C \rightarrow gi, \\ D &\rightarrow n, \quad U \rightarrow kW, \quad V \rightarrow baXXX, \\ V &\rightarrow oV, \quad W \rightarrow c, \quad X \rightarrow fV, \quad Y \rightarrow Yhm \end{aligned}$$

Primero se aplica el algoritmo primero.

Inicialmente V' tiene las variables $V' = \{B, D, C, W\}$.

En la siguiente iteración añadimos a V' las variables que se alcanzan desde estas: A y W . V' queda igual a $\{B, D, C, W, A, U\}$.

En la siguiente $V' = \{B, D, C, W, A, U, S\}$.

En la siguiente $V' = \{B, D, C, W, A, U, S\}$.

Como V' no cambia ya se ha terminado el ciclo. Estas son las variables desde las que se alcanza una cadena de símbolos terminales. El restos de las variables: X, Y, V , son inútiles y se pueden eliminar. También se eliminan las producciones asociadas. La gramática resultante es:

$$\begin{aligned} S &\rightarrow gAe, \quad A \rightarrow ooC, \\ B &\rightarrow dd, \quad B \rightarrow D, \quad C \rightarrow gi, \\ D &\rightarrow n, \quad U \rightarrow kW, \quad W \rightarrow c \end{aligned}$$

A esta gramática le aplicamos el segundo algoritmo

$J = \{S\}, V'' = \{S\}, T' = \emptyset$

Tomando S de J y ejecutando las instrucciones del ciclo principal nos queda

$J = \{A\}, V'' = \{S, A\}, T' = \{g, e\}$

Tomando A de J , tenemos

$J = \{C\}, V'' = \{S, A, C\}, T' = \{g, e, o\}$

Sacando C de J , obtenemos

$J = \emptyset, V'' = \{S, A, C\}, T' = \{g, e, o, i\}$

Como $J = \emptyset$ se acaba el ciclo. Solo nos queda eliminar las variables inútiles: B, D, U, W , los símbolos terminales inútiles: n, k, c, d , y las producciones donde estos aparecen. La gramática queda

$$S \rightarrow gAe, \quad A \rightarrow ooC, \quad C \rightarrow gi$$

En esta gramática solo se puede generar una palabra: googige.

Si el lenguaje generado por una gramática es vacío, esto se detecta en que la variable S resulta inútil en el primer algoritmo. En ese caso se pueden eliminar directamente todas las producciones, pero no el símbolo S .

Ejemplo 57 eliminar símbolos y producciones inútiles de la gramática

$$S \rightarrow aSb, S \rightarrow ab, S \rightarrow bcD,$$

$$S \rightarrow cSE, E \rightarrow aDb, F \rightarrow abc, E \rightarrow abF$$

4.2.2. Producciones Nulas

Las producciones nulas son las de la forma $A \rightarrow \epsilon$. Vamos a tratar de eliminarlas sin que cambie el lenguaje generado por la gramática ni la estructura de los árboles de derivación. Evidentemente si $\epsilon \in L(G)$ no vamos a poder eliminarlas todas sin que la palabra nula deje de generarse. Así vamos a dar un algoritmo que dada una gramática G , construye una gramática G' equivalente a la anterior sin producciones nulas y tal que $L(G') = L(G) - \{\epsilon\}$. Es decir, si la palabra nula era generada en la gramática original entonces no puede generarse en la nueva gramática. Primero se calcula el conjunto de las variables anulables:

H es el conjunto de las variables anulables

1. $H = \emptyset$
2. Para cada producción $A \rightarrow \epsilon$, se hace $H = H \cup \{A\}$
3. Mientras H cambie
 4. Para cada producción $B \rightarrow A_1A_2\dots A_n$,
donde $A_i \in H$ para todo $i = 1, \dots, n$, se hace $H = H \cup \{B\}$

Una vez calculado el conjunto, H , de las variables anulables, se hace lo siguiente:

1. Se eliminan todas las producciones nulas de la gramática
2. Para cada producción de la gramática de la forma
 $A \rightarrow \alpha_1 \dots \alpha_n$, donde $\alpha_i \in V \cup T$.
 3. Se elimina la producción $A \rightarrow \alpha_1 \dots \alpha_n$
 4. Se añaden todas las producciones de la forma $A \rightarrow \beta_1 \dots \beta_n$

donde $\beta_i = \alpha_i$ si $\alpha_i \notin H$
 $(\beta_i = \alpha_i) \vee (\beta_i = \epsilon)$ si $\alpha \in H$
y no todos los β_i puedan ser nulos al mismo tiempo

G' es la gramática resultante después de aplicar estos dos algoritmos.

Si G generaba inicialmente la palabra nula, entonces, a partir de G' , podemos construir una gramática G'' con una sola producción nula y que genera el mismo lenguaje que G . para ello se añade una nueva variable, S' , que pasa a ser el símbolo inicial de la nueva gramática, G'' . También se añaden dos producciones:

$$S' \rightarrow S, \quad S' \rightarrow \epsilon$$

Ejemplo 58 Eliminar las producciones nulas de la siguiente gramática:

$$\begin{aligned} S &\rightarrow ABb, S \rightarrow ABC, C \rightarrow abC, \\ B &\rightarrow bB, B \rightarrow \epsilon, A \rightarrow aA, \\ A &\rightarrow \epsilon, C \rightarrow AB \end{aligned}$$

Las variables anulables después de ejecutar el paso 2 del primer algoritmo son B y A . Al ejecutar el paso 3, resulta que C y S son también anulables, es decir $H = \{A, B, C, S\}$. Al ser S anulable la palabra vacía puede generarse mediante esta gramática. En la gramática que se construye con el segundo algoritmo esta palabra ya no se podrá generar. Al ejecutar los pasos 3 y 4 del segundo algoritmo para las distintas producciones no nulas de la gramática resulta

3. Se elimina la producción $S \rightarrow ABb$
4. Se añade $S \rightarrow ABb, S \rightarrow Ab, S \rightarrow Bb$
3. Se elimina $S \rightarrow ABC$
4. Se añade $S \rightarrow ABC, S \rightarrow AB, S \rightarrow AC,$
 $S \rightarrow BC, S \rightarrow A, S \rightarrow B, S \rightarrow C$
3. Se elimina $C \rightarrow abC$
4. Se añade $C \rightarrow abC, C \rightarrow ab$
3. Se elimina $B \rightarrow bB$
4. Se añade $B \rightarrow bB, B \rightarrow b$
3. Se elimina $A \rightarrow aA$
4. Se añade $A \rightarrow aA, A \rightarrow a$
3. Se elimina $C \rightarrow AB$

4. Se añade $C \rightarrow AB, C \rightarrow A, C \rightarrow B$

En definitiva, la gramática resultante tiene las siguientes producciones:

$$\begin{aligned} S &\rightarrow ABb, \quad S \rightarrow Ab, \quad S \rightarrow Bb, \quad S \rightarrow ABC, \quad S \rightarrow AB, \\ S &\rightarrow AC, \quad S \rightarrow BC, \quad S \rightarrow A, \quad S \rightarrow B, \quad S \rightarrow C, \\ C &\rightarrow abC, \quad C \rightarrow ab, \quad B \rightarrow bB, \quad B \rightarrow b, \quad A \rightarrow aA, \\ A &\rightarrow a, \quad C \rightarrow AB, \quad C \rightarrow A, \quad C \rightarrow B \end{aligned}$$

Ejemplo 59 Sea la gramática

$$S \rightarrow aHb, \quad H \rightarrow aHb, \quad H \rightarrow \epsilon$$

La única variable anulable es H . Y la gramática equivalente, que resulta de aplicar el algoritmo 2 es:

$$S \rightarrow aHb, \quad H \rightarrow aHb, \quad S \rightarrow ab, \quad H \rightarrow ab$$

4.2.3. Producciones Unitarias

Las producciones unitarias son las que tienen la forma

$$A \rightarrow B$$

donde $A, B \in V$.

Veamos como se puede transformar una gramática G , en la que $\epsilon \notin L(G)$, en otra gramática equivalente en la que no existen producciones unitarias.

Para ello, hay que partir de una gramática sin producciones nulas.

Entonces, se calcula el conjunto H de parejas (A, B) tales que B se puede derivar a partir de A : $A \Rightarrow B$. Eso se hace con el siguiente algoritmo

1. $H = \emptyset$
2. Para toda producción de la forma $A \rightarrow B$,
la pareja (A, B) se introduce en H .
3. Mientras H cambie
4. Para cada dos parejas $(A, B), (B, C)$
5. Si la pareja (A, C) no est. en H
 (A, C) se introduce en H
6. Se eliminan las producciones unitarias
7. Para cada producción $A \rightarrow \alpha$

-
8. Para cada pareja $(B, A) \in H$
 9. Se añade una producción $B \rightarrow \alpha$

Ejemplo 60 Consideremos la gramática resultante en el ejemplo 58 del apartado anterior

$$\begin{aligned} S &\rightarrow ABb, \quad S \rightarrow Ab, \quad S \rightarrow Bb, \quad S \rightarrow ABC, \quad S \rightarrow AB, \\ S &\rightarrow AC, \quad S \rightarrow BC, \quad S \rightarrow A, \quad S \rightarrow B, \quad S \rightarrow C, \\ C &\rightarrow abC, \quad C \rightarrow ab, \quad B \rightarrow bB, \quad B \rightarrow b, \quad A \rightarrow aA, \\ A &\rightarrow a, \quad C \rightarrow AB, \quad C \rightarrow A, \quad C \rightarrow B \end{aligned}$$

El conjunto H est formado por las parejas $\{(S, A), (S, B), (S, C), (C, A), (C, B)\}$. Entonces se eliminan las producciones

$$S \rightarrow A, \quad S \rightarrow B, \quad S \rightarrow C, \quad C \rightarrow A, \quad C \rightarrow B$$

Se añaden a continuacion las producciones (no se consideran las repetidas)

$$\begin{aligned} S &\rightarrow a, \quad S \rightarrow aA, \quad S \rightarrow bB, \quad S \rightarrow b, \quad S \rightarrow abC, \\ S &\rightarrow ab, \quad C \rightarrow aA, \quad C \rightarrow a, \quad C \rightarrow bB, \quad C \rightarrow b \end{aligned}$$

Ejemplo 61 El lenguaje $L = \{a^n b^n : n \geq 1\} \cup \{a^n b a^n : n \geq 1\}$ viene generado por la siguiente gramática de tipo 2:

$$S \rightarrow A, \quad S \rightarrow B, \quad A \rightarrow ab, \quad A \rightarrow aHb$$

$$H \rightarrow ab, \quad H \rightarrow aHb, \quad B \rightarrow aBa, \quad B \rightarrow b$$

Las parejas resultantes en el conjunto H son $\{(S, A), (S, B)\}$. La gramática sin producciones unitarias equivalente es:

$$\begin{aligned} A &\rightarrow ab, \quad A \rightarrow aHb, \quad H \rightarrow ab, \quad H \rightarrow aHb, \quad B \rightarrow aBa, \\ B &\rightarrow b, \quad S \rightarrow ab, \quad S \rightarrow aHb, \quad S \rightarrow aBa, \quad S \rightarrow b. \end{aligned}$$

4.3. Formas Normales

4.3.1. Forma Normal de Chomsky

Una gramática de tipo 2 se dice que esté en forma normal de Chomsky si y solo si todas las producciones tienen la forma

$$A \rightarrow BC, \quad A \rightarrow a,$$

donde $A, B, C \in V, a \in T$.

Toda gramática de tipo 2 que no acepte la palabra vacía se puede poner en forma normal de Chomsky. Para ello lo primero que hay que hacer es suprimir las producciones nulas y unitarias. A continuación se puede ejecutar el siguiente algoritmo:

1. Para cada producción de la forma $A \rightarrow \alpha_1 \dots \alpha_n, \quad \alpha_i \in (V \cup T), n \geq 2$
2. Para cada α_i , si α_i es terminal: $\alpha_i = a \in T$
 3. Se añade la producción $C_a \rightarrow a$
 4. Se cambia α_i por C_a en $A \rightarrow \alpha_1 \dots \alpha_n$
5. Para cada producción de la forma $A \rightarrow B_1, \dots, B_m, \quad m \geq 3$
6. Se añaden $(m - 2)$ variables D_1, D_2, \dots, D_{m-2} (distintas para cada producción)
7. La producción $A \rightarrow B_1 \dots B_m$ se reemplaza por

$$A \rightarrow B_1 D_1, \quad D_1 \rightarrow B_2 D_2, \quad \dots, \quad D_{m-2} \rightarrow B_{m-1} B_m$$

Ejemplo 62 Sea la Gramática $G = (\{S, A, B\}, \{a, b\}, P, S)$ dada por las producciones

$$S \rightarrow bA \mid aB, \quad A \rightarrow bAA \mid AS \mid a, \quad B \rightarrow aBB \mid bS \mid b$$

Para pasarl a forma normal de Chomsky, en el ciclo asociado al paso 1 se añaden las producciones

$$C_a \rightarrow a, \quad C_b \rightarrow b$$

y las anteriores se transforman en

$$S \rightarrow C_b A \mid C_a B, \quad A \rightarrow C_b AA \mid AS \mid C_a, \quad B \rightarrow C_a BB \mid C_b S \mid C_b b$$

Al aplicar el paso asociado al paso 5, la gramática queda

$$\begin{aligned} S &\rightarrow C_b A \mid C_b B, \quad A \rightarrow C_b D_1 \mid AS \mid a, \quad D_1 \rightarrow AA, \\ B &\rightarrow C_a E_1 \mid C_b S \mid b, \quad E_1 \rightarrow B, \quad C_a \rightarrow a, \quad C_b \rightarrow b \end{aligned}$$

Con esto la gramática ya está en forma normal de Chomsky.

4.3.2. Forma Normal de Greibach

Una gramática se dice que est en forma normal de Greibach si y solo si todas las producciones tienen la forma

$$A \rightarrow a\alpha$$

donde $a \in T$, $\alpha \in V^*$. Toda gramática de tipo 2 que no acepte la palabra vacía se puede poner en forma normal de Greibach. Para ello hay que partir de una gramática en forma normal de Chomsky y aplicarle el siguiente algoritmo. En realidad no es necesario que la gramática esté en forma normal de Chomsky. Basta que todas las producciones sean de uno de los tipos siguientes:

- $A \rightarrow a\alpha$, $a \in T, \alpha \in V^*$.
- $A \rightarrow \alpha$, $\alpha \in V^*$.

Claro está, en una gramática en forma normal de Chomsky, todas las producciones son de alguno de estos dos tipos.

En este algoritmo se supone que el conjunto de variables de la gramática está numerado: $V = \{A_1, \dots, A_m\}$.

El algoritmo se basa en dos operaciones básicas. La primera es eliminar una producción, $A \rightarrow B\alpha$ de la gramática G , donde $A \neq B$. Esto se hace con los siguientes pasos:

1. Eliminar $A \rightarrow B\alpha$
2. Para cada producción $B \rightarrow \beta$
3. Añadir $A \rightarrow \beta\alpha$

La idea básica de esta operación es la siguiente. Si aplicamos $A \rightarrow B\alpha$, entonces, después tenemos que realizar una sustitución de B , mediante una producción $B \rightarrow \beta$. Uniendo las dos sustituciones, obtenemos $A \Rightarrow B\alpha \Rightarrow \beta\alpha$. Es decir, A se puede sustituir por $\beta\alpha$. Si eliminamos $A \rightarrow B\alpha$, esta sustitución ya no se puede hacer, por lo que si no queremos perder la posibilidad de eliminar ninguna palabra después de esta eliminación, tenemos que añadir producciones que nos permitan hacer esta sustitución. Por ese motivo añadimos $A \rightarrow \beta\alpha$, que permite realizar los dos pasos de antes en uno sólo.

La otra operación básica consiste en eliminar todas las producciones del tipo $A \rightarrow A\alpha$ donde $\alpha \in V^*$. Esto se hace siguiendo los siguientes pasos:

1. Añadir una nueva variable B_A
2. Para cada producción $A \rightarrow A\alpha$
3. Añadir $B_A \rightarrow \alpha$ y $B_A \rightarrow \alpha B_A$

4. Eliminar $A \rightarrow A\alpha$
5. Para cada producción $A \rightarrow \beta$ β no empieza por A
6. Añadir $A \rightarrow \beta B_A$

Llamemos $\text{ELIMINA}_1(A \rightarrow B\alpha)$ a la función que realiza el primer paso y $\text{ELIMINA}_2(A)$ a la función que realiza el segundo paso. Si si llama a $\text{ELIMINA}_2(A_j)$, la variable que añadimos la notaremos como B_j .

En estas condiciones vamos a realizar un algoritmo, al final del cual todas las producciones tengan una forma que corresponda a alguno de los patrones siguientes:

- $A \rightarrow a\alpha$, $a \in T, \alpha \in V^*$.
- $A_i \rightarrow A_j\alpha$, $j > i, \alpha \in V^*$.
- $B_j \rightarrow A_i\alpha$, $\alpha \in V^*$

El algoritmo es como sigue:

1. Para cada $k = 1, \dots, m$
2. Para cada $j = 1, \dots, k - 1$
 3. Para cada producción $A_k \rightarrow A_j\alpha$
 4. $\text{ELIMINA}_1(A_k \rightarrow A_j\alpha)$
5. Si existe alguna producción de la forma $A_k \rightarrow A_k\alpha$
6. $\text{ELIMINA}_2(A_k)$

A continuación se puede eliminar definitivamente la recursividad por la izquierda con el siguiente algoritmo pasando a forma normal de Greibach

1. Para cada $i = m - 1, \dots, 1$
2. Para cada producción de la forma $A_i \rightarrow A_j\alpha$, $j > i$
 3. $\text{ELIMINA}_1(A_i \rightarrow A_j\alpha)$
4. Para cada $i = 1, 2, \dots, m$
5. Para cada producción de la forma $B_j \rightarrow A_i\alpha$.
6. $\text{ELIMINA}_1(B_j \rightarrow A_i\alpha)$

El resultado del segundo algoritmo es ya una gramática en forma normal de Greibach.

Ejemplo 63 Pasar a forma normal de Greibach la gramática dada por las producciones

$$A_1 \rightarrow A_2A_3, \quad A_2 \rightarrow A_3A_1, \quad A_2 \rightarrow b, \quad A_3 \rightarrow A_1A_2, \quad A_3 \rightarrow a$$

Aplicamos ELIMINA₁ a $A_3 \rightarrow A_1A_2$.

Se elimina esta producción y se añade: $A_3 \rightarrow A_2A_3A_2$

Queda:

$$A_1 \rightarrow A_2A_3, \quad A_2 \rightarrow A_3A_1, \quad A_2 \rightarrow b,$$

$$A_3 \rightarrow a, \quad A_3 \rightarrow A_2A_3A_2$$

Aplicamos ELIMINA₁ a $A_3 \rightarrow A_2A_3A_2$

Se elimina esta producción y se añaden: $A_3 \rightarrow A_3A_1A_3A_2, \quad A_3 \rightarrow bA_3A_2$

Queda:

$$A_1 \rightarrow A_2A_3, \quad A_2 \rightarrow A_3A_1, \quad A_2 \rightarrow b,$$

$$A_3 \rightarrow a, \quad A_3 \rightarrow A_3A_1A_3A_2, \quad A_3 \rightarrow bA_3A_2$$

Aplicamos ELIMINA₂ a A_3

Se añade B_3 y las producciones $B_3 \rightarrow A_1A_3A_2, \quad B_3 \rightarrow A_1A_3A_2B_3$

Se elimina $A_3 \rightarrow A_3A_1A_3A_2$.

Se añaden las producciones: $A_3 \rightarrow aB_3, \quad A_3 \rightarrow bA_3A_2B_3$

Queda:

$$\begin{array}{llll} A_1 \rightarrow A_2A_3, & A_2 \rightarrow A_3A_1, & A_2 \rightarrow b, & A_3 \rightarrow a, \\ A_3 \rightarrow bA_3A_2 & B_3 \rightarrow A_1A_3A_2, & B_3 \rightarrow A_1A_3A_2B_3 & A_3 \rightarrow aB_3, \\ A_3 \rightarrow bA_3A_2B_3 & & & \end{array}$$

Se aplica ELIMINA₁ a $A_2 \rightarrow A_3A_1$.

Se elimina esta producción y se añaden:

$$A_2 \rightarrow aA_1, \quad A_2 \rightarrow aB_3A_1, \quad A_2 \rightarrow bA_3A_2B_3A_1, \quad A_2 \rightarrow bA_3A_2A_1$$

Queda:

$$\begin{array}{llll} A_1 \rightarrow A_2A_3, & A_2 \rightarrow b, & A_2 \rightarrow aA_1, & A_2 \rightarrow aB_3A_1, \\ A_2 \rightarrow bA_3A_2B_3A_1, & A_2 \rightarrow bA_3A_2A_1, & A_3 \rightarrow a, & A_3 \rightarrow bA_3A_2, \\ B_3 \rightarrow A_1A_3A_2, & B_3 \rightarrow A_1A_3A_2B_3 & A_3 \rightarrow aB_3, & A_3 \rightarrow bA_3A_2B_3 \end{array}$$

Se aplica ELIMINA₁ a $A_1 \rightarrow A_2A_3$.

Se elimina esta producción y se añaden:

$$\begin{aligned} A_1 &\rightarrow bA_3, \quad A_1 \rightarrow aA_1A_3, \quad A_1 \rightarrow aB_3A_1A_3, \\ A_1 &\rightarrow bA_3A_2B_3A_1A_3, \quad A_1 \rightarrow bA_3A_2A_1A_3 \end{aligned}$$

Queda:

$$\begin{array}{lll} A_2 \rightarrow b, & A_2 \rightarrow aA_1 & A_2 \rightarrow aB_3A_1, \\ A_2 \rightarrow bA_3A_2B_3A_1, & A_2 \rightarrow bA_3A_2A_1, & A_3 \rightarrow a, \\ A_3 \rightarrow bA_3A_2, & B_3 \rightarrow A_1A_3A_2, & B_3 \rightarrow A_1A_3A_2B_3, \\ A_3 \rightarrow aB_3, & A_3 \rightarrow bA_3A_2B_3, & A_1 \rightarrow bA_3, \\ A_1 \rightarrow aA_1A_3, & A_1 \rightarrow aB_3A_1A_3, & A_1 \rightarrow bA_3A_2B_3A_1A_3, \\ A_1 \rightarrow bA_3A_2A_1A_3 \end{array}$$

Se aplica ELIMINA₁ a $B_3 \rightarrow A_1A_3A_2$. Se elimina esta producción y se añaden:

$$\begin{array}{lll} B_3 \rightarrow bA_3A_3A_2, & B_3 \rightarrow aA_1A_3A_3A_2, & B_3 \rightarrow aB_3A_1A_3A_3A_2, \\ B_3 \rightarrow bA_3A_2B_3A_1A_3A_3A_2, & B_3 \rightarrow aB_3A_1A_3A_3A_2, & \end{array}$$

Queda:

$$\begin{array}{lll} A_2 \rightarrow b, & A_2 \rightarrow aA_1 & A_2 \rightarrow aB_3A_1, \\ A_2 \rightarrow bA_3A_2B_3A_1, & A_2 \rightarrow bA_3A_2A_1 & A_3 \rightarrow a, \\ A_3 \rightarrow bA_3A_2, & B_3 \rightarrow A_1A_3A_2B_3, & A_3 \rightarrow aB_3, \\ A_3 \rightarrow bA_3A_2B_3, & A_1 \rightarrow bA_3, & A_1 \rightarrow aA_1A_3, \\ A_1 \rightarrow aB_3A_1A_3, & A_1 \rightarrow bA_3A_2B_3A_1A_3, & A_1 \rightarrow bA_3A_2A_1A_3 \\ B_3 \rightarrow bA_3A_3A_2, & B_3 \rightarrow aA_1A_3A_3A_2, & B_3 \rightarrow aB_3A_1A_3A_3A_2, \\ B_3 \rightarrow bA_3A_2B_3A_1A_3A_3A_2, & B_3 \rightarrow aB_3A_1A_3A_3A_2, & \end{array}$$

Se aplica ELIMINA₁ a $B_3 \rightarrow A_1A_3A_2B_3$. Se elimina esta producción y se añaden:

$$\begin{array}{lll} B_3 \rightarrow bA_3A_3A_2B_3, & B_3 \rightarrow aA_1A_3A_3A_2B_3, & B_3 \rightarrow aB_3A_1A_3A_3A_2B_3, \\ B_3 \rightarrow bA_3A_2B_3A_1A_3A_3A_2B_3, & B_3 \rightarrow aB_3A_1A_3A_3A_2B_3, & \end{array}$$

Resultado:

$$\begin{array}{lll} A_2 \rightarrow b, & A_2 \rightarrow aA_1, & A_2 \rightarrow aB_3A_1, \\ A_2 \rightarrow bA_3A_2B_3A_1, & A_2 \rightarrow bA_3A_2A_1 & A_3 \rightarrow a, \\ A_3 \rightarrow bA_3A_2, & A_3 \rightarrow aB_3, & A_3 \rightarrow bA_3A_2B_3, \\ A_1 \rightarrow bA_3, & A_1 \rightarrow aA_1A_3, & A_1 \rightarrow aB_3A_1A_3, \\ A_1 \rightarrow bA_3A_2B_3A_1A_3, & A_1 \rightarrow bA_3A_2A_1A_3, & B_3 \rightarrow bA_3A_3A_2, \\ B_3 \rightarrow aA_1A_3A_3A_2, & B_3 \rightarrow aB_3A_1A_3A_3A_2, & B_3 \rightarrow bA_3A_2B_3A_1A_3A_3A_2, \\ B_3 \rightarrow aB_3A_1A_3A_3A_2, & B_3 \rightarrow bA_3A_3A_2B_3, & B_3 \rightarrow aA_1A_3A_3A_2B_3, \\ B_3 \rightarrow aB_3A_1A_3A_3A_2B_3, & B_3 \rightarrow bA_3A_2B_3A_1A_3A_3A_2B_3, & B_3 \rightarrow aB_3A_1A_3A_3A_2B_3 \end{array}$$

Capítulo 5

Autómatas con Pila

5.1. Definición de Autómata con Pila

Los lenguajes generados por las gramáticas libres de contexto también tienen un autómata asociado que es capaz de reconocerlos. Estos autómatas son parecidos a los autómatas finitos determinísticos, solo que ahora tendrán un dispositivo de memoria de capacidad ilimitada: una pila. A continuación daremos la definición formal de autómata con pila no determinístico (APND). Al contrario que en los autómatas finitos, los autómatas con pila no determinísticos y determinísticos no aceptan las mismas familias de lenguajes. Precisamente son los no determinísticos los asociados con los lenguajes libres de contexto. Los determinísticos aceptan una familia más restringida de lenguajes.

Definición 43 *Un autómata con pila no determinístico (APND) es una septupla $(Q, A, B, \delta, q_0, Z_0, F)$ en la que*

- Q es un conjunto finito de estados
- A es un alfabeto de entrada
- B es un alfabeto para la pila
- δ es la función de transición

$$\delta : Q \times (A \cup \{\epsilon\}) \times B \longrightarrow \wp(Q \times B^*)$$

- q_0 es el estado inicial
- Z_0 es el símbolo inicial de la pila
- F es el conjunto de estados finales

La función de transición aplica cada estado, cada símbolo de entrada (incluyendo la cadena vacía) y cada símbolo tope de la pila en un conjunto de posibles movimientos. Cada movimiento parte de un estado, un símbolo de la cinta de entrada y un símbolo tope de la pila. El movimiento en sí consiste en un cambio de estado, en la lectura del símbolo de entrada y en la substitución del símbolo tope de la pila por una cadena de símbolos.

Ejemplo 64 Sea el autómata $M = (\{q_1, q_2\}, \{0, 1, c\}, \{R, B, G\}, \delta, q_1, R, \emptyset)$ donde

$$\begin{array}{ll} \delta(q_1, 0, R) = \{(q_1, BR)\} & \delta(q_1, 1, R) = \{(q_1, GR)\} \\ \delta(q_1, 0, B) = \{(q_1, BB)\} & \delta(q_1, 1, B) = \{(q_1, GB)\} \\ \delta(q_1, 0, G) = \{(q_1, BG)\} & \delta(q_1, 1, G) = \{(q_1, GG)\} \\ \delta(q_1, c, R) = \{(q_2, R)\} & \delta(q_1, c, B) = \{(q_2, B)\} \\ \delta(q_1, c, G) = \{(q_2, G)\} & \delta(q_2, 0, B) = \{(q_2, \epsilon)\} \\ \delta(q_2, 1, G) = \{(q_2, \epsilon)\} & \delta(q_2, \epsilon, R) = \{(q_2, \epsilon)\} \end{array}$$

La interpretación es que si el autómata est en el estado q_1 y lee un 0 entonces permanece en el mismo estado y añade una B a la pila; si lo que lee es un 1, entonces añade una G; si lee una c pasa a q_2 . En q_2 se saca una B por cada 0, y una G por cada 1.

Se llama descripción instantánea o configuración de un autómata con pila a una tripleta

$$(q, u, \alpha) \in Q \times A^* \times B^*$$

en la que q es el estado en el se encuentra el autómata, u es la parte de la cadena de entrada que queda por leer y α el contenido de la pila (el primer símbolo es el tope de la pila).

Definición 44 Se dice que de la configuración $(q, au, Z\alpha)$ se puede llegar a la configuración $(p, u, \beta\alpha)$ y se escribe $(q, au, Z\alpha) \vdash (p, u, \beta\alpha)$ si y solo si

$$(p, \beta) \in \delta(q, a, Z)$$

donde a puede ser cualquier símbolo de entrada o la cadena vacía.

Definición 45 Si C_1 y C_2 son dos configuraciones, se dice que se puede llegar de C_1 a C_2 mediante una sucesión de pasos de cálculo y se escribe $C_1 \xrightarrow{*} C_2$ si y solo si existe una sucesión de configuraciones T_1, \dots, T_n tales que

$$C_1 = T_1 \vdash T \vdash T_{n-1} \vdash T_n = C_2$$

Definición 46 Si M es un APND y $u \in A^*$, se llama configuración inicial correspondiente a esta entrada, u , a (q_0, u, Z_0) donde q_0 es el estado inicial y Z_0 el símbolo inicial de la pila.

Ejemplo 65 En el caso del autómata con pila del ejemplo anterior tenemos

$$\begin{aligned} (q_1, 011c110, R) &\vdash (q_1, 11c110, BR) \vdash (q_1, 1c110, GBR) \vdash (q_1, c110, GGBR) \vdash \\ (q_2, 110, GGBR) &\vdash (q_2, 10, GBR) \vdash (q_2, 0, BR) \vdash (q_2, \epsilon, R) \vdash (q_2, \epsilon, \epsilon) \end{aligned}$$

5.1.1. Lenguaje aceptado por un autómata con pila

Existen dos criterios para determinar el lenguaje aceptado por un APND:

a) Lenguaje aceptado por estados finales:

$$L(M) = \{w \in A^* : (q_0, w, Z_0) \xrightarrow{*} (p, \epsilon, \gamma), p \in F, \gamma \in B^*\}$$

b) Lenguaje aceptado por pila vacía:

$$N(M) = \{w \in A^* : (q_0, w, Z_0) \xrightarrow{*} (p, \epsilon, \epsilon), p \in Q\}$$

En el primer caso, una palabra es aceptada, si se puede llegar a un estado final después de consumir la entrada. En el segundo criterio, los estados finales no tienen ningún significado, y una palabra se acepta si cuando se termina de leer la entrada la pila se queda vacía.

Ejemplo 66 En el caso del ejemplo 64, la palabra 011c110 es aceptada por el autómata por el criterio de pila de vacía. Es decir, $011c110 \in N(M)$. De hecho, el lenguaje aceptado por este autómata, según el criterio de la pila vacía es $N(M) = \{wcw : w \in \{0,1\}^*\}$. Para este mismo autómata $L(M) = \emptyset$, ya que al no haber estados finales, nunca se puede verificar la condición de aceptación. Una pequeña modificación del APND, lo transformaría en una autómata que aceptase el mismo lenguaje, pero ahora por el criterio de estados finales. Solo hay que considerar $M = (\{q_1, q_2, q_3\}, \{0, 1, c\}, \{R, B, G\}, \delta, q_1, R, \{q_3\})$ donde

$$\begin{array}{ll} \delta(q_1, 0, R) = \{(q_1, BR)\} & \delta(q_1, 1, R) = \{(q_1, GR)\} \\ \delta(q_1, 0, B) = \{(q_1, BB)\} & \delta(q_1, 1, B) = \{(q_1, GB)\} \\ \delta(q_1, 0, G) = \{(q_1, BG)\} & \delta(q_1, 1, G) = \{(q_1, GG)\} \\ \delta(q_1, c, R) = \{(q_2, R)\} & \delta(q_1, c, B) = \{(q_2, B)\} \\ \delta(q_1, c, G) = \{(q_2, G)\} & \delta(q_2, 0, B) = \{(q_2, \epsilon)\} \\ \delta(q_2, 1, G) = \{(q_2, \epsilon)\} & \delta(q_2, \epsilon, R) = \{(q_3, R)\} \end{array}$$

Teorema 15 a) Si M es un APND entonces existe otro autómata M' , tal que $N(M) = L(M')$

b) Si M es un APND entonces existe otro autómata M' , tal que $L(M) = N(M')$.

Demostración.-

a) Si $M = (Q, A, B, \delta, q_0, Z_0, F)$, entonces el autómata M' se construye a partir de M siguiendo los siguientes pasos:

- Se añaden dos estados nuevos, q'_0 y q_f . El estado inicial de M' será q'_0 y q_f será estado final de M' .
- Se añade un nuevo símbolo a B : Z'_0 . Este será el nuevo símbolo inicial de la pila.
- Se mantienen todas las transiciones de M , añadiéndose las siguientes:

- $\delta(q'_0, \epsilon, Z'_0) = \{(q_0, Z_0 Z'_0)\}$
- $\delta(q, \epsilon, Z'_0) = \{(q_f, Z'_0)\}, \quad \forall q \in Q$

- b) Si $M = (Q, A, B, \delta, q_0, Z_0, F)$, entonces el autómata M' se construye a partir de M siguiendo los siguientes pasos:
- Se añaden dos estados nuevos, q'_0 y q_s . El estado inicial de M' será q'_0 .
 - Se añade un nuevo símbolo a B : Z'_0 . Este será el nuevo símbolo inicial de la pila.
 - Se mantienen todas las transiciones de M , añadiéndose las siguientes:
 - $\delta(q'_0, \epsilon, Z'_0) = \{(q_0, Z_0 Z'_0)\}$
 - $\delta(q, \epsilon, H) = \{(q_s, H)\}, \quad \forall q \in F, H \in B \cup \{Z'_0\}$
 - $\delta(q_s, \epsilon, H) = \{(q_s, \epsilon)\}, \quad \forall H \in B \cup \{Z'_0\}$

■

5.2. Autómatas con Pila y Lenguajes Libres de Contexto

Teorema 16 *Si un lenguaje es generado por una gramática libre del contexto, entonces es aceptado por un Autómata con Pila No-Determinístico.*

Demostración.- Supongamos que la gramática no acepta la palabra vacía. En caso de que acepte la palabra vacía se le eliminaría y después se podría transformar el autómata para añadir la palabra vacía al lenguaje aceptado por el autómata.

Transformemos entonces la gramática a forma normal de Greibach. El autómata con pila correspondiente es $M = (\{q\}, T, V, \delta, q, S, \emptyset)$ donde la función de transición viene dada por

$$(q, \gamma) \in \delta(q, a, A) \Leftrightarrow A \rightarrow a\gamma \in P$$

Este autómata acepta por pila vacía el mismo lenguaje que genera la gramática. ■

■

Ejemplo 67 *Para la gramática en forma normal de Greibach:*

$$S \rightarrow aAA$$

$$A \rightarrow aS \mid bS \mid a$$

el autómata es

$$M = (\{q\}, \{a, b\}, \{A, S\}, \delta, q, S, \emptyset)$$

donde

$$\delta(q, a, S) = \{(q, AA)\}$$

$$\delta(q, a, A) = \{(q, S), (q, \epsilon)\}$$

$$\delta(q, b, A) = \{(q, S)\}$$

Teorema 17 Si $L = N(M)$ donde M es un APND, existe una gramática libre del contexto G , tal que $L(G) = L$.

Demostración.-

Sea $M = (Q, A, B, \delta, q_0, Z_0, \emptyset)$, tal que $L = N(M)$. La gramática $G = (V, A, P, S)$ se construye de la siguiente forma:

- V será el conjunto de los objetos de la forma $[q, C, p]$, donde $p, q \in Q$ y $C \in B$, además de la variable S que será la variable inicial.
- P será el conjunto de las producciones de la forma
 1. $S \rightarrow [q_0, Z, q]$ para cada $q \in Q$.
 2. $[q, C, q_m] \rightarrow a[p, D_1, q_1][q_1, D_2, q_2] \dots [q_{m-1}, D_m, q_m]$
donde $a \in A \cup \epsilon$, y $C, D_1, \dots, D_m \in B$ tales que

$$(p, D_1 D_2 \dots D_m) \in \delta(q, a, C)$$

(si $m = 0$, entonces la producción es $[q, A, p] \rightarrow a$).

Esta gramática genera precisamente el lenguaje $N(M)$. La idea de la demostración es que la generación de una palabra en esta gramática simula el funcionamiento del autómata no determinístico. En particular, se verificar que $[q, C, p]$ generar la palabra x si y solo si el autómata partiendo del estado q y llegando al estado p , puede leer la palabra x eliminando el símbolo C de la pila. ■

Ejemplo 68 Si partimos del autómata $M = (\{q_0, q_1\}, \{0, 1\}, \{X, Z\}, \delta, q_0, Z_0, \emptyset)$, donde

$$\begin{aligned} \delta(q_0, 0, Z_0) &= \{(q_0, XZ_0)\}, & \delta(q_1, 1, X) &= \{(q_1, \epsilon)\} \\ \delta(q_0, 0, X) &= \{(q_0, XX)\}, & \delta(q_1, \epsilon, X) &= \{(q_1, \epsilon)\} \\ \delta(q_0, 1, X) &= \{(q_1, \epsilon)\}, & \delta(q_1, \epsilon, Z_0) &= \{(q_1, \epsilon)\} \end{aligned}$$

las producciones de la gramática asociada son

$$\begin{aligned}
S &\rightarrow [q_0, Z_0, q_0] \\
S &\rightarrow [q_0, Z_0, q_1] \\
[q_0, Z_0, q_0] &\rightarrow 0[q_0, X, q_0][q_0, Z_0, q_0] \\
[q_0, Z_0, q_1] &\rightarrow 0[q_0, X, q_0][q_0, Z_0, q_1] \\
[q_0, Z_0, q_0] &\rightarrow 0[q_0, X, q_1][q_1, Z_0, q_0] \\
[q_0, Z_0, q_1] &\rightarrow 0[q_0, X, q_1][q_1, Z_0, q_1] \\
[q_0, X, q_0] &\rightarrow 0[q_0, X, q_0][q_0, X, q_0] \\
[q_0, X, q_1] &\rightarrow 0[q_0, X, q_0][q_0, X, q_1] \\
[q_0, X, q_0] &\rightarrow 0[q_0, X, q_1][q_1, X, q_0] \\
[q_0, X, q_1] &\rightarrow 0[q_0, X, q_1][q_1, X, q_1] \\
[q_0, X, q_1] &\rightarrow 1 \\
[q_1, X, q_1] &\rightarrow 1 \\
[q_1, X, q_1] &\rightarrow \epsilon \\
[q_1, Z_0, q_1] &\rightarrow \epsilon
\end{aligned}$$

Elimando símbolos y producciones inútiles queda

$$\begin{aligned}
S &\rightarrow [q_0, Z_0, q_1] \\
[q_0, Z_0, q_1] &\rightarrow 0[q_0, X, q_1][q_1, Z_0, q_1] \\
[q_0, X, q_1] &\rightarrow 0[q_0, X, q_1][q_1, X, q_1] \\
[q_1, X, q_1] &\rightarrow 1 \\
[q_1, X, q_1] &\rightarrow \epsilon \\
[q_1, Z_0, q_1] &\rightarrow \epsilon
\end{aligned}$$

Después de esta equivalencia un lenguaje independiente del contexto puede definirse indistintamente como aquel que es generado por una gramática de tipo 2 o bien como aquel que es aceptado por un autómata con pila.

5.3. Lenguajes Independientes del Contexto Deterministas

Definición 47 Un autómata con pila se dice que es **determinista** (APD) si y solo si se verifican las dos condiciones siguientes

1. $\forall q \in Q, Z \in B$, si $\delta(q, \epsilon, Z) \neq \emptyset$ entonces $\delta(q, a, Z) = \emptyset$, $\forall a \in A$.
2. $\forall a \in A \cup \epsilon, z \in B, q \in F$, $\delta(q, a, Z)$ nunca contiene más de un elemento.

El autómata del Ejemplo 64 es un autómata determinista.

La clase de lenguajes aceptados por un autómata con pila determinista es más reducida que la clase de los lenguajes aceptados por los autómatas con pila genéricos. Además, en el caso de los autómatas con pila deterministas, los dos criterios (pila vacía y estados finales) no son equivalentes.

Definición 48 Un lenguaje L se dice que es **independiente del contextos determinista** si y solo si es aceptado por un autómata con pila determinista por el criterio de estados finales.

Ejemplo 69 El lenguaje aceptado por el autómata del Ejemplo 64 era $L = \{wcw^{-1} : w \in \{0, 1\}^*\}$. El autómata era determinista, pero el criterio usado era el de estados finales. Sin embargo, el autómata se puede convertir en otro también determinista y que acepte L por el criterio de estados finales.

Sea el autómata $M = (\{q_1, q_2\}, \{0, 1, c\}, \{R, B, G\}, \delta, q_1, R, \{q_3\})$ donde

$$\begin{array}{ll} \delta(q_1, 0, R) = \{(q_1, BR)\} & \delta(q_1, 1, R) = \{(q_1, GR)\} \\ \delta(q_1, 0, B) = \{(q_1, BB)\} & \delta(q_1, 1, B) = \{(q_1, GB)\} \\ \delta(q_1, 0, G) = \{(q_1, BG)\} & \delta(q_1, 1, G) = \{(q_1, GG)\} \\ \delta(q_1, c, R) = \{(q_2, R)\} & \delta(q_1, c, B) = \{(q_2, B)\} \\ \delta(q_1, c, G) = \{(q_2, G)\} & \delta(q_2, 0, B) = \{(q_2, \epsilon)\} \\ \delta(q_2, 1, G) = \{(q_2, \epsilon)\} & \delta(q_2, \epsilon, R) = \{(q_3, \epsilon)\} \end{array}$$

El autómata tiene un sólo estado final y lo que hemos hecho es que, al mismo tiempo que dejamos la pila vacía, pasamos a ese estado final.

Lo que hemos hecho en el ejemplo anterior es siempre posible y se puede demostrar el siguiente resultado.

Teorema 18 Si L es aceptado por un autómata con pila determinista M por el criterio de pila vacía, entonces existe otro autómata con pila determinista M' que acepta el mismo lenguaje por el criterio de estados finales.

Demostración.- La demostración es la misma que la del apartado a) del Teorema 15. Sólo hay que tener en cuenta que en la transformación que se realiza, si partimos de un autómata con pila determinista M , entonces el autómata con pila que se obtiene, M' es también determinista. ■

El resultado inverso no es cierto. Hay lenguajes que son aceptados por autómatas con pila por el criterio de estados finales, que no pueden ser aceptados por el criterio de pila vacía.

Ejemplo 70 El siguiente autómata determinista acepta por el criterio de estados finales el lenguaje de las palabras del lenguaje sobre $\{0, 1\}$ con la misma cantidad de ceros que de unos, donde q_2 es el estado final y R el símbolo inicial de la pila.

$$\begin{aligned}\delta(q_1, 0, X) &= \{(q_1, XX)\} & \delta(q_1, 1, Y) &= \{(q_1, YY)\} \\ \delta(q_1, 1, X) &= \{(q_1, \epsilon)\} & \delta(q_1, 0, Y) &= \{(q_1, \epsilon)\} \\ \delta(q_1, \epsilon, R) &= \{(q_2, R)\} & \delta(q_2, 0, R) &= \{(q_1, XR)\} \\ \delta(q_2, 1, R) &= \{(q_1, YR)\}\end{aligned}$$

Sin embargo, no puede ser aceptado por un autómata determinista por el criterio de pila vacía. El problema es el siguiente: si fuese posible, después de leer 0011 se quedaría la pila vacía, pero entonces la palabra formada por 001101 no podría ser aceptada, porque al ser el autómata determinista, después de leer la primera parte de la parte, 0011, la pila se quedaría vacía y como un autómata no puede seguir funcionando con la pila vacía, la palabra completa 001101 no podría ser leída y, por tanto no es aceptada.

El razonamiento del ejemplo anterior se puede generalizar a cualquier lenguaje en el que se de la misma situación: dos palabras u y v del lenguaje distintas tal que una es un prefijo de la otra (una subcadena del principio de la palabra). De hecho esta propiedad es fundamental para caracterizar a los lenguajes aceptados por un autómata con pila deterministas por el criterio de estados finales que lo son también por el criterio de pila vacía. En lo que sigue, formalizamos lo que que acabamos de decir.

Definición 49 Un lenguaje L se dice que tiene la propiedad prefijo si no existen dos palabras $u, v \in L$ tales que $u \neq v$ y una es un prefijo de la otra.

El siguiente teorema (que enunciamos sin demostración) establece la relación entre los dos criterios de aceptación en autómatas con pila deterministas.

Teorema 19 Un lenguaje L puede ser aceptado por un autómata con pila determinista por el criterio de pila vacía si y solo si puede ser aceptado por un autómata con pila determinista por el criterio de estados finales y tiene la propiedad prefijo.

Si tenemos un lenguaje, L , independiente del contexto determinista que no cumple la propiedad prefijo es fácil transformarlo en otro lenguaje muy similar que cumple la propiedad prefijo y puede ser aceptado por un autómata con pila determinista por el criterio de pila vacía. Basta con considerar un símbolo $\$ \notin L$ y entonces el lenguaje viene dado por

$$L\{\$\} = \{u\$: u \in L\}$$

Es como añadirle un símbolo de fin de palabra (\$) a todas las palabras de L .

Finalmente, es importante que quede claro que hay lenguajes que son independientes del contexto, pero no deterministas. Como ejemplo, podemos considerar $L = \{u \in \{0, 1\}^*: u = u^{-1}\}$. No vamos a demostrar formalmente que este lenguaje no es deterministas. Intuitivamente, las palabras de este lenguaje se reconocen metiendo símbolos en la pila hasta la mitad de la palabra (primera fase), y después de la mitad se sacan símbolos de la pila (segunda fase). Si aparece el símbolo inicial de la pila se pasa a un estado final y la palabra es aceptada. El problema es que cuando se está leyendo la palabra símbolo a símbolo no se sabe cuando estamos en la mitad de la palabra y cuando hay que pasar de meter símbolos a sacar. Si permitemos que haya no determinismo esto se resuelve dando, en todo momento de la primera fase, la opción de cambiar de la primera a la segunda fase.

Capítulo 6

Propiedades de los Lenguajes Libres del Contexto

6.1. Lema de Bombeo

Comenzamos esta sección con un lema que nos da una condición necesaria que deben de cumplir todos los lenguajes libres de contexto. Nos sirve para demostrar que un lenguaje dado no es libre de contexto, comprobando que no cumple esta condición necesaria.

Lema 2 (Lema de Bombeo para lenguajes libres de contexto) *Sea L un lenguaje libre de contexto. Entonces, existe una constante n , que depende solo de L , tal que si $z \in L$ y $|z| \geq n$, z se puede escribir de la forma $z = uvwxy$ de forma que*

1. $|vx| \geq 1$
2. $|vwx| \leq n$, y
3. $\forall i \geq 0, uv^iwx^i y \in L$

Demostración.-

Sólo vamos a indicar una idea de cómo es la demostración. Supongamos que la gramática no tiene producciones nulas ni unitarias (si existiesen siempre se podrían eliminar).

Supongamos un árbol de derivación de una palabra u generada por la gramática. Es fácil ver que si la longitud de u es suficientemente grande, en su árbol de derivación debe de existir un camino de longitud mayor que el número de variables. Sea N un número que garantice que se verifica esta propiedad. En dicho camino, al menos debe de haber una variable repetida. Supongamos que esta variable es A , y que la figura 6.1 representa el árbol de derivación y dos apariciones consecutivas de A . ■

Ejemplo 71 Vamos a utilizar el lema de bombeo para probar que el lenguaje $L = \{a^i b^i c^i \mid i \geq 1\}$ no es libre de contexto.

Supongamos que L fuese libre de contexto y sea n la constante especificada en el Lema de Bombeo. Consideremos la palabra $z = a^n b^n c^n \in L$, que tiene una longitud mayor que n . Consideremos que z se puede descomponer de la forma $z = uvxy$, verificando las condiciones del lema de bombeo.

Como $|vwx| \leq n$, no es posible para vx tener símbolos a y c al mismo tiempo: entre la última a y la primera c hay n símbolos. En estas condiciones se pueden dar los siguientes casos:

- $|vx|$ contiene solamente símbolos a . En este caso para $i = 0$, $uv^0wx^0y = uw$ debería de pertenecer a L por el lema de bombeo. Pero uw contiene n símbolos b , n símbolos c , menos de n símbolos a , con lo que no podría pertenecer a L y se obtiene una contradicción.

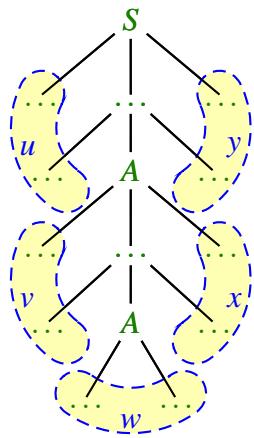


Figura 6.1: Árbol de Derivación en el Lema de Bombeo

- $|vx|$ contiene solamente símbolos b . Se llega a una contradicción por un procedimiento similar al anterior.
- $|vx|$ contiene solamente símbolos c . Se llega a una contradicción por un procedimiento similar.
- $|vx|$ contiene símbolos a y b . En este caso, uwv tendría más símbolos c que a o b , con lo que se llegaría de nuevo a una contradicción.
- $|vx|$ contiene símbolos b y c . En este caso, uwv tendría más símbolos a que b o c , con lo que se llegaría también a una contradicción.

En todo caso se llega a una contradicción y el lema de bombeo no puede cumplirse, con lo que L no puede ser libre de contexto.

Es importante señalar que el lema de bombeo no es una condición suficiente. Es solo necesaria. Así si un lenguaje verifica la condición del lema de bombeo no podemos garantizar que sea libre de contexto. Un ejemplo de uno de estos lenguajes es

$$L = \{a^i b^j c^k d^l | (i=0) \vee (j=k=l)\}$$

Ejemplo 72 Demostrar que el lenguaje $L = \{a^i b^j c^i d^j : i, j \geq 0\}$ no es libre de contexto.

Ejemplo 73 Demostrar que el lenguaje $L = \{a^i b^j c^k : i \geq j \geq k \geq 0\}$ no es libre de contexto.

6.2. Propiedades de Clausura de los Lenguajes Libres de Contexto

Teorema 20 Los lenguajes libres de contexto son cerrados para las operaciones:

- Unión
- Concatenación
- Clausura

Demostración.-

Sean $G_1 = (V_1, T_1, P_1, S_1)$ y $G_2 = (V_2, T_2, P_2, S_2)$ dos gramáticas libres de contexto y L_1 y L_2 los lenguajes que generan. Supongamos que los conjuntos de variables son disjuntos. Demostremos que los lenguajes $L_1 \cup L_2$, $L_1 L_2$ y L_1^* son libres de contexto, encontrando gramáticas de tipo 2 que los generen.

- $L_1 \cup L_2$. Una gramática que genera este lenguaje es $G_3 = (V_1 \cup V_2 \cup \{S_3\}, T_1 \cup T_2, P_3, S_3)$, donde S_3 es una nueva variable, y $P_3 = P_1 \cup P_2$ más las producciones $S_3 \rightarrow S_1$ y $S_3 \rightarrow S_2$.
- $L_1 L_2$. Una gramática que genera este lenguaje es $G_4 = (V_1 \cup V_2 \cup \{S_4\}, T_1 \cup T_2, P_4, S_4)$, donde S_4 es una nueva variable, y $P_4 = P_1 \cup P_2$ más la producción $S_4 \rightarrow S_1 S_2$.
- L_1^* . Una gramática que genera este lenguaje es $G_5 = (V_1 \cup \{S_5\}, T_1, P_5, S_5)$, donde P_5 es P_1 más las producciones $S_5 \rightarrow S_1 S_5$ y $S_5 \rightarrow \epsilon$.

■

Algunas propiedades de clausura de los lenguajes regulares no se verifican en la clase de los lenguajes libres de contexto, como las que expresan el siguiente teorema y corolario.

Teorema 21 La clase de los lenguajes libres de contexto no es cerrada para la intersección.

Demostración.- Sabemos que el lenguaje $L = \{a^i b^i c^i \mid i \geq 1\}$ no es libre de contexto. Por otra parte los lenguajes $L_2 = \{a^i b^i c^j \mid i \geq 1 \text{ y } j \geq 1\}$ y $L_3 = \{a^i b^j c^j \mid i \geq 1 \text{ y } j \geq 1\}$ si lo son. El primero de ellos es generado por la gramática:

$$S \rightarrow AB$$

$$A \rightarrow aAb \mid ab$$

$$B \rightarrow cB \mid c$$

y el segundo, por la gramática:

$$S \rightarrow CD$$

$$C \rightarrow aC|a$$

$$D \rightarrow bDc|bc$$

Como $L_2 \cap L_3 = L_1$, se deduce que la clase de lenguajes libres de contexto no es cerrada para la intersección. ■

Corolario 1 *La clase de lenguajes libres de contexto no es cerrada para el complementario.*

Demostración.-

Es inmediato, ya que como la clase es cerrada para la unión, si lo fuese para el complementario, se podría demostrar, usando las leyes DeMorgan que lo es también para la intersección. ■

6.3. Algoritmos de Decisión para los Lenguajes Libres de Contexto

Existen una serie de problemas interesantes que se pueden resolver en la clase de los lenguajes libres de contexto. Por ejemplo, existen algoritmos que nos dicen si un Lenguaje Libre de Contexto (dado por una gramática de tipo 2 o un autómata con pila no determinístico) es vacío, finito o infinito. Sin embargo, en la clase de lenguajes libres de contexto comienzan a aparecer algunas propiedades indecidibles. A continuación, veremos algoritmos para las propiedades decidibles y mencionaremos algunas propiedades indecidibles importantes.

Teorema 22 *Existen algoritmos para determinar si un lenguaje libre de contexto es*

- a) vacío
- b) finito
- c) infinito

Demostración.-

- a) De hecho, ya hemos visto un algoritmo para determinar si el lenguaje generado por una gramática de tipo 2 es vacío. En la primera parte del algoritmo para eliminar símbolos y producciones inútiles de una gramática, se determinaban las variables que podían generar una cadena formada exclusivamente por símbolos terminales. El lenguaje generado es vacío si y solo si la variable inicial S es eliminada: no puede generar una palabra de símbolos terminales.
- b) y c) Para determinar si el lenguaje generado por una gramática de tipo 2 es finito o infinito pasamos la gramática a forma normal de Chomsky, sin símbolos ni producciones inútiles. En estas condiciones todas las producciones son de la forma:

$$A \rightarrow BC, A \rightarrow a$$

Se construye entonces un grafo dirigido en el que los vértices son las variables y en el que para cada producción de la forma $A \rightarrow BC$ se consideran dos arcos: uno de A a B y otro de A a C . Se puede comprobar que el lenguaje generado es finito si y solo si el grafo construido de esta forma no tiene ciclos dirigidos.

■

Ejemplo 74 Consideremos la gramática con producciones,

$$S \rightarrow AB$$

$$A \rightarrow BC|a$$

$$B \rightarrow CC|b$$

$$C \rightarrow a$$

El grafo asociado es el de la figura 6.2. No tiene ciclos y el lenguaje es finito. Si añadimos la producción $C \rightarrow AB$, el grafo tiene ciclos (figura 6.3) y el lenguaje generado es infinito.

6.3.1. Algoritmos de Pertenencia

Estos algoritmos tratan de resolver el siguiente problema: dada una gramática de tipo 2, $G = (V, T, P, S)$ y una palabra $u \in T^*$, determinar si la palabra puede ser generada por la gramática. Esta propiedad es indecidible en la clase de lenguajes recursivamente enumerables, pero es posible encontrar algoritmos para la clase de lenguajes libres de contexto.

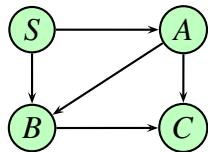


Figura 6.2: grafo asociado a una gramática de tipo 2 con lenguaje finito

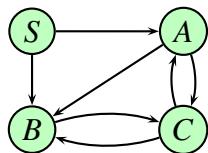


Figura 6.3: grafo asociado a una gramática de tipo 2 con lenguaje infinito

Un algoritmo simple, pero ineficiente se aplica a gramáticas en forma normal de Greibach (si una gramática no está en esta forma se pasa, teniendo en cuenta que hemos podido dejar de aceptar la palabra vacía). La pertenencia de una palabra no vacía se puede comprobar en esta forma normal de Greibach de la siguiente forma: Como cada producción añade un símbolo terminal a la palabra generada, sabemos que una palabra, u , de longitud $|u|$ ha de generarse en $|u|$ pasos. El algoritmo consistiría en enumerar todas las generaciones por la izquierda de longitud $|u|$, tales que los símbolos que se vayan generando coincidan con los de la palabra u , y comprobar si alguna de ellas llega a generar la palabra u en su totalidad. Este algoritmo para, ya que el número de derivaciones por la izquierda de una longitud dada es finito. Sin embargo puede ser muy ineficiente (exponencial en la longitud de la palabra). Para comprobar la pertenencia de la palabra vacía se puede seguir el siguiente procedimiento:

- Si no hay producciones nulas, la palabra vacía no pertenece.
- Si hay producciones nulas, la palabra vacía pertenece si y solo si al aplicar el algoritmo que elimina las producciones nulas, en algún momento hay que eliminar la producción $S \rightarrow \epsilon$.

El Algoritmo de Cocke-Younger-Kasami

Existen algoritmos de pertenencia con una complejidad $O(n^3)$, donde n es la longitud de la palabra de la que se quiere comprobar la pertenencia. Nosotros vamos a ver el algoritmo

de Cocke-Younger-Kasami (CYK). Este algoritmo se aplica a palabras en forma normal de Chomsky. Este consta de los siguientes pasos (n es la longitud de la palabra).

1. Para $i = 1$ hasta n
 2. Calcular $V_{i1} = \{A \mid A \rightarrow a \text{ es una produccion y el simbolo } i\text{-esimo de } u \text{ es } a\}$
 3. Para $j = 2$ hasta n
 4. Para $i = 1$ hasta $n - j + 1$
 5. $V_{ij} = \emptyset$
 6. Para $k = 1$ hasta $j - 1$
- $$V_{ij} = V_{i1} \cup \{A \mid A \rightarrow BC \text{ es una produccion, } B \in V_{ik} \text{ y } C \in V_{i+k, j-k}\}$$

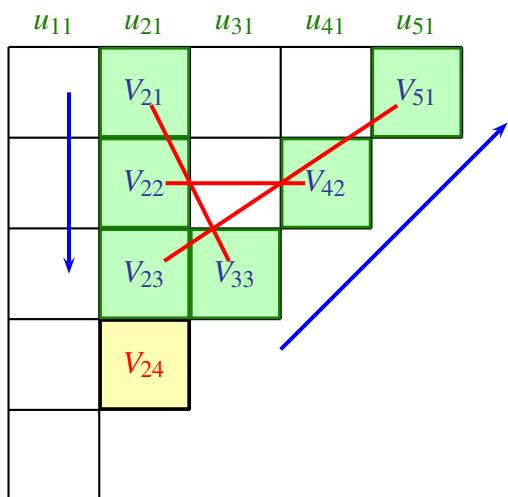
Este algoritmo calcula para todo i, j ($i \in \{1, \dots, n\}, j \leq n - i + 1$), el conjunto de variables V_{ij} que generan u_{ij} , donde u_{ij} es la subcadena de u que comienza en el símbolo que ocupa la posición i y que contiene j símbolos. La palabra u será generada por la gramática si la variable inicial S pertenece al conjunto V_{1n} .

Los cálculos se pueden organizar en una tabla como la de la siguiente figura (para una palabra de longitud 5):

u_{11}	u_{21}	u_{31}	u_{41}	u_{51}
V_{11}	V_{21}	V_{31}	V_{41}	V_{51}
V_{12}	V_{22}	V_{32}	V_{42}	
V_{13}	V_{23}	V_{33}		
V_{14}	V_{24}			
V_{15}				

En ella, cada $V_{i,j}$ se coloca en una casilla de la tabla. En la parte superior se ponen los símbolos de la palabra para la que queremos comprobar la pertenencia. La ventaja es que es fácil localizar los emparejamientos de variables que hay que comprobar para calcular cada conjunto V_{ij} . Se comienza en la casilla que ocupa la misma columna y está en la parte superior de la tabla, y la casilla que está en la esquina superior derecha, emparejando todas las variables de estas dos casillas. A continuación elegimos como primera casilla la que está justo debajo de la

primera anterior, y como segunda casilla la que ocupa la esquina superior derecha de la segunda anterior. Este proceso se continúa hasta que se eligen como primera casilla todas las que están encima de la que se está calculando. La siguiente figura ilustra el proceso que se sigue en las emparejamientos (para un elemento de la cuarta fila, en una palabra de longitud 5).



Ejemplo 75 Consideremos la gramática libre de contexto dada por las producciones

$$S \rightarrow AB|BC$$

$$A \rightarrow BA|a$$

$$B \rightarrow CC|b$$

$$C \rightarrow AB|a$$

Comprobar la pertenencia de las palabras *baaba*, *aaaaaa*, *aaaaaaaa* al lenguaje generado por la gramática.

El Algoritmo de Early

El algoritmo de Early es también de complejidad $O(n^3)$ en el caso general, pero es lineal para gramáticas $LR(1)$, que son las que habitualmente se emplean para especificar la sintaxis de los lenguajes de programación.

Al contrario que el algoritmo de Cocke-Younger-Kasami que trata de obtener las palabras de abajo hacia arriba (desde los símbolos terminales al símbolo inicial, el algoritmo de Early comenzará en el símbolo inicial (funciona de arriba hacia abajo).

Sea G una gramática con símbolo inicial S y que no tenga producciones nulas ni unitarias.

Supondremos que $u[i..j]$ es la subcadena de u que va de la posición i a la posición j . El algoritmo producirá registros de la forma (i, j, A, α, β) , donde i y j son enteros y $A \rightarrow \alpha\beta$ es una producción de la gramática. La existencia de un registro indicará un hecho y un objetivo. El hecho es que $u[i+1..j]$ es derivable a partir de α y el objetivo es encontrar todos los k takes que β deriva a $u[j+1..k]$. Si encontramos uno de estos k sabemos que A deriva $u[i+1..k]$.

Para cada j , $REGISTROS[j]$ contendrá todos los registros existentes de la forma (i, j, A, α, β) .

El algoritmo consta de los siguientes pasos:

P1 *Inicialización*.- Sea

- $REGISTROS[0] = \{(0, 0, S, \epsilon, \beta) : S \rightarrow \beta \text{ es una producción}\}$
- $REGISTROS[j] = \emptyset$ para $j = 1, \dots, n$.
- $j = 0$

P2 *Clausura*.- Para cada registro $(i, j, A, \alpha, B\gamma)$ en $REGISTROS[j]$ y cada producción $B \rightarrow \delta$, crear el registro $(j, j, B, \epsilon, \delta)$ e insertarlo en $REGISTROS[j]$. Repetir la operación recursivamente para los nuevos registros insertados.

P3 *Avance*.- Para cada registro $(i, j, A, \alpha, c\gamma)$ en $REGISTROS[j]$, donde c es un símbolo terminal que aparece en la posición $j+1$ de u , crear $(i, j+1, A, \alpha c, \gamma)$ e insertarlo en $REGISTROS[j+1]$.

Hacer $j = j + 1$.

P4 *Terminación*.- Para cada par de registros de la forma $(i, j, A, \alpha, \epsilon)$ en $REGISTROS[j]$ y $(h, i, B, \gamma A, \delta)$ en $REGISTROS[i]$, crear el nuevo registro $(h, j, B, \gamma A, \delta)$ e insertarlo en $REGISTROS[j]$.

P5 Si $j < n$ ir a P2.

P6 Si en $REGISTROS[n]$ hay un registro de la forma $(0, n, S, \alpha, \epsilon)$, entonces u es generada. En caso contrario no es generada.

Ejemplo 76 Comprobar mediante el algoritmo de Early si la palabra baa es generada por la gramática con producciones:

$$\begin{array}{llll} S \rightarrow AB, & S \rightarrow BC, & A \rightarrow BA, & A \rightarrow a, \\ B \rightarrow CC, & B \rightarrow b, & C \rightarrow AB, & C \rightarrow a \end{array}$$

Después de aplicar el paso de inicialización, el contenido de $REGISTROS[0]$ es:

$REGISTROS[0] : (0, 0, S, \epsilon, AB), (0, 0, S, \epsilon, BC)$

$(0, 0, A, \epsilon, BA)$,
 $(0, 0, A, \epsilon, a)$, $(0, 0, B, \epsilon, CC)$, $(0, 0, B, \epsilon, b)$, $(0, 0, C, \epsilon, AB)$, $(0, 0, C, \epsilon, a)$,
REGISTROS[1] : $(0, 1, B, b, \epsilon)$, $(0, 1, S, B, C)$, $(0, 1, A, B, A)$, $(1, 1, C, \epsilon, AB)$,
 $(1, 1, C, \epsilon, a)$, $(1, 1, A, \epsilon, BA)$, $(1, 1, A, \epsilon, a)$, $(1, 1, B, \epsilon, CC)$, $(1, 1, B, \epsilon, b)$
REGISTROS[2] : $(1, 2, C, a, \epsilon)$, $(1, 2, A, a, \epsilon)$, $(0, 2, S, BC, \epsilon)$, $(0, 2, A, BA, \epsilon)$,
 $(1, 2, C, A, B)$, $(1, 2, B, C, C)$, $(0, 2, S, A, B)$, $(0, 2, C, A, B)$, $(2, 2, B, \epsilon, CC)$,
 $(2, 2, B, \epsilon, b)$, $(2, 2, C, \epsilon, AB)$, $(2, 2, C, \epsilon, a)$, $(2, 2, A, \epsilon, BA)$, $(2, 2, A, \epsilon, a)$
REGISTROS[3] : $(2, 3, C, a, \epsilon)$, $(2, 3, A, a, \epsilon)$, $(1, 3, B, CC, \epsilon)$, $(2, 3, B, C, C)$
 $(2, 3, C, A, B)$, $(1, 3, A, B, A)$

Como $(0, 3, S, \alpha, \epsilon)$ no está en REGISTROS[3], la palabra baa no es generada

$S \rightarrow T$, $S \rightarrow S + T$, $T \rightarrow F$, $T \rightarrow T * F$, $F \rightarrow a$, $F \rightarrow b$, $F \rightarrow (S)$

*Palabra: $(a + b) * a$*

REGISTROS[0] : $(0, 0, S, \epsilon, T)$, $(0, 0, S, \epsilon, S + T)$, $(0, 0, T, \epsilon, F)$, $(0, 0, T, \epsilon, T * F)$, $(0, 0, F, \epsilon, a)$, $(0, 0, F, \epsilon, b)$,
REGISTROS[1] : $(0, 1, F, (S))$, $(1, 1, S, \epsilon, T)$, $(1, 1, S, \epsilon, S + T)$, $(1, 1, T, \epsilon, F)$,
 $(1, 1, T, \epsilon, T * F)$, $(1, 1, F, \epsilon, a)$, $(1, 1, F, \epsilon, b)$, $(1, 1, F, \epsilon, (S))$,
REGISTROS[2] : $(1, 2, F, a, \epsilon)$, $(1, 2, T, F, \epsilon)$, $(1, 2, S, T, \epsilon)$, $(0, 2, F, (S,))$, $(1, 2, S, S, +T)$
REGISTROS[3] : $(1, 3, S, S +, T)$, $(3, 3, T, \epsilon, F)$, $(3, 3, T, \epsilon, T * F)$, $(3, 3, F, \epsilon, a)$, $(3, 3, F, \epsilon, b)$, $(3, 3, F, \epsilon, (S))$

REGISTROS[4] : $(3, 4, F, b, \epsilon)$, $(3, 4, T, F, \epsilon)$, $(1, 4, S, S + T, \epsilon)$, $(3, 4, T, T, *F)$, $(0, 4, F, (S,))$, $(1, 4, S, S, +T)$

REGISTROS[5] : $(0, 5, F, (S), \epsilon)$, $(0, 5, T, F, \epsilon)$, $(0, 5, S, T, \epsilon)$, $(0, 5, T, T, *F)$, $(0, 5, S, S, +T)$,

REGISTROS[6] : $(0, 6, T, T *, F)$, $(6, 6, F, \epsilon, a)$, $(6, 6, F, \epsilon, b)$, $(6, 6, F, \epsilon, (S))$,
REGISTROS[7] : $(6, 7, F, a, \epsilon)$, $(0, 7, T, T * F, \epsilon)$, **(0, 7, S, T, \epsilon)**, $(0, 7, T, T, *F)$, $(0, 7, S, S, +T)$

*Como tenemos $(0, 7, S, T, \epsilon)$, entonces la palabra $(a + b) * c$ es generada.*

6.3.2. Problemas Indecidibles para Lenguajes Libres de Contexto

Para terminar el apartado de algoritmos de decisión para gramáticas libres de contexto daremos algunos problemas que son indecidibles, es decir, no hay ningún algoritmo que los resuelva. En ellos se supone que G, G_1 y G_2 son gramáticas libres de contexto dadas y R es un lenguaje regular.

- Saber si $L(G_1) \cap L(G_2) = \emptyset$.
- Determinar si $L(G) = T^*$, donde T es el conjunto de símbolos terminales.
- Comprobar si $L(G_1) = L(G_2)$.
- Determinar si $L(G_1) \subseteq L(G_2)$.
- Determinar si $L(G_1) = R$.

- Comprobar si $L(G)$ es regular.
- Determinar si G es ambigua.
- Conocer si $L(G)$ es inherentemente ambiguo.
- Comprobar si $L(G)$ puede ser aceptado por una autómata determinístico con pila.

Bibliografía

- [1] A.V. Aho, J.D. Ullman, *Foundations of Computer Science*. W.H. Freeman and Company, New York (1992).
- [2] R.V. Book, F. Otto, *String rewriting systems*. Springer-Verlag, Nueva York (1993).
- [3] J.G. Brookshear, *Teoría de la Computación. Lenguajes formales, autómatas y complejidad*. Addison Wesley Iberoamericana (1993).
- [4] J. Carroll, D. Long, *Theory of Finite Automata with an Introduction to Formal Languages*. Prentice Hall (1989)
- [5] D.I. Cohen, *Introduction to Computer Theory*. John Wiley, Nueva York (1991).
- [6] M.D. Davis, E.J. Weyuker, *Computability, Complexity, and Languages*. Academic Press (1983)
- [7] M.D. Davis, R. Sigal, E.J. Weyuker, *Computability, Complexity, and Languages, 2 Edic.*. Academic Press (1994)
- [8] D. Grune, C.J. Ceriel, *Parsing techniques: a practical guide*. Ellis Horwood, Chichester (1990).
- [9] M. Harrison, *Introduction to Formal Language Theory*. Addison-Wesley (1978)
- [10] J.E. Hopcroft, R. Motwani, J.D. Ullman, *Introducción a la Teoría de Autómatas, Lenguajes y Programación*, 2^a Ed. Addison Wesley (2002).
- [11] J.E. Hopcroft, J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley (1979)
- [12] J.M. Howie, *Automata and Languages*. Oxford University Press, Oxford (1991)
- [13] H.R. Lewis, C.H. Papadimitriou, *Elements of the Theory of Computation*. Prentice Hall (1981)
- [14] B.I. Plotkin, J.L. Gvarami, *Algebraic structures in automata and database theory* World Scientific, River Edge (1992).
- [15] G.E. Revesz, *Introduction to Formal Languages*. Dover Publications, Nueva York (1991)
- [16] T.A. Sudkamp, *Languages and Machines*. Addison Wesley, Reading (1988)

Lectura 14. Un Nuevo Modelo para Procesos de Computación con Palabras en Toma de Decisión Lingüística

UNIVERSIDAD DE JAÉN

Escuela Politécnica Superior de Jaén
Departamento de Informática



Un Nuevo Modelo para Procesos de
Computación con Palabras en Toma de
Decisión Lingüística

DIPLOMA DE ESTUDIOS AVANZADOS

LÍNEA DE INVESTIGACIÓN:

*Integración de Información,
Toma de Decisiones, DSS y Sistemas Multiagente*

Rosa M^a Rodríguez Domínguez

Jaén, Junio de 2010

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

Lectura 15. Informática teórica. Elementos propedéuticos.

Informática teórica. Elementos propedéuticos. Raúl Gómez Marín y Andrés Sicard Ramírez, publicado por el Fondo Editorial de la Universidad EAFIT, 2002.

Nuestro texto está agotado. La versión que usted está leyendo corresponde a un reimpresión para la *Web* en la cual hemos realizado algunas correcciones que eran de nuestro conocimiento.

Si el lector desea realizar algún comentario (sugerencia, corrección, etc.) con relación a los contenidos del texto, lo puede hacer en la dirección de correo electrónico asicard@eafit.edu.co.

Última actualización: 1 de febrero de 2009

Correciones:

Págs. 195-197: El problema del isomorfismo de grafos no es un problema *NP*-completo (es un problema *NPI*). Por lo tanto el teorema 6.22 es falso.

A Lisa y Andrea, por un tiempo que era nuestro

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados.**

EDICTI_GUIA_MEX Editorial Guías Mexico

Índice general

Prefacio	13
0. Introducción	17
1. Computabilidad	25
1.1. Descripción Informal de la Máquina de Turing	26
1.2. Descripción Formal de la Máquina de Turing	27
1.3. Funciones Turing-Computables	31
1.4. m-funciones	37
1.5. Codificación de las Máquinas de Turing	40
1.5.1. Codificación de Turing	40
1.5.2. Codificación de Gödel	43
1.6. Máquina Universal de Turing	48
1.7. El Problema de la Parada	49
1.8. Ejercicios	54
1.9. Notas Bibliográficas	55
2. Recursividad	57
2.1. Funciones y Relaciones Numérico-Teóricas	57
2.2. Funciones Primitivas Recursivas	58
2.3. Construcción de Funciones Recursivas Primitivas	60
2.4. Predicados Primitivos Recursivos	63
2.5. Funciones Definidas Mediante Condiciones	65
2.6. Funciones Recursivas	67
2.7. Funciones Recursivas Parciales	71
2.8. Funciones Definidas por Minimalización Acotada	72
2.9. Conjuntos Recursivos	74
2.10. Conjuntos Recursivamente Enumerables	76
2.11. Computabilidad y Recursividad	80
2.12. Tesis de Church-Turing	93
2.13. Ejercicios	95
2.14. Notas Bibliográficas	98

3. Lenguajes y Gramáticas	99
3.1. Alfabetos y Lenguajes	99
3.1.1. Definiciones preliminares	99
3.1.2. Operaciones sobre palabras	100
3.1.3. Relación entre los conceptos de monoide y lenguaje	102
3.1.4. Operaciones entre lenguajes	104
3.2. Sistemas Formales y Sistemas Combinatorios	105
3.3. Gramáticas Formales o de Frase Estructurada	107
3.3.1. Problema de la representación	107
3.3.2. Gramáticas	107
3.3.3. Taxonomía de las gramáticas	110
3.3.4. Notación alternativa para las gramáticas	111
3.3.5. Algunos aspectos sobre las gramáticas	112
3.3.6. Algunos teoremas sobre gramáticas	120
3.4. Expresiones Regulares	123
3.5. Ejercicios	125
3.6. Notas Bibliográficas	132
4. Autómatas de Estado Finito	133
4.1. Máquinas de Estado Finito	134
4.2. Autómatas de Estado Finito	140
4.3. Reconocedor Finito	142
4.4. Algunas Clases de Autómatas	145
4.4.1. Autómatas de estado finito deterministas	145
4.4.2. Autómatas de estado finito no deterministas	146
4.5. Álgebra y Autómatas	148
4.5.1. Monoïdes asociados con un autómata	148
4.5.2. Comportamiento entrada-estados de un autómata	151
4.5.3. Relación de equirrespuesta de un autómata	152
4.5.4. Relaciones de congruencia	153
4.5.5. Relación equirrespuesta de un reconocedor finito	154
4.6. Álgebra y Lenguajes	156
4.6.1. Relación de congruencia derecha inducida por un lenguaje	156
4.6.2. Condición para que un lenguaje sea aceptado por un reconocedor finito	157
4.7. Ejercicios	158
4.8. Notas Bibliográficas	163
5. Autómatas de Pila	165
5.1. Autómata de pila no determinista	166
5.2. Autómatas de pila y reconocedores	168
5.3. Lenguajes independientes del contexto y autómatas de pila	171
5.4. Ejercicios	177
5.5. Notas Bibliográficas	181

6. Complejidad algorítmica	183
6.1. Máquinas de Turing k -cintas	183
6.2. Lenguajes recursivos y lenguajes recursivamente enumerables	184
6.3. Complejidad temporal determinista	185
6.4. Notación asintótica	186
6.5. Relaciones de complejidad temporal determinista	188
6.6. Máquinas de Turing $(k, 1)$ -cintas	191
6.7. Complejidad espacial determinista	191
6.8. Relaciones de complejidad espacial entre los modelos de computación determinista .	193
6.9. Máquina de Turing no determinista	194
6.10. Complejidad temporal y espacial no determinista	194
6.11. Relaciones entre clases de complejidad	196
6.12. Problemas intratables	199
6.13. Ejercicios	206
6.14. Notas Bibliográficas	207
Bibliografía	209
Índice de Materias	213

Índice de figuras

1.	Relaciones y dependencia entre capítulos.	18
3.1.	Curva de Koch.	106
3.2.	Árbol de análisis sintáctico para $\alpha \equiv 1 + 2 + 3$	113
3.3.	Árbol de derivación por la derecha y por la izquierda para $\alpha \equiv 3 + 1$	114
3.4.	Árboles de análisis sintáctico para $\alpha \equiv 1 - 2 + 3$	115
3.5.	Primer árbol de análisis sintáctico con base en una gramática ambigua	116
3.6.	Segundo árbol de análisis sintáctico con base en una gramática ambigua	117
3.7.	Árbol de análisis sintáctico con base en una gramática no ambigua	118
3.8.	Recursividad por la derecha para $\alpha \equiv 10101$	119
3.9.	Recursividad por la izquierda para $\alpha \equiv 10101$	120
3.10.	Digrafo para la gramática lineal derecha \mathcal{G}	122
3.11.	Digrafo para la gramática lineal izquierda \mathcal{G}'	123
4.1.	Representación de un sistema (1).	133
4.2.	Representación de un sistema (2).	134
4.3.	Sumador binario.	135
4.4.	Diagrama de transición para un sumador binario (1).	135
4.5.	Diagrama de transición para un sumador binario (2).	135
4.6.	Máquina de Mealy para un sumador binario.	138
4.7.	Máquina de Moore para un sumador binario.	139
4.8.	Autómata como generador.	140
4.9.	Autómata como reconocedor.	140
4.10.	Diagrama de transición para un autómata de estado finito.	141
4.11.	Reconocedor finito para $\mathcal{L} = \{a^n b^m; n, m \geq 1\}$	143
4.12.	Reconocedor finito para $\mathcal{L} = \{1(01)^n; n \geq 0\}$	144
4.13.	Autómata de estado finito no determinista.	146
4.14.	Ejemplo AFN.	147
4.15.	Construcción de un \mathcal{AFD} a partir de un \mathcal{AFN} (1).	149
4.16.	Construcción de un \mathcal{AFD} a partir de un \mathcal{AFN} (2).	150
4.17.	Expansión de la función $f_a, a \in \Sigma$ en $f_\alpha, \alpha \in \Sigma^*$	151
4.18.	Homomorfismo entre los monoides $\langle \Sigma^*, \bullet, \varepsilon \rangle$ y $\langle \Gamma^\Gamma, \circ, f_\varepsilon \rangle$	152
6.1.	$T(n) = \max(n+1, 'T(n)').$	187
6.2.	$f(n) = O(g(n))$	188

6.3.	$f(n) = \Omega(g(n))$	189
6.4.	$f(n) = \Theta(g(n))$	189
6.5.	$S(n) = \max(1, 'S(n)')$	192
6.6.	Árbol de computación: máquinas de Turing no deterministas.	194
6.7.	Línea de computación: máquinas de Turing.	195
6.8.	Árbol de computación para una \mathcal{MTN} para el problema <i>SAT</i>	201
6.9.	Ejemplo digrafo finito.	204
6.10.	Ejemplo grafo no dirigido finito.	204
6.11.	\mathcal{G}_1 isomorfo al grafo de la figura 6.12.	205
6.12.	\mathcal{G}_2 isomorfo al grafo de la figura 6.11.	205
6.13.	\mathcal{G}_1 no isomorfo al grafo de la figura 6.14.	206
6.14.	\mathcal{G}_2 no isomorfo al grafo de la figura 6.13.	206
6.15.	\mathcal{G}_a no isomorfo al grafo de la figura 6.16.	206
6.16.	\mathcal{G}_b no isomorfo al grafo de la figura 6.15.	207

Índice de cuadros

1.1.	Tabla simulación máquina \mathcal{MT} .	29
1.2.	Representación entrada de datos (1).	30
1.3.	Representación entrada de datos (2).	30
1.4.	Representación entrada de datos (3).	30
1.5.	Representación salida de los datos.	31
1.6.	Notación no tradicional: eliminación de un símbolo.	37
1.7.	Notación no tradicional: escritura del símbolo α sobre la cinta.	38
1.8.	Notación no tradicional: varios símbolos en la columna de operaciones (1).	38
1.9.	Escritura en la notación actual para la tabla 1.8.	38
1.10.	Notación no tradicional: Varios símbolos en la columna de operaciones (2).	39
1.11.	Escritura en la notación actual para la tabla 1.10.	39
1.12.	Notación no tradicional: múltiples operaciones.	40
1.13.	Estado de la cinta para los ejemplos 1.10, 1.11 y 1.12.	40
1.14.	m-función $F(\mathbf{S}, \mathbf{B}, \alpha)$.	41
1.15.	m-función $F_1(\mathbf{S}, \mathbf{B}, \alpha)$.	41
1.16.	m-función $F(\mathbf{S}, \mathbf{B}, \alpha)$.	42
1.17.	Expansión m-función $F(\mathbf{S}, \mathbf{B}, \alpha)$.	42
1.18.	m-función $PE(\mathbf{S}, \beta)$.	43
1.19.	Expansión m-función $PE(\mathbf{S}, \beta)$.	43
1.20.	m-función $PE_2(\mathbf{S}, \alpha, \beta)$.	44
1.21.	Descripción estándar de instrucciones.	44
1.22.	Entrada para la máquina universal de Turing.	48
1.23.	$\Sigma(n)$ y $S(n)$ para $1 \leq n \leq 6$.	52
2.1.	Número de llamadas a la función de Ackermann.	71
2.2.	Simulación \mathcal{MT} que calcula la función $f(2, 1) = 0$.	91
2.3.	Cálculo de $p(2, 1)$.	93
3.1.	Taxonomía de las gramáticas.	110
3.2.	Gramáticas, lenguajes y reconocedores.	111
4.1.	Tabla de transición para un sumador binario.	136
4.2.	Tabla de transición para un autómata de estado finito.	142
5.1.	Tabla de transición δ para el ejemplo 5.1.	167

5.2. Tabla de transición δ para el ejemplo 5.3.	170
6.1. Aceptación para \mathcal{L} palíndromo por \mathcal{MT}_2 (1).	185
6.2. Aceptación para \mathcal{L} palíndromo por \mathcal{MT}_2 (2).	185
6.3. Aceptación para \mathcal{L} palíndromo por \mathcal{MT}_2 (3).	186
6.4. Decisión para \mathcal{L} palíndromo por \mathcal{MT}_2 (4).	186
6.5. Decisión para \mathcal{L} palíndromo por una \mathcal{MT}	190
6.6. Relaciones entre las clases de complejidad.	199

Prefacio

Si escribir un texto, cualquiera sea éste, es una faena difícil, escribirlo a varias voces todavía lo es más. Con todo, la dificultad es dinamo de reto, de provocación, de osadía incluso, máxime si se trata de dos profesores cuya manera de ocuparse del saber no es extraña a una búsqueda permanente de momentos o encuentros afortunados que propicien el compartir saberes y construir perspectivas confluientes de trabajo, de suerte que éstas emanen del tramo de sus experiencias didácticas, lógicas, teóricas y humanas. La obra que aquí ofrecemos es el producto de un haz de confluencias, realizado a partir de encuentros afortunados que nos han permitido compartir nuestros conocimientos y experimentaciones como profesores de los diversos cursos del área de Matemáticas Especiales, cursos que hemos impartido en la Facultad de Ingeniería de Sistemas de la Universidad EAFIT, cuya sede se encuentra en la ciudad de Medellín, Colombia. Es así como nos hemos dado a la tarea de darle un cierto corpus didáctico y teórico a nuestras experimentaciones en este campo específico.

En efecto, no sin cierto rigor y esmero en la formalización, hemos querido reunir diversos temas que se han desarrollado en varios campos del conocimiento, temas que hoy se agrupan en lo que se podría llamar Informática Teórica. Anotemos además, que buena parte de los distintos “objetos” que definen el corpus teórico de esta obra emergen del oficio del saber lógico-matemático. Y en lo que a nosotros hace relación, aquéllos se yerguen esencialmente de una arraigada inquietud de saber, compartir lo sabido, y reconocer que todo saber sabido es siempre precario, inquietud que nosotros consideramos de orden ético-estético. Ello, primero que todo, porque esta inquietud está en el núcleo de lo humano y debe ser el “motor inmóvil” de nuestra manera de existir y, en segundo lugar, porque esta inquietud y en parte estos “objetos”, no han cesado de trabajar en nosotros y hacernos trabajar, marcando de este modo la que mal que bien podríamos llamar “dinámica del curso” de Matemáticas Especiales III, de la citada facultad.

No es del todo fácil develar los propósitos de un autor. Cuando se escribe, cabe la pregunta, ¿qué se pone realmente en disposición y a disposición del otro? Nosotros no sabríamos arriesgar una respuesta. No obstante, digamos que creemos poner en disposición un cierto corpus de nociones —cuya filiación marcamos meticulosamente—, un cierto tramo, analítico y a la vez sintético, constituido por objetos teóricos y demostraciones hechas muy a nuestra manera, así como objetos y consideraciones que forman parte del quehacer

formativo de nuestros estudiantes y profesores (y de manera muy específica de aquellos que transitan por las Facultades de Ingeniería de Sistemas de algunas universidades de nuestro entorno). Así contorneada la respuesta, no sobra destacar una variable importante en nuestras pretensiones. Queremos contribuir a llenar un cierto vacío bibliográfico en nuestro medio. Ciertamente existe una buena bibliografía, pero sus enfoques y contenidos están esencialmente orientados a cursos de postgrado. Pocos textos hay en el ámbito universitario que tengan una orientación hacia una propedéutica acorde con las condiciones y pretensiones de nuestros pregrados. En realidad son escasos los textos que tengan una orientación hacia estos niveles y que reúnan y traten adecuadamente y con cierta didaxis los temas aquí presentados. Cabría destacar en esta especie de vacío el texto de Lógica y Calculabilidad, escrito por el profesor colombiano Xavier Caicedo Ferrer; los demás, referenciados en nuestras notas bibliográficas, sin dejar de ser importantes, mantienen una dimensión bastante intuitiva y poco formalizada para lo que en nuestro medio, al menos en nuestra visión de las cosas, se busca con este género de cursos de pregrado.

Las referencias contextuales anteriores no pretenden escamotear los valores de los otros textos que circulan en nuestro medio, ni mucho menos abrogarnos una cuota de cabal originalidad. No, lejos estamos de tal intención. Los puntos de vista que aquí desarrollamos están bastante difundidos, como puede constatarse en el esfuerzo que hemos realizado a lo largo de este texto, por señalar la fuente específica desde la cual asumimos una determinada mirada, definición, demostración o ejemplo. Allí, en ese juego de despeje, de reconocimiento y asunción queda nuestro esfuerzo de originalidad. Ciertamente hay un poco de originalidad, pero no queremos entrar en ese juego de imaginarios; más bien, respetuosamente dejamos este asunto a cargo de los lectores avisados en estos campos. Lo que en rigor el texto busca, no sin deliberada intención, es generar un trazado entre intuición y formalización, así, como contornear o solamente diagramatizar ciertos temas; diagramatizar en el sentido de introducir iconos de relación que permitan, en el caso del lector, engendrar una visión amplia y comprensiva del tema, y ello no sin ciertas precisiones de detalle, técnicas formales, cuando así lo amerite la perspectiva que deseamos proponer. Con esta última alusión también queremos aclarar que ciertos temas; quizá la mayoría no pretende ir más allá de los límites que el texto configura, dada justamente su pretensión de ampliar los campos de formación en pregrado a la vez que otorgarles un cierto límite. Así las cosas, presentamos un texto que recoge y ordena metódicamente ciertos temas que habitualmente no se consideran o integran en los currículos de Ingeniería de Sistemas en nuestro medio; y otros temas que son formulados clásica e intuitivamente en otros textos, se desarrollan en el nuestro en función de una metodología que con un cierto grado de síntesis y de desarrollo, creemos, abre vías de lecturas que acompañarán el proceso de formación y desarrollo (más científico y riguroso) de un Ingeniero de Sistemas.

Finalmente, conviene afirmar que al avanzar en la elaboración de este texto, no cesamos de hallar problemas y vías alternas de evolución en su producción. Al fin de cuentas, nos percatamos de que el sentido que jalona una escritura no debe estar alejado de poder pensarla como una vía múltiple, es decir, como una vía que, al vislumbrar el fin, no puede

menos de pretextarlo. Nuestra sentida apuesta es, entonces, por un convidar al lector a “ver” en esta obra un trabajo que no cesa de rehacerse, esto es, a ver en ella una obra abierta. Abierta en diversos sentidos y grados. Abierta en ella misma, abierta al lector, campo abierto de problemas que ella contiene y que, quizás, contribuye a generar. Por último, para cerrar con un mandato ético de nuestro proceder, queremos señalar que toda argumentación, imprecisión, problemas inherentes a la producción de este libro y que los cuales permanecen sin resolver, son enteramente nuestra responsabilidad. Como diría el pensador chileno Humberto Maturana: “Nosotros nos hacemos cargo de lo dicho, en la esperanza de que el lector se haga cargo del sentido construido.”

Para mantener una comunicación constante con nuestros lectores —con base en los medios informáticos actuales—, el texto cuenta con una página *web*, la cual puede ser accesada desde la dirección www1.eafit.edu.co/asicard. Esta página mantendrá información actualizada respecto a correcciones, modificaciones o actualizaciones al texto. Por otra parte, si el lector desea realizar algún comentario (sugerencia, corrección, etc.) con relación a los contenidos del texto, lo puede hacer en la dirección de correo electrónico asicard@eafit.edu.co.

No queremos cerrar este prefacio sin ofrecer algunas notas de agradecimiento. A nuestros colegas, Juan Carlos Agudelo Agudelo, Orlando García Jaimes y Hugo Guarín Vásquez, quienes al seguir una versión preliminar de este texto en sus cursos, formularon algunas sugerencias al mismo. A nuestro estudiante Carlos Andrés Ardila por la elaboración del ejemplo 2.24 (pág. 90). A nuestros revisores por sus correcciones y sugerencias. Al fondo editorial de la Universidad EAFIT y a su directora Leticia Bernal. A nuestros estudiantes, quienes, semestre a semestre, por espacio de seis años, nos acompañaron en esta labor, bien en calidad de escuchas concernidos, bien en calidad de escuchas silenciosos. A la Universidad EAFIT, de una forma u otra nuestra alma mater, por concedernos el tiempo y el espacio requeridos para sacar adelante este pequeño sueño hoy vuelto realidad. Y, de contera, a otras presencias, aquí no nombradas, pero cuyos nombres sabemos con certeza que nos acompañan en nuestra memoria.

Los autores.
Universidad EAFIT, Medellín
15 de febrero del 2001

Capítulo 0

Introducción

A continuación presentamos un texto que contiene una cierta estructura didáctica y teórica. Ciertamente hay cambios en el orden canónico usual. No obstante, hay que decir que la experiencia en la enseñanza de estos temas nos ha revelado lo fructífero de tal ordenamiento.

Hemos organizado ciertos aspectos formales básicos para la computación en una estructura que contempla seis capítulos. En el primero nos ocuparemos de los elementos que consideramos básicos para una compresión de la teoría de las máquinas de Turing. El segundo, por su parte, lo consagraremos al estudio de la teoría de las funciones parciales recursivas. En el tercero nos adentraremos en el estudio de aspectos básicos de las teorías de lenguajes y gramáticas formales. El cuarto, en íntima conexión con el anterior, nos permitirá acercarnos a la teoría de los autómatas de estado finito. El quinto, lo consagremos al estudio de la teoría de los autómatas de pila. Finalmente, cerraremos el texto con un capítulo consagrado a presentar los elementos básicos de la teoría de la complejidad algorítmica. La figura 1 ilustra claramente la estructura del texto, esto es, nos representa las relaciones y dependencias establecidas entre los contenidos del texto.

A continuación presentamos algunas consideraciones que contribuyan a revelar la importancia y pertinencia de estos temas en la formación de una buena parte de ingenieros y tecnólogos, la misma que realizaremos para cada uno de los capítulos del texto.

Capítulo 1: Computabilidad

La teoría de la computabilidad clásica, tal como la conocemos hoy en día, fue establecida en la década de los 30's por los trabajos fundacionales realizados por Kurt Gödel, Jacques Herbrand y Stephen Kleene en funciones recursivas, Alonso Church y Stephen Kleene en funciones λ -definibles, y Alan Turing y Emil Post en funciones computables. Entre estos trabajos se destaca el realizado por Turing, debido a que no es necesario realizar un gran esfuerzo para observar la compenetración existente entre, por un lado, la idea intuitiva de

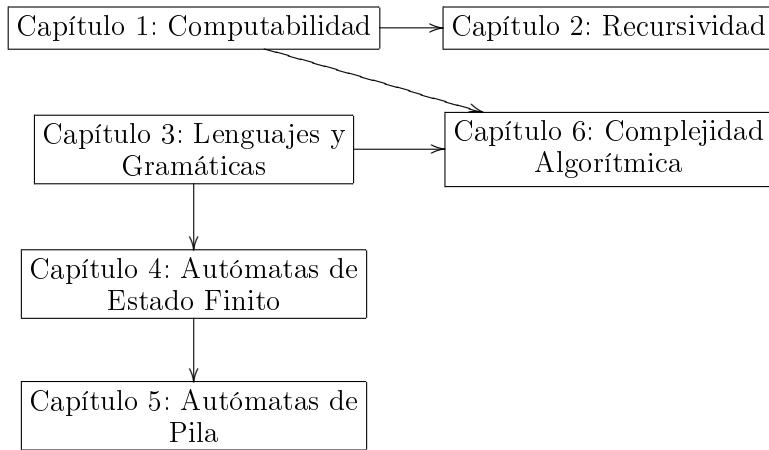


Figura 1: Relaciones y dependencia entre capítulos.

procedimiento calculable (es decir, aquel procedimiento que se pueda llevar a buen término siguiendo un conjunto de pasos establecidos) y, por otro lado, la formalización realizada por Turing para dar cuenta de esta clase de procedimientos, justamente lo que hoy denominamos máquinas de Turing.

El capítulo 1 aborda el estudio de la teoría de la computabilidad desde la perspectiva de las máquinas de Turing, obteniendo así la “teoría de la Turing-computabilidad”. Esta teoría, además de aportar conceptos muy importantes al estudio de las propiedades metamáticas de los sistemas formales, se constituyó (al igual que otras teorías) de la mano del propio Turing (y por supuesto de la mano de muchos otros) en la fundamentación formal de la ciencias de la computación. Inicialmente identificamos la noción de algoritmo con la noción de máquina de Turing.

Una vez establecida una definición formal de procedimiento computable como el que es computable por una máquina de Turing, podemos clasificar los objetos (números, funciones o procesos) como computables o no computables. Dentro de estos procesos no computables se destaca el problema de la parada de una máquina de Turing, el cual consiste en determinar si una máquina de Turing se detendrá o no con una entrada seleccionada de un conjunto posible de ellas. La insolubilidad del problema de la parada es fraseable, en términos de la ciencias de la computación, afirmando la imposibilidad de construir un depurador de programas que detecte si un programa podría o no entrar en un ciclo infinito.

Debido a que el conjunto de instrucciones de una máquina de Turing está compuesto por instrucciones muy simples, la “programación” de una máquina de Turing es usualmente un proceso muy tedioso, pudiéndose incluso comparar con la programación en lenguaje ensamblador. Para soslayar esta dificultad, Turing estableció el concepto de m-función.

Este concepto no es más que el predecesor formal de los llamados macros en los lenguajes ensambladores o de los llamados procedimientos en los lenguajes de programación de mayor nivel.

La noción de máquina de Turing universal es la noción formal subyacente a un computador, en el sentido que éste puede ser pensado como una máquina que ejecuta un algoritmo para ejecutar algoritmos. La máquina de Turing universal se constituye, pues, en el modelo formal de nuestras actuales máquinas de cómputo; de allí que se afirme que cualquier computador es una máquina de Turing con las limitaciones físicas que aquélla no tiene, limitaciones físicas que hacen referencia a la capacidad de memoria no acotada de una máquina de Turing universal.

En fin, en este capítulo iniciaremos un recorrido por la teoría de la computabilidad, recorrido que no dejará de gozar de un cierto nivel de formalización y abstracción.

Capítulo 2: Recursividad

Una vez hayamos terminado el recorrido con el concepto de computabilidad (en el capítulo 1), retomaremos este concepto en el capítulo 2, pero esta vez no en el sentido de Turing, sino en el sentido de Gödel, Herbrand y Kleene. El concepto de computabilidad es una idea bastante importante. ¿Por qué? Justamente porque existen operaciones, incluso aritméticas, que no son computables. En este capítulo probaremos por ejemplo, que existen funciones numéricas que no son computables.

El concepto de computabilidad es realmente un concepto abstracto, es decir, va más allá de cualquier realización concreta, en el sentido de que existen diversas formas de abordar la idea de computabilidad (λ -cálculo, funciones recursivas parciales, máquinas de Turing, etc.)

En este capítulo abordaremos la teoría de la computabilidad desde la perspectiva de la teoría de las funciones recursivas parciales, justamente para trabajar el concepto de computabilidad de una manera más matemática, lo cual nos dará cierta independencia de formalismos concretos y nos permitirá, mediante la teoría de conjuntos, obtener ciertos resultados sorprendentes sobre lo computable y lo no computable. Mostraríamos, por ejemplo, la equivalencia entre las funciones recursivas parciales y las funciones Turing-computables (lo cual nos permitirá otra formalización de la noción de algoritmo); probaremos igualmente que el conjunto de las funciones computables es del orden infinito enumerable, mientras que el conjunto de las no computables es del orden infinito no enumerable. Una pregunta o problema nos hará ciertamente trabajar bastante: se trata de saber si todo lo computable en sentido intuitivo es computable en el sentido formal, esto es, en el sentido de una función recursiva parcial. Esta es, justamente, una de las formas como podemos expresar la tesis de Church-Turing.

Es importante resaltar que, desde la perspectiva de las ciencias de la computación, la teoría de las funciones recursivas se constituye en la teoría formal subyacente al paradigma

de programación denominado “programación funcional”, así lenguajes de programación tales como *LISP* y *Mathematica* obtienen sus fundamentos formales de esta teoría.

En fin, en este capítulo haremos un recorrido axiomático y constructivo, de un cierto rigor, que nos ofrecerá de nuevo posibilidades de adentrarnos aún más en el fascinante universo de los problemas de la computabilidad.

Capítulo 3: Lenguajes y Gramáticas

Los lenguajes admiten una clasificación en lenguajes formales y lenguajes naturales. Una diferencia importante existente entre ellos, estriba en que en los primeros es posible desarrollar un trabajo en sintaxis y semántica mucho más riguroso que en los segundos. Ejemplos de lenguajes formales lo constituyen la lógica, la matemática y los lenguajes de programación.

En los lenguajes formales infinitos tendremos uno de los primeros aspectos importantes para considerar; se trata de la posibilidad de generación de un lenguaje por un conjunto finito de reglas, denominadas “reglas de producción”. Este conjunto finito de reglas de producción constituye lo que se conoce como la gramática del lenguaje. Noam Chomsky estableció una taxonomía para aquellos lenguajes formales susceptibles de ser generados por una gramática, con base en una clasificación de las reglas de producción. Trabajos posteriores establecieron relaciones del tipo lenguaje-reconocedor, asignando a cada tipo de lenguaje un tipo de reconocedor. Así dichas las cosas, dedicaremos este capítulo al desarrollo de la teoría de los lenguajes y de las gramáticas formales.

Uno de los aspectos más importantes de este capítulo, reside en el hecho de la clase de lenguajes formales denominados “lenguajes independientes del contexto”. Esta clase de lenguajes es de suma importancia en la ciencias de la computación, debido a que justamente, gran parte de los lenguajes de programación actuales como *C++*, *Java*, *Fortran*, entre otros, pertenecen a esta clase y por lo tanto comparten propiedades formales, la más importante de la cuales es quizás, que son reconocidos por máquinas teóricas denominadas “autómatas de pila”. Estos autómatas serán estudiados en el capítulo 5.

Una vez conocida la grámatica de un lenguaje es posible estudiar algunas propiedades y características de la misma, y por ende, algunas propiedades y características del lenguaje generado por ésta. Dentro de estas propiedades se destaca la propiedad de ambigüedad o no del lenguaje, propiedad que se revela de gran importancia debido a sus efectos nocivos en la manipulación automática del mismo, es decir, en la generación, interpretación o el reconocimiento del mismo.

En fin, este capítulo tiene como objetivo fundamental el estudio formal y riguroso de los lenguajes formales a partir de sus generadores, o sea, a partir de las gramáticas formales.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Capítulo 4: Autómatas de estado finito

La palabra ‘autómata’ tiene diversos sentidos. Puede evocar, por ejemplo, un dispositivo que simula los movimientos de un ser vivo. En su referencia a la informática, lo esencial de esta palabra consiste en permitirnos pensar en una simulación de los procesos para manipular información, cuyo paradigma típico es, obviamente, el computador, en la medida que éste puede ser visto como una máquina que manipula símbolos.

Este capítulo lo consagraremos a la teoría de autómatas de estado finito, también llamada teoría algebraica de máquinas. Si entendemos por un autómata una máquina secuencial (esto es, que opera sobre secuencias de símbolos), entonces el objetivo aquí es formalizar esta idea mediante elementos de la teoría de conjuntos y del álgebra abstracta.

La teoría de autómatas de estado finito que vamos a desarrollar nos dota de conocimientos y métodos para los problemas de análisis y de síntesis de las máquinas secuenciales. Un aspecto muy importante que desarrollaremos en este capítulo es el concerniente al problema de la decidibilidad, vía autómatas de estado finito, de un lenguaje y sus conexiones con las gramáticas formales. Puede pensarse, por ejemplo, en un autómata reconocedor para un cierto lenguaje. Pero esto último no siempre es posible garantizarlo para todo lenguaje formal, por ello en este capítulo estudiaremos bajo qué condiciones podemos garantizar el que un lenguaje dado sea aceptado por un reconocedor finito.

Aunque sabemos que desde la teoría de la computabilidad los autómatas de estado finito son modelos más limitados que las máquinas de Turing o las funciones recursivas, su importancia reside en que desde la perspectiva de las ciencias de la computación estos autómatas son usados para modelar procesos cuya complejidad admite un modelo más simple que el modelo general de computabilidad. Ejemplos de tales procesos los hallamos en la especificación de ciertas partes de los lenguajes de programación y en la modelación de ciertas subetapas de algunas de las etapas de Ingeniería del *software*.

En fin, este capítulo tiene como objetivo fundamental presentar los elementos básicos de la teoría de autómatas de estado finito, desarrollando algunos ejemplos que nos permitan ver su utilidad en campos diversos, especialmente los relacionados con la informática y la computabilidad.

Capítulo 5: Autómatas de pila

Una clase de máquinas formales más potentes que los autómatas de estado finito, pero menos potente que las máquinas de Turing, la constituye la clase de los autómatas de pila. Intuitivamente expresado, podemos decir que esta diferencia está sustentada en la capacidad de memoria de las tres máquinas formales mencionadas; así, una máquina de Turing tiene memoria no acotada y sin restricciones de acceso, un autómata de pila también tiene memoria no acotada pero el acceso a ella es restringido a las manipulaciones que se puedan realizar sobre la estructura de datos denominada “pila”, y un autómata de estado finito tiene memoria finita.

Como mencionamos en la sección correspondiente al capítulo 3, los autómatas de pila son los reconocedores de los lenguajes independientes del contexto (y de este tipo son la mayoría de los lenguajes de programación actuales). Por lo tanto, no es de extrañar que los automátas de pila sean uno de los principales elementos formales subyacente al área de las ciencias de la computación denominada “teoría de la compilación”. Es más, usualmente cuando compilamos un programa estamos ejecutando una implementación de un autómata de pila diseñado para reconocer palabras (algoritmos en este caso) que pertenecen al lenguaje de programación en cuestión.

De nuevo, un aspecto muy importante que desarrollaremos en este capítulo es el relacionado con el problema de la decidibilidad, en este caso desde la perspectiva de los autómatas de pila, estableciendo qué lenguajes pueden o no ser reconocidos por estos autómatas.

En fin, este capítulo será una primera aproximación formal a la teoría de los autómatas de pila, teoría que como mencionamos está en los cimientos de nuestros actuales compiladores.

Capítulo 6: Complejidad Algorítmica

La teoría de la computabilidad establece qué procesos pueden o no ser computados. Para ellos la teoría de la complejidad algorítmica establece clasificaciones de acuerdo con los recursos necesarios (tiempo y memoria) para computar los procesos computables. Este capítulo se consagrará entonces al desarrollo de los elementos básicos de la teoría de la complejidad algorítmica.

Para las ciencias de la computación no sólo es necesario saber si un problema es o no computable, sino que, además, es necesario establecer la factibilidad de ejecutar dicha computación (es decir, la posibilidad real de su ejecución), pues si la computación requiere un período de tiempo excesivo (quizás años o siglos) o si requiere una cantidad de memoria imposible de suministrar (desde un punto de vista práctico), la computación no se puede realizar. De otra parte, el estudio y desarrollo de algoritmos cada vez mejores está sustentado, entre otros aspectos, en la reducción de los recursos requeridos en su ejecución.

Una clasificación establecida *de facto*, para los diferentes problemas computables, es la que clasifica los problemas en tratables e intratables. Intuitivamente expresado esto último, problemas tratables, en cuanto a algún recurso, son problemas que tienen una demanda del recurso en cuestión, expresable como una función polinómica del tamaño de la entrada al problema, y problemas intratables son aquellos para los cuales esta función es exponencial.

Un aspecto de fundamental importancia para la teoría de la complejidad algorítmica lo constituye el modo de computación (determinista o no determinista). Mientras que en la teorías de la Turing-computabilidad, de los autómatas de estado finito y de los autómatas de pila no existe diferencia fundamental en cuanto a si ellos son o no deterministas, desde la perspectiva de la complejidad algorítmica el operar en modo determinista o el operar en modo no determinista puede establecer la diferencia entre que un problema sea

tratable (complejidad polinómica) o no tratable (complejidad exponencial). De hecho, uno de los problemas matemáticos actuales de mayor trascendencia consiste en decidir si para problemas de complejidad temporal polinómica existe o no diferencia en cuanto al modo de computación empleado.

En fin, en este capítulo haremos una introducción a esa área de la informática teórica que goza de dos características importantes: por una lado, la complejidad algorítmica es uno de los campos de investigación de mayor dinámica desde el punto de vista teórico, y por otro lado, es uno de los campos teóricos con mayor potencial para ofrecer posibilidad de obtener resultados aplicados.

Capítulo 1

Computabilidad

En la actualidad contamos con una definición muy precisa del concepto de *algoritmo*, palabra que procede del nombre del matemático persa del siglo IX, Abu Ja'far Mohammed ibd Mâsa al-Khowâriz; pero la situación no era la misma a principios de este siglo.

Como consecuencia del descubrimiento (a finales del siglo pasado y a comienzos de éste) de las paradojas o antinomias en la teoría de conjuntos y la situación que esto generó (situación muy importante dado el papel de teoría base que desempeña la teoría de conjuntos en la matemática), se abordó un problema más amplio que comprendía la fundamentación de la matemática y la lógica. En 1900, en el Congreso Internacional de Matemáticos realizado en París, Francia, Hilbert propuso 23 problemas que deberían marcar el desarrollo de las matemáticas en los años siguientes. El décimo de estos problemas era: ¿es posible encontrar un procedimiento mecánico para calcular la solución de una clase particular de ecuaciones (ecuaciones diofánticas, es decir, ecuaciones algebraicas con una o más incógnitas, de coeficientes enteros y de las que interesa únicamente las soluciones enteras)? El mismo Hilbert generalizó el problema y lo presentó en 1928 en el Congreso Internacional de Matemáticos realizado en Bolonia, Italia; esta generalización es conocida actualmente como *el problema de la decisión (Entscheidungsproblem)*, (algunos autores mencionan que el problema de la decisión es una generalización del segundo problema planteado por Hilbert en el congreso de 1900; este problema consistía en demostrar que los axiomas de la aritmética eran consistentes entre sí) y su enunciado es: ¿existe algún método o procedimiento mecánico, que pueda *en principio*, decidir (resolver) *todas* las preguntas (problemas) matemáticos? Expresado en términos más “formalistas” el problema de la decisión consistía en encontrar un *método efectivo general* para determinar si un fórmula era o no verdadera en un sistema formal dado.

Alan Mathison Turing demostró el carácter irresoluble del problema de la decisión, por un camino diferente al empleado por Kurt Gödel, presentado en su famoso teorema sobre la incompletitud de los sistemas formales. Turing sintió la necesidad de contar con una noción más precisa del concepto de *procedimiento efectivo de decisión* para el cual concibió

Esta guía de estudio, fue vendida por www.guiaaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

la noción de *computabilidad*. Creemos que la idea de que el procedimiento de decisión es un procedimiento mecánico, es decir, puede ser ejecutado por un ente sin inteligencia, siempre y cuando tenga acceso a las instrucciones indicadas por él (éstas sí creadas con inteligencia) llevó a Turing a concebir la idea de una máquina abstracta para describir este concepto. Esta máquina abstracta se conoce actualmente como la *máquina de Turing* y corresponde a la noción formal del concepto de algoritmo. Turing replanteó el problema de la decisión en términos de sus máquinas y demostró que este problema no tenía solución.

1.1. Descripción Informal de la Máquina de Turing

Alan Mathison Turing construyó una máquina abstracta, o si se prefiere, una máquina matemática conocida en nuestros días como *máquina de Turing*, en la cual capturó la noción de *algoritmo*.

Un algoritmo recibe algunos datos de entrada, los procesa y genera unos resultados. Usualmente, cuando ejecutamos un algoritmo a mano, utilizamos el papel como mecanismo de entrada-salida del algoritmo. Veamos inicialmente cuál es el mecanismo de entrada-salida utilizado por la máquina de Turing.

La máquina de Turing utiliza una *cinta infinita* como mecanismo de entrada-salida. La cinta está dividida en celdas las cuales pueden contener sólo un *símbolo* de un *alfabeto finito de símbolos* indivisibles de la máquina (el símbolo vacío se representa por \square y por convención hace parte de este alfabeto). En lenguaje técnico decimos que la cinta es una cinta unidimensional bi-infinita (infinita en ambos extremos).

En cada instante *discreto* de tiempo, la máquina, “observa” una celda de la cinta y tiene la capacidad de leer-escribir un símbolo sobre ésta. Adicionalmente la máquina tiene la capacidad de *desplazarse* sobre la cinta, es decir, se puede desplazar una celda a la derecha, o una celda a la izquierda, o no desplazarse con respecto a su posición actual. Aunque hemos mencionado que es la máquina la que se desplaza sobre la cinta, no existe tampoco ningún inconveniente, desde el punto de vista conceptual, el considerar que es la cinta la que se desplaza sobre la máquina. Nuestra convención estipula que es la máquina la que se desplaza sobre la cinta.

Además del *alfabeto* (conjunto finito de símbolos indivisibles), la máquina tiene también un *conjunto finito de estados*, los cuales representan, como su nombre lo indica, el estado en el que está la máquina en algún instante discreto de tiempo.

Definimos una *situación* de la máquina, como una dupla formada por un *estado* (perteneciente al conjunto finito de estados) y un *símbolo* (perteneciente al alfabeto). Para cada instante de tiempo, la *situación actual* de la máquina está determinada por el *estado actual* en el cual se encuentra y por el *símbolo* que está en la celda, es decir, aquélla sobre la cual la máquina está situada (la acción de leer el símbolo de la celda es automática). Es necesario entonces, antes de comenzar la ejecución de la máquina, definir su *situación inicial* (estado inicial, posición inicial sobre la cinta).

Esta guía de estudio, fue vendida por www.guiaconeval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaconeval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

El elemento más importante de una máquina de Turing lo constituye el *conjunto finito de instrucciones*. Este conjunto representa el comportamiento de la máquina. Cada *instrucción* está compuesta por dos partes; la primera indica *cuándo* se debe ejecutar la instrucción, es decir, en qué situación (estado, símbolo); la segunda parte indica *qué hace* la instrucción, lo cual consiste en: escribir un símbolo (en la posición actual), ejecutar un movimiento sobre la cinta y colocar la máquina en un nuevo estado. En otras palabras, una instrucción está compuesta por cinco elementos, a saber: *estado-actual*, *símbolo-leer*, *símbolo-escribir*, *movimiento*, *estado-siguiente*; donde los dos primeros elementos indican *cuándo* y los tres restantes *qué hacer*.

Veamos cómo opera una máquina de Turing: La máquina busca en su conjunto de instrucciones una instrucción que concuerde (la primera parte) con la situación actual (estado actual, símbolo actual) de la máquina. Si la encuentra, la ejecuta, escribiendo el símbolo, realizando un movimiento y pasando al estado indicado por la instrucción hallada. En este momento la máquina se encuentra en una nueva situación actual y repite el proceso. Si por el contrario la máquina no encuentra una instrucción que concuerde con la situación actual, la máquina se detiene y se da por finalizada su ejecución. Éste es un contexto adecuado para la siguiente pregunta: ¿Es posible que la máquina tenga más de una instrucción para una situación en particular? De acuerdo con la respuesta obtenemos dos clases diferentes de máquina de Turing, a saber: Las *máquinas de Turing determinísticas*, las cuales no permiten definir más de una instrucción para una situación particular, y las *máquinas de Turing no determinísticas*, que sí permiten tal cosa.

1.2. Descripción Formal de la Máquina de Turing

Definimos formalmente una máquina de Turing determinista mediante la estructura matemática $\mathcal{MT} = \langle Q, \Sigma, M, I \rangle$, donde:

$Q = \{q_0, q_1, q_2, \dots, q_n\}$: Conjunto finito de estados de la máquina ($Q \neq \emptyset$).

$\Sigma = \{s_0, s_1, s_2, \dots, s_m\}$: Alfabeto o conjunto finito de símbolos de entrada- salida. Adoptamos por convención que $s_0 = \square$ (símbolo vacío) ($\Sigma - \{s_0\} \neq \emptyset$).

$M = \{L, R, N\}$: Conjunto de movimientos (L : izquierda, R : derecha, N : no movimiento).

I : Es una función definida de un subconjunto $Q \times \Sigma$ en $\Sigma \times M \times Q$. Se debe notar que la función I está definida sobre un subconjunto de $Q \times \Sigma$, puesto que no es necesario que exista una instrucción para cada una de las situaciones (estado, símbolo) teóricas (el conjunto formado por $Q \times \Sigma$) en las que puede estar la máquina. Además, el concepto de función refleja la característica de las máquinas de Turing determinísticas. La función I también puede ser definida como un conjunto *finito* de instrucciones $I = \{i_0, i_1, i_2, \dots, i_p\}$, donde cada i_j es una quíntupla de la forma: $q_m \ s_m \ s_n \ m \ q_n$, donde $q_m, q_n \in Q$; $s_m, s_n \in \Sigma$; $m \in M$.

No obstante la distinción establecida, nuestro trabajo en este capítulo hará referencia principalmente a máquinas de Turing deterministas; por la tanto, la palabra “deterministas” se omitirá desde este momento.

Esta guía de estudio, fue vendida por www.guiaconeval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Es importante anotar que la información inicial que contiene la cinta, esto es, el estado inicial y la posición inicial de la máquina sobre ésta, será suministrada “informalmente” antes de comenzar la ejecución de la máquina.

Si una máquina \mathcal{MT} se encuentra en la situación actual (q_m, s_m) y encuentra una instrucción $i_j : q_m, s_m, s_n, m, q_n$, entonces \mathcal{MT} cambia s_m por s_n ; realiza el movimiento indicado por m y pasa al estado q_n (puede ocurrir que $q_m = q_n$ o $s_m = s_n$); en un caso contrario, la máquina finaliza su ejecución.

Pasemos ahora a considerar algunos ejemplos de máquinas de Turing:

Ejemplo 1.1. Sea $\mathcal{MT} = \langle Q, \Sigma, M, I \rangle$ una máquina de Turing definida por:

$Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$ (recordamos que, por convención el símbolo vacío \square pertenece al alfabeto), $I = \{i_0, i_1, i_2, i_3\}$ donde:

$i_0: q_0 \square 0 R q_1$

$i_1: q_1 \square \square R q_2$

$i_2: q_2 \square 1 R q_3$

$i_3: q_3 \square \square R q_0$

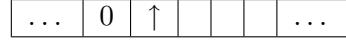
Antes de comenzar la ejecución de \mathcal{MT} necesitamos definir la secuencia de símbolos iniciales en la cinta, el estado inicial en el cual se encuentra la máquina y la posición inicial de \mathcal{MT} en dicha cinta. Inicialmente la cinta se encuentra vacía y la máquina está en alguna posición de la cinta (representada por la celda a la cual apunta el símbolo \uparrow); además, la máquina se encuentra en su estado inicial q_0 . Al inicio tenemos:

estado actual: q_0



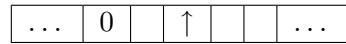
Comencemos la ejecución: la máquina busca en su conjunto de instrucciones una que comience por la situación actual (q_0, \square) y encuentra la instrucción “ $q_0 \square 0 R q_1$ ” (instrucción i_0), entonces, la máquina procede a cambiar \square por 0, se mueve una posición a la derecha y pasa al estado q_1 . Después de ejecutar la primera instrucción tenemos:

estado actual: q_1



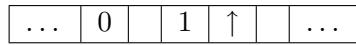
Ahora la máquina está en la situación actual (q_1, \square) ; como a esta situación le corresponde la instrucción “ $q_1 \square \square R q_2$ ” (instrucción i_1), entonces la máquina se mueve a la derecha y pasa al estado q_2 (porque la instrucción indica que cambie \square por \square lo que se traduce en no escribir nada). Ahora tenemos:

estado actual: q_2



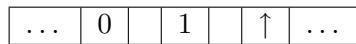
Como la situación actual (q_2, \square) corresponde la instrucción “ $q_2 \square 1 R q_3$ ” (instrucción i_2), entonces la máquina cambia \square por 1, se mueve a la derecha y pasa al estado q_3 . Ahora tenemos:

estado actual: q_3



Ahora, para la situación actual (q_3, \square) corresponde la instrucción “ $q_3 \square \square R q_0$ ” (instrucción i_3), la cual le indica a la máquina que se mueva a la derecha y pase al estado q_0 , es decir:

estado actual: q_0



En este momento la máquina se encuentra de nuevo en la situación actual (q_0, \square) y necesariamente repite el ciclo indefinidamente.

La tabla 1.1 muestra el comportamiento de la máquina \mathcal{MT} para la ejecución de los siete primeros “pasos”.

Paso	EA ^a	IE ^b									
0	q_0			...	\uparrow						...
1	q_1	i_0		...	0	\uparrow					...
2	q_2	i_1		...	0		\uparrow				...
3	q_3	i_2		...	0		1	\uparrow			...
4	q_0	i_3		...	0		1		\uparrow		...
5	q_1	i_0		...	0		1		0	\uparrow	...
6	q_2	i_1		...	0		1		0		\uparrow
7	q_3	i_2		...	0		1		0		1
:											...

^aEA: Estado Actual

^bIE: Instrucción Ejecutada

Cuadro 1.1: Tabla simulación máquina \mathcal{MT} .

Observación. Nótese que particularmente esta máquina no se detiene nunca. Además utiliza la convención propuesta por Turing de “trabajar” en celdas intercaladas, con el propósito de utilizar los espacios entre las celdas ocupadas para colocar “marcas” temporales que se necesiten durante la ejecución. Pero señalemos que no todas las máquinas tienen estas características.

Ejemplo 1.2. A continuación vamos a construir una \mathcal{MT} que calcula la suma de dos números naturales representados en el código “palitos”.

Uno de los aspectos que se debe tener en cuenta en el diseño de una máquina de Turing es la forma como están representados en la cinta los datos de entrada. Supongamos que deseamos realizar la suma de 1 y 2.

Para codificar los números naturales $\mathbb{N} = \{0, 1, 2, \dots\}$ en el alfabeto de “palitos”, estableceremos la siguiente convención:

$$\begin{aligned} 0 &= / \\ 1 &= // \\ 2 &= /// \\ 3 &= //// \\ &\vdots \\ n &= /^{n+1} = \underbrace{/// \dots /}_{n+1 \text{ veces}} \end{aligned}$$

Una representación de la diversas representaciones que podríamos tener viene dada por la tabla 1.2. En este caso los números que se van a sumar están separados por un espacio en blanco.

...			/		/		/		/		/		...
-----	--	--	---	--	---	--	---	--	---	--	---	--	-----

Cuadro 1.2: Representación entrada de datos (1).

Otra representación viene dada por la tabla 1.3. En este caso los números que se van a sumar están separados por un signo ‘+’.

...			/		/		+		/		/		...
-----	--	--	---	--	---	--	---	--	---	--	---	--	-----

Cuadro 1.3: Representación entrada de datos (2).

Otra posible representación viene dada por la tabla 1.4. En este caso los números que se van a sumar están separados por n espacios en blanco.

/			...		/		/		/		...
---	--	--	-----	--	---	--	---	--	---	--	-----

Cuadro 1.4: Representación entrada de datos (3).

Seleccionemos la primera representación (tabla 1.2) para construir nuestra máquina sumadora de palitos.

Otro de los aspectos que se debe considerar en el diseño de una \mathcal{MT} , es la forma o convención elegida para representar en la cinta la respuesta generada por la máquina.

En este caso, vamos a suponer que la salida de la máquina está dada por el número de “palitos” en la cinta. Entonces para nuestro ejemplo de la suma de 1 + 2 la salida estará

representada como lo indica la tabla 1.5.

...		/	/	/				...
-----	--	---	---	---	--	--	--	-----

Cuadro 1.5: Representación salida de los datos.

Precisemos ahora algunos detalles concernientes a la máquina en cuestión. Digamos de entrada que nuestra máquina está definida por $\mathcal{MT} = \langle Q, \Sigma, M, I \rangle$ donde, $Q = \{q_0, q_1, q_2, q_3, q_4, \text{stop}\}$ (conjunto de estados internos de la máquina).

Antes de continuar, hablemos un poco acerca de este estado “stop” en nuestra máquina. En la construcción de una gran parte de máquinas, se espera que éstas finalicen (a diferencia de nuestro primer ejemplo). Es necesario conocer si dicha finalización fue exitosa o no, es decir, si la máquina finalizó porque estaba programada para ello, o finalizó porque no encontró una instrucción para la situación actual en la que se encontraba. Para resolver esta incertidumbre se acostumbra dotar a la máquina de un estado de parada exitosa, normalmente llamado “stop”. El cual, cuando es alcanzado por la máquina, finaliza su ejecución (pues no existe ninguna instrucción que contemple a “stop” como su primer componente). Ciertamente, para el diseñador de la máquina esta es una parada exitosa. Observe el lector que el uso de esta convención no modifica para nada lo dicho hasta el momento. Hechas las anteriores consideraciones, culminemos el diseño de nuestra máquina.

Inicialmente la cinta contiene dos números naturales representados en el código “palitos”, separados por un espacio en blanco. La máquina se encuentra situada sobre el primer palito de izquierda a derecha y el estado inicial es q_0 .

$\Sigma = \{| \}$ (alfabeto de entrada-salida), $I = \{i_0, i_1, i_2, i_3, i_4, i_5, i_6\}$ (conjunto de instrucciones), donde:

$i_0: q_0 \mid \mid R q_0$; recorre el primer número hasta el espacio en blanco

$i_1: q_0 \square \mid R q_1$

$i_2: q_1 \mid \mid R q_1$; recorre el segundo número hasta el espacio en blanco

$i_3: q_1 \square \square L q_2$

$i_4: q_2 \mid \square L q_3$

$i_5: q_3 \mid \square L q_4$

$i_6: q_4 \mid \square N \text{ stop}$; la máquina se detiene en el estado stop

1.3. Funciones Turing-Computables

Precisemos que, en este contexto, el conjunto de los números naturales \mathbb{N} está definido por los números enteros no negativos, es decir $\mathbb{N} = \{0, 1, 2, \dots\}$. Señalemos igualmente que la teoría de la computabilidad opera sobre funciones de la forma $f: \mathbb{N}^n \rightarrow \mathbb{N}$, es decir, funciones $f(x_1, x_2, \dots, x_n) = y$ (la función f asigna a un n-tupla (x_1, x_2, \dots, x_n) un único número natural y). Como un caso particular se tienen las funciones de la forma $f: \mathbb{N} \rightarrow \mathbb{N}$;

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

esto es, funciones tales que $f(x) = y$ (la función f asigna a un número natural x un único número natural y). Estas funciones son denominadas funciones numérico-teóricas.

A lo largo de nuestra experiencia como estudiantes de matemáticas, hemos encontrado la idea de que una función es una relación que asocia a todos y cada uno de los elementos de su dominio una y sólo una imagen. No obstante, es importante para nuestro trabajo en la teoría de la computabilidad, que hagamos una distinción en el conjunto de las funciones numérico-teóricas, a saber, funciones totales y funciones parciales.

Definición 1.1 (Función total). Sea $f(x_1, \dots, x_n)$ una función n-ádica. Si f está definida para toda $(x_1, \dots, x_n) \in \mathbb{N}^n$, entonces diremos que f es una función total.

Definición 1.2 (Función parcial). Sea $f(x_1, \dots, x_n)$ una función n-ádica. Si f no está definida para al menos una $(x_1, \dots, x_n) \in \mathbb{N}^n$, entonces diremos que f es una función parcial.

Para poder asociar a una máquina de Turing \mathcal{MT} una función f_M y así, poder hablar de funciones Turing-computables, es necesario contar con una descripción completa de la ejecución de la máquina, es decir, necesitamos conocer la evolución temporal de la máquina, y para ello, necesitamos conocer los símbolos sobre la cinta, la posición actual sobre la cinta, la instrucción que se ejecuta y los cambios efectuados por la ejecución de dicha instrucción.

Las definiciones que presentamos a continuación, nos permitirán identificar los elementos antes mencionados.

Definición 1.3 (Sucesión). Sea $\mathcal{MT} = \langle Q, \Sigma, M, I \rangle$ una máquina de Turing. Una sucesión es una palabra finita (posiblemente vacía) construida sobre el alfabeto formado por $Q \cup \Sigma \cup M$.

Sea \mathcal{MT} una máquina de Turing tal que: $Q = \{q_1, q_2, \dots, q_n\}$, $\Sigma = \{s_0, s_1, \dots, s_m\}$ y $M = \{L, R, N\}$. Algunas posibles sucesiones son: Λ (la palabra vacía), $\alpha \equiv q_1 q_1 L L s_2 s_3$, $\beta \equiv N N N s_0, q_1$, $\gamma \equiv s_2 s_3 L R q_4$.

Observe el lector que, según la definición anterior, el conjunto de instrucciones I puede ser definido como un conjunto de sucesiones con una características particulares, en cuanto al número de símbolos (longitud) en cada sucesión y en cuanto al dominio al cual deben pertenecer estos símbolos.

Definición 1.4 (Descripción instantánea). Sea $\mathcal{MT} = \langle Q, \Sigma, M, I \rangle$ una máquina de Turing. Una descripción instantánea α de \mathcal{MT} , es una sucesión tal que:

1. Contiene exactamente un símbolo $q_i \in Q$.
2. No contiene ningún símbolo $m \in M$.
3. El símbolo q_i en α no es el último símbolo de izquierda a derecha.
4. Todos los s_i que ocurren en α pertenecen a Σ .

Definición 1.5 (Sucesión de la cinta). Una sucesión que consta únicamente de símbolos $s_i \in \Sigma$ es llamada una sucesión de la cinta. Las sucesiones de la cinta se representarán por los símbolos P y Q .

Las definiciones de descripción instantánea y sucesión de la cinta, junto con las instrucciones de una máquina de Turing, nos permiten formalizar la evolución temporal de la misma. Para tal propósito presentamos las siguientes definiciones.

Definición 1.6 (Cambio descripción instantánea). Sea una máquina de Turing \mathcal{MT} y sean α, β descripciones instantáneas de \mathcal{MT} . El paso de la descripción instantánea α a la descripción instantánea β se representa por $\alpha \xrightarrow{\mathcal{MT}} \beta$, lo cual significa que alguna de las siguientes condiciones se satisface:

1. Existen sucesiones P y Q (sucesiones de la cinta, posiblemente vacías) tales que:

$$\alpha \equiv P q_i s_j Q$$

$$\beta \equiv P q_j s_k Q$$

y \mathcal{MT} contiene una instrucción $i_a : q_i s_j s_k N q_j$

2. Existen sucesiones P y Q (sucesiones de la cinta, posiblemente vacías) tales que:

$$\alpha \equiv P q_i s_j s_k Q$$

$$\beta \equiv P s_l q_j s_k Q$$

y \mathcal{MT} contiene una instrucción $i_a : q_i s_j s_l R q_j$

3. Existen sucesiones P y Q (sucesiones de la cinta, posiblemente vacías) tales que:

$$\alpha \equiv P q_i s_j Q$$

$$\beta \equiv P s_l q_j s_0 Q \text{ donde } s_0 = \square$$

y \mathcal{MT} contiene una instrucción $i_a : q_i s_j s_l R q_j$

4. Existen sucesiones P y Q (sucesiones de la cinta, posiblemente vacías) tales que:

$$\alpha \equiv P s_j q_i s_k Q$$

$$\beta \equiv P q_j s_j s_l Q$$

y \mathcal{MT} contiene una instrucción $i_a : q_i s_k s_l L q_j$

5. Existen sucesiones P y Q (sucesiones de la cinta, posiblemente vacías) tales que:

$$\alpha \equiv P q_i s_j Q$$

$$\beta \equiv P q_j s_0 s_l Q \text{ donde } s_0 = \square$$

y \mathcal{MT} contiene una instrucción $i_a : q_i s_j s_l L q_j$

Definición 1.7 (Descripción terminal). Sea \mathcal{MT} un máquina de Turing. Una descripción instantánea α es llamada una descripción terminal, con respecto a \mathcal{MT} , si no existe una descripción instantánea β tal que: $\alpha \xrightarrow{\mathcal{MT}} \beta$.

Definición 1.8 (Computación de una máquina de Turing). Una computación de una máquina de Turing \mathcal{MT} es una secuencia finita de descripciones instantáneas $\alpha_1, \alpha_2, \dots, \alpha_n$,

tal que: $\alpha_i \xrightarrow{MT} \alpha_{i+1}$, para $1 \leq i < n - 1$, y tal que, α_n es una descripción terminal con respecto a MT . En este caso escribimos: $\alpha_n = Res_{MT}(\alpha_1)$ y llamamos a α_n el resultado de α_1 con respecto a MT .

Ejemplo 1.3. Para la máquina MT presentada en el ejemplo 1.2 (pág. 29), tenemos la siguiente computación:

$$\begin{aligned}\alpha_1 &\equiv q_0||\square||| \\ \rightarrow \alpha_2 &\equiv |q_0|\square||| \\ \rightarrow \alpha_3 &\equiv ||q_0\square||| \\ \rightarrow \alpha_4 &\equiv |||q_1||| \\ \rightarrow \alpha_5 &\equiv ||||q_1|| \\ \rightarrow \alpha_6 &\equiv |||||q_1| \\ \rightarrow \alpha_7 &\equiv |||||q_1\square \\ \rightarrow \alpha_8 &\equiv |||||q_2|\square \\ \rightarrow \alpha_9 &\equiv |||||q_3|\square\square \\ \rightarrow \alpha_{10} &\equiv |||||q_4|\square\square\square \\ \rightarrow \alpha_{11} &\equiv ||||stop\square\square\square\square.\end{aligned}$$

Entonces, $\alpha_{11} = Res_{MT}(\alpha_1)$.

Una vez formalizada la evolución temporal de una máquina de Turing y el resultado de ejecutar dicha máquina a partir de una descripción instantánea inicial, procedemos a dotar del carácter funcional el resultado obtenido en la ejecución de la máquina. Para ello utilizaremos una codificación similar a la presentada en el ejemplo 1.2 (pág. 29).

Definición 1.9 (Codificación números naturales). Cada número $n \in \mathbb{N}$ se asocia con una sucesión \vec{n} (en la cinta), donde:

$$\vec{n} = |^{n+1} = \underbrace{||| \dots |}_{n+1 \text{ veces}}.$$

Definición 1.10 (Codificación n-tuplas de números naturales). Cada n-tupla de números naturales $(n_1, n_2, \dots, n_k) \in \mathbb{N}^k$ se asocia una sucesión $(\vec{n}_1, \vec{n}_2, \dots, \vec{n}_k)$ (en la cinta) donde:

$$(\vec{n}_1, \vec{n}_2, \dots, \vec{n}_k) = \vec{n}_1 \square \vec{n}_2 \square \dots \square \vec{n}_k = \vec{n}_1 s_0 \vec{n}_2 s_0 \dots s_0 \vec{n}_k.$$

Por ejemplo, la terna $(0, 1, 2)$ se asocia con la sucesión en la cinta:

$$\begin{aligned}(\overrightarrow{0, 1, 2}) &= |\square||\square||| \\ &= |s_0||s_0|||.\end{aligned}$$

Definición 1.11 (Números de | en una sucesión). Sea γ una sucesión. El número de | en γ lo representamos por $< \gamma >$. Por ejemplo, $< |\square|\square||| > = 6$.

Definición 1.12 (Función asociada a una máquina de Turing). Sea \mathcal{MT} una máquina de Turing. Por cada $n \in \mathbb{N}$ se asocia con la máquina \mathcal{MT} una función $f_M^{(n)}(x_1, x_2, \dots, x_n)$, definida por:

Para cada n-tupla (m_1, m_2, \dots, m_n) se selecciona una descripción instantánea $\alpha_1 \equiv q_1(\overrightarrow{m_1, m_2, \dots, m_n})$ y se distinguen dos casos:

1. Existe una computación de \mathcal{MT} , $\alpha_1, \alpha_2, \dots, \alpha_p$. En este caso, tenemos que:

$$\begin{aligned} f_M^{(n)}(m_1, m_2, \dots, m_n) &= < \alpha_p > \\ &= < \text{Res}_M(\alpha_1) >. \end{aligned}$$

2. No existe una computación de \mathcal{MT} , $\alpha_1, \alpha_2, \dots, \alpha_p$. En este caso, tenemos que $f_M^{(n)}(m_1, m_2, \dots, m_n)$ no está definida.

Definición 1.13 (Función parcial Turing-computable). Sea $f(x_1, \dots, x_n)$ una función parcial. $f(x_1, \dots, x_n)$ es una función parcial Turing-computable si y sólo si, existe una máquina de Turing \mathcal{MT} , tal que:

$$f(x_1, \dots, x_n) = f_M^{(n)}(x_1, \dots, x_n).$$

En tal caso se dice que \mathcal{MT} computa a la función f .

Definición 1.14 (Función total Turing-computable). Sea $f(x_1, x_2, \dots, x_n)$ una función total. $f(x_1, x_2, \dots, x_n)$ es una función total Turing-computable si y sólo si, existe una máquina de Turing \mathcal{MT} , tal que:

$$f(x_1, x_2, \dots, x_n) = f_M^{(n)}(x_1, x_2, \dots, x_n).$$

Se dice, igualmente, que \mathcal{MT} computa a la función f .

Ejemplo 1.4. La función $f(x) = 0$, donde $x \in \mathbb{N}$ es una función Turing-computable. Para comprobarlo es necesario construir una máquina de Turing \mathcal{MT} , tal que: $f(x) = f_{\mathcal{MT}}^{(1)}(x)$. Sea \mathcal{MT} la máquina de Turing dada por:

$$\begin{aligned} i_1 : q_1 | \square R q_1 \\ i_2 : q_1 \square \square R stop \end{aligned}$$

Así, para $x = 2$ comenzando en, $\alpha_1 \equiv q_1(\overrightarrow{x})$, tenemos la siguiente computación:

$$\begin{aligned}
 \alpha_1 &\equiv q_1 // \\
 \rightarrow \alpha_2 &\equiv \square q_1 // \\
 \rightarrow \alpha_3 &\equiv \square \square q_1 / \\
 \rightarrow \alpha_4 &\equiv \square \square \square q_1 \square \\
 \rightarrow \alpha_5 &\equiv \square \square \square \square q_1 \square.
 \end{aligned}$$

Entonces,

$$\begin{aligned}
 f_{\mathcal{MT}}^{(1)}(2) &= \langle \text{Res}_{\mathcal{MT}}(\alpha_1) \rangle \\
 &= \langle \text{Res}_{\mathcal{MT}}(q_1 //) \rangle \\
 &= \langle \alpha_5 \rangle \\
 &= \langle \square \square \square \square q_1 \square \rangle \\
 &= 0 \\
 &= f(2).
 \end{aligned}$$

La demostración de que $f(x) = f_{\mathcal{MT}}^{(1)}(x)$ se presenta en el lema 2.2 (pág. 80).

Ejemplo 1.5. Mostrar que la función $f(x) = x + 1$ es una función Turing-computable. Construyamos una máquina de Turing \mathcal{MT} tal que $f(x) = f_{\mathcal{MT}}^{(1)}(x)$. Sea \mathcal{MT} la máquina de Turing dada por: $i_1 : q_1 \square \square N q_1$

Así, para $x = 5$ y comenzando en, $\alpha_1 \equiv q_1(\vec{x})$, la computación estaría dada por:

$$\alpha_1 \equiv q_1 // // //.$$

Es decir, la máquina no realiza ningún paso de computación, por lo tanto:

$$\begin{aligned}
 f_{\mathcal{MT}}^{(1)}(5) &= \langle \text{Res}_{\mathcal{MT}}(\alpha_1) \rangle \\
 &= \langle \text{Res}_{\mathcal{MT}}(q_1 // // //) \rangle \\
 &= \langle \alpha_1 \rangle \\
 &= \langle q_1 // // // \rangle \\
 &= 6 \\
 &= f(x).
 \end{aligned}$$

Una demostración de que $f(x) = f_{\mathcal{MT}}^{(1)}(x)$ se presenta en el lema 2.3 (pág. 81).

1.4. m-funciones

En la actualidad, el comportamiento de cualquier máquina de Turing se acostumbra expresarlo en la notación de instrucciones. Cada instrucción está constituida por una quíntupla: $q_i s_j s_k m q_l$ donde:

q_i : estado actual

s_j : símbolo que se lee

s_k : símbolo que se escribe

m : movimiento que se realiza

q_l : estado final

Aunque la notación anterior fue propuesta por Turing en su artículo fundacional sobre las máquinas de Turing, las máquinas construidas por él en dicho artículo (en particular la máquina universal de Turing), no están descritas en la “notación de instrucciones”; por el contrario, están descritas en una notación que permite simplificar considerablemente el número de instrucciones necesarias para describir el comportamiento de una máquina; dicha notación tiene como elemento principal el concepto de *m-función*.

Existen ciertos tipos de tareas tales como copiar una determinada secuencia de símbolos, buscar un determinado símbolo en la cinta, imprimir una determinada cadena de símbolos al final de la cinta, entre otras, que son de uso muy frecuente en la construcción de máquinas de Turing de cierta complejidad. La idea entonces es contruir una máquina de Turing *générica* para cada una de estas tareas, de tal forma que pueda ser invocada por otras máquinas de Turing que necesiten realizar dicha tarea. El concepto de m-función corresponde a estas máquinas de Turing genericas.

Presentamos a continuación algunos elementos necesarios para poder formalizar (parcialmente) y ejemplificar el uso de las m-funciones. Además, se indicarán los procedimientos requeridos para expresar en la notación de instrucciones cualquier m-función, lo cual nos permitirá paladejar la simplicidad-complejidad ofrecida por dichas m-funciones.

Como mencionamos con anterioridad, la descripción de algunas de las máquinas propuestas por Turing se realiza con base en las m-funciones y en una notación no tradicional (diferente a la notación de instrucciones). Esta notación no tradicional se presenta por medio de los siguientes ejemplos:

Ejemplo 1.6. Es frecuente, en Alan Turing, el uso de una notación especial para escribir o borrar un símbolo (tal como lo indican las tablas 1.6 y 1.7).

Estado Inicial	Símbolo	Operación	Estado Final
B	E		B

Cuadro 1.6: Notación no tradicional: eliminación de un símbolo.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Estado Inicial	Símbolo	Operación	Estado Final
B		$P\alpha$	B

Cuadro 1.7: Notación no tradicional: escritura del símbolo α sobre la cinta.

Ejemplo 1.7. Turing también utilizaba una notación que permitía en la columna de operaciones varios símbolos (tal como lo indica la tabla 1.8).

Estado Inicial	Símbolo	Operación	Estado Final
B	Ninguno	$P0, L$	B

Cuadro 1.8: Notación no tradicional: varios símbolos en la columna de operaciones (1).

En la notación actual, la tabla 1.8 se podría escribir como lo indica la tabla 1.9.

Estado Actual	Ac-	Símbolo Leer	Símbolo escribir	Movimiento	Estado Siguiente	Si-
B		□	0	L	B	

Cuadro 1.9: Escritura en la notación actual para la tabla 1.8.

Ejemplo 1.8. Un ejemplo adicional del uso de varios símbolos en la columna de operaciones se indica en la tabla 1.10.

En la notación actual, la tabla 1.10 se podría escribir como lo indica la tabla 1.11.

Este ejemplo permite entrever la dificultad que se presenta al traducir la notación utilizada por Alan Turing a la notación de instrucciones. El doble movimiento indicado por la operación “ $R, R, P1$ ”, para ser escrito en la notación tradicional, exige conocer el conjunto de símbolos que pueden estar sobre la cinta. Este conjunto de símbolos se denota por el símbolo ‘Cualquiera’.

Ejemplo 1.9. La notación no tradicional permite escribir en una sola instrucción el número de operaciones que se desee (como lo permite observar la tabla 1.12).

En este caso, a partir del estado B , sin importar sobre cuál símbolo del alfabeto se encuentre la máquina, ésta imprime el símbolo e , se mueve a la derecha, imprime de nuevo el símbolo e , se mueve a la derecha, imprime el símbolo 0, se mueve dos casillas a la derecha, imprime el símbolo 0, finalmente se mueve dos casillas a la izquierda y pasa al estado C (mirar ejercicio 1.7).

Definición 1.15 (m-función). Una m-función es una máquina de Turing que permite el uso de parámetros para los estados y los símbolos que la conforman.

Estado Inicial	Símbolo	Operación	Estado Final
B	0	$R, R, P1$	B

Cuadro 1.10: Notación no tradicional: Varios símbolos en la columna de operaciones (2).

Estado Actual	Símbolo Leer	Símbolo Escribir	Movimiento	Estado Siguiente
q_i	0	0	R	q_{i+1}
q_{i+1}	Cualquiera	Cualquiera	R	q_{i+2}
q_{i+2}	Cualquiera	1	N	q_i

Cuadro 1.11: Escritura en la notación actual para la tabla 1.10.

El uso de parámetros permite que la m-función sea llamada por una máquina de Turing, instanciando éstos de acuerdo con sus necesidades específicas. El uso y la combinación de estas máquinas de Turing parametrizadas o m-funciones simplifica la construcción de las máquinas de Turing.

Definición 1.16 (Expansión m-función). El proceso de la traducción final de una m-función a la notación de instrucciones (instrucciones de la forma: $q_i s_j s_k m q_l$), exige conocer el valor de los parámetros con los cuales es invocada, y además conocer el alfabeto de la máquina de la cual la m-función hace parte. Aunque esta traducción no es posible de realizarla con base en la definición de la m-función, sí es posible hacer una traducción parcial a la “nomenclatura de instrucciones” que facilite el proceso de traducción final, una vez se conozca la información (valor de los parámetros y alfabeto de la máquina) requerida para ello; este proceso de traducción parcial lo denominamos la *expansión* de la m-función.

Los ejemplos 1.10 (pág. 39), 1.11 (pág. 40) y 1.12 (pág. 40) operan bajo la hipótesis de que existe un símbolo e como el símbolo más a la izquierda en la cinta; además suponen que los símbolos principales (denotados con s_i) están intercalados con símbolos auxiliares (denotados con m_i), tal como lo indica la tabla 1.13.

Ejemplo 1.10. Dado que existe un símbolo e como el símbolo más a la izquierda de la cinta, la m-función $\mathbf{F}(\mathbf{S}, \mathbf{B}, \alpha)$ busca el símbolo α ; si lo encuentra, pasa al estado \mathbf{S} , si no lo encuentra, pasa al estado \mathbf{B} . La m-función $\mathbf{F}(\mathbf{S}, \mathbf{B}, \alpha)$ está dada por la tabla 1.14.

Es frecuente que una m-función -en este caso la m-función $\mathbf{F}(\mathbf{S}, \mathbf{B}, \alpha)$ - esté compuesta por varias m-funciones: para este caso, $\mathbf{F}_1(\mathbf{S}, \mathbf{B}, \alpha)$ y $\mathbf{F}_2(\mathbf{S}, \mathbf{B}, \alpha)$. De las definiciones de las m-funciones $\mathbf{F}_1(\mathbf{S}, \mathbf{B}, \alpha)$ y $\mathbf{F}(\mathbf{S}, \mathbf{B}, \alpha)$ se observa que la definición de $\mathbf{F}_1(\mathbf{S}, \mathbf{B}, \alpha)$ presentada en la tabla 1.15, indica el comportamiento para el símbolo ‘Ninguno’. Por el contrario, la definición para $\mathbf{F}(\mathbf{S}, \mathbf{B}, \alpha)$ presentada en la tabla 1.16, no lo indica. En este caso el comportamiento del símbolo ‘Ninguno’ está implícito en el comportamiento para el “símbolo”

EI	S	Operación	EF
B		$Pe, R, Pe, R, P0, R, R, P0, L, L$	C

Cuadro 1.12: Notación no tradicional: múltiples operaciones.

...	e	s_i	m_i	s_i	s_i	m_i	s_i	s_i	...
-----	---	-------	-------	-------	-------	-------	-------	-------	-----

Cuadro 1.13: Estado de la cinta para los ejemplos 1.10, 1.11 y 1.12.

‘no e’. Señalemos además que la expansión de la m-función $F(S, B, \alpha)$ se presenta en la tabla 1.17.

Ejemplo 1.11. Dado que existe un símbolo e como el símbolo más a la izquierda de la cinta, la m-función $PE(S, \beta)$ imprime el símbolo β al final de la secuencia de símbolos y pasa al estado S . La m-función $PE(S, \beta)$ está dada por la tabla 1.18.

Para simplicar la expansión de la m-función $F(PE_1(S, \beta), S, e)$ se supondrá que el símbolo ‘e’ está en la cinta. Note el lector que la expansión de la m-función $PE(S, \beta)$ se presenta en la tabla 1.19.

Ejemplo 1.12. Con base en que existe un símbolo e como el símbolo más a la izquierda de la cinta, la m-función $PE_2(S, \alpha, \beta)$ imprime los símbolos α y β al final de la secuencia de símbolos y entonces pasa al estado S . Nótese además que la m-función $PE_2(S, \alpha, \beta)$ está dada por la tabla 1.20 (mirar ejercicio 1.8).

1.5. Codificación de las Máquinas de Turing

1.5.1. Codificación de Turing

Para desarrollar la solución al caso general del *problema de la decisión*, Turing utilizó una enumeración muy particular de sus máquinas. Esta enumeración tiene como base un mecanismo de codificación de las mismas. Por otro lado, la codificación de las máquinas es necesaria para definir un procedimiento que sea capaz de ejecutar cualquier máquina de Turing (en la sección correspondiente a la máquina de Turing universal desarrollaremos esta idea).

Presentemos entonces la codificación y la enumeración de máquinas propuesta por Turing. De acuerdo con la definición formal de una máquina de Turing tenemos el siguiente “esquema” para una instrucción: $q_i s_j s_k m q_l$; donde $i, j, k, l \geq 0$ y $m \in M = \{L, R, N\}$. Con base en esta representación, reemplazamos cada instrucción de acuerdo con la siguiente codificación:

q_i : Se reemplaza por una D seguida de A repetida i veces

s_j : Se reemplaza por una D seguida de C repetida j veces

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Estado Inicial	Símbolo	Operación	Estado Final
$F(S, B, \alpha)$	e	L	$F_1(S, B, \alpha)$
	$No\ e$	L	$F(S, B, \alpha)$
$F_1(S, B, \alpha)$	α		S
	$No\ \alpha$	R	$F_1(S, B, \alpha)$
	$Ninguno$	R	$F_2(S, B, \alpha)$
$F_2(S, B, \alpha)$	α		S
	$No\ \alpha$	R	$F_1(S, B, \alpha)$
	$Ninguno$	R	B

Cuadro 1.14: m-función $F(S, B, \alpha)$.

Estado Inicial	Símbolo	Operación	Estado Final
$F_1(S, B, \alpha)$	α		B
	$No\ \alpha$	R	$F_1(S, B, \alpha)$
	$Ninguno$	R	$F_2(S, B, \alpha)$

Cuadro 1.15: m-función $F_1(S, B, \alpha)$.

Acto seguido, todas las instrucciones de la máquina se reescriben utilizando este código y se separan por un punto y coma (;).

Definición 1.17 (Descripción estándar). Cuando describimos el conjunto de instrucciones de nuestra máquina utilizando este sistema de codificación, decimos que la máquina está representada en su descripción estándar. Las instrucciones de la descripción estándar estarán construidas con símbolos del alfabeto $\Sigma = \{A, C, D, R, L, N, ;\}$.

De forma similar, asociamos a cada símbolo del alfabeto Σ un número natural mediante la siguiente función $f: \Sigma \rightarrow \mathbb{N}$; donde:

$$f(s) = \begin{cases} 1 & \text{si } s = A, \\ 2 & \text{si } s = C, \\ 3 & \text{si } s = D, \\ 4 & \text{si } s = L, \\ 5 & \text{si } s = R, \\ 6 & \text{si } s = N, \\ 7 & \text{si } s = ;. \end{cases}$$

Luego reemplazamos cada símbolo de la descripción estándar de una máquina de Turing por el valor numérico que tiene asociado.

Estado Inicial	Símbolo	Operación	Estado Final
$F(S, B, \alpha)$	e	L	$F_1(S, B, \alpha)$
	$No\ e$	L	$F(S, B, \alpha)$

Cuadro 1.16: m-función $F(S, B, \alpha)$.

EA ^a	SL ^b	SE ^c	M ^d	ES ^e	Explicación
q_i	e	e	L	q_{i+1}	Expansión m-función $F(S, B, \alpha)$
q_i	$No\ e$	$No\ e$	L	q_i	
q_i	\square	\square	L	q_i	
q_{i+1}	α	α	N	S	Expansión m-función $F_1(S, B, \alpha)$
q_{i+1}	$No\ \alpha$	$No\ \alpha$	R	q_{i+1}	
q_{i+1}	\square	\square	R	q_{i+2}	
q_{i+2}	α	α	N	S	Expansión m-función $F_2(S, B, \alpha)$
q_{i+2}	$No\ \alpha$	$No\ \alpha$	R	q_{i+1}	
q_{i+2}	\square	\square	R	B	

^aEA: Estado Actual^bSL: Símbolo Leer^cSE: ímbole Escribir^dM: Movimiento^eES: Estado SiguienteCuadro 1.17: Expansión m-función $F(S, B, \alpha)$.

Definición 1.18 (Número de descripción). El número así obtenido es denominado el *número de descripción* de la máquina codificada.

Ejemplo 1.13. Hallemos la descripción estándar y el número de descripción para la máquina \mathcal{MT} :

$i_1: q_0 \square / R q_1$

$i_2: q_1 / / R q_1$

$i_3: q_1 \square \square L q_2$

$i_4: q_2 / \square N stop$

Renombremos los símbolos utilizados por la \mathcal{MT} así: \square por s_0 , y $/$ por s_1 . Además renombremos el estado *stop* con el símbolo q_3 . Ahora podemos escribir de nuevo las instrucciones y obtener la descripción estándar de cada una de ellas como lo indica la tabla 1.21.

Realizado este proceso de codificación, tendríamos la siguiente descripción estándar para la máquina \mathcal{MT} :

$DDDCRDA; DADCDCRDA; DADDLDAA; DAADCDNDAAA;$

y su número de descripción sería, entonces:

333253173132325317313343117311323631117.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Estado Inicial	Símbolo	Operación	Estado Final
$PE(S, \beta)$			$F(PE_1(S, \beta), S, e)$
$PE_1(S, \beta)$	Cualquiera	R, R	$PE_1(S, \beta)$
$PE_1(S, \beta)$	Ninguno	$P\beta$	S

Cuadro 1.18: m-función $PE(S, \beta)$.

EA	SL	SE	M	ES	Explicación
q_i	e	e	N	q_{i+1}	Expansión m-función $PE(S, \beta)$
q_i	No e	No e	L	q_i	
q_i	\square	\square	L	q_i	
q_{i+1}	C^a	C	R	q_{i+2}	
q_{i+1}	\square	β	N	S	
q_{i+2}	C	C	R	q_{i+1}	
q_{i+2}	\square	\square	R	q_{i+1}	

^aC: CualquieraCuadro 1.19: Expansión m-función $PE(S, \beta)$.

Podemos observar que para cada máquina de Turing existe un número natural que la representa, mas no todo número natural representa una máquina de Turing. Además, de acuerdo con la enumeración anterior, es posible demostrar que el conjunto de las máquinas de Turing es enumerable (y en particular infinito), y naturalmente lo es el conjunto de los algoritmos que se pueden construir sobre algún lenguaje. Esto debe ser así, si somos consistentes con la idea de que cualquier algoritmo puede ser representado por una máquina de Turing.

1.5.2. Codificación de Gödel

Existen otros procedimientos de codificación para las máquinas de Turing diferentes al propuesto por Turing. Presentamos entonces, uno de estos procedimientos, basado en los números de Gödel.

Para la demostración de su famoso teorema de incompletitud de la aritmética, Kurt Gödel, utilizó la idea hoy conocida como codificación numérica o códigos de números. La potencia de esta técnica produjo su estandarización en el campo de la lógica matemática y teorías afines. En otros términos, Gödel se sirvió de una codificación numérica de las cadenas o palabras del lenguaje de la aritmética. Es decir, mediante una previa ordenación de los símbolos del alfabeto, logró generar una técnica para codificar éste y en general, cualquier lenguaje cuyo alfabeto sea enumerable. Gödel nos mostró, en detalle, cómo codificar las reglas de inferencia de un sistema formal, así como el conjunto de sus axiomas y teoremas.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Estado Inicial	Símbolo	Operación	Estado Final
$\text{PE}_2(S, \alpha, \beta)$			$\text{PE}(\text{PE}(S, \beta), \alpha)$

Cuadro 1.20: m-función $\text{PE}_2(S, \alpha, \beta)$.

	Instrucción original	Instrucción renombrada	Instrucción codificada
i_1	$q_0 \square / R q_1$	$q_0 s_0 s_1 R q_1$	$DDDCRDA$
i_2	$q_1 / / R q_1$	$q_1 s_1 s_1 R q_1$	$DADCDCRDA$
i_3	$q_1 \square \square L q_2$	$q_1 s_0 s_0 L q_2$	$DADDLDAA$
i_4	$q_2 / \square N stop$	$q_2 s_1 s_0 N q_3$	$DAADCDNDAAA$

Cuadro 1.21: Descripción estándar de instrucciones.

El objetivo de la presente sección se limitará a presentar, no lo que Gödel hizo, sino lo esencial de su técnica de codificación y su aplicación a las máquinas de Turing.

Definición 1.19 (Codificación de Gödel). Supongamos que $\Sigma = \{b_1, b_2, \dots\}$ es un alfabeto enumerable y sean Σ^* y Σ^n los conjuntos de palabras y n -tuplas construibles sobre el alfabeto Σ . Llamaremos codificación de Gödel, del conjunto Σ^n , a toda función inyectiva $f: \Sigma^n \rightarrow \mathbb{N}$. Igualmente, la función $f: \Sigma^* \rightarrow \mathbb{N}$ la llamaremos la codificación de Gödel para Σ^* .

El procedimiento o técnica de codificación (o aritmétización) de Gödel está sustentado en el teorema fundamental de la aritmética. Este procedimiento lo podemos dividir en dos partes, a saber:

Primeramente, definimos una función, digamos NG , sobre el conjunto Σ (el alfabeto); posteriormente, definimos una extensión de esta función (o del proceso), apoyándonos en el teorema fundamental de la aritmética. Así, para el caso del alfabeto $\Sigma = \{b_1, b_2, \dots\}$, tenemos

1. $NG(b_i) = i$.
2. Si $\alpha \in \Sigma^*$ y $\alpha \equiv s_1 s_2 \dots s_k$, sea $Pn(k)$ la función que calcula el k -ésimo primo, entonces

$$\begin{aligned} NG(\alpha) &= Pn(1)^{NG(s_1)} \cdot Pn(2)^{NG(s_2)} \dots Pn(k)^{NG(s_k)} \\ &= \prod_{i=1}^k Pn(i)^{NG(s_i)} \end{aligned}$$

La función NG , transforma la palabra $\alpha \in \Sigma^*$ en un número natural $NG(\alpha)$. Este número se conoce como el número de Gödel para α , según la enumeración o codificación NG .

Ejemplo 1.14.

1. Consideremos el alfabeto Σ cuyos elementos son los símbolos de la siguiente lista:

$$+, \cdot, (,), =, 0, 1, \dots, 9, a, b, x_1, x_2, \dots, x_i, \dots$$

2. Definimos la función NG , que codifica los símbolos de Σ , como sigue:

$$NG(s) = \begin{cases} 1 & \text{si } s = + \\ 2 & \text{si } s = \cdot \\ 3 & \text{si } s = (\\ 4 & \text{si } s =) \\ 5 & \text{si } s == \\ 6 & \text{si } s = 0 \\ 7 & \text{si } s = 1 \\ \vdots & \\ 15 & \text{si } s = 9 \\ 16 & \text{si } s = a \\ 17 & \text{si } s = b \\ 18 & \text{si } s = x_1 \\ \vdots & \end{cases}$$

3. ¿Cuál es el código o número de Gödel de la expresión α ?

$$\begin{aligned}\alpha &\equiv s_1 s_2 s_3 s_4 s_5 s_6 \\ &\equiv 2 \cdot 5 = 10.\end{aligned}$$

De acuerdo con esta codificación o enumeración la expresión α tendría el siguiente número de Gödel:

$$\begin{aligned}
NG(2 \cdot 5 = 10) &= \prod_{i=6}^k Pn(i)^{NG(s_i)} \\
&= 2^{NG(2)} \cdot 3^{NG(\cdot)} \cdot 5^{NG(5)} \cdot 7^{NG(=)} \cdot 11^{NG(1)} \cdot 13^{NG(0)} \\
&= 2^8 \cdot 3^2 \cdot 5^{11} \cdot 7^5 \cdot 11^7 \cdot 13^6 \\
&= 177849083895509989462500000000.
\end{aligned}$$

Como la función NG es inyectiva, podemos garantizar la unicidad del código y la existencia de códigos diferentes para palabras diferentes de Σ^* . Igualmente, podemos saber si dado un $n \in \mathbb{N}$ existe una palabra de Σ^* codificada por dicho número. Y en caso de que exista, podemos determinar tal palabra; para lograrlo es suficiente hallar la factorización prima de n en \mathbb{N} .

Ejemplo 1.15. Para la función de codificación presentada en el ejemplo 1.14 (pág. 45), ¿Existe una palabra de Σ^* , cuyo código sea $n = 2250$?

Solución:

1. Factorizamos a n , esto es, $n = 2^1 \cdot 3^2 \cdot 5^3$.
2. Hallamos, si existen, $s_1, s_2, s_3 \in \Sigma$, tales que $NG(s_1) = 1$; $NG(s_2) = 2$ y $NG(s_3) = 3$.
Luego $s_1 = +$, $s_2 = \cdot$ y $s_3 = ($
3. Luego, $\alpha \equiv + \cdot ($

Definición 1.20 (Codificación de Gödel: instrucción). El alfabeto Σ para una instrucción de una máquina de Turing está dado por:

$$\Sigma = \{R, L, N, s_0, \dots, s_m, q_0, \dots, q_n\}.$$

Definimos la función NG que codifica los símbolos de Σ por:

$$NG(a) = \begin{cases} 3 & \text{si } a = R, \\ 5 & \text{si } a = L, \\ 7 & \text{si } a = N, \\ 9 & \text{si } a = s_0, \\ 11 & \text{si } a = q_0, \\ 13 & \text{si } a = s_1, \\ 15 & \text{si } a = q_1, \\ \vdots & \\ 4i + 9 & \text{si } s = s_i, \\ 4i + 11 & \text{si } s = q_i. \end{cases}$$

El esquema para un instrucción de una máquina de Turing es: $q_i \ s_j \ s_k \ m \ q_l$, donde $i, j, k, l \geq 0$ y $m \in M = \{L, R, N\}$. Si definimos los símbolos a_1, a_2, a_3, a_4, a_5 por: $a_1 = q_i$, $a_2 = s_j$, $a_3 = s_k$, $a_4 = m$ y $a_5 = q_l$, entonces de acuerdo con la codificación anterior para el alfabeto Σ , el número de Gödel asociado con una instrucción, está dado por:

$$NG(i) = \prod_{k=1}^5 Pn(k)^{NG(a_k)}.$$

Ejemplo 1.16. Sean las instrucciones:

$i_1: q_0 \ s_0 \ s_1 \ R \ q_1$

$i_2: q_1 \ s_1 \ s_1 \ R \ q_1$.

Su número de Gödel está dado por:

$$\begin{aligned} NG(i_1) &= \prod_{k=1}^5 Pn(k)^{NG(a_k)} \\ &= 2^{NG(q_0)} \cdot 3^{NG(s_0)} \cdot 5^{NG(s_1)} \cdot 7^{NG(R)} \cdot 11^{NG(q_1)} \\ &= 2^{11} \cdot 3^9 \cdot 5^{13} \cdot 7^3 \cdot 11^{15} \\ &= 705043151787065817777975000000000000. \end{aligned}$$

$$\begin{aligned} NG(i_2) &= \prod_{k=1}^5 Pn(k)^{NG(a_k)} \\ &= 2^{NG(q_1)} \cdot 3^{NG(s_1)} \cdot 5^{NG(s_1)} \cdot 7^{NG(R)} \cdot 11^{NG(q_1)} \\ &= 2^{15} \cdot 3^{13} \cdot 5^{13} \cdot 7^3 \cdot 11^{15} \\ &= 91373592471603729984025560000000000000. \end{aligned}$$

Definición 1.21 (Codificación de Gödel: máquina de Turing). Sea \mathcal{MT} una máquina de Turing compuesta por un conjunto de instrucciones $\{i_1, i_2, \dots, i_n\}$. El número de Gödel para \mathcal{MT} está dado por:

$$NG(\mathcal{MT}) = \prod_{k=1}^n Pn(k)^{NG(i_k)}.$$

Ejemplo 1.17. Sea \mathcal{MT} una máquina de Turing compuesta por las instrucciones:

$i_1: q_0 \ s_0 \ s_1 \ R \ q_1$

$i_2: q_1 \ s_1 \ s_1 \ R \ q_1$.

El número de Gödel de la máquina de Turing \mathcal{MT} , está dado por:

1.6. Máquina Universal de Turing

Alan Mathison Turing no sólo definió formalmente el concepto de algoritmo por medio de sus máquinas de Turing, sino que además (en el mismo artículo donde describió la noción de máquina de Turing), construyó el modelo teórico de nuestros computadores actuales por medio de una máquina abstracta conocida en nuestros días como *máquina universal de Turing*. Posteriormente a estos trabajos, Turing dirigió su interés hacia el área de la Inteligencia Artificial. En esta área se destaca su test para verificar si un computador goza de la propiedad de “inteligencia”, test hoy conocido como la *verificación de Turing*.

La máquina universal de Turing es una máquina de Turing muy particular. Esta máquina recibe como entrada una máquina de Turing y una secuencia de datos iniciales; así, la máquina universal de Turing realiza la ejecución de la máquina de Turing (dada como entrada) con la secuencia de datos iniciales. Expresaremos, en notación funcional, esta noción como sigue: sean \mathcal{U} , la máquina universal de Turing, \mathcal{MT} una máquina de Turing, α una secuencia de datos iniciales y β una secuencia de datos de salida, entonces: $\mathcal{U}(\mathcal{MT}, \alpha) = \beta$, si y sólo si $\mathcal{MT}(\alpha) = \beta$.

En el lenguaje informático lo expresamos por: un computador (máquina universal de Turing) ejecuta un algoritmo (máquina de Turing) con unos datos de entrada (secuencia de datos iniciales) y genera unos datos de salida (secuencia de datos salida).

Como mencionamos anteriormente, la máquina universal de Turing es una máquina de Turing muy particular: antes de comenzar su ejecución, los datos de entrada (máquina de Turing, secuencia de datos iniciales) deben estar en la cinta. La máquina de Turing actúa como “primer” dato de entrada, y debe estar expresada en su descripción estándar; a su derecha se escribe la secuencia de datos iniciales, como lo indica la tabla 1.22.

... Descripción estándar de una MT secuencia datos iniciales ...

Cuadro 1.22: Entrada para la máquina universal de Turing.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados.

Esta guia fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
EDICT. GUIA MEX - Editorial Guias Mexico

La máquina universal de Turing está compuesta por un conjunto de instrucciones que “saben” cómo “ejecutar” la máquina de Turing (expresada en su descripción estándar) con la secuencia de datos iniciales. La descripción exacta del comportamiento de la máquina universal de Turing es bastante compleja (por ello remitimos al lector interesado a la bibliografía).

En resumen podemos sintetizar lo dicho como sigue: la máquina universal de Turing debe “capturar” la situación actual (estado actual, símbolo actual) de la máquina de Turing. Entonces, debe ir a buscar una instrucción para esta situación actual (en la sección de la descripción estándar de la máquina); si la encuentra, debe realizar lo que esta instrucción indica (en la sección de la secuencia de datos iniciales), almacenando cuál era la posición de la máquina (que está ejecutando) sobre los mismos; luego va de nuevo y busca una instrucción para la nueva situación actual y así continúa su proceso hasta que la máquina de Turing se detenga (si éste fuese el caso).

1.7. El Problema de la Parada

Antes de realizar la presentación formal del problema de la parada, consideremos las siguientes definiciones y “propiedades” fundamentales para nuestro trabajo.

1. Una función $f(x)$ es efectivamente calculable si y sólo si existe un algoritmo que la calcula.
2. Dado el hecho de que los algoritmos se presentan como constructos de un lenguaje en particular, podemos pensar en una ordenación de los mismos. Primero, tendríamos los algoritmos de longitud uno (si existen), luego los algoritmos de longitud dos, y, así sucesivamente. Para los algoritmos de longitud n , podemos realizar un ordenamiento lexicográfico, ordenamiento con el cual obtendríamos finalmente una enumeración de todos los algoritmos. Denotemos esta enumeración por: $A_1, A_2, \dots, A_N, \dots$
3. Sea $f_i(x)$, la función efectivamente calculable, calculada por el algoritmo A_i .
4. Definamos una nueva función $g(x) = f_x(x) + 1$, para toda x . Podemos observar que $g(x)$ es una función efectivamente calculable, dado que se obtiene sumando uno al algoritmo usado para calcular la función $f_x(x)$. Este procedimiento es algorítmico.
5. Dado que $g(x)$ es una función efectivamente calculable, debe existir una función $f_i(x)$ tal que, $f_i(x) = g(x)$. Es decir, debe existir un algoritmo E_i que calcula la función $g(x)$.
- 6.

Teorema 1.1. *No existe $f_i(x)$ tal que, $f_i(x) = g(x)$. Es decir, $g(x)$ es una función que no es efectivamente calculable.*

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Demostración.

- a) $g(x) = f_x(x) + 1$ para todo x .
- b) Sea $f_i(x) = g(x) = f_x(x) + 1$ para todo x .
- c) Para $x = i$ tenemos $f_i(i) = g(i) = f_i(i) + 1$, lo cual es una contradicción.
- d) Por lo tanto, no existe $f_i(x)$, tal que $f_i(x) = g(x)$.

□

Por una parte, las propiedades y razonamientos implicados en los numerales 1 a 5 nos indican que $g(x)$ es una función efectivamente calculable, pero el numeral (6) nos indica que $g(x)$ no es una función efectivamente calculable. ¿Cuál de los razonamientos es incorrecto?

El razonamiento incorrecto es el realizado en los numerales 1 a 5. En particular el numeral 2 supone la existencia del ordenamiento de los posibles algoritmos. Dado varios algoritmos de longitud n , no vemos inconveniente en realizar un orden lexicográfico sobre ellos. Entonces, ¿cuál es el error en este punto? El error consiste en suponer que contamos con un criterio de decidibilidad para los algoritmos, es decir, que dada una secuencia de palabras, se pueda identificar si éstas forman o no un algoritmo. La noción intuitiva de algoritmo nos exige la ejecución de un número finito de instrucciones en un tiempo finito. Con respecto al número finito de instrucciones, no existe inconveniente en su detección. El problema consiste en que no contamos con un procedimiento efectivo que nos permita decidir si un conjunto de instrucciones finitas se detendrá o no a partir de una secuencia de datos iniciales y, al no contar con este procedimiento efectivo, entonces no podemos identificar el conjunto de instrucciones como un algoritmo (se detiene) o como un no algoritmo (no se detiene). Este problema de determinar si un conjunto de instrucciones se detiene o no, es conocido como el problema de la parada.

El *problema de la decisión*, en su caso general, fue replanteado por Turing en los siguientes términos: ¿Es posible construir una máquina de Turing capaz de responder si una máquina de Turing se detendrá o no para una secuencia de datos iniciales determinada? Expresado en otros términos:

Sean, \mathcal{MT} una máquina de Turing y α una secuencia de datos iniciales. Existe una máquina de Turing \mathcal{P} , tal que:

$$\mathcal{P}(\mathcal{MT}, \alpha) = \begin{cases} 1 & \text{si } \mathcal{MT} \text{ se detiene con } \alpha, \\ 0 & \text{si } \mathcal{MT} \text{ no se detiene con } \alpha. \end{cases}$$

Esta formulación (de Turing) se conoce como *el problema de la parada*.

El lector observará que para ciertas máquinas de Turing y para ciertas secuencias de datos iniciales es posible determinar si la máquina se detendrá o no. Por ejemplo, para la máquina \mathcal{MT} construida en el ejemplo 1.1 (pág. 28) y la secuencia de datos iniciales vacía, la máquina no se detiene. Por el contrario, la máquina \mathcal{MT} construida en el ejemplo 1.2

(pág. 29) y para una secuencia de datos iniciales constituida por dos números naturales, expresados en el código de “palitos”, sí se detiene. Pero estas, son soluciones al problema de la parada para un caso particular, y nos interesa la solución al problema de la parada para su caso general, es decir: ¿es posible construir una máquina de Turing \mathcal{P} que resuelva el problema de la parada, para cualquier máquina de Turing y para cualquier secuencia de datos iniciales?

Teorema 1.2. *Si \mathcal{MT} es una máquina de Turing y α es una secuencia de datos iniciales, entonces no existe una máquina de Turing \mathcal{P} , tal que:*

$$\mathcal{P}(\mathcal{MT}, \alpha) = \begin{cases} 1 & \text{si } \mathcal{MT} \text{ se detiene con } \alpha, \\ 0 & \text{si } \mathcal{MT} \text{ no se detiene con } \alpha. \end{cases}$$

Demostración. La demostración procederá por *reduction ad absurdum*. Supongamos que existe \mathcal{P} , entonces podemos definir una nueva máquina de Turing \mathcal{H} por:

$$\mathcal{H}(\alpha) = \begin{cases} 1 & \text{si } \mathcal{P}(\mathcal{MT}_\alpha, \alpha) = 0, \\ \text{no se detiene} & \text{si } \mathcal{P}(\mathcal{MT}_\alpha, \alpha) = 1. \end{cases}$$

Como \mathcal{H} es una máquina de Turing le corresponde un número de descripción; denominemos este número \mathcal{MT}_β , e invoquemos a \mathcal{H} con el parámetro β .

Entonces, $\mathcal{H}(\beta) = 1$ si $\mathcal{P}(\mathcal{MT}_\beta, \beta) = 0$. De acuerdo con la definición de \mathcal{P} , $\mathcal{P}(\mathcal{MT}_\beta, \beta) = 0$, lo cual significa que la máquina \mathcal{MT}_β con la entrada β no se detiene. Pero la máquina \mathcal{MT}_β es la máquina \mathcal{H} y estamos afirmando que $\mathcal{H}(\beta) = \mathcal{MT}_\beta(\beta) = 1$, es decir, que la máquina \mathcal{MT}_β con la entrada β se detiene, y esto es una contradicción. Un razonamiento similar puede ser construido para, $\mathcal{H}(\beta) = 0$ no se detiene. Podemos entonces concluir que no existe la máquina $\mathcal{P}(\mathcal{MT}, \alpha)$. \square

El teorema anterior expresado en términos del problema de la decisión nos permite afirmar el problema de la decisión; el caso general, no tiene solución.

El problema de la parada no es el único problema que no puede ser resuelto por una máquina de Turing. Como un ejemplo adicional presentamos el problema conocido como *el problema del castor afanoso (the busy beaver problem)*.

Ejemplo 1.18. Sea $\Sigma(n)$ el número máximo de “unos” que puede escribir una máquina de Turing con n estados antes de detenerse (sin contar con el estado donde se detiene), comenzando con la cinta en blanco, con un alfabeto dado por $\{\}$ y en la cual sólo se permiten dos movimientos, a izquierda o a derecha. Por otra parte, la función $S(n)$ representa el número máximo de movimientos realizados por una máquina de Turing con las características anteriores.

Hallar el valor de la función $\Sigma(n)$ es conocido como el problema del castor afanoso. La función $\Sigma(n)$ y la función $S(n)$ fueron sugeridas por Tibor Rado en 1962. La tabla 1.23 muestra los valores conocidos para las funciones $\Sigma(n)$ y $S(n)$.

<i>n</i>	<i>Σ(n)</i>	<i>S(n)</i>
1	1	1
2	4	6
3	6	21
4	13	107
5	$\geq 4,098$	$\geq 47,176,870$
6	$\geq 95,524,079$	$\geq 8,690,333,381,690,951$

Cuadro 1.23: $\Sigma(n)$ y $S(n)$ para $1 \leq n \leq 6$.

A manera de ejemplo, la máquina de Turing que calcula $\Sigma(3)$ está dada por las siguientes instrucciones:

- $i_1: q_1 \square \mid R q_2$
- $i_2: q_1 \mid \mid L q_3$
- $i_3: q_2 \square \mid L q_1$
- $i_4: q_2 \mid \mid R q_2$
- $i_5: q_3 \square \mid L q_2$
- $i_6: q_3 \mid \mid R stop$

El número de máquinas de Turing de n estados con las características descritas para el problema del castor afanoso es $(4(n+1))^{2n}$, dado que para cada estado de no parada hay dos posibles transiciones; entonces hay $2n$ transiciones y cada transición tiene: 2 posibilidades de escribir un símbolo ($\square, |$), dos posibilidades de movimiento (L, R) y $(n + 1)$ posibles estados para ir (contando el estado de parada).

Solucionar el problema del castor afanoso para un n en particular es muy difícil por dos razones: la primera es que el espacio de búsqueda es muy extenso ($(4(n + 1))^{2n}$ máquinas) y la segunda, algunas de estas máquinas pueden no parar y detectar esto puede ser muy difícil.

Para el caso general, el problema del castor afanoso es imposible de solucionar tal como lo indica el siguiente teorema.

Teorema 1.3. *La función $\Sigma(n)$ no es Turing-computable.*

Demostración. La idea de la demostración es demostrar que si $f(x)$ es una función Turing-computable, existe x_0 tal que $\Sigma(x) > f(x)$ para $x \geq x_0$.

1. Sea $f(x)$ una función Turing-computable.
2. $g(x) = \sum_{0 \leq i \leq x} (f(i) + i^2)$ es una función Turing-computable.
3. Sea M_g una máquina de Turing de n estados que calcula la función g . Supongamos que M_g comienza con un bloque de x unos y finaliza con un bloque de $g(x)$ unos, separados al menos por una celda del bloque inicial.

4. Sea \mathcal{M} una máquina que realiza tres cosas: (a) Inicialmente escribe x unos a partir de una cinta en blanco, lo cual puede hacerse con x estados; (b) Ejecuta la máquina \mathcal{M}_g ; por lo tanto en la cinta hay un bloque de x unos y un bloque de $g(x)$ unos, lo cual puede hacerse con $x + n$ estados, (c) Ejecuta de nuevo la máquina \mathcal{M}_g , por lo tanto en la cinta hay un bloque de x unos, un bloque de $g(x)$ unos y un bloque de $g(g(x))$ unos, lo cual puede hacerse con $x + 2n$ estados.
5. $\Sigma(x + 2n) \geq x + g(x) + g(g(x))$. Porque la máquina que calcula $\Sigma(x)$ debe escribir por lo menos tantos unos como la máquina \mathcal{M} .
6. Existe c_1 tal que $x^2 > x + 2n$, para todo $x \geq c_1$ y de la definición de $g(x)$ tenemos que $g(x) \geq x^2$, luego $g(x) > x + 2n$, para todo $x \geq c_1$.
7. De la definición de $g(x)$ tenemos que $g(x) > g(y)$ si $x > y$, luego $g(g(x)) > g(x + 2n)$, para todo $x \geq c_1$.
8. Luego, $\Sigma(x + 2n) \geq x + g(x) + g(g(x)) > g(g(x)) > g(x + 2n) > f(x + 2n)$, para todo $x \geq c_1$.
9. Dado que la función $f(x)$ es arbitraria y la función $\Sigma(x)$ eventualmente es mayor que la función $f(x)$, se sigue que la función $\Sigma(x)$ no es Turing-computable.

□

Como consecuencia del teorema anterior, se sigue que la función $S(n)$ no es Turing-computable.

Teorema 1.4. *La función $S(n)$ no es Turing-computable.*

Demostración.

1. Sea \mathcal{M}_Σ una máquina de Turing de n estados que escribe $\Sigma(n)$ unos antes de detenerse.
2. La máquina \mathcal{M}_Σ debe realizar por lo menos $\Sigma(n)$ movimientos.
3. Entonces $S(n) \geq \Sigma(n)$.
4. Como $\Sigma(n)$ es eventualmente más grande que cualquier función Turing-computable, entonces $S(n)$ también lo es.
5. Luego, $S(n)$ no es una función Turing-computable.

□

1.8. Ejercicios

Ejercicio 1.1. Sea $\Sigma = \{a, b, c, d\}$ un alfabeto y α una palabra de Σ^* sobre la cinta. Diseñe una máquina de Turing, tal que:

1. Elimine la palabra α sobre la cinta.
2. Elimine todos los símbolos a de la palabra α .
3. Elimine todos los símbolos a y el segundo símbolo b de la palabra α .
4. Elimine todos los símbolos a y el último símbolo b de la palabra α .
5. Invierta la palabra α .

Ejercicio 1.2. Dado un símbolo $s \in \Sigma$ sobre la cinta. Diseñe una máquina de Turing que encuentre éste, sin importar en qué posición inicial sobre la cinta, comienza su ejecución.

Ejercicio 1.3. Dos máquinas de Turing son equivalentes si para la misma entrada α sobre la cinta, se obtiene la misma salida β sobre la cinta. Demuestre que para toda máquina de Turing \mathcal{MT} existe una máquina de Turing equivalente \mathcal{MT}' que nunca visita las celdas a la izquierda de la celda desde la cual comienza la máquina \mathcal{MT} su ejecución.

Ejercicio 1.4. Diseñe máquinas de Turing que realicen las conversiones necesarias entre los códigos binario, decimal y “palitos”.

Ejercicio 1.5. Sean $x_1, x_2 \in \mathbb{N}$, demuestre que la función $f(x_1, x_2) = x_1 + x_2$ es una función Turing-computable.

Ejercicio 1.6. Diseñe una máquina de Turing \mathcal{MT} tal que $f_{\mathcal{MT}} = f$, donde la función f está dada por:

1. $f(x_1, x_2, x_3) = x_2$.
2. $f(x_1, x_2, x_3, x_4) = x_4$.
3. $f(x) = 2x$.
4. $f(x, y) = \max(x, y)$.
5. $f(x, y) = \min(x, y)$.
6. $f(x, y) = \text{mcd}(x, y)$ (use el algoritmo de Euclides para el mcd).
7. $f(x, y) = xy$.

Ejercicio 1.7. Traducir la tabla 1.12 (pág. 40) a la “nomenclatura de instrucciones”.

Ejercicio 1.8. Realizar la expansión de la m-función $\mathbf{PE}_2(\mathbf{S}, \alpha, \beta)$ presentada en el ejemplo 1.12 (pág. 40).

Ejercicio 1.9. Construya las siguientes m-funciones:

1. $\mathbf{L}(\mathbf{S})$: Se desplaza una celda a la izquierda y pasa al estado \mathbf{S} .
2. $\mathbf{F}^0(\mathbf{S}, \mathbf{B}, \alpha)$: Similar a la m-función $\mathbf{F}(\mathbf{S}, \mathbf{B}, \alpha)$, pero antes de pasar al estado \mathbf{S} se desplaza a la izquierda.
3. $\mathbf{CP}(\mathbf{S}, \mathbf{A}, \mathbf{C}, \alpha, \beta, \gamma)$: A partir de un símbolo inicial a la izquierda γ , compara las dos primeras ocurrencias de los símbolos α y β . Si alguno de los dos símbolos no está, pasa al estado \mathbf{C} ; si los símbolos son diferentes, pasa al estado \mathbf{A} y si los dos símbolos son iguales, pasa al estado \mathbf{S} .

Ejercicio 1.10. Decimos que una máquina de Turing acepta una palabra α si comenzando con α sobre la cinta, la máquina se detiene con la cinta en blanco. Construya una máquina de Turing que acepte únicamente palabras de la forma $10, 1100, 111000, \dots, 1^n0^n, \dots$, para $n \geq 1$.

Ejercicio 1.11. Proponga una definición de una máquina de Turing que satisfaga las siguientes condiciones:

1. No determinística.
2. Probabilística.
3. Que opere sobre dos cintas.
4. Que opere sobre n cintas.
5. Que opere con dos cabezas de lectura-escritura.
6. Que opere con n cabezas de lectura-escritura.
7. 2-dimensional.
8. n -dimensional.

El artículo fundacional sobre las máquinas de Turing es [38]. Por su parte, [7, 16, 21, 26, 27, 29, 30, 33, 34, 36], entre otros, ofrecen algunos elementos contextuales para la noción de máquina de Turing. La definición para las funciones Turing-computables fue adaptada de [7]. La codificación de las máquinas de Turing y la definición de m-función fueron adaptadas

de [33], el cual a su vez es una adaptación de [38]. Con relación a la máquina universal de Turing, [29] presenta una construcción alternativa a la propuesta por Turing y [32] ofrece algunos elementos para construir ésta, a partir de las m-funciones propuestas por Turing. La situación presentada como antecedente al problema de la parada fue tomada de [7]. El problema del castor afanoso, presentado en el ejemplo 1.18 (pág. 51) fue tomado de [31, 25].

Capítulo 2

Recursividad

En esta sección consagraremos nuestros esfuerzos al desarrollo de una teoría intuitiva, teoría que construiremos para una cierta clase de predicados y funciones de la teoría de números. El concepto de función recursiva primitiva, los conceptos de función parcial y función recursiva general o función μ -recursiva, así como las herramientas teóricas que presentaremos en esta sección, constituyen, a bien decir, elementos de mucha importancia tanto en la lógica como en la informática teórica.

En los inicios de la década de los treinta, en busca de la noción de procedimiento efectivo, se introduce la teoría de las funciones recursivas. En ese entonces se hace de la noción de función recursiva general un eje central. En particular Kurt Gödel utiliza una subclase específica de esta noción (las funciones primitivas recursivas) para codificar la teoría de números; Gödel constituye esta noción como estrategia clave para la construcción de la demostración de su teorema de incompletitud.

Anotemos inicialmente que para infinitas funciones numérico-teóricas no es posible hallar una expresión algebraica que defina su regla de asociación. Es justamente este aspecto el que le confiere a la recursión toda su potencia metodica.

2.1. Funciones y Relaciones Numérico-Teóricas

Definición 2.1 (Relaciones numérico-teóricas). Sea $R(x_1, x_2, \dots, x_n)$ una relación n -ádica tal que los argumentos o evaluaciones de x_1, x_2, \dots, x_n sean números naturales. Tal relación la denominaremos relación numérico-teórica, es decir, R es una relación numérico-teórica, si y sólo si, $R \subseteq \mathbb{N}^n$; $n \geq 1$.

Ejemplo 2.1. Las siguientes son relaciones numérico-teóricas:

- $R_1 = \{(x, y, z) \mid x^2 - y = z; x, y, z \in \mathbb{N}\}$.
- $R_2 = \{(x_1, x_2, x_3, x_4, x_5) \in \mathbb{N}^5 \mid x_1 - x_3 = 0 \wedge 2x_4 = x_5\}$.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Definición 2.2 (Funciones numérico-teóricas). Una función numérico-teórica es una función definida en \mathbb{N}^n y evaluada en \mathbb{N} , es decir, f es una función numérico-teórica, si y sólo si, $f: \mathbb{N}^n \rightarrow \mathbb{N}$. La clase de las funciones numérico-teóricas \mathcal{F} , la definimos como: $\mathcal{F} = \bigcup_{n < \omega} \mathcal{F}_n$, donde:

$$\begin{aligned}\mathcal{F}_0 &= \{f \mid f: \mathbb{N} \rightarrow \mathbb{N}\}, \\ \mathcal{F}_1 &= \{f \mid f: \mathbb{N}^2 \rightarrow \mathbb{N}\}, \\ &\vdots \\ \mathcal{F}_n &= \{f \mid f: \mathbb{N}^{n+1} \rightarrow \mathbb{N}\}, \\ &\vdots\end{aligned}$$

Ejemplo 2.2. Las siguientes funciones, son funciones numérico-teóricas:

1. $f: \mathbb{N}^3 \rightarrow \mathbb{N}$, donde $f(x, y, z) = 2x + y + z$.
2. $f: \mathbb{N}^4 \rightarrow \mathbb{N}$, donde $f(x_1, x_2, x_3, x_4) = (x_4)^2 - x_1$.

2.2. Funciones Primitivas Recursivas

Nuestro objetivo en esta sección estará centrado en la construcción de una subclase de funciones numérico-teóricas que llamaremos funciones primitivas recursivas (FPR).

Antes de introducirnos en la teoría de la sección, es útil tener en cuenta que para definir una función o un predicado por recursión, procedemos mediante el siguiente esquema: digamos que para φ ,

$$\begin{aligned}\varphi(0) &= h(x), \\ \varphi(n+1) &= g(n, \varphi(n)),\end{aligned}$$

donde, g y h son funciones numérico-teóricas.

Siguiendo el procedimiento de definición por construcción, veremos que las funciones que deseamos reconocer como primitivas recursivas, son justamente aquellas que constructivamente generamos mediante el uso de esquemas de construcción aplicados, un número finito de veces, sobre un conjunto de funciones de base (o axiomas) que, de entrada, declaramos como FPR.

Para definir, constructivamente, el conjunto de las FPR emplearemos el siguiente procedimiento.

1. Definimos un conjunto de axiomas o condiciones límite.

Las siguientes funciones, llamadas funciones de base, serán nuestras condiciones límite. Por ello las declaramos axiomáticamente como FPR.

- a) $z(x) = 0$, la función cero.
- b) $s(x) = x + 1$, la función sucesor.
- c) $I_k^n(x_1, x_2, \dots, x_n) = x_k$, el esquema de funciones k-ésima proyección.
2. Dotamos al sistema de dos esquemas de construcción, con el propósito de construir nuevas funciones a partir de funciones ya construidas. Los esquemas de construcción son:

- a) Esquema de composición o de sustitución.

Este esquema lo definimos mediante la siguiente expresión:

$$f(x_1, x_2, \dots, x_n) = g(h_1(x_1, x_2, \dots, x_n), \dots, h_m(x_1, x_2, \dots, x_n)),$$

donde, $g: \mathbb{N}^m \rightarrow \mathbb{N}$ y cada $h_i: \mathbb{N}^n \rightarrow \mathbb{N}$ son funciones bien definidas.

Es decir, si la función $g(y_1, \dots, y_m)$ y cada función $h_i(z_1, \dots, z_n)$ son FPR, entonces, la función $f(x_1, x_2, \dots, x_n)$, definida por el esquema anterior, es una FPR.

El valor de f , para una interpretación de x_1, x_2, \dots, x_n , se obtiene evaluando a g en los valores obtenidos para las h_i con la interpretación de las x_i .

Si una función proviene de la aplicación del esquema de composición, decimos que f está definida por composición de las funciones g y h_1, h_2, \dots, h_m .

- b) Esquema de recurrencia primitiva.

Este esquema lo definimos mediante la siguiente expresión:

$$\begin{aligned} f(x_1, x_2, \dots, x_n, 0) &= g(x_1, x_2, \dots, x_n), \\ f(x_1, x_2, \dots, x_n, k+1) &= h(x_1, x_2, \dots, x_n, k, f(x_1, x_2, \dots, x_n, k)), \end{aligned}$$

donde, $g: \mathbb{N}^n \rightarrow \mathbb{N}$ y $h: \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ son funciones bien construidas. Es decir, si las funciones $g(y_1, \dots, y_n)$ y $h(z_1, \dots, z_n, z_{n+1}, z_{n+2})$ son FPR, entonces, la función $f(x_1, x_2, \dots, x_n, k)$, definida por el esquema anterior, es una FPR.

En este caso, diremos que f está definida o construida por recurrencia primitiva, mediante las funciones g y h .

Si no existe ambigüedad respecto al contexto de trabajo, simplificaremos la notación introduciendo el símbolo \vec{x}_n en lugar de la n-tupla x_1, x_2, \dots, x_n ; así, el esquema de recurrencia primitiva, quedaría expresado por:

$$\begin{aligned} f(\vec{x}_n, 0) &= g(\vec{x}_n), \\ f(\vec{x}_n, k+1) &= h(\vec{x}_n, k, f(\vec{x}_n, k)). \end{aligned}$$

Definición 2.3 (Función primitiva recursiva (FPR)). Una función f numérico-teórica se dice primitiva recursiva, si y sólo si f es una función de base, o, f se puede obtener a partir de las funciones de base mediante la aplicación de un número finito de veces del esquema de composición y/o del esquema de definición por recurrencia primitiva. Es decir, f es FPR si y sólo si existe una sucesión de funciones f_1, f_2, \dots, f_n tales que:

1. $f_n = f$.
2. Para cada $i \leq n$, f_i es una función de base, o f_i se obtiene de sucesiones anteriores, mediante aplicación finita del esquema de composición y/o del esquema de definición por recurrencia primitiva.

2.3. Construcción de Funciones Recursivas Primitivas

En esta sección presentaremos una serie de funciones recursivas primitivas que ilustran el proceso de construcción de la teoría y que constituyen, por así, decirlo, las herramientas básicas de la teoría.

1. La siguientes funciones son FPR.

- a) $i(x) = x$, ya que, $i(x) = I_1^1(x)$.
- b) $f_k(x) = k$, ya que, $f_k(x) = \underbrace{S(S(S(\dots(Z(x)))))}_{k \text{ veces}}$.
- c) $f(x) = x + 2$, ya que, $f(x) = S(S(x))$.

2. La adición, $+(x, y) = x + y$

Si deseamos adicionarle al número x el número y , podemos pensarlo como $(\dots(((x+0)+1)+1)+\dots+1)$, lo que nos sugiere una construcción por recursividad. Así:

$$\begin{aligned} +(x, 0) &= x \\ +(x, n+1) &= +(+((x, n), 1)) = (x + n) + 1. \end{aligned}$$

Para llevarla al esquema de definición por recurrencia primitiva, lo expresamos por:

$$\begin{aligned} +(x, 0) &= I_1^1(x), \\ +(x, y+1) &= S(I_3^3(x, y, +(x, y))), \end{aligned}$$

la función $+$, así definida, es FPR, puesto que queda definida por el esquema de definición por recurrencia primitiva aplicado sobre funciones primitivas recursivas.

3. Multiplicación y potenciación

Si observamos los primeros y segundos miembros de las ecuaciones dadas a continuación, podremos constatar que las funciones son FPR, ya que están construidas por recurrencia primitiva sobre funciones primitivas recursivas.

$$a) \cdot(x, y) = x \cdot y$$

$$\begin{aligned}\cdot(x, 0) &= Z(x), \\ \cdot(x, y + 1) &= h(x, y, \cdot(x, y)),\end{aligned}$$

donde, $h(x, y, z) = +(I_3^3(x, y, z), I_1^3(x, y, z))$. Luego $\cdot(x, y) = x \cdot y$ es una FPR.

$$b) g(x, y) = x^y$$

$$\begin{aligned}g(x, 0) &= x^0 = S(Z(x)), \\ g(x, y + 1) &= x^{y+1} = x^y \cdot x = h(x, y, g(x, y)),\end{aligned}$$

donde, $h(x, y, z) = \cdot(I_3^3(x, y, z), I_1^3(x, y, z))$. Luego $g(x, y) = x^y$ es una FPR.

Observación. De acuerdo con esta definición $0^0 = 1$.

4. Función factorial, $f(x) = x!$

$$\begin{aligned}f(0) &= 0! = 1 = S(0), \\ f(n + 1) &= (n + 1)! \\ &= n!(n + 1) \\ &= \cdot(n + 1, f(n)) \\ &= \cdot(S(n), f(n)).\end{aligned}$$

Entonces, $f(x) = x!$ es FPR. Hemos obviado expresar a $f(n + 1)$ mediante una función de tres variables, en aras de la simplificación.

5. Función permutación de variables

Sea $f(x_1, x_2, \dots, x_n) = g(x_{G(1)}, x_{G(2)}, \dots, x_{G(n)})$, entonces, si g es recursiva primitiva f también es FPR.

Sea $f(x_1, x_2, \dots, x_n) = g(I_{G(1)}^n, I_{G(2)}^n, \dots, I_{G(n)}^n)$.
Si por ejemplo,

$$G = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{pmatrix},$$

entonces $f(x_1, x_2, \dots, x_n) = g(x_3, x_1, x_2, x_4)$.

6. Función predecesor: Pr

La función predecesor asocia con cada número natural su predecesor, salvo con cero al que le asocia cero. Definimos la función Pr , por recurrencia primitiva, como sigue:

$$\begin{aligned}Pr(0) &= Z(x), \\ Pr(n + 1) &= I_1^2(n, Pr(n)).\end{aligned}$$

7. Funciones diferencia truncada y diferencia absoluta

a) La función diferencia truncada

$$\dot{-}(x, y) = \begin{cases} x - y & \text{sii } x \geq y, \\ 0 & \text{sii } x < y. \end{cases}$$

Podemos expresar esta función mediante el esquema de definición por recurrencia primitiva sobre FPR, así

$$\begin{aligned}\dot{-}(x, 0) &= I_1^1(x) = x, \\ \dot{-}(x, n + 1) &= Pr(I_3^3(x, n, \dot{-}(x, n))).\end{aligned}$$

Por ejemplo calculemos, $\dot{-}(4, 2) = 2$.

$$\begin{aligned}\dot{-}(4, 2) &= \dot{-}(4, 1 + 1) \\ &= Pr(I_3^3(4, 1, \dot{-}(4, 1))) \\ &= Pr(\dot{-}(4, 1)) \\ &= Pr(Pr(I_3^3(4, 0, \dot{-}(4, 0)))) \\ &= Pr(Pr(\dot{-}(4, 0))) \\ &= Pr(Pr(4)) \\ &= Pr(3) \\ &= 2.\end{aligned}$$

b) La función diferencia absoluta

$$|x - y| = \begin{cases} x - y & \text{si } x \geq y, \\ y - x & \text{si } y \geq x. \end{cases}$$

Observemos que $|x - y|$, es FPR ya que $|x - y| = +(\dot{-}(x, y), \dot{-}(y, x))$.

8. La función cero test \overline{sg} y su complementaria, la función cero test inversa sg

La función cero test está definida por:

$$\overline{sg}(x) = \begin{cases} 1 & \text{si } x = 0, \\ 0 & \text{si } x \neq 0. \end{cases}$$

La función \overline{sg} es recursiva primitiva ya que

$$\begin{aligned}\overline{sg}(0) &= S(0), \\ \overline{sg}(n + 1) &= Z(I_1^2(n, \overline{sg}(n))).\end{aligned}$$

Así,, $\overline{sg}(0) = 1$, $\overline{sg}(1) = Z(0) = 0$, $\overline{sg}(2) = Z(1) = 0$.

La función cero test inversa sg está definida por:

$$sg(x) = \begin{cases} 1 & \text{si } x \neq 0, \\ 0 & \text{si } x = 0. \end{cases}$$

La función sg es FPR, porque:

$$\begin{aligned} sg(x) &= 1 \doteq \overline{sg}(x) \\ &= f_1 \doteq \overline{sg}(x) \\ &= S(Z(x)) \doteq \overline{sg}(x). \end{aligned}$$

9. La función paridad: P

La función paridad está definida por

$$P(x) = \begin{cases} 1 & \text{si } x \text{ es par,} \\ 0 & \text{si } x \text{ es impar.} \end{cases}$$

La función $P(x)$ es FPR porque:

$$\begin{aligned} P(0) &= 1 = f_1(x), \\ P(n+1) &= \overline{sg}(I_2^2(n, P(n))). \end{aligned}$$

2.4. Predicados Primitivos Recursivos

Definición 2.4 (Predicado primitivo recursivo). Un predicado $P(\vec{x})$ se dice primitivo recursivo (PPR), si y sólo si la función característica de $P(\vec{x})$ es primitiva recursiva. La función C_P , designará la función característica de $P(\vec{x})$, donde,

$$C_P(\vec{x}) = \begin{cases} 0 & \text{si } P(\vec{x}), \\ 1 & \text{si } \neg P(\vec{x}). \end{cases}$$

Ejemplo 2.3. Los siguientes son ejemplos de predicados primitivos recursivos.

1. El predicado $P(x, y) : x = y$, es PPR.

$$C_P(x, y) = \begin{cases} 0 & \text{si } x = y, \\ 1 & \text{si } x \neq y, \end{cases} = sg(|x - y|).$$

2. El predicado: *ser menor que*; $<(x, y)$ o $x < y$, es PPR.

$$C_<(x, y) = \begin{cases} 0 & \text{si } x < y, \\ 1 & \text{si } x \geq y, \end{cases} = \overline{sg}(y \dashv x).$$

Teorema 2.1. Si $P(\vec{x})$ y $Q(\vec{x})$ son predicados primitivos recursivos, entonces los predicados: $\neg P(\vec{x})$, $P(\vec{x}) \vee Q(\vec{x})$ y $P(\vec{x}) \wedge Q(\vec{x})$, son predicados primitivos recursivos.

Demostración.

$$\begin{aligned} C_{P \vee Q} &= C_P C_Q, \\ C_{P \wedge Q} &= (C_P + C_Q) \dot{-} C_P C_Q, \\ C_{\neg P} &= 1 \dot{-} C_P. \end{aligned}$$

□

Definición 2.5 (Cuantificadores acotados). Si $P(\vec{x}, y)$ es un predicado de la teoría de números, podemos definir nuevos predicados mediante cuantificación acotada. Para todo entero n tenemos:

$$\begin{aligned} \forall_{y \leq n}(P(\vec{x}, y)) &\stackrel{\text{def}}{\equiv} P(\vec{x}, 0) \wedge P(\vec{x}, 1) \wedge \dots \wedge P(\vec{x}, n) \\ &\stackrel{\text{def}}{\equiv} \forall y(y \leq n \Rightarrow P(\vec{x}, y)); \end{aligned}$$

$$\begin{aligned} \exists_{y \leq n}(P(\vec{x}, y)) &\stackrel{\text{def}}{\equiv} P(\vec{x}, 0) \vee P(\vec{x}, 1) \vee \dots \vee P(\vec{x}, n) \\ &\stackrel{\text{def}}{\equiv} \exists y(y \leq n \wedge P(\vec{x}, y)). \end{aligned}$$

Teorema 2.2. Si $P(\vec{x}, y)$ es un predicado primitivo recursivo y definimos

$$\begin{aligned} R(\vec{x}) &= \exists_{y \leq n}(P(\vec{x}, y)), \\ S(\vec{x}) &= \forall_{y \leq n}(P(\vec{x}, y)), \end{aligned}$$

entonces $R(\vec{x})$ y $S(\vec{x})$ son predicados primitivos recursivos.

Demostración.

1. $C_P(\vec{x}, y)$ es FPR.
2. Sea $C_R(\vec{x}) = \prod_{i=0}^n C_P(\vec{x}, i)$.
3. Sea $C_S(\vec{x}) = sg(\sum_{i=0}^n C_P(\vec{x}, i))$.
4. Las funciones $C_R(\vec{x})$ y $C_S(\vec{x})$ son FPR.

□

Teorema 2.3. Sea $P(\vec{x}, y)$ un PPR y sea $f(\vec{x})$ una FPR. Si definimos

$$\begin{aligned} R(\vec{x}) &= \exists_{y \leq f(\vec{x})} (P(\vec{x}, y)), \\ S(\vec{x}) &= \forall_{y \leq f(\vec{x})} (P(\vec{x}, y)), \end{aligned}$$

entonces $R(\vec{x})$ y $S(\vec{x})$ son predicados primitivos recursivos.

Demostración.

1. $C_P(\vec{x}, y)$ es FPR.
2. Sea $C_R(\vec{x}) = \prod_{i=0}^{f(\vec{x})} C_P(\vec{x}, i)$.
3. C_R es FPR.
4. Sea $C_S(\vec{x}) = sg(\sum_{i=0}^{f(\vec{x})} C_P(\vec{x}, i))$.
5. C_S es FPR.

□

2.5. Funciones Definidas Mediante Condiciones

Las funciones definidas por condiciones son bastante familiares en matemáticas. Una instancia posible es la siguiente: Sea $P(x)$ el predicado x es par, entonces:

$$g(x) = \begin{cases} 2x & \text{si } P(x), \\ 3x & \text{si } \neg P(x). \end{cases}$$

Observemos que si expresamos a $\neg P(x)$ como $P_1(x)$, entonces, $P(x)$ y $P_1(x)$ definen conjuntos que forman una partición de \mathbb{N} . Además, estos dos predicados son PPR. De allí que g está definida mediante condiciones y funciones que son recursivas primitivas. El siguiente teorema generaliza este procedimiento y nos garantiza que la función así definida es recursiva primitiva.

Teorema 2.4. Si $g_1(\vec{x}), g_2(\vec{x}), \dots, g_m(\vec{x})$; son m funciones FPR, y $P_1(\vec{x}), P_2(\vec{x}), \dots, P_m(\vec{x})$; son m predicados PPR, tales que los predicados P_1, P_2, \dots, P_m forman una partición de \mathbb{N}^n , entonces la siguiente función es recursiva primitiva:

$$f(\vec{x}) = \begin{cases} g_1(\vec{x}) & \text{si } P_1(\vec{x}), \\ g_2(\vec{x}) & \text{si } P_2(\vec{x}), \\ \vdots & \\ g_m(\vec{x}) & \text{si } P_m(\vec{x}). \end{cases}$$

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Demostración.

1. Sea C_{P_i} la función característica del predicado P_i .
2. Cada \vec{x} satisface un y sólo un predicado P_k , para $1 \leq k \leq m$.
3. Luego, $f(\vec{x}) = g_k(\vec{x})\overline{sg}(C_{P_k}(\vec{x})) = g_k(\vec{x})$, para algún k tal que $P_k(\vec{x})$.
4. Luego, para cualquier $\vec{x} \in \mathbb{N}^n$ se tiene que

$$f(\vec{x}) = \sum_{i=1}^m g_i(\vec{x})\overline{sg}(C_{P_i}(\vec{x})).$$

5. La afirmación de que f es FPR está sustentada en el teorema siguiente.

□

El siguiente teorema nos proporciona herramientas para construir FPR mediante iteraciones acotadas.

Teorema 2.5. *Si $g(\vec{x}, y)$ es una FPR, entonces las siguientes funciones son FPR.*

1. $f(\vec{x}, y) = \sum_{i=0}^y g(\vec{x}, i).$
2. $h(\vec{x}, y) = \prod_{i=0}^y g(\vec{x}, i).$

Demostración.

1. Función $f(\vec{x}, y)$
 - a) $f(\vec{x}, 0) = g(\vec{x}, 0).$
 - b) $f(\vec{x}, n + 1) = \sum_{i=0}^{n+1} g(\vec{x}, i) = g(\vec{x}, n + 1) + \sum_{i=0}^n g(\vec{x}, i).$
 - c) $f(\vec{x}, n + 1) = +(g(\vec{x}, n + 1), f(\vec{x}, n)).$
 - d) f es FPR.
2. Función $h(\vec{x}, y)$
Ejercicio 2.11.

□

2.6. Funciones Recursivas

Definición 2.6 (Función regular). La función $g(\vec{x}, y)$ es una función regular si g verifica la condición: $\forall \vec{x} \exists y (g(\vec{x}, y) = 0)$.

Ejemplo 2.4. Sea $P(y)$ el predicado: *y es un número primo*. Sea $f(x, y)$ la siguiente función:

$$f(x, y) = \begin{cases} 0 & \text{si } y > x \wedge P(y), \\ 1 & \text{si } y \leq x \vee \neg P(y). \end{cases}$$

Verifiquemos que $f(x, y)$ es regular.

1. $\forall x (x \in \mathbb{N} \Rightarrow \exists y (y \in \mathbb{N} \wedge y > x))$.
2. Dado un $x_0 \in \mathbb{N}$, existen infinitos números primos y , tales que $y > x_0$.
3. Luego, $\forall x \exists y (f(x, y) = 0)$.

Ejemplo 2.5. La función $f(x, y) = x \dot{-} y$ es regular, ya que:

1. $\forall x (x \in \mathbb{N} \Rightarrow \exists y (y \in \mathbb{N} \wedge y \geq x))$.
2. Dado un $x_0 \in \mathbb{N}$, existen infinitos y_0 , tales que $x_0 \dot{-} y_0$.
3. Luego, $\forall x \exists y (f(x, y) = 0)$.

Definición 2.7 (Operador de minimalización μ). Si $f(\vec{x}, y)$ es una función, entonces definimos el operador μ (no restringido) por: $g(\vec{x}) = \mu_y (f(\vec{x}, y) = 0)$, que leemos: *g asocia a \vec{x} , el menor y tal que $f(\vec{x}, y) = 0$* .

Si la función $f(\vec{x}, y)$ es una función regular, entonces la función $g(\vec{x})$ es una función total, pero si $f(\vec{x}, y)$ no es una función regular, entonces el operador de minimalización $\mu_y (f(\vec{x}, y) = 0)$ no está definido para toda \vec{x} , y por lo tanto, $g(\vec{x})$ es una función parcial.

Ejemplo 2.6.

1. La función $f(x, y, z) = (x \dot{-} z) + (y \dot{-} z)$ es regular para la variable z (ejercicio 2.13).
2. Podemos construir una función $g(x, y)$ por minimalización como sigue: $g(x, y) = \mu_z (f(x, y, z) = 0)$.
3. Obsérvese que $g(x, y) = \max(x, y)$.

Ejemplo 2.7. Dada la función regular $f(x, y) = x \dot{-} y$, podemos definir, por minimalización (operador μ), la función $g(x) = \mu_z (f(x, z) = 0)$.

Para construir la clase de las funciones recursivas totales (llamadas funciones recursivas), introduciremos un tercer esquema de construcción de funciones (mediante el operador μ , o sea, mediante la operación de minimalización).

Definición 2.8 (Funciones recursivas). La clase de las funciones recursivas es la menor clase de funciones que contiene a las funciones de base y que es estable bajo las operaciones:

R-1 Composición.

R-2 Recurrencia primitiva.

R-3 El operador μ (es decir bajo la operación de minimalización), esto es,

$$f(\vec{x}) = \mu_y(g(\vec{x}, y) = 0), \text{ si } g \text{ es regular.}$$

Observemos que para obtener la clase de funciones recursivas hemos añadido a la definición de las FPR el esquema de minimalización.

En otras palabras, una función $f(\vec{x})$ es recursiva si y sólo si:

1. Es una función de base, o
2. Puede producirse a partir de la clase de funciones de base mediante un número finito de aplicaciones de los esquemas de composición y/o de recurrencia primitiva y/o de minimalización.

Para lograr mayor claridad indicaremos el esquema de minimalización, sobre todo si tenemos en cuenta el dicho esquema u operación es quien introduce un corte de distinción entre las FPR y aquellas funciones que son recursivas pero que no son primitivas recursivas.

Esquema de minimalización: sea $f(\vec{x}, y)$ una función recursiva y regular. Entonces la función,

$$g(\vec{x}) = \mu_y(f(\vec{x}, y) = 0),$$

es una función recursiva.

Ejemplo 2.8. La función $\max(x, y)$ es recursiva.

1. $f(x, y, z) = (x \dotminus z) + (y \dotminus z)$ es una función recursiva.
2. $f(x, y, z)$ es una función recursiva (cuando $z = \max(x, y)$, $f(x, y, z) = 0$).
3. $h(x, y) = \mu_z(f(x, y, z) = 0)$.
4. Pero, $h(x, y) = \max(x, y)$. Luego, $\max(x, y)$ es recursiva.

Observemos que al minimalizar una función, eventualmente se puede obtener una FPR. Así, $\max(x, y) = (x \dotminus y) + y$, la cual es una FPR.

De acuerdo con la distinción que introduce el operador μ , sabemos que existen funciones recursivas que no son funciones primitivas recursivas. Presentamos sin demostración un ejemplo de una de tales funciones.

Ejemplo 2.9. La función de Ackermann, denotada por $A(x, y)$, es una función recursiva que no es una función primitiva recursiva. $A(x, y)$ está definida por:

$$\begin{aligned} A(0, y) &= y + 1, \\ A(x + 1, 0) &= A(x, 1), \\ A(x + 1, y + 1) &= A(x, A(x + 1, y)). \end{aligned}$$

Como ejemplo calculemos $A(2, 3)$.

$$\begin{aligned}
A(2, 3) &= A(1, A(2, 2)) \\
&= A(1, A(1, A(2, 1))) \\
&= A(1, A(1, A(1, A(2, 0)))) \\
&= A(1, A(1, A(1, A(1, 1)))) \\
&= A(1, A(1, A(1, A(1, A(0, A(1, 0)))))) \\
&= A(1, A(1, A(1, A(0, A(0, 1)))))) \\
&= A(1, A(1, A(1, A(0, 2)))) \\
&= A(1, A(1, A(1, 3))) \\
&= A(1, A(1, A(0, A(1, 2)))) \\
&= A(1, A(1, A(0, A(0, A(1, 1)))))) \\
&= A(1, A(1, A(0, A(0, A(0, A(1, 0)))))) \\
&= A(1, A(1, A(0, A(0, A(0, A(0, 1)))))) \\
&= A(1, A(1, A(0, A(0, A(0, 2)))))) \\
&= A(1, A(1, A(0, A(0, 3)))) \\
&= A(1, A(1, A(0, 4))) \\
&= A(1, A(1, 5)) \\
&= A(1, A(0, A(1, 4))) \\
&= A(1, A(0, A(0, A(1, 3)))) \\
&= A(1, A(0, A(0, A(0, A(1, 2)))))) \\
&= A(1, A(0, A(0, A(0, A(0, A(1, 1)))))) \\
&= A(1, A(0, A(0, A(0, A(0, A(0, A(1, 0))))))) \\
&= A(1, A(0, A(0, A(0, A(0, A(0, A(0, 1))))))) \\
&= A(1, A(0, A(0, A(0, A(0, A(0, 2)))))) \\
&= A(1, A(0, A(0, A(0, A(0, 3)))))) \\
&= A(1, A(0, A(0, A(0, 4)))) \\
&= A(1, A(0, A(0, 5))) \\
&= A(1, A(0, 6)) \\
&= A(1, 7) \\
&= A(0, A(1, 6)) \\
&= A(0, A(0, A(1, 5))) \\
&= A(0, A(0, A(0, A(1, 4)))) \\
&= A(0, A(0, A(0, A(0, A(1, 3)))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(1, 2)))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(0, A(1, 1))))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(1, 0))))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(1, 0)))))))) \\
&= A(0, 1)))))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(0, 2)))))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(0, 3))))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(0, 4)))))) \\
&= A(0, A(0, A(0, A(0, 5)))) \\
&= A(0, A(0, A(0, 6))) \\
&= A(0, A(0, 7)) \\
&= A(0, 8) \\
&= 9
\end{aligned}$$

Aunque no es difícil implementar la función de Ackermann, debido a su rápido crecimiento, es imposible calcularla para valores relativamente pequeños de x y de y . Así por ejemplo, $A(4, 2)$ es un número con 19,728 dígitos. Por otra parte, parece ser que el número de llamadas a la función de Ackermann para calcular un valor inicial, crece más rápido que la misma función, tal como lo ilustra la tabla 2.1.

$A(x, y)$	Valor	Número de llamadas
$A(3, 0)$	5	15
$A(3, 1)$	13	106
$A(3, 2)$	29	541
$A(3, 3)$	61	2,432
$A(3, 4)$	125	10,307
$A(3, 5)$	253	42,438
$A(3, 6)$	509	172,233
$A(3, 7)$	1,021	693,964
$A(3, 8)$	2,045	2,785,999

Cuadro 2.1: Número de llamadas a la función de Ackermann.

2.7. Funciones Recursivas Parciales

En el contexto de las funciones recursivas es necesario también que hagamos la distinción entre función total (definición 1.1 (pág. 32)) y función parcial (definición 1.2 (pág. 32)). De hecho, ya hemos trabajado con las funciones parciales en el contexto de las máquinas de Turing, donde permitimos que las reglas de construcción de la máquina nos condujeran a un computar que nunca termina, o en otros términos, la función asociada con la máquina queda indefinida para algunas entradas, esto es, no originando así ningún valor de salida o imagen de la función. El interés de nuestra anterior distinción reside en el hecho de que podemos transladarla al contexto de las funciones recursivas.

Consideremos el caso de una función recursiva $h(x, y)$ que sea una función total, pero que no sea necesariamente una función regular. Si aplicamos el operador de minimalización a la función h , entonces, para ciertos valores de x , la función generada por minimalización no estará definida, y por ende la función h será una función parcial. Es decir, la función $f(x) = \mu_y(h(x, y) = 0)$ no estará definida para toda $x \in \mathbb{N}$. La función $f(x)$ la podemos expresar como sigue:

$$f(x) = \begin{cases} \mu_y(h(x, y) = 0) & \text{si, un tal } y \text{ existe,} \\ \text{indefinida} & \text{si, no existe tal } y. \end{cases}$$

Ejemplo 2.10.

1. $f(x, y) = (x + y) \dot{-} 17$, es una función recursiva total y no regular.
2. $g(x) = \mu_y(f(x, y) = 0)$, es una función recursiva parcial.
3. Observamos que para $x \geq 18$, tendríamos $g(x)$ indefinida. Por ejemplo, para $x = 18$, $g(18) = \text{indefinida}$, ya que, $(18 + 0) \dot{-} 17 \neq 0$, $(18 + 1) \dot{-} 17 \neq 0$, etc.

Entonces diremos que al aplicar el operador de minimalización a una función recursiva total, obtendremos, en general, una función recursiva parcial.

Definición 2.9 (Funciones recursivas parciales). La clase de las funciones recursivas parciales es la menor clase de funciones que contiene a las funciones de base y que es estable bajo las operaciones composición, recurrencia primitiva y el operador μ (es decir bajo la operación de minimalización

$$f(\vec{x}) = \mu_y(g(\vec{x}, y) = 0), \text{ si tal } y \text{ existe, de lo contrario, indefinida}.$$

Cerraremos la presente sección estableciendo la siguiente cadena de inclusiones propias entre las diversas clases de funciones que hemos estudiado. Denotemos por \mathcal{F} la clase de funciones numérico-teóricas, por \mathcal{F}_0 la clase de funciones recursivas parciales, por \mathcal{F}_1 la clase de funciones recursivas y por \mathcal{F}_2 la clase de funciones primitivas recursivas. Entonces se puede establecer la cadena $\mathcal{F} \supset \mathcal{F}_0 \supset \mathcal{F}_1 \supset \mathcal{F}_2$. La relaciones de inclusión en general se deducen del hecho mismo de las definiciones de cada clase. Ahora:

1. $\mathcal{F}_2 \subset \mathcal{F}_1$, puesto que existen funciones recursivas que no son primitivas recursivas. Es decir, existen funciones computables que no son primitivas recursivas. Para ello, basta probar que si $f_0, f_1, \dots, f_k, \dots$ es una enumeración de \mathcal{F}_2 , entonces la función $g(n) = f_n(n)$ no está en la clase \mathcal{F}_2 , luego no es FPR.
2. $\mathcal{F}_1 \subset \mathcal{F}_0$, es decir, toda función recursiva es recursiva parcial, pero existen funciones recursivas parciales que no son recursivas totales. En general, existen funciones recursivas parciales, estrictamente parciales que no pueden llevarse a una función recursiva total. Si se considera, por ejemplo, una enumeración de todas las funciones recursivas parciales de una variable: g_0, g_1, \dots , entonces la función $h(n) = g_n(n)$ (para el caso en que g_n este definida, e indefinida cuando no lo esté) no puede ser llevada a una función recursiva total.
3. $\mathcal{F}_0 \subset \mathcal{F}$. Un argumento de cardinalidad lo prueba inmediatamente, ya que $\overline{\overline{\mathcal{F}_0}} = \aleph_0$ y $\overline{\overline{\mathcal{F}}} > \aleph_0$.

2.8. Funciones Definidas por Minimalización Acotada

Definición 2.10 (Operador de minimalización acotada). Cuando acotamos el alcance del operador de minimalización (μ) obtenemos una nueva operación que llamaremos minimalización acotada. Escribimos, para un entero n y una función $g(\vec{x}, y)$ la expresión

$\mu_{y \leq n}(g(\vec{x}, y) = 0)$, para señalar o simbolizar al menor y tal que sea menor o igual que n , y tal que $g(\vec{x}, y) = 0$.

Sin que $g(\vec{x}, y)$ sea necesariamente una función regular, podemos siempre definir el operador de minimalización acotada, así:

$$h(\vec{x}) = \mu_{y \leq n}(g(\vec{x}, y) = 0) = \begin{cases} \text{al menor } y \leq n, \text{ tal que } g(\vec{x}, y) = 0, \text{ si tal } y \text{ existe,} \\ 0 \quad \text{si no existe tal } y. \end{cases}$$

Si la función g es recursiva, entonces para un n especificado se calcula:

1. $g(\vec{x}, 0), g(\vec{x}, 1), \dots, g(\vec{x}, n)$.
2. Si para algún $k \leq n$, $g(\vec{x}, k) = 0$, entonces, $\mu_{y \leq n}(g(\vec{x}, y) = 0) = k$, donde k fue el primer número tal que $g(\vec{x}, k) = 0$.
3. Si no existe ningún cero en la sucesión $g(\vec{x}, 0), g(\vec{x}, 1), \dots, g(\vec{x}, n)$, entonces, $\mu_{y \leq n}(g(\vec{x}, y) = 0) = 0$.

Teorema 2.6. Si $g(\vec{x}, y)$ es una FPR, entonces la función $f(\vec{x})$ definida por: $f(\vec{x}) = \mu_{y \leq m}(g(\vec{x}, y) = 0)$, es una FPR.

Demostración.

1. Para demostrar este resultado introducimos la función $h(\vec{x}, m)$ definida por el siguiente esquema de recursión:

$$h(\vec{x}, 0) = 0,$$

$$h(\vec{x}, m + 1) = \begin{cases} h(\vec{x}, m) & \text{si } h(\vec{x}, m) \neq 0, \\ m + 1 & \text{si } h(\vec{x}, m) = 0 \wedge g(\vec{x}, m + 1) = 0, \\ 0 & \text{si } h(\vec{x}, m) = 0 \wedge g(\vec{x}, m + 1) \neq 0. \end{cases}$$

2. Si utilizamos la función de identidad $i(x) = I_1^1(x)$, es posible demostrar que la función $h(\vec{x}, m)$ es FPR.
3. Se puede verificar que

$$f(\vec{x}) = h(\vec{x}, m),$$

por lo tanto, $f(\vec{x})$ es FPR.

□

Teorema 2.7. Si $g(\vec{x}, y)$ y $h(\vec{x})$ son FPR, entonces la función $f(\vec{x})$ definida por: $f(\vec{x}) = \mu_{y \leq h(\vec{x})}(g(\vec{x}, y) = 0)$, es una FPR.

Demostración.

1. Se prueba que $j(\vec{x}, n) = \mu_{y \leq n}(g(\vec{x}, y) = 0)$ es FPR.
2. Hacemos $f(\vec{x}) = j(\vec{x}, n)$.

□

Ejemplo 2.11. La función $p(n) = (n + 1)$ -énesimo primo es FPR.

1. Definimos la función $p(n)$ como sigue:

$$\begin{aligned} p(0) &= 2, \\ p(n+1) &= \mu_{y \leq h(n, p(n))}[y > p(n) \wedge p_1(y)]. \end{aligned}$$

2. Donde $h(x, k) = (x + 1)^{k+1}$ es FPR.
3. Donde $p_1(y)$ es el predicado: “ y es un número primo”.
4. La función

$$f(x, y) = \begin{cases} 0 & \text{si } y > p(x) \wedge p_1(y), \\ 1 & \text{si no es el caso,} \end{cases}$$

es FPR.

5. Luego, $p(n+1) = \mu_{y \leq h(n, p(n))}(f(n, y) = 0)$ es FPR.

2.9. Conjuntos Recursivos

La noción de recursividad puede aplicarse igualmente a los conjuntos y, específicamente a conjuntos numéricos. La importancia de la noción de conjunto recursivo está vinculada a los problemas de decidibilidad. Presentaremos tal noción en el contexto de subconjuntos de \mathbb{N}^N .

Definición 2.11 (Conjunto (primitivo) recursivo). Dado $A \subseteq \mathbb{N}^N$, diremos que el conjunto A es (primitivo) recursivo, si y sólo si, existe una función (primitiva) recursiva f tal que:

$$f(x_1, x_2, \dots, x_n) = \begin{cases} 0 & \text{si } (x_1, x_2, \dots, x_n) \in A, \\ 1 & \text{si } (x_1, x_2, \dots, x_n) \notin A. \end{cases}$$

Esto es, la función característica de la relación: “ $\in A$ ”, es una función (primitiva) recursiva. Notemos que la función f decide la pertenencia o no de una cierta n -tupla al conjunto A . En este sentido, la función f es la función característica del conjunto A . Por ello decimos con frecuencia que f es la función decisión para el conjunto A .

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Ejemplo 2.12. El conjunto \mathbb{N}^N es recursivo. La función $f(\vec{x}) = 0$ es recursiva.

Ejemplo 2.13. Todo conjunto finito A es recursivo (primitivo recursivo). Si $A = \{a_1, \dots, a_n\}$, entonces la función característica de A es FPR. Así,

$$C_A(x) = \prod_{i=1}^n sg(|x - a_i|).$$

Ejemplo 2.14. El conjunto de los números impares es recursivo. Para comprobarlo se demuestra que la función:

$$f(x) = \begin{cases} 0 & \text{si } x \text{ es impar,} \\ 1 & \text{si } x \text{ es par;} \end{cases}$$

es recursiva. Así: $f(0) = 1$ y $f(x+1) = \overline{sg}(I_2^2(x, f(x)))$.

Ejemplo 2.15. El conjunto de los números primos es recursivo. Esto es, si designamos con A el conjunto de los números primos, entonces la función

$$f(n) = \begin{cases} 0 & \text{si } n \text{ es primo,} \\ 1 & \text{si } n \text{ es compuesto;} \end{cases}$$

es recursiva, lo cual es verdadero ya que $f(n) = |d(n) - S(S(Z(n)))|$; donde $d(n)$ es el número de divisores de n , luego f FPR.

Decíamos que f es una función de decisión. Efectivamente, la noción de conjunto recursivo es una formulación de la noción intuitiva de conjunto decidable. Intuitivamente hablando, decimos que un conjunto A es decidable, si y sólo si, existe un algoritmo o procedimiento α que nos permita decidir acerca de si un determinado elemento x , del universo de referencia de A , pertenece o no al conjunto A .

Ejemplo 2.16. Para el conjunto de los números naturales, si A es el conjunto de los números pares; ¿es n un elemento de A ?

Efectivamente, sabemos que existe un algoritmo que proporciona una respuesta. Así, dado cualquier $n \in \mathbb{N}$

1. Divida n por 2.
2. Hallar el residuo r .
3. Si $r = 0$, entonces $n \in A$. Si $r \neq 0$, entonces $n \notin A$.

Entonces, el conjunto $A = \{x \mid x \text{ es par}\}$ es recursivo.

Ejemplo 2.17. El conjunto P de los números primos es decidable. Es decir, dado un $n \in \mathbb{N}$, existe un algoritmo α que proporciona una respuesta a la pregunta ¿es n un número primo?

Ciertamente, sabemos que la aritmética elemental contiene procedimientos o algoritmos que nos permiten decidir si un número natural dado es primo o compuesto (mirar ejercicio 2.17). Por ejemplo:

1. Hallar todos los $n_i \in \mathbb{N}$ tales que $n_i < n$ y $n_i \neq 1$.
2. Dividir a n por cada n_i y escribir el residuo respectivo r_i .
3. Si todos los $r_i \neq 0$, entonces $n \in P$, de lo contrario, $n \notin P$.

Por supuesto que existen algoritmos en la aritmética mucho más eficientes que el anterior. El conjunto P de los números primos es recursivo como lo probamos en el ejemplo 2.15 (pág. 75).

Ejemplo 2.18. El conjunto de las fórmulas de la lógica de enunciados es un conjunto decidable. Efectivamente, existe un algoritmo que nos permite decidir si una palabra $\alpha \in \mathbb{L}(\Sigma)$ es o no es una fórmula (ejercicio 2.18).

Como puede observarse, la noción de conjunto recursivo se instituye como una tentativa de formalización de la noción intuitiva de conjunto decidable. Esto es, la definición de conjunto recursivo corresponde con la noción intuitiva de conjunto decidable.

2.10. Conjuntos Recursivamente Enumerables

Intuitivamente pensamos un conjunto enumerable como aquel conjunto cuyos elementos pueden ser escritos o listados como una sucesión infinita (con eventuales repeticiones). Tal noción la podemos formalizar valiéndonos de la noción de aplicación o función sobreyectiva.

Definición 2.12 (Conjunto enumerable). Decimos que un conjunto A es enumerable, si y sólo si, A es vacío o existe una aplicación sobreyectiva de \mathbb{N} en el conjunto A . Esto es,

$$A \text{ es enumerable} \underset{\text{def.}}{\equiv} (A = \emptyset) \vee (\exists f)(f: \mathbb{N} \longrightarrow A).$$

Si A es enumerable, decimos que la función f es una enumeración de A , es decir, f escribe los elementos de A como la sucesión: $f(0), f(1), \dots, f(n), \dots$

Ejemplo 2.19. \mathbb{N} es enumerable. La enumeración es: $i(x) = x$.

Ejemplo 2.20. El conjunto de los números pares es enumerable. La enumeración es la función sobreyectiva $f(n) = 2n$.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Ejemplo 2.21. Todo conjunto finito es enumerable. Sea $A = \{a_1, a_2, \dots, a_n\}$ y, sea $f: \mathbb{N} \rightarrow A$, tal que:

$$f(x) = \begin{cases} a_x & \text{si } x < n, \\ a_n & \text{si } x \geq n; \end{cases}$$

f es una función sobreyectiva de \mathbb{N} en A . Luego A es un conjunto enumerable.

Una pregunta de bastante importancia, la cual nos ocupará en el resto de esta sección, es la siguiente: ¿es posible hallar para toda enumeración f de un conjunto A un algoritmo o procedimiento efectivo que nos permita computar $f(n)$, para todo $n \in \mathbb{N}$?

Definición 2.13 (Conjunto recursivamente enumerable). Sea el conjunto A un subconjunto de \mathbb{N} . Decimos que A es un conjunto recursivamente enumerable (RE) si A es el rango de una función recursiva, o si A es vacío.

La anterior definición implica que si $A \neq \emptyset$ es recursivamente enumerable, entonces existe una enumeración recursiva de A , es decir, existe una función recursiva tal que: $f(0), f(1), \dots, f(n), \dots$ es una lista exhaustiva de los elementos de A (con eventuales repeticiones). Esto es, $f(\mathbb{N}) = A$. Expresado de otra manera:

$$A \text{ es RE} \underset{\text{def.}}{\equiv} (A = \emptyset) \vee ((\exists f)(f: \mathbb{N} \rightarrow A) \wedge f \text{ es recursiva}).$$

Por ello decimos también que la función f genera el conjunto A .

Ejemplo 2.22. Todo conjunto finito es RE Si $A = \{a_1, a_2, \dots, a_n\}$, entonces, sea la función $f: \mathbb{N} \rightarrow A$, tal que:

$$f(x) = \begin{cases} a_x & \text{si } x < n, \\ a_n & \text{si } x \geq n; \end{cases}$$

f es una función sobreyectiva y recursiva de \mathbb{N} en A (ejercicio 2.21).

Unas preguntas que podemos formularnos en este contexto son las siguientes: ¿son las nociones de conjunto recursivo y conjunto recursivamente enumerable diferentes?; ¿existe algún conjunto que sea recursivamente enumerable y no sea recursivo, o viceversa?

Teorema 2.8. *Todo conjunto recursivo (subconjunto de \mathbb{N}) es recursivamente enumerable.*

Demostración.

1. Supongamos que $A \neq \emptyset$ es un conjunto recursivo (si $A = \emptyset$, A es RE por definición).
2. Existe una función recursiva f tal que:

$$f(x) = \begin{cases} 0 & \text{si } x \in A, \\ 1 & \text{si } x \notin A. \end{cases}$$

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

3. Sean $g: \mathbb{N} \rightarrow A$ y $x_0 \in A$ (fijo) tales que:

$$g(x) = \begin{cases} x & \text{si } x \in A, \\ x_0 & \text{si } x \notin A. \end{cases}$$

4. Los predicados $x \in A$ y $x \notin A$, son recursivos (puesto que f es una función recursiva).
5. Luego, la función g es recursiva (puesto que está definida mediante condiciones recursivas).

□

El recíproco del teorema anterior no es válido, es decir, existen conjuntos RE que no son recursivos. Pero si adicionamos una condición al complemento del conjunto RE, éste sí es recursivo.

Teorema 2.9. *Un conjunto A es recursivo, si y sólo si el conjunto A y su complemento son recursivamente enumerables.*

Demostración. (primera parte)

1. Supongamos que A es recursivo.
2. Luego, A es RE por el teorema 2.8 (pág. 77).
3. Denotemos por A^c el complemento de A .
4. $f(x) = 1 \dot{-} C_A(x)$ es recursiva.
5. A^c es recursivo.
6. Luego, A^c es RE por el teorema 2.8 (pág. 77).

□

Demostración. (segunda parte)

1. Supongamos que A y A^c son RE
2. Existen enumeraciones recursivas de A y A^c .
3. Sean éstas respectivamente $f: \mathbb{N} \longleftrightarrow A$ y $g: \mathbb{N} \longleftrightarrow A^c$.
4. Sea $h: \mathbb{N} \rightarrow A$ tal que:

$$h(x) = \begin{cases} f([\frac{x}{2}]) & \text{si } x \text{ es par,} \\ g([\frac{x}{2}]) & \text{si } x \text{ es impar.} \end{cases}$$

5. La función h escribe la sucesión $f(0), g(0), f(1), g(1), \dots$
6. La función h es recursiva.
7. Definamos la función $k(y, z) = |h(y) - z|$, la cual es regular para la variable z .
8. La función $m(y) = \mu_z(k(y, z) = 0)$ es recursiva.
9. La función p , característica de los números pares es recursiva.
10. Definamos la función C_A , característica del conjunto A , como la función $C_A(x) = p(m(x))$.
11. C_A es una función recursiva.
12. Luego, el conjunto A es recursivo.

□

Es conocido que la noción de conjunto recursivamente enumerable es una formalización de la noción intuitiva de conjunto efectivamente enumerable. Intuitivamente hablando, decimos que un conjunto A es efectivamente enumerable, si y sólo si, existe un enumeración f calculable de A . Los valores $f(0), f(1), \dots$ son determinados mediante un algoritmo o procedimiento efectivo. En otras palabras se trata de considerar la noción intuitiva de función calculable mediante un algoritmo. Como se afirma con frecuencia, el sí una cierta descripción matemática particular de la noción de algoritmo corresponde exactamente a la idea intuitiva, no es algo que se pueda demostrar. No obstante, hay buenos argumentos para suponer que las descripciones matemáticas que se han hecho del concepto de algoritmo, son lo bastante generales como para incluir todos los algoritmos intuitivos. Dos ejemplos de ello nos lo proporcionan la clase de las funciones recursivas y la clase de las funciones Turing-computables. Por otro lado, es bastante razonable afirmar que una función parcial es calculable mediante un algoritmo, si existe uno que determine explícitamente el valor de la función cuando ésta está definida.

Una función $f(x_1, x_2, \dots, x_n)$ es efectivamente calculable si existe un procedimiento mecánico o algoritmo que permita determinar el valor $f(a_1, a_2, \dots, a_n)$ para cualquier n -tupla de elementos de \mathbb{N} .

Ejemplo 2.23. El conjunto de los números enteros es efectivamente enumerable, puesto que existe una enumeración efectiva de \mathbb{Z} , $g: \mathbb{N} \rightarrow \mathbb{Z}$ dada por $g(n) = (-1)^n(\lfloor \frac{n+1}{2} \rfloor)$. Esta fórmula es un algoritmo de enumeración de \mathbb{Z} .

Lema 2.1. *Todo entero positivo $n > 1$ puede ser factorizado de forma única como $n = P_1^{n_1} \times P_2^{n_2} \times \dots \times P_m^{n_m}$ donde $P_1 < P_2 < \dots < P_m$, son los m primeros primos y $n_m \neq 0$. Esta forma de expresar el número n se llama la forma normal de Cantor.*

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Demostración. Ejercicio 2.22. □

Teorema 2.10. *El conjunto \mathbb{N}^f de las sucesiones finitas de números naturales es un conjunto efectivamente enumerable.*

Demostración. Para probar el teorema, definamos la función f mediante el siguiente algoritmo.

1. Sea $n \in \mathbb{N}$ un elemento cualquiera.
2. Expresar m como $n + 2$.
3. Expresar m en la forma normal de Cantor (lema 2.1 (pág. 79)), es decir, $m = P_1^{\alpha_1} \times P_2^{\alpha_2} \times \cdots \times P_k^{\alpha_k}$.
4. Escribir la sucesión: $\langle \alpha_1, \alpha_2, \dots, \alpha_k - 1 \rangle$.
5. Sea $f(n) = \langle \alpha_1, \alpha_2, \dots, \alpha_k - 1 \rangle$.
6. Para probar que f es una enumeración efectiva exhaustiva, es decir, sobreyectiva, invierta el proceso de escribir la sucesión $\langle \alpha_1 \alpha_2 \dots \alpha_k - 1 \rangle$. Así, dado a_1, a_2, \dots, a_s , $n = P_1^{a_1} \times P_2^{a_2} \times \cdots \times P_s^{a_s+1} - 2$.

□

2.11. Computabilidad y Recursividad

Esta sección establece la coexistencia entre el conjunto de las de funciones Turing-computables y el de las funciones recursivas. La coexistencia es presentada por dos teoremas; el primero de ellos indica que las funciones recursivas son funciones Turing-computables; el segundo indica que las funciones Turing-computables son funciones recursivas.

Inicialmente presentaremos seis lemas que serán utilizados en la demostración de que una función recursiva es una función Turing-computable.

Lema 2.2. *La función cero, $z(x) = 0$, es una función Turing-computable.*

Demostración.

1. Sea \mathcal{MT} la máquina de Turing presentada en el ejemplo 1.4 (pág. 35) que está definida por:
 $i_1 : q_1 | \square R q_1$
 $i_2 : q_1 \square \square R stop$

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

2. Sea α_1 la descripción instantánea inicial definida por $\alpha_1 \equiv q_1 \vec{x}$. Entonces la computación de la máquina \mathcal{MT} está dada por (en donde $|^n$ representa n palitos):

$$\begin{aligned}\alpha_1 &\equiv q_1 \vec{x} = q_1|^{x+1} \\ &\rightarrow \square q_1|^x \\ &\rightarrow \square \square q_1|^{x-1} \\ &\vdots \\ &\rightarrow \square \square \dots \square q_1| \\ &\rightarrow \square \square \dots \square \square q_1.\end{aligned}$$

3. Por lo tanto, la función $f_{\mathcal{MT}}^{(1)}(x)$ asociada con la máquina de Turing \mathcal{MT} está definida por:

$$\begin{aligned}f_{\mathcal{MT}}^{(1)}(x) &= <Res_{\mathcal{MT}}(\alpha_1)> \\ &= <\square \square \dots \square \square q_1> \\ &= 0 \\ &= z(x).\end{aligned}$$

□

Lema 2.3. *La función sucesor, $s(x) = x + 1$, es una función Turing-computable.*

Demostración.

1. Sea \mathcal{MT} la máquina de Turing, presentada en el ejemplo 1.5 (pág. 36), que está definida por: $i_1 : q_1 \square \square N q_1$.
2. Sea α_1 la descripción instantánea inicial definida por $\alpha_1 \equiv q_1 \vec{x}$. Entonces la computación de la máquina \mathcal{MT} está dada por (en donde $|^n$ representa n palitos):

$$\begin{aligned}\alpha_1 &\equiv q_1 \vec{x} = q_1|^{x+1} \\ &\rightarrow q_1|^{x+1}.\end{aligned}$$

3. Por lo tanto, la función $f_{\mathcal{MT}}^{(1)}(x)$ asociada con la máquina de Turing \mathcal{MT} está definida por:

$$\begin{aligned}
 f_{\mathcal{MT}}^{(1)}(x) &= \langle \text{Res}_{\mathcal{MT}}(\alpha_1) \rangle \\
 &= \langle q_1 |^{x+1} \rangle \\
 &= x + 1 \\
 &= s(x).
 \end{aligned}$$

□

Lema 2.4. Las funciones k -ésima proyección, $I_k^n(x_1, x_2, \dots, x_n) = x_k$, son funciones Turing-computables.

Demostración. Vamos a construir una máquina de Turing genérica para una función $I_k^n(x_1, x_2, \dots, x_n) = x_k$.

1. Sea \mathcal{MT} una máquina de Turing definida por:

$q_1 | \square N q_{2n+1}$; estas instrucciones borran el argumento x_1
 $q_1 \square \square R q_2$
 $q_{2n+1} \square \square R q_1$

$q_2 | \square N q_{2n+2}$; estas instrucciones borran el argumento x_2

$q_2 \square \square R q_3$
 $q_{2n+2} \square \square R q_2$
 \vdots

$q_{k-1} | \square N q_{2n+(k-1)}$; estas instrucciones borran el argumento x_{k-1}

$q_{k-1} \square \square R q_k$
 $q_{2n+(k-1)} \square \square R q_{k-1}$
 $q_k | \square N q_k$, esta instrucción elimina un palito del argumento x_k
 $q_k \square \square R q_{k+1}$, esta instrucción envía a borrar los argumentos $x_{j>k}$
 $q_{k+1} | \square N q_{2n+(k+1)}$; estas instrucciones borran el argumento x_{k+1}
 $q_{k+1} \square \square R q_{k+2}$
 $q_{2n+(k+1)} \square \square R q_{k+1}$
 \vdots
 $q_n | \square N q_{2n+n}$; estas instrucciones borran el argumento x_n
 $q_n \square \square R stop$
 $q_{2n+n} \square \square R q_n$

2. De la construcción anterior se concluye que la función $f_{\mathcal{MT}}^{(n)}(x_1, x_2, \dots, x_n)$ asociada con la máquina de Turing \mathcal{MT} es igual a $I_k^n(x_1, x_2, \dots, x_n) = x_k$.

□

Lema 2.5. *El esquema de composición o sustitución preserva la propiedad de Turing-computabilidad. Es decir, si $g: \mathbb{N}^m \rightarrow \mathbb{N}$ y cada $h_i: \mathbb{N}^n \rightarrow \mathbb{N}$ son funciones Turing-computables (totales, parciales), entonces*

$$f(x_1, x_2, \dots, x_n) = g(h_1(x_1, x_2, \dots, x_n), \dots, h_m(x_1, x_2, \dots, x_n)),$$

es una función Turing-computable (total, parcial).

Demostración. La idea es construir una máquina \mathcal{MT}_f que envíe a computar cada una de las funciones $h_i(x_1, x_2, \dots, x_n)$, $1 \leq i \leq m$ y con los valores obtenidos, envíe a computar la función $g(h_1(x_1, x_2, \dots, x_n), \dots, h_m(x_1, x_2, \dots, x_n))$.

1. Supongamos que la máquina \mathcal{MT}_g computa la función $g: \mathbb{N}^m \rightarrow \mathbb{N}$ y que las máquinas \mathcal{MT}_{h_i} computan la funciones $h_i: \mathbb{N}^n \rightarrow \mathbb{N}$.
2. Inicialmente la máquina \mathcal{MT}_f ejecuta la máquina \mathcal{MT}_{h_1} con la descripción inicial:
 $\alpha_1 \equiv q_1 \xrightarrow{x_1, \dots, x_n};$
y obtenemos a la salida $\overrightarrow{h_1(x_1, x_2, \dots, x_n)}$.
3. Después la máquina \mathcal{MT}_f ejecuta la máquina \mathcal{MT}_{h_2} con la descripción inicial:
 $\alpha_1 \equiv q_1 \xrightarrow{x_1, \dots, x_n};$
y adicionando algunas instrucciones, obtenemos la salida:
 $\overrightarrow{h_1(x_1, x_2, \dots, x_n)} \square \overrightarrow{h_2(x_1, x_2, \dots, x_n)}.$
4. La máquina \mathcal{MT}_f continúa con la ejecución de las máquinas $\mathcal{MT}_{h_3}, \dots, \mathcal{MT}_{h_m}$ con la descripción inicial:
 $\alpha_1 \equiv q_1 \xrightarrow{x_1, \dots, x_n};$
y con la adición de las instrucciones necesarias, obtenemos finalmente la salida:
 $\overrightarrow{h_1(x_1, x_2, \dots, x_n)} \square \overrightarrow{h_2(x_1, x_2, \dots, x_n)} \square \dots \square \overrightarrow{h_m(x_1, x_2, \dots, x_n)}.$
5. Entonces la máquina \mathcal{MT}_f ejecuta la máquina \mathcal{MT}_g con la descripción inicial:
 $\alpha_1 \equiv q_1 \xrightarrow{h_1(x_1, x_2, \dots, x_n)} \square \xrightarrow{h_2(x_1, x_2, \dots, x_n)} \square \dots \square \xrightarrow{h_m(x_1, x_2, \dots, x_n)};$
y obtenemos a la salida el valor de la función $f(x_1, x_2, \dots, x_n)$.

□

Lema 2.6. *El esquema de recurrencia primitiva preserva la propiedad de Turing-computabilidad. Es decir, si $g: \mathbb{N}^n \rightarrow \mathbb{N}$ y $h: \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ son funciones Turing-computables (totales, parciales), entonces*

$$f(x_1, x_2, \dots, x_n, 0) = g(x_1, x_2, \dots, x_n),$$

$$f(x_1, x_2, \dots, x_n, k+1) = h(x_1, x_2, \dots, x_n, k, f(x_1, x_2, \dots, x_n, k)),$$

es una función Turing-computable (total, parcial).

Demostración. La idea es construir una máquina \mathcal{MT}_f que compute el valor de $f(x_1, x_2, \dots, x_n, 0)$ a partir de $g(x_1, x_2, \dots, x_n)$, y que compute cada uno de los valores

$$f(x_1, x_2, \dots, x_n, 1), \dots, f(x_1, x_2, \dots, x_n, k+1)$$

a partir de los valores de

$$h(x_1, x_2, \dots, x_n, 0, f(x_1, x_2, \dots, x_n, 0)), \dots, h(x_1, x_2, \dots, x_n, k, f(x_1, x_2, \dots, x_n, k))$$

respectivamente. Es decir, inicialmente se computa el valor de $f(x_1, x_2, \dots, x_n, 0)$, después el valor de $f(x_1, x_2, \dots, x_n, 1)$ y así sucesivamente hasta computar el valor de $f(x_1, x_2, \dots, x_n, k+1)$.

1. Supongamos que la máquina \mathcal{MT}_g computa la función $g(x_1, x_2, \dots, x_n)$ y la máquina \mathcal{MT}_h computa la función $h(x_1, x_2, \dots, x_n, k, z)$.

2. Inicialmente la máquina \mathcal{MT}_f calcula el valor de $f(x_1, x_2, \dots, x_n, 0)$, es decir, ejecuta la máquina \mathcal{MT}_g con la descripción inicial:

$$\alpha_1 \equiv q_1 \overrightarrow{x_1, \dots, x_n} \square \overrightarrow{0},$$

$$\text{de donde obtenemos que } \overrightarrow{f(x_1, x_2, \dots, x_n, 0)} = \overrightarrow{g(x_1, \dots, x_n)}.$$

3. Con el valor de $f(x_1, x_2, \dots, x_n, 0)$ la máquina \mathcal{MT}_f calcula el valor de $f(x_1, x_2, \dots, x_n, 1)$, es decir, ejecuta la máquina \mathcal{MT}_h con la descripción inicial:

$$\alpha_1 \equiv q_1^h \overrightarrow{x_1, \dots, x_n} \square \overrightarrow{0} \square \overrightarrow{f(x_1, x_2, \dots, x_n, 0)}$$

de donde obtenemos que:

$$\overrightarrow{f(x_1, x_2, \dots, x_n, 1)} = \overrightarrow{h(x_1, x_2, \dots, x_n, 0, f(x_1, x_2, \dots, x_n, 0))};$$

4. La máquina \mathcal{MT}_f continúa con el proceso anterior hasta calcular el valor de $f(x_1, x_2, \dots, x_n, k+1)$ a partir de la máquina \mathcal{MT}_h con la descripción inicial dada por:

$$\alpha_1 \equiv q_1 \overrightarrow{x_1, \dots, x_n} \square \overrightarrow{k} \square \overrightarrow{f(x_1, x_2, \dots, x_n, k)};$$

de donde finalmente obtenemos que:

$$\overrightarrow{f(x_1, x_2, \dots, x_n, k+1)} = \overrightarrow{h(x_1, x_2, \dots, x_n, k, f(x_1, x_2, \dots, x_n, k))};$$

□

Lema 2.7. *El esquema de minimalización preserva la propiedad de Turing-computabilidad. Es decir, si $g(\vec{x}, y)$ es una función Turing-computable (total y regular, total), entonces*

$$f(\vec{x}) = \mu_y(g(\vec{x}, y) = 0),$$

es una función Turing-computable (total, parcial).

Demostración. La idea es construir una máquina \mathcal{MT}_f que compute incrementalmente el valor de y hasta encontrar (si es el caso) que $g(\vec{x}, y) = 0$, entonces la máquina \mathcal{MT}_f retorna este valor de y .

1. Supongamos que la máquina \mathcal{MT}_g computa la función $g(\vec{x}, y)$.
2. Inicialmente la máquina \mathcal{MT}_f ejecuta la máquina \mathcal{MT}_g con la descripción inicial:

$$\alpha_1 \equiv q_1 \overrightarrow{x_1, \dots, x_n} \square \overrightarrow{0},$$
y obtenemos a la salida $\overrightarrow{g(x_1, \dots, x_n, 0)}$.
3. Después la máquina \mathcal{MT}_f ejecuta la máquina \mathcal{MT}_g con la descripción inicial:

$$\alpha_1 \equiv q_1 \overrightarrow{x_1, \dots, x_n} \square \overrightarrow{1},$$
y obtenemos a la salida $\overrightarrow{g(x_1, \dots, x_n, 1)}$.
4. La máquina \mathcal{MT}_f continua ejecutando sucesivamente la máquina \mathcal{MT}_g hasta encontrar la salida $\overrightarrow{g(x_1, \dots, x_n, y)} = \overrightarrow{0}$ y retorna el valor y . Si tal salida no existe, entonces la máquina \mathcal{MT}_f no se detiene.

□

Ahora podemos presentar el primer teorema relacionado con la coexistencia entre el conjunto de las funciones recursivas y el conjunto de las funciones Turing-computables.

Teorema 2.11. *Las funciones recursivas (totales, parciales) son funciones Turing-computables (totales, parciales).*

Demostración. La demostración está sustentada en los elementos involucrados en la definición de una función recursiva, es decir, las funciones de base y los esquemas de composición, recurrencia primitiva y minimalización.

Los lemas 2.2 (pág. 80), 2.3 (pág. 81) y 2.4 (pág. 82) muestran que la función cero $z(x) = 0$, la función sucesor $s(x) = x + 1$ y las funciones k-ésima proyección $I_k^n(x_1, x_2, \dots, x_n) = x_k$ son funciones Turing-computables. Por otra parte, los lemas 2.5 (pág. 83), 2.6 (pág. 83) y 2.7 (pág. 84) muestran que la propiedad de Turing-computabilidad es preservada en la aplicación de los esquemas de composición, recurrencia primitiva y minimalización. □

A continuación presentamos el segundo teorema relacionado con la coexistencia entre el conjunto de las funciones recursivas y el conjunto de las funciones Turing-computables.

Teorema 2.12. *Las funciones Turing-computables (totales, parciales) son funciones recursivas (totales, parciales).*

Demostración.

1. La demostración se realizará para una función Turing-computable (total, parcial) $f(x_1, x_2)$. La restricción a dos argumentos no es importante, dado que es posible hacer las generalizaciones necesarias para funciones de n argumentos.
2. Inicialmente la cinta contiene la dupla (x_1, x_2) codificada tal como lo indica la definición 1.10 (pág. 34), es decir,

$$\begin{aligned} (\overrightarrow{x_1}, \overrightarrow{x_2}) &= \overrightarrow{x_1} \square \overrightarrow{x_2} \\ &= \underbrace{||| \dots |}_{x_1+1 \text{ veces}} \square \underbrace{||| \dots |}_{x_2+1 \text{ veces}}. \end{aligned}$$

Al final la cinta contiene el valor de $f(x_1, x_2)$ codificado por:

$$\overrightarrow{f(x_1, x_2)} = \underbrace{||| \dots |}_{f(x_1, x_2)+1 \text{ veces}}.$$

Es necesario observar que hemos modificado la forma de salida de la máquina. Es decir, en lugar de que la máquina al final contenga $f(x_1, x_2)$ palitos sobre la cinta, la máquina contendrá $f(x_1, x_2) + 1$ palitos sobre la cinta. Esto debido al uso de la función *lo* descrita en el numeral 7 de esta demostración.

3. Los contenidos de la cinta y de la celda visitada en cualquier momento se representarán en notación binaria de la siguiente forma: Si se considera que $\square \equiv 0$, los contenidos de las celdas a la izquierda de la celda visitada pueden ser pensados como un número binario; éste será denotado por *inum*. El contenido de la celda visitada y las celdas a su derecha, pueden ser pensados como un número binario escrito inversamente; este número será denotado por *dnum*.
4. Veamos qué sucede en los números *inum* y *dnum* de acuerdo con los posibles movimientos de la máquina (de nuevo se va a considerar $\square \equiv 0$):
 - a) La máquina no se mueve:
 - 1) La máquina cambia el símbolo 1 por 0:
El número *dnum* se decrementa en uno y el número *inum* no cambia.
 - 2) La máquina cambia el símbolo 0 por 1:
El número *dnum* se incrementa en uno y el número *inum* no cambia.
 - b) La máquina se mueve a la izquierda:
 - 1) Si *inum* es impar: $inum = \frac{inum-1}{2}$; $dnum = 2dnum + 1$.
 - 2) Si *inum* es par: $inum = \frac{inum}{2}$; $dnum = 2dnum$.
 - c) La máquina se mueve a la derecha:
 - 1) Si *dnum* es impar: $inum = 2inum + 1$; $dnum = \frac{dnum-1}{2}$.

- 2) Si $dnum$ es par: $inum = 2inum$; $dnum = \frac{dnum}{2}$.
5. Veamos los valores de $inum$ y $dnum$ al comienzo y al final de la computación. Al comienzo la máquina está en el primer 1 correspondiente a $\vec{x_1}$, luego

$$\begin{aligned}inum &= 0, \\dnum &= (2^{x_2+1}-1)2^{x_1+2} + (2^{x_1+1}-1).\end{aligned}$$

Al final la máquina está en el primer | correspondiente a $\overrightarrow{f(x_1, x_2)}$, luego

$$\begin{aligned}inum &= 0 \\dnum &= 2^{f(x_1, x_2)+1} - 1.\end{aligned}$$

6. La función $k(x_1, x_2) = (2^{x_2+1}-1)2^{x_1+2} + (2^{x_1+1}-1)$ es una función primitiva recursiva.
7. La función

$$d(x, y) = \begin{cases} 0 & \text{si } 2^y \leq x \\ 1 & \text{si } 2^y > x, \end{cases}$$

es primitiva recursiva.

La función

$$Mx_w(f(\vec{x}_n, y)) = \begin{cases} \text{mayor } y \text{ entre } 0 \text{ y } w \text{ inclusive, tal que } f(\vec{x}_n, y) = 0, \\ 0 \text{ si tal } y \text{ no existe,} \end{cases}$$

es primitiva recursiva si la función $f(\vec{x}_n, y)$ lo es.

Entonces la función $lo(x) = Mx_y(d(x, y))$ es primitiva recursiva.

Como $lo(2^{z+1}-1) = z$, entonces cuando la máquina se detiene en el cálculo de $f(x_1, x_2)$ tenemos que:

$$\begin{aligned}f(x_1, x_2) &= lo(dnum) \\&= lo(2^{f(x_1, x_2)+1} - 1).\end{aligned}$$

8. Codificamos la dinámica de la máquina mediante dos funciones α y q , de la siguiente forma:

- a) Cada estado q_i es codificado por el número i .
- b) El símbolo \square es codificado por el número 0 y el símbolo 1 por el número 1.
- c) Si la máquina de Turing ejecuta una instrucción de la forma $q_i \square \square N q_k$ entonces $\alpha(i, 0) = 0$ y $q(i, 0) = k$.

- d) Si la máquina de Turing ejecuta una instrucción de la forma $q_i \square 1 N q_k$ entonces $a(i, 0) = 1$ y $q(i, 0) = k$.
- e) Si la máquina de Turing ejecuta una instrucción de la forma $q_i \square \square L q_k$ entonces $a(i, 0) = 2$ y $q(i, 0) = k$.
- f) Si la máquina de Turing ejecuta una instrucción de la forma $q_i \square \square R q_k$ entonces $a(i, 0) = 3$ y $q(i, 0) = k$.
- g) Si la máquina de Turing ejecuta una instrucción de la forma $q_i 1 \square N q_k$ entonces $a(i, 1) = 0$ y $q(i, 1) = k$.
- h) Si la máquina de Turing ejecuta una instrucción de la forma $q_i 1 1 N q_k$ entonces $a(i, 1) = 1$ y $q(i, 1) = k$.
- i) Si la máquina de Turing ejecuta una instrucción de la forma $q_i 1 1 L q_k$ entonces $a(i, 1) = 2$ y $q(i, 1) = k$.
- j) Si la máquina de Turing ejecuta una instrucción de la forma $q_i 1 1 R q_k$ entonces $a(i, 1) = 3$ y $q(i, 1) = k$.
- k) En otro caso, $a(i, x) = q(i, x) = 0$.

De acuerdo con lo anterior, las funciones $a(x, y)$ y $q(x, y)$ son funciones primitivas recursivas definidas por casos.

9. Definimos ahora algunas funciones primitivas recursivas:

- a) $tpl(x, y, z) = 2^x 3^y 5^z$
- b) $lft(w) =$ al mayor $x \leq w$ tal que 2^x divide a w .
- c) $crt(w) =$ al mayor $x \leq w$ tal que 3^x divide a w .
- d) $rft(w) =$ al mayor $x \leq w$ tal que 5^x divide a w .
- e) $e(x) = \begin{cases} 0 & \text{si } x \text{ es par,} \\ 1 & \text{si } x \text{ es impar.} \end{cases}$

La función tpl codifica una tripleta de números (x, y, z) en un único número $tpl(x, y, z)$, y las funciones lft , crt y rft obtienen de este número codificado los correspondientes valores de x , y y z .

Si la celda visitada contiene un 0, entonces $e(dnum) = 0$ y si contiene un 1 entonces $e(dnum) = 1$.

10. Vamos a construir una función primitiva recursiva $g(x_1, x_2, t)$ que indique el comportamiento de la máquina en el paso de ejecución t (que no es el paso en el cual la máquina se detiene). Esta función está definida por:

$$g(x_1, x_2, t) = \text{tpl}(\text{inum en } t, \\ \text{estado de la máquina en } t, \\ \text{dnum en } t).$$

La definición de $g(x_1, x_2, t)$ se hará usando el esquema de recurrencia primitiva. Para $g(x_1, x_2, 0)$, como la máquina comienza a ejecutar desde el estado q_1 y está parada en el primer 1 de \vec{x}_1 tenemos que:

$$g(x_1, x_2, 0) = \text{tpl}(0, 1, k(x_1, x_2)).$$

Para $g(x_1, x_2, t + 1)$ es necesario considerar varios casos de acuerdo con las posibles situaciones e instrucciones que pueda realizar la máquina. Las convenciones utilizadas son las siguientes:

$$\begin{aligned} l &= \text{lft}(g(x_1, x_2, t)), \\ c &= \text{crt}(g(x_1, x_2, t)), \\ r &= \text{rft}(g(x_1, x_2, t)), \\ q &= q(c, e(r)); \end{aligned}$$

entonces la función $g(x_1, x_2, t + 1)$ está definida por:

$$g(x_1, x_2, t + 1) = \begin{cases} \text{tpl}(l, q, r) & \text{si } a(c, e(r)) = 0 \wedge e(r) = 0, \\ \text{tpl}(l, q, r - 1) & \text{si } a(c, e(r)) = 0 \wedge e(r) = 1, \\ \text{tpl}(l, q, r + 1) & \text{si } a(c, e(r)) = 1 \wedge e(r) = 0, \\ \text{tpl}(l, q, r) & \text{si } a(c, e(r)) = 1 \wedge e(r) = 1, \\ \text{tpl}(l/2, q, 2r) & \text{si } a(c, e(r)) = 2 \wedge e(l) = 0, \\ \text{tpl}((l - 1)/2, q, 2r + 1) & \text{si } a(c, e(r)) = 2 \wedge e(l) = 1, \\ \text{tpl}(2l, q, r/2) & \text{si } a(c, e(r)) = 3 \wedge e(l) = 0, \\ \text{tpl}(2l + 1, q, (r - 1)/2) & \text{si } a(c, e(r)) = 3 \wedge e(l) = 1, \\ 0 & \text{si de otro modo.} \end{cases}$$

Como la función $g(x_1, x_2, t)$ está definida a partir del esquema de recurrencia primitiva sobre funciones recursivas primitivas, entonces g es una función primitiva recursiva.

11. Para definir $f(x_1, x_2)$ como una función recursiva es necesario realizar algunos construcciones adicionales.
 - a) Si la máquina se detiene en el estado t , entonces $\text{ctr}(g(x_1, x_2, y)) \neq 0$ para todo $y \leq t$ y $\text{ctr}(g(x_1, x_2, y)) = 0$ para todo $y > t$.

- b) Entonces la máquina para de computar a $f(x_1, x_2)$ si y sólo si t es el menor y tal que $ctr(g(x_1, x_2, y + 1)) = 0$.
- c) La función $h(x_1, x_2, y) = ctr(g(x_1, x_2, y + 1))$ es primitiva recursiva.
- d) Todas las funciones definidas hasta el momento son primitivas recursivas. La siguiente función ofrece el carácter de recursividad a las funciones Turing-computables.
- e) La función $p(x_1, x_2) = \mu_y(h(x_1, x_2, y) = 0)$ es recursiva.
- f) Sí la función $f(x_1, x_2)$ es Turing-computable total, la función $p(x_1, x_2)$ es una función recursiva total y si, la función $f(x_1, x_2)$ es Turing-computable parcial, la función $p(x_1, x_2)$ es una función recursiva parcial.
12. Finalmente definimos $f(x_1, x_2) = lo(rft(g(x_1, x_2, p(x_1, x_2))))$, como f está formada por composición de funciones recursivas (totales, parciales), entonces f es una función recursiva (total, parcial).

□

Ejemplo 2.24. Como ejemplo del teorema 2.12 (pág. 85) demostraremos que la función Turing-computable $f(x, y) = 0$ es una función primitiva recursiva.

1. Definición de la máquina de Turing que calcula $f(x, y) = 0$.

Vamos a definir una máquina de Turing \mathcal{MT} tal que la función asociada con la máquina de Turing sea la función $f(x, y) = 0$, es decir, $f_{\mathcal{MT}}^{(2)}(x, y) = f(x, y) = 0$. Como caso particular vamos a realizar el análisis para la instancia de la función $f(2, 1) = 0$. Se debe tener en cuenta que la máquina de Turing \mathcal{MT} que calcula la función $f(2, 1) = 0$ debe estar implementada de manera que concuerde con la dinámica expuesta en el paso (8) de la demostración, es decir, debe poseer instrucciones simples. Además la máquina debe comenzar en el estado q_1 para que concuerde con la función primitiva recursiva g definida en el paso (10) de la demostración y debe finalizar sólo con un palito en la cinta, dado que el número de palitos al finalizar debe ser $f(2, 1) + 1$. De acuerdo con lo anterior, la máquina de Turing \mathcal{MT} que calcula la función $f(x_1, x_2) = 0$ está definida por: $I = \{i_1, i_2, i_3, i_4, i_5, i_6\}$ donde:

- $i_1: q_1 \ 1 \square N \ q_2$
- $i_2: q_2 \square \square R \ q_1$
- $i_3: q_1 \square \square R \ q_3$
- $i_4: q_3 \ 1 \square N \ q_4$
- $i_5: q_4 \square \square R \ q_3$
- $i_6: q_3 \square 1 \ N \ q_5$

2. Seguimiento del *inum* y del *dmun*

La descripción instantánea inicial es $q_1 \vec{x_1} \square \vec{x_2}$ que para el caso de $f(2, 1)$ es

$q_1 \ 1 \ 1 \ 1 \square \ 1 \ 1$. Entonces $inum = 0$ y $dnum = 110111_2 = 55$, donde 110111_2 significa que el número está en binario. La máquina comienza entonces la ejecución de las instrucciones. La simulación de la ejecución, la descripción instantánea que se obtiene, el valor de $inum$ y de $dnum$ se presentan en la tabla 2.2. De acuerdo con la tabla 2.2 los valores finales de $inum$ y $dnum$ son $inum = 0$, $dnum = 2^{f(2,1)+1} - 1 = 1$. Luego $f(2, 1) = lo(dnum_{final}) = lo(1) = 0$.

Instrucción	Descripción instantánea	$inum$	$dnum$
i_1	$q_2 \square \ 1 \ 1 \ \square \ 1 \ 1$	0	$110110_2 = 54$
i_2	$\square \ q_1 \ 1 \ 1 \ \square \ 1 \ 1$	0	$11011_2 = 27$
i_1	$\square \ q_2 \ \square \ 1 \ \square \ 1 \ 1$	0	$11010_2 = 26$
i_2	$\square \ \square \ q_1 \ 1 \ \square \ 1 \ 1$	0	$1101_2 = 13$
i_1	$\square \ \square \ q_2 \ \square \ \square \ 1 \ 1$	0	$1100_2 = 12$
i_2	$\square \ \square \ q_1 \ \square \ 1 \ 1$	0	$110_2 = 6$
i_3	$\square \ \square \ \square \ \square \ q_3 \ 1 \ 1$	0	$11_2 = 3$
i_4	$\square \ \square \ \square \ \square \ q_4 \ \square \ 1$	0	$10_2 = 2$
i_5	$\square \ \square \ \square \ \square \ \square \ q_3 \ 1$	0	$1_2 = 1$
i_4	$\square \ \square \ \square \ \square \ \square \ q_4 \ \square$	0	$0_2 = 0$
i_5	$\square \ \square \ \square \ \square \ \square \ \square \ q_3 \ \square$	0	$0_2 = 0$
i_6	$\square \ \square \ \square \ \square \ \square \ \square \ q_5 \ 1$	0	$1_2 = 1$

Cuadro 2.2: Simulación \mathcal{MT} que calcula la función $f(2, 1) = 0$.

3. Dinámica de la máquina de Turing \mathcal{MT}

Con base en el paso (8) de la demostración, codificamos la dinámica de la máquina \mathcal{MT} de la siguiente manera:

- $i_1: q_1 \ 1 \ \square \ N \ q_2 \quad \alpha(1, 1) = 0 \text{ y } q(1, 1) = 2 \quad \text{por (g)}$
- $i_2: q_2 \ \square \ \square \ R \ q_1 \quad \alpha(2, 0) = 3 \text{ y } q(2, 0) = 1 \quad \text{por (f)}$
- $i_3: q_1 \ \square \ \square \ R \ q_3 \quad \alpha(1, 0) = 3 \text{ y } q(1, 0) = 3 \quad \text{por (f)}$
- $i_4: q_3 \ 1 \ \square \ N \ q_4 \quad \alpha(3, 1) = 0 \text{ y } q(3, 1) = 4 \quad \text{por (g)}$
- $i_5: q_4 \ \square \ \square \ R \ q_3 \quad \alpha(4, 0) = 3 \text{ y } q(4, 0) = 3 \quad \text{por (f)}$
- $i_6: q_3 \ \square \ 1 \ N \ q_5 \quad \alpha(3, 0) = 1 \text{ y } q(3, 0) = 5 \quad \text{por (d)}$

4. Construcción de la función primitiva recursiva $f(2, 1) = 0$

La construcción de la dinámica de la máquina \mathcal{MT} del paso anterior se realizó para implementar la función primitiva recursiva:

$$g(2, 1, t) = tpl(inun \text{ en } t, \text{estado de la máquina en } t, dnum \text{ en } t),$$

donde t es una variable que se incrementa cada vez que se ejecuta una instrucción de la máquina \mathcal{MT} y $tpl(x, y, z)$ es la función primitiva recursiva definida por $tpl(x, y, z) =$

$2^x3^y5^z$. La función primitiva recursiva g está definida por casos en el paso (10) de la demostración. De acuerdo con el paso (11) de la demostración, se define la función $p(2, 1)$ de la siguiente manera:

$$\begin{aligned} p(2, 1) &= \mu_y(h(2, 1, y) = 0) \\ &= \mu_y(ctr(g(2, 1, y + 1)) = 0), \end{aligned}$$

donde la función ctr devuelve el número y de $tpl(x, y, z)$. La función $p(2, 1)$ calcula el menor $y+1$ tal que $ctr(g(2, 1, y+1)) = 0$. Finalmente definimos la función $f(2, 1)$ como $f(2, 1) = lo(rft(g(2, 1, p(2, 1))))$, donde la función lo y la función rft son funciones primitivas recursivas definidas en los pasos (7) y (9) de la demostración.

5. Cálculo de $p(2, 1)$

Para calcular el valor de $p(2, 1)$ y de allí el valor de $f(2, 1)$ procedemos de la siguiente manera: suponemos $p(2, 1)$ conocido y operamos de una manera recursiva, ya que la función g depende del estado de la configuración t anterior, hasta un $t = 0$. Sea $p(2, 1) = n$, entonces para calcular $f(2, 1) = lo(rft(g(2, 1, n)))$ necesitamos $g(2, 1, n)$; para calcular $g(2, 1, n)$ necesitamos $g(2, 1, n - 1)$; para calcular $g(2, 1, n - 1)$ necesitamos $g(2, 1, n - 2)$ y así sucesivamente hasta calcular $g(2, 1, 0) = tpl(0, 1, k(2, 1)) = tpl(0, 1, 55)$ definido en el paso (10) de la demostración. Procedemos entonces a realizar estos cálculos.

a) Para $t = 0$:

$$g(2, 1, 0) = tpl(0, 1, 55)$$

b) Para $t = 1$:

$$\begin{aligned} c &= ctr(g(2, 1, 0)) = ctr(tpl(0, 1, 55)) = 1 \\ r &= rft(g(2, 1, 0)) = rft(tpl(0, 1, 55)) = 55 \\ l &= ltf(g(2, 1, 0)) = ltf(tpl(0, 1, 55)) = 0 \\ e(r) &= 1 \quad (\text{Paridad}) \\ \alpha(c, e(r)) &= \alpha(1, 1) = 0 \\ q(c, e(r)) &= q(1, 1) = 2 \end{aligned}$$

$$\text{Entonces } g(2, 1, 1) = tpl(0, 2, 55 \div 1) = tpl(0, 2, 54)$$

c) Los cálculos necesarios para $t = 2$ hasta $t = 13$ se presentan en la tabla 2.3.

d) Por primera vez en el seguimiento $ctr(g(2, 1, 13)) = ctr(tpl(0, 0, 0)) = 0$. Esto implica que $n = 12$, es decir, $p(2, 1) = 12$.

6. Cálculo de $f(2, 1)$

<i>t</i>	<i>c</i>	<i>r</i>	<i>l</i>	<i>e(r)</i>	$\alpha(c, e(r))$	<i>q(c, e(r))</i>	<i>g(2, 1, t)</i>
2	2	54	0	0	3	1	<i>tpl(0, 1, 27)</i>
3	1	27	0	1	0	2	<i>tpl(0, 2, 26)</i>
4	2	26	0	0	3	1	<i>tpl(0, 1, 13)</i>
5	1	13	0	1	0	2	<i>tpl(0, 2, 12)</i>
6	2	12	0	0	3	1	<i>tpl(0, 1, 6)</i>
7	1	6	0	0	3	3	<i>tpl(0, 3, 3)</i>
8	3	3	0	1	0	4	<i>tpl(0, 4, 2)</i>
9	4	2	0	0	3	3	<i>tpl(0, 3, 1)</i>
10	3	1	0	1	0	4	<i>tpl(0, 4, 0)</i>
11	4	0	0	0	3	3	<i>tpl(0, 3, 0)</i>
12	3	0	0	0	1	5	<i>tpl(0, 5, 1)</i>
13	5	1	0	1	0	0	<i>tpl(0, 0, 0)</i>

Cuadro 2.3: Cálculo de $p(2, 1)$.

Dado que $p(2, 1) = 12$, entonces

$$\begin{aligned} f(2, 1) &= lo(rft(g(2, 1, 12))) \\ &= lo(rft(tpl(0, 5, 1))) \\ &= lo(1) \\ &= 0. \end{aligned}$$

Sugerimos comparar las tablas 2.2 y 2.3 para observar que la computación realizada es la misma.

7. Hemos demostrado entonces que la función Turing-computable $f(2, 1) = 0$ se puede definir como una función primitiva recursiva.

2.12. Tesis de Church-Turing

La tesis de Church-Turing, según la cual una función es efectivamente calculable si y sólo si es una función recursiva, ha conocido bastantes discusiones, tanto por filósofos como por matemáticos.

El hecho crucial de que las diversas caracterizaciones de las funciones calculables por un algoritmo hayan conducido a la clase de las funciones recursivas (Turing-computables) y el hecho de que aún no se haya producido una función efectivamente calculable que no sea recursiva, constituyen argumentos de peso o evidencia en favor de la conjetura conocida como tesis de Church-Turing (extendida), tesis que afirma que la clase de las funciones parciales calculables es idéntica a la clase de las funciones recursivas parciales.

De hecho, aceptar como válido este enunciado (puesto que no puede ser probado) implica establecer de una vez por todas que la noción intuitiva de algoritmo corresponde a las descripciones matemáticas que han sido dadas (máquinas de Turing, recursividad, etc.). En otras palabras, aceptar la tesis equivale a afirmar categóricamente que: toda función parcial calculable mediante un algoritmo es una función recursiva parcial.

La potencia de la tesis extendida de Church-Turing radica en el hecho de que se puedan usar estrategias o técnicas matemáticas para probar la existencia o no de algoritmos para un cierta clase particular de problemas. Y, recíprocamente, sabido de la existencia de un procedimiento mecánico o algoritmo particular, poder afirmar que el conjunto o función correspondiente es recursivo.

Ejemplo 2.25. Como un conjunto es recursivamente enumerable si existe una función recursiva f tal $f(0), f(1), \dots$ es una lista exhaustiva del conjunto, entonces la tesis de Church-Turing conduce a la idea de que recursivamente enumerable es equivalente a efectivamente enumerable.

Ejemplo 2.26.

1. Es sabido que: el conjunto A de los números de Gödel de las fórmulas de la teoría de números \mathcal{N} que son verdaderas, no es recursivo.
2. Aplicando la tesis de Church-Turing podemos entonces afirmar: no existe ningún algoritmo que sirva para responder la pregunta: ¿es la fórmula α verdadera en \mathcal{N} ?
3. Luego, no existe una función de decisión f para el conjunto A . Así, A no es un conjunto decidable (en sentido intuitivo)

Un sistema formal es recursivamente indecidible si el conjunto de los números de Gödel de los teoremas del sistema no es recursivo. Usando la tesis de Church-Turing podemos observar que un sistema formal es indecidible si y sólo si no existe un algoritmo que pueda responder la cuestión: ¿es la fórmula α un teorema del sistema?

Teorema 2.13. *Existe un subconjunto de \mathbb{N} que es recursivamente enumerable pero no es recursivo.*

Demostración. Presentaremos en este contexto una justificación incompleta, dejando el resto al cuidado del lector.

1. Sea \mathcal{N} el sistema formal de la teoría de números.
2. El conjunto A de los axiomas del sistema formal \mathcal{N} es recursivo (a partir de los números de Gödel de los mismos).
3. Puede diseñarse un procedimiento efectivo para listar el conjunto de teoremas de \mathcal{N} a partir del conjunto de axiomas A .

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

4. Por la tesis de Church-Turing, el conjunto de los teoremas de \mathcal{N} es recursivamente enumerable.
5. \mathcal{N} es recursivamente indecidible.
6. Luego, el conjunto de los teoremas de \mathcal{N} no es recursivo.

□

2.13. Ejercicios

Ejercicio 2.1. Demuestre que las funciones test de desigualdad denotada por ϵ y test de igualdad denotada por δ son funciones primitivas recursivas.

$$\epsilon(x, y) = \begin{cases} 0 & \text{si } x = y, \\ 1 & \text{si } x \neq y. \end{cases} \quad \delta(x, y) = \begin{cases} 1 & \text{si } x = y, \\ 0 & \text{si } x \neq y. \end{cases}$$

Ejercicio 2.2. Demuestre que las siguientes funciones son primitivas recursivas:

1. $rm(x, y)$ = residuo de la división de y por x .
2. $qt(x, y)$ = cociente de la división de y por x .
3. $[\sqrt{x}]$ = mayor entero $\leq \sqrt{x}$.

Ejercicio 2.3. Demuestre que la intersección y unión de dos conjuntos primitivos recursivos y el complemento de un conjunto primitivo recursivo, es un conjunto primitivo recursivo.

Ejercicio 2.4. Demuestre que los siguientes predicados son primitivos recursivos:

1. \leq .
2. $>$.
3. \geq .
4. $|$ divide
5. $P(x)$: x es par.
6. $I(x)$: x es impar.
7. $Pr(x)$: x es primo.

Ejercicio 2.5. Demuestre que las siguientes funciones son primitivas recursivas:

$$1. \ max(x_1, x_2, \dots, x_n).$$

$$2. \ min(x_1, x_2, \dots, x_n).$$

Ejercicio 2.6. Demuestre que la sucesión de Fibonacci definida por $f(0) = 1$, $f(1) = 2$, $f(k + 2) = f(k) + f(k + 1)$ para $k \geq 0$, es primitiva recursiva.

Ejercicio 2.7. Demuestre que las siguientes funciones son primitivas recursivas:

$$1. \ f(0) = 2, \ f(1) = 4, \ f(k + 2) = 3f(k + 1) - (2f(k) + 1).$$

$$2. \ f(0) = 5, \ f(n + 1) = f(n) \cdot f(n - 2) + f(n - 5).$$

$$3. \ f(0) = 5, \ f(n + 1) = (n + 1)^{f(n)} + n \cdot f(n - 3).$$

Ejercicio 2.8. El siguiente programa calcula c empleando a como entrada. Expresar c como función de a

```
b := 3 + a;
c := 2 + a;
c := c * b;
```

Ejercicio 2.9. Expresar el valor final de la variable *suma* como una función primitiva recursiva.

```
for i := 1 to n do suma := suma + i * i
```

Ejercicio 2.10. Expresar la función $if(x, y, z)$ como una función primitiva recursiva.

$$if(x, y, z) = \begin{cases} y & \text{si } x > 0, \\ z & \text{si } x = 0. \end{cases}$$

Ejercicio 2.11. Demuestre la segunda parte del teorema 2.5 (pág. 66).

Ejercicio 2.12. Las siguientes funciones no son regulares. Justifíquelo.

$$1. \ f(x, y) = x + y.$$

$$2. \ g(x, y) = (x - y) + y.$$

Ejercicio 2.13. Verifique que la función $f(x, y, z) = (x - z) + (y - z)$ es regular para la variable z .

Ejercicio 2.14. Implemente en algún lenguaje de programación la función de Ackermann y observe el crecimiento de la función para algunos valores (x, y) contra el crecimiento del número de llamadas necesarias a la función para calcular dichos valores.

Ejercicio 2.15. La función de Ackermann es un ejemplo de una función recursiva que no es primitiva recursiva. La siguiente demostración afirma que la función de Ackermann es una función primitiva recursiva.

Demostración. Cada “brazo” de la función de Ackermann puede ser expresado de la siguientes forma:

1. $A(0, y) = s(y)$, donde $s(y)$ es la función sucesor.
2. $A(x+1, 0) = A(Pr(x+1), s(z(0)))$ donde Pr es la función predecesor y z es la función cero.
3. $A(x+1, y+1) = A(Pr(x+1), A(x+1, Pr(y+1))).$

Como cada “brazo” de la función de Ackermann puede ser expresado por una función primitiva recursiva, o por composición de funciones primitivas recursivas, entonces la función de Ackermann es primitiva recursiva. \square

Explicar por qué la demostración anterior no es válida.

Ejercicio 2.16. Presente un ejemplo de una función estrictamente recursiva parcial (que no sea total).

Ejercicio 2.17. Suponga que h es una función parcial recursiva la cual no es total y g es una función recursiva. Sea $f = h \circ g$, ¿puede f ser una función total?

Ejercicio 2.18. Para el ejemplo 2.17 (pág. 76) hallar un algoritmo más eficiente que decida si un número $n \in \mathbb{N}$ es o no es un número primo.

Ejercicio 2.19. Para el conjunto del ejemplo 2.18 (pág. 76), ¿cuál es el procedimiento de decisión?

Ejercicio 2.20. El conjunto de las tautologías de la lógica de enunciados es un conjunto decidable. ¿Cuál es el procedimiento de decisión?

Ejercicio 2.21. Demuestre que la función del ejemplo 2.22 (pág. 77) es sobreyectiva y recursiva.

Ejercicio 2.22. Demuestre el lema 2.1 (pág. 79).

2.14. Notas Bibliográficas

Los textos [3, 4, 7, 15, 21, 26, 27, 34, 36, 39], entre otros, presentan una contextualización a las funciones recursivas. Las funciones y relaciones numérico-teóricas son presentadas por [26]. En relación con la función de Ackermann, [15] presenta una demostración de que no es una función primitiva recursiva; por otra parte, algunos de los datos del número de llamadas de la función de Ackermann, fueron tomados de [9]. Las nociones de conjunto enumerable y conjunto efectivamente enumerable son presentas por [4], así como los ejemplos 2.15 (pág. 75) y 2.23 (pág. 79) y el algoritmo presentado en el teorema 2.10 (pág. 80). Para la demostración de los lemas 2.3 (pág. 81), 2.4 (pág. 82) y 2.7 (pág. 84) seguimos a [7] y para el lema 2.5 (pág. 83) seguimos a [4]. Para la demostración del teorema 2.12 (pág. 85) seguimos la presentación realizada por [3]. El texto [21] y en especial [7], ofrecen una demostración mucho más elaborada del mismo. En cuanto a la tesis de Church-Turing, [21, 34, 36] ofrecen algunos elementos, tanto históricos como técnicos.

Capítulo 3

Lenguajes y Gramáticas

3.1. Alfabetos y Lenguajes

Un alfabeto será para nosotros, en este contexto, cualquier conjunto de objetos que llamaremos símbolos indivisibles. Un ejemplo de ello es el alfabeto romano $\Sigma = \{a, \dots, z, A, \dots, Z\}$. Un alfabeto básico en la computación es el conjunto $\Sigma = \{0, 1\}$.

3.1.1. Definiciones preliminares

Definición 3.1 (Alfabeto). Un alfabeto es un conjunto enumerable (finito o infinito) de símbolos indivisibles.

Ejemplo 3.1. Ejemplos de alfabetos:

$$\Sigma_1 = \{0, 1\},$$

$$\Sigma_2 = \{/ \},$$

$$\Sigma_3 = \{(, \neg\} \cup \{p_i \mid i \in \omega\}.$$

Definición 3.2 (Palabra). Una palabra es una sucesión finita de símbolos de un alfabeto Σ . Como palabra de Σ se incluye la palabra vacía denotada por ε .

Ejemplo 3.2. Para $\Sigma_1 = \{0, 1\}$ son palabras $\alpha \equiv 00000$, $\beta \equiv 0101010101$.

Para $\Sigma_3 = \{\vee, \neg\} \cup \{p_i \mid i \in \omega\}$ son palabras:

$$\alpha_1 \equiv \neg(p_1 \vee p_2),$$

$$\alpha_2 \equiv \neg\neg\neg,$$

ε .

Definición 3.3 (Lenguaje universal). El conjunto Σ^* de todas las palabras que se pueden construir sobre un alfabeto Σ se denomina lenguaje universal para Σ , es decir: $\Sigma^* = \{a_1 a_2 \dots a_n \mid n \in \omega\} \cup \{\varepsilon\}$, $a_i \in \Sigma$.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

La definición que presentamos a continuación (lenguaje sobre un alfabeto) es válida, pero poco útil, puesto que lo que nos interesa en este contexto es el poder dar una representación finita o descripción finita de un lenguaje infinito. No obstante, para estos mismos propósitos es importante que presentemos tal definición.

Definición 3.4 (Lenguaje). Sea Σ un alfabeto. Un lenguaje \mathcal{L} sobre Σ denotado por $\mathcal{L}(\Sigma)$ es un subconjunto de Σ^* , es decir, $\mathcal{L}(\Sigma)$ es un lenguaje sobre Σ , si y sólo si, $\mathcal{L}(\Sigma) \subseteq \Sigma^*$.

Ejemplo 3.3. Si Σ es un alfabeto, entonces, Σ^* y \emptyset son lenguajes sobre Σ .

Ejemplo 3.4. Dado $\Sigma = \{a, b, \dots, z\}$, los siguientes conjuntos son lenguajes finitos y por ende tienen una descripción finita.

1. $\mathcal{L}_1(\Sigma) = \{casa, hola, perro\}$.
2. $\mathcal{L}_2(\Sigma) = \{aaa, dfg, asd, jkl\}$.

Si el lenguaje $\mathcal{L}(\Sigma)$ es infinito no podemos, bajo estas consideraciones, describir todas sus palabras, lo cual nos obligará más tarde a buscar otros métodos para su representación finita (si tal cosa es posible).

Ejemplo 3.5. Los siguientes lenguajes sobre $\Sigma = \{0, 1\}$ son infinitos:

1. $\mathcal{L}_1(\Sigma) = \{0, 01, 011, 0111, 01111, \dots\}$.
2. $\mathcal{L}_2(\Sigma) = \{\alpha \in \Sigma^* \mid l(\alpha) = 3k, k \in \omega\}$.
3. $\mathcal{L}_3(\Sigma) = \{\alpha \in \Sigma^* \mid \alpha \equiv 0\beta, \beta \in \Sigma^*\}$.

3.1.2. Operaciones sobre palabras

Sobre el conjunto de palabras que pueden construirse sobre un alfabeto dado, podemos definir una función matemática que asigne a cada palabra el número de símbolos de que consta, número que llamaremos longitud de dicha palabra.

Definición 3.5 (Longitud de una palabra). Dada un alfabeto Σ , existe una función $l: \Sigma^* \rightarrow \omega$ tal que:

$$\begin{aligned} l(\varepsilon) &= 0, \\ l(\alpha a) &= l(\alpha) + 1; \quad \alpha \in \Sigma^*, a \in \Sigma. \end{aligned}$$

Ejemplo 3.6. Sea $\Sigma = \{a, b\}$ entonces

1. $l(\varepsilon) = 0$.
2. $l(\varepsilon a) = l(\varepsilon) + 1 = 1, \quad a \in \Sigma$,

3. $l(ab) = l(a) + 1 = 1 + 1 = 2.$
4. $l(abc) = l(ab) + 1 = 2 + 1 = 3.$

Definición 3.6 (Conjunto de palabras de longitud n). Llamaremos Σ^n al conjunto de todas las palabras de longitud n construibles sobre un alfabeto Σ , luego:

$$\begin{aligned}\Sigma^0 &= \{\varepsilon\}, \\ \Sigma^1 &= \{\alpha \mid l(\alpha) = 1\}, \\ &\vdots \\ \Sigma^n &= \{\alpha \mid l(\alpha) = n\}.\end{aligned}$$

El conjunto Σ^* es el conjunto infinito: $\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$, es decir:

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n.$$

Dos palabras sobre un mismo alfabeto pueden ser combinadas para formar una tercera mediante la operación de concatenación. La concatenación $\alpha \bullet \beta$, de las palabras α y β , es la palabra obtenida escribiendo β inmediatamente después de α .

Definición 3.7 (Concatenación de palabras). La concatenación \bullet es una ley de composición interna sobre Σ^* , es decir, es una aplicación $\bullet: \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ definida por:
 $\bullet(a_1a_2\dots a_n, b_1b_2\dots b_m) = a_1a_2\dots a_nb_1b_2\dots b_m$.

En otros términos, sean $\alpha \equiv a_1a_2\dots a_n$, y $\beta \equiv b_1b_2\dots b_m$, palabras de Σ^* , entonces:
 $\alpha \bullet \beta = \alpha\beta \equiv a_1a_2\dots a_nb_1b_2\dots b_m$.

Ejemplo 3.7. Sea $\Sigma = \{a, b, c, d\}$, entonces:

1. $aaa \bullet bbb = aaabbb.$
2. $abcd \bullet dcba = abcd dcba.$

Definición 3.8 (Potencia de una palabra). Sea α una palabra sobre un alfabeto Σ , entonces:

$$\begin{aligned}\alpha^0 &= \varepsilon, \\ \alpha^1 &= \alpha, \\ \alpha^2 &= \alpha\alpha, \\ \alpha^3 &= \alpha^2\alpha, \\ &\vdots \\ \alpha^n &= \alpha^{n-1}\alpha \quad \text{para } n > 0.\end{aligned}$$

Ejemplo 3.8. Si $\Sigma = \{1, 2, 3\}$ entonces:

1. $\alpha \equiv 12$ y $\alpha^2 \equiv 1212$.
2. $\beta \equiv 23$ y $\beta^3 = \beta^2\beta \equiv (23)^2(23) = 232323$.

Teorema 3.1. Si $\alpha, \beta \in \Sigma^*$, entonces $l(\alpha \bullet \beta) = l(\alpha) + l(\beta)$.

Demostración. Ejercicio 3.7. □

Teorema 3.2. Si $\alpha \in \Sigma^*$, entonces $l(\alpha^n) = nl(\alpha)$.

Demostración. La demostración se hará por inducción.

1. Si $n = 0$ entonces

$$l(\alpha^0) = l(\varepsilon) = 0 = 0 \cdot 0 = 0 \cdot l(\varepsilon).$$
2. Suponemos que si $n = k$, entonces, $l(\alpha^k) = kl(\alpha)$.
3. Para $n = k + 1$:

$$\begin{aligned} l(\alpha^{k+1}) &= l(\alpha^k \bullet \alpha) \\ &= l(\alpha^k) + l(\alpha) \\ &= kl(\alpha) + l(\alpha) \\ &= (k + 1)l(\alpha). \end{aligned}$$

□

Definición 3.9 (Reflexión de una palabra). Sea α una palabra sobre un alfabeto Σ , y $\alpha \equiv a_1a_2 \dots a_{n-1}a_n$. Definimos la reflexión de α , denotada por α^{-1} , a la palabra: $\alpha^{-1} \equiv a_na_{n-1} \dots a_2a_1$.

Ejemplo 3.9. Sea $\Sigma = \{a, b, c, o\}$ y una palabra $\alpha \equiv acob$ entonces $\alpha^{-1} \equiv boca$.

3.1.3. Relación entre los conceptos de monoide y lenguaje

Pasemos ahora a estudiar la estructura algebraica del lenguaje universal. Inicialmente presentamos las definiciones de semigrupo y monoide.

Definición 3.10 (Semigrupo). Sea $\mathcal{L} = \{\ast\}$ un lenguaje de primer orden, compuesto por un símbolo de función de aridez dos (\ast). Un semigrupo $\mathcal{S} = \langle A, \ast \rangle$ es un modelo de \mathcal{L} que satisface el siguiente axioma:

$$\mathcal{S} \models \forall x \forall y \forall z ((x * y) * z = x * (y * z)) \quad (\text{asociativa}) \tag{S-1}$$

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Definición 3.11 (Monoide). Sea $\mathcal{L} = \{*, e\}$ un lenguaje de primer orden, compuesto por un símbolo de función de aridez dos (*) y por un símbolo de constante (e). Un monoide $\mathcal{M} = < A, *, e >$ es un modelo de \mathcal{L} que satisface los siguientes axiomas:

$$\mathcal{M} \models \forall x \forall y \forall z ((x * y) * z = x * (y * z)) \quad (\text{asociativa}) \quad (\text{M-1})$$

$$\mathcal{M} \models \forall x (x * e = e * x = x) \quad (\text{existencia elemento neutro}) \quad (\text{M-2})$$

Ejemplo 3.10. $< \mathbb{N}, * >$ y $< \mathbb{Z}, + >$ son ejemplos de semigrupos. $< \mathbb{N}, *, 1 >$ y $< \mathbb{Z}, +, 0 >$ son ejemplos de monoides.

Presentamos entonces el teorema que prescribe la estructura algebraica de un lenguaje universal.

Teorema 3.3. Si Σ es un alfabeto y si $< \Sigma^*, \bullet, \varepsilon >$ es una estructura donde; Σ^* es el lenguaje universal para Σ , \bullet es la concatenación de palabras, y ε es la palabra vacía, entonces $< \Sigma^*, \bullet, \varepsilon >$ es un monoide, llamado el monoide libre generado por Σ .

Demostración.

1. \bullet es una operación cerrada bajo Σ^* , es decir:

$$< \Sigma^*, \bullet, \varepsilon > \models \forall x \forall y (x \bullet y \in \Sigma^*).$$

2. $< \Sigma^*, \bullet, \varepsilon > \models \forall \alpha \forall \beta \forall \gamma ((\alpha \bullet \beta) \bullet \gamma = \alpha \bullet (\beta \bullet \gamma))$ (axioma de asociatividad).

3. $< \Sigma^*, \bullet, \varepsilon > \models \forall \alpha (\alpha \bullet \varepsilon = \varepsilon \bullet \alpha = \alpha)$ (axioma existencia de elemento neutro).

□

El teorema anterior afirma que $< \Sigma^*, \bullet, \varepsilon >$ es un monoide. ¿Será esto cierto si reemplazamos Σ^* por \mathcal{L} ? es decir, ¿cuálquier lenguaje \mathcal{L} sobre un alfabeto Σ , con la operación de concatenación y la palabra vacía, forman un monoide? Los siguientes ejemplos responderán a la pregunta.

Ejemplo 3.11. Sea $\mathcal{L} = \{\alpha \in \Sigma^* \mid l(\alpha) = 3k, k \in \omega\}$. Entonces $< \mathcal{L}, \bullet, \varepsilon >$ es un monoide. Veamos que \bullet es cerrada bajo \mathcal{L} . Sean $\alpha, \beta \in \mathcal{L}$, entonces:

$$\begin{aligned} l(\alpha \bullet \beta) &= l(\alpha) + l(\beta) \\ &= 3k_1 + 3k_2; \quad k_1, k_2 \in \omega \\ &= 3(k_1 + k_2); \quad k_1, k_2 \in \omega \\ &= 3p, p \in \omega, \quad \text{donde } p = k_1 + k_2. \end{aligned}$$

Luego, \bullet es cerrada bajo \mathcal{L} . Además \mathcal{L} hereda la asociativa de Σ^* y e opera como elemento neutro (ε , donde $l(\varepsilon) = 0 = 3(0)$ luego $\varepsilon \in \mathcal{L}$). Podemos entonces afirmar que $< \mathcal{L}, \bullet, \varepsilon >$ es un monoide.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Ejemplo 3.12. Sea $\mathcal{L} = \{\alpha \in \Sigma^* \mid l(\alpha) = 2k + 1, k \in \omega\}$. Entonces $\langle \mathcal{L}, \bullet, \varepsilon \rangle$ no es un monoide. Veamos que \bullet no es cerrada bajo \mathcal{L} . Sean $\alpha, \beta \in \mathcal{L}$ tales que, $l(\alpha) = 5$ y $l(\beta) = 7$, entonces:

$$\begin{aligned} l(\alpha \bullet \beta) &= l(\alpha) + l(\beta) \\ &= 5 + 7 = 12; \end{aligned}$$

pero, 12 no se puede expresar como $2k + 1$ con $k \in \omega$. Luego \bullet no es cerrada bajo \mathcal{L} .

3.1.4. Operaciones entre lenguajes

Sean \mathcal{L} y \mathcal{L}' dos lenguajes sobre un alfabeto Σ . Definimos las siguientes operaciones entre ellos:

1. Reflexión: $\mathcal{L}^{-1} = \{\alpha^{-1} \mid \alpha \in \mathcal{L}\}$.
2. Unión: $\mathcal{L} \cup \mathcal{L}' = \{\alpha \mid \alpha \in \mathcal{L} \vee \alpha \in \mathcal{L}'\}$.
3. Intersección: $\mathcal{L} \cap \mathcal{L}' = \{\alpha \mid \alpha \in \mathcal{L} \wedge \alpha \in \mathcal{L}'\}$.
4. Complemento: $\overline{\mathcal{L}} = \{\alpha \in \Sigma^* \mid \alpha \notin \mathcal{L}\}$
5. Concatenación: $\mathcal{L} \bullet \mathcal{L}' = \{\alpha \bullet \beta \mid \alpha \in \mathcal{L} \wedge \beta \in \mathcal{L}'\}$.
6. Clausura de Kleene:

La clausura de Kleene de un lenguaje \mathcal{L} , denotada por \mathcal{L}^* , es el conjunto de todas las palabras finitas construibles con los elementos de \mathcal{L} (incluyendo la palabra vacía), esto es:

$$\mathcal{L}^* = \{\alpha \in \Sigma^* \mid \alpha = \alpha_1 \bullet \alpha_2 \bullet \cdots \bullet \alpha_n \wedge n \geq 0 \wedge \alpha_i \in \mathcal{L}\}.$$

Podemos crear la sucesión:

$$\begin{aligned} \mathcal{L}^0 &= \{\varepsilon\}, \\ \mathcal{L}^{k+1} &= \mathcal{L}^k \bullet \mathcal{L}; \end{aligned}$$

Luego, $\mathcal{L}^* = \bigcup_{n \in \omega} \mathcal{L}^n$, donde $\omega = \{0, 1, 2, \dots\}$.

Observemos que si $\mathcal{L} = \Sigma$, entonces, $\mathcal{L}^* = \Sigma^*$.

7. Clausura positiva:

La clausura positiva de un lenguaje \mathcal{L} , denotada por \mathcal{L}^+ , es el conjunto de todas las palabras finitas construibles con los elementos de \mathcal{L} , sin incluir la palabra vacía, es decir:

$$\mathcal{L}^+ = \bigcup_{n \in \mathbb{N}} \mathcal{L}^n, \text{ donde } \mathbb{N} = \{1, 2, \dots\}.$$

Ejemplo 3.13. Sea $\mathcal{L} = \{otla, cola, avu, aa\}$ entonces $\mathcal{L}^{-1} = \{alto, aloc, uva, aa\}$.

Ejemplo 3.14. Sea $\mathcal{L}(\Sigma) = \{\alpha\}$, donde $\alpha \in \Sigma^*$ entonces $\mathcal{L}(\Sigma) \bullet \Sigma^* = \{\alpha \bullet \beta \mid \beta \in \Sigma^*\}$. Este lenguaje es el conjunto de todas las palabras que tienen prefijo α .

Ejemplo 3.15. Si $\Sigma = \{0, 1\}$ y $\mathcal{L}(\Sigma) = \{01, 1, 100\}$ entonces:

$$\mathcal{L}^0(\Sigma) = \{\varepsilon\},$$

$$\mathcal{L}^1(\Sigma) = \mathcal{L}(\Sigma) = \{01, 1, 100\},$$

$$\mathcal{L}^2(\Sigma) = \{0101, 011, 01100, 101, 11, 1100, 10001, 1001, 100100\},$$

$$\mathcal{L}^3(\Sigma) = \mathcal{L}^2(\Sigma) \bullet \mathcal{L}(\Sigma),$$

$$\mathcal{L}^*(\Sigma) = \bigcup_{n \in \omega} \mathcal{L}^n(\Sigma).$$

Ejemplo 3.16. Sean $\mathcal{L} = \{A, \dots, Z, a, \dots, z\}$ y $\mathcal{L}' = \{0, \dots, 9\}$, entonces:

$$\mathcal{L} \cup \mathcal{L}' = \{\alpha \mid \alpha \text{ es una letra o es un dígito}\}.$$

$$\mathcal{L}\mathcal{L}' = \{\alpha\beta \mid \alpha \text{ es una letra y } \beta \text{ es un dígito}\}.$$

$$\mathcal{L}^* = \{\alpha \mid \alpha \text{ es una secuencia de letras (incluyendo la secuencia vacía)}\}.$$

$$\mathcal{L}'^+ = \{\alpha \mid \alpha \text{ es una secuencia de dígitos (no incluye la secuencia vacía)}\}.$$

$$\mathcal{L}(\mathcal{L} \cup \mathcal{L}')^* = \{\alpha \mid \alpha \text{ es una secuencia de letras o dígitos y además } \alpha \text{ comienza por una letra}\}.$$

3.2. Sistemas Formales y Sistemas Combinatorios

Definición 3.12 (Sistema formal). Un sistema formal es una estructura matemática $\mathcal{SF} = < \Sigma, \Gamma, \Psi, \Delta >$, donde:

Σ : Alfabeto.

Γ : Conjunto de fórmulas (obtenido a partir de unas reglas de formación de fórmulas).

Ψ : Conjunto de reglas de transformación de fórmulas.

Δ : Conjunto de axiomas.

Ejemplo 3.17. Definamos un sistema formal para la lógica de enunciados por $\mathcal{LE} = < \Sigma, \Gamma, \Psi, \Delta >$ donde:

$$\Sigma = \{\neg, \vee, \wedge, \rightarrow, \leftrightarrow\} \cup \{p_i \mid i \in \omega\} \cup \{(,)\}.$$

Γ está determinado por las siguientes reglas de formación de fórmulas:

R-1 $\forall i \in \omega (p_i \in \Gamma)$.

R-2 Si $\alpha \in \Gamma$ entonces $\neg(\alpha) \in \Gamma$.

R-3 Si $\alpha, \beta \in \Gamma$ entonces $(\alpha) \vee (\beta) \in \Gamma$.

Ψ está formado por una única regla llamada regla de separación:

$$(\alpha \wedge (\alpha \rightarrow \beta)) \rightarrow \beta.$$

Un conjunto de axiomas Δ está formado por:

$$\text{Axioma-1 } (\alpha \vee \alpha) \rightarrow \alpha.$$

$$\text{Axioma-2 } \alpha \rightarrow (\alpha \vee \beta)$$

$$\text{Axioma-3 } (\alpha \vee \beta) \rightarrow (\beta \vee \alpha).$$

Axioma-4 $(\alpha \vee \beta) \rightarrow ((\gamma \vee \alpha) \rightarrow (\gamma \vee \beta)).$

Definición 3.13 (Sistema combinatorio). Un sistema combinatorio es una estructura matemática $\mathcal{SC} = < \Sigma_p, \Sigma_a, \Gamma, \Psi, \Delta >$, donde:

Σ_p : Alfabeto principal.

Σ_a : Alfabeto auxiliar.

Γ : Conjunto de fórmulas (obtenido a partir de una reglas de formación de fórmulas).

Ψ : Conjunto de reglas de transformación de fórmulas.

Δ : Conjunto unitario de axiomas.

Ejemplo 3.18. Si examinamos las figuras 3.1(a), 3.1(b), 3.1(c) y 3.1(d), podemos descubrir que cada figura se construye usando como base la figura anterior. Observemos el paso de la figura 3.1(a) a la figura 3.1(b); este paso es una transformación de cada uno de los segmentos de recta, que es la figura 3.1(a), en un conjunto de segmentos de recta. Si observamos la figura 3.1(c), veremos que ésta se obtiene de la figura 3.1(b) efectuando la misma transformación a cada uno de los segmentos de recta que la componen. La figura 3.1(d) es producto de hacer la misma transformación a la figura 3.1(c) y así, sucesivamente.

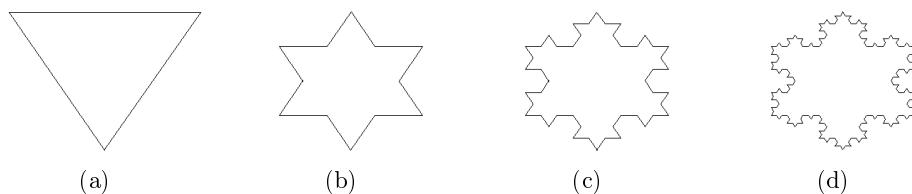


Figura 3.1: Curva de Koch.

Éste es un ejemplo de lo que se conoce como un fractal (tipo L). Vamos a definir el fractal anterior conocido como curva de Koch por medio de un sistema combinatorio $\mathcal{SC} = < \Sigma_p, \Sigma_a, \Gamma, \Psi, \Delta >$, donde:

$\Sigma_p = \{L, +, -\}$. La interpretación que damos a los símbolos de Σ_p es la siguiente: L : segmento de recta, $+$: giro de 45° en el sentido de las manecillas del reloj y $-$: giro de 45° en sentido inverso al de las manecillas del reloj.

$\Sigma_a = \{\rightarrow\}$.

Γ : Conjunto de fractales obtenidos.

Ψ está compuesto por una única regla:

$L \rightarrow L - L + +L - L$.

$\Delta = \{L\}$.

3.3. Gramáticas Formales o de Frase Estructurada

3.3.1. Problema de la representación

Con respecto al tratamiento de los lenguajes (artificiales o naturales), un objetivo que se plantea es el de formalizarlos. Por lo general los lenguajes “interesantes” son infinitos; entonces surge el problema: ¿cómo representar un lenguaje infinito? Hay dos posibles soluciones, a saber:

1. Representación analítica: máquinas abstractas.
2. Representación generadora: sistema de generación finito con un conjunto finito de reglas (gramática generativa).

3.3.2. Gramáticas

Definición 3.14 (Gramática). Una gramática está definida por una estructura matemática $\mathcal{G} = \langle \mathbf{N}, \mathbf{T}, \mathbf{P}, \mathbf{I} \rangle$, donde:

N: Conjunto finito de símbolos llamados *No Terminales*.

T: Conjunto finito de símbolos llamados *Terminales*.

P: Conjunto de reglas de producción.

I: Axioma de inicio o símbolo inicial.

Los elementos de una gramática presentan las siguientes características:

G-1. Los conjuntos **N** y **T** son disjuntos ($\mathbf{N} \cap \mathbf{T} = \emptyset$).

G-2. $\mathbf{I} \notin (\mathbf{N} \cup \mathbf{T})$.

G-3. Las producciones tienen la forma:

- a) $\alpha A\beta \rightarrow \gamma\delta\eta$ donde $\alpha, \beta, \gamma, \delta, \eta \in (\mathbf{N} \cup \mathbf{T})^*$; $A = \mathbf{I} \vee A \in \mathbf{N}$.
- b) La palabra $\alpha A\beta$ recibe el nombre de “lado izquierdo” de la producción;
- c) La palabra $\gamma\delta\eta$ recibe el nombre de “lado derecho” de la producción.
- d) El símbolo “ \rightarrow ” se lee: $\alpha A\beta$ deriva en $\gamma\delta\eta$.

Con respecto a la notación, se utilizan letras mayúsculas para denotar los símbolos no terminales y letras minúsculas para denotar los símbolos terminales.

Definición 3.15 (Lenguaje generado por una gramática). El lenguaje generado por una gramática \mathcal{G} , denotado por $\mathcal{L}(\mathcal{G})$, es el conjunto de secuencias de símbolos terminales que se pueden derivar a partir de **I**, es decir, $\mathcal{L}(\mathcal{G}) = \{\alpha \in \mathbf{T}^* \mid \mathbf{I} \xrightarrow{*} \alpha\}$ donde, $\xrightarrow{*}$ representa cero o más producciones, es decir, α se puede obtener por sucesivas derivaciones a partir de **I**. Para efectos de facilitar la notación (cuando no haya lugar a confusión), vamos a denotar $\mathcal{L}(\mathcal{G})$ por \mathcal{L} .

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Ejemplo 3.19. Sea $\mathcal{G} = \langle N, T, P, I \rangle$ donde:

$$I = I.$$

$N = \{\text{ORACIÓN, SUJETO, PREDICADO, ARTÍCULO, ADVERBIO, SUSTANTIVO, VERBO}\}$.

$T = \{\text{la, el, mujer, hombre, lee, escribe, adecuadamente, inadecuadamente}\}$.

El conjunto P está formado por las siguientes reglas de producción:

$$P_1: I \rightarrow \text{ORACIÓN}$$

$$P_2: \text{ORACIÓN} \rightarrow \text{SUJETO PREDICADO}$$

$$P_3: \text{SUJETO} \rightarrow \text{ARTÍCULO SUSTANTIVO}$$

$$P_4: \text{ARTÍCULO} \rightarrow \text{el}$$

$$P_5: \text{ARTÍCULO} \rightarrow \text{la}$$

$$P_6: \text{SUSTANTIVO} \rightarrow \text{hombre}$$

$$P_7: \text{SUSTANTIVO} \rightarrow \text{mujer}$$

$$P_8: \text{PREDICADO} \rightarrow \text{VERBO ADVERBIO}$$

$$P_9: \text{VERBO} \rightarrow \text{lee}$$

$$P_{10}: \text{VERBO} \rightarrow \text{escribe}$$

$$P_{11}: \text{ADVERBIO} \rightarrow \text{adecuadamente}$$

$$P_{12}: \text{ADVERBIO} \rightarrow \text{inadecuadamente}$$

Observemos que cada una de las producciones satisface el formato definido para las mismas. Las palabras (en realidad son oraciones) que componen a \mathcal{L} son:

$$\alpha_1 \equiv \text{la mujer lee adecuadamente}$$

$$\alpha_2 \equiv \text{la mujer lee inadecuadamente}$$

$$\alpha_3 \equiv \text{la mujer escribe adecuadamente}$$

$$\alpha_4 \equiv \text{la mujer escribe inadecuadamente}$$

$$\alpha_5 \equiv \text{el hombre lee adecuadamente}$$

$$\alpha_6 \equiv \text{el hombre lee inadecuadamente}$$

$$\alpha_7 \equiv \text{el hombre escribe adecuadamente}$$

$$\alpha_8 \equiv \text{el hombre escribe inadecuadamente}$$

Observemos que por ejemplo:

$\beta \equiv \text{"el hombre lee"}$, no es una palabra de \mathcal{L} . Para que lo fuera, sería necesario que ADVERBIO se pudiera derivar en vacío.

Ejemplo 3.20. Sea $\mathcal{G} = \langle \{A, B\}, \{0, 1\}, P, I \rangle$, donde P está formado por las siguientes reglas de producción:

$$I \rightarrow 1B$$

$$I \rightarrow 1$$

$$B \rightarrow 0A$$

$$A \rightarrow 1B$$

$$A \rightarrow 1$$

De acuerdo con las reglas de producción, tenemos que:

$$I \rightarrow 1B \rightarrow 10A \rightarrow 101B \rightarrow 1010A \rightarrow \dots \rightarrow 10\dots10; \text{ es decir, } \mathcal{L} = (10)^*1.$$

Ejemplo 3.21. Sea $\mathcal{G} = <\{A\}, \{a, b\}, \{I \rightarrow A, A \rightarrow aAb, A \rightarrow ab\}, I >$. Con base en las reglas de producción, tenemos:

$$I \rightarrow A \rightarrow aAb \rightarrow aaAbb \rightarrow aaaAbbb \rightarrow \dots \rightarrow a_1a_2\dots a_nabb_1b_2\dots b_n; \text{ es decir, } \mathcal{L} = \{a^n b^n \mid n \geq 1\}.$$

Ejemplo 3.22. Sea \mathcal{G} una gramática definida por las siguientes producciones:

$$\begin{aligned} I &\xrightarrow{1} A \\ A &\xrightarrow{2} aABC \\ A &\xrightarrow{3} abC \\ CB &\xrightarrow{4} BC \\ bB &\xrightarrow{5} bb \\ bC &\xrightarrow{6} bc \\ cC &\xrightarrow{7} cc \end{aligned}$$

El lenguaje asociado con esta gramática es, $\mathcal{L} = \{a^n b^n c^n \mid n \geq 1\}$. Construyamos inicialmente la derivación para la palabra $\alpha \equiv aabbcc$. En este caso indicamos la producción utilizada en cada paso de la derivación.

$$I \xrightarrow{1} A \xrightarrow{2} aABC \xrightarrow{3} aabCBC \xrightarrow{4} aabBCC \xrightarrow{5} aabbCC \xrightarrow{6} aabbcC \xrightarrow{7} aabbcc.$$

Ejemplo 3.23. Sea \mathcal{G} una gramática definida por las siguientes producciones:

$$\begin{aligned} I &\xrightarrow{1} ACaB \\ Ca &\xrightarrow{2} aaC \\ CB &\xrightarrow{3} DB \\ CB &\xrightarrow{4} E \\ AD &\xrightarrow{5} AC \\ aD &\xrightarrow{6} Da \\ aE &\xrightarrow{7} Ea \\ AE &\xrightarrow{8} \varepsilon \end{aligned}$$

El lenguaje asociado con esta gramática es $\mathcal{L} = \{a^{2^k}\}$. Para $a^{2^2} = a^4$ se obtiene la siguiente derivación:

$$\begin{aligned} \mathbf{I} &\xrightarrow{1} ACaB \xrightarrow{2} AaaCB \xrightarrow{3} AaaDB \xrightarrow{6} AaDaB \xrightarrow{6} ADaaB \xrightarrow{5} ACaaB \xrightarrow{2} AaaCaB \xrightarrow{2} \\ &AaaaaCB \xrightarrow{4} AaaaaE \xrightarrow{7} AaaaEa \xrightarrow{7} AaaEaa \xrightarrow{7} AaEaaa \xrightarrow{7} AEaaaa \xrightarrow{8} aaaa. \end{aligned}$$

3.3.3. Taxonomía de las gramáticas

Sean $\mathcal{G} = \langle N, T, P, I \rangle$ una gramática, $\alpha, \beta, \gamma \in (N \cup T)^*$, $A \in N \cup \{I\}$, $B \in N$, $a \in T$, y ε es la palabra vacía.

La taxonomía de las gramáticas está sustentada sobre el patrón de producciones que pueden tener, tal como está indicado en la tabla 3.1.

Tipo	Nombre	Producciones	Comentarios
0	Gramáticas no restringidas	$\alpha A \beta \rightarrow \alpha \gamma \beta$.	Admiten producciones que implican decrecimiento. La única restricción, es que no permite producciones de la forma $\varepsilon \rightarrow \gamma$.
1	Gramáticas sensibles al contexto	$\alpha A \beta \rightarrow \alpha \gamma \beta; \gamma \neq \varepsilon$.	No tiene producciones compresoras. Admite $I \rightarrow \varepsilon$ (elegancia en el lenguaje que genera).
2	Gramáticas independientes del contexto	$A \rightarrow \gamma; \gamma \neq \varepsilon$.	El contexto es obligatoriamente vacío. De este tipo son la mayoría de los lenguajes de programación. Admite $I \rightarrow \varepsilon$.
3	Gramáticas regulares, gramáticas de Kleene o k-gramáticas	Lineales o recursivas a la derecha: $A \rightarrow aB$ y $A \rightarrow a$. Lineales o recursivas a la izquierda: $A \rightarrow Ba$ y $A \rightarrow a$.	Pueden contener a lo sumo un símbolo no terminal en la parte derecha de la producción. Admite $I \rightarrow \varepsilon$.

Cuadro 3.1: Taxonomía de las gramáticas.

Para cada tipo de gramática existe un tipo de lenguaje y para cada tipo de lenguaje existe un tipo de máquina (abstracta) que reconoce las palabras de ese lenguaje. La tabla 3.2 esquematiza esta situación.

Finalmente diremos que $\mathcal{G}_3 \subseteq \mathcal{G}_2 \subseteq \mathcal{G}_1 \subseteq \mathcal{G}_0$, luego, $\mathcal{L}(\mathcal{G}_3) \subseteq \mathcal{L}(\mathcal{G}_2) \subseteq \mathcal{L}(\mathcal{G}_1) \subseteq \mathcal{L}(\mathcal{G}_0)$.

Tipo gramática	Tipo lenguaje	Reconocedor
Gramáticas no restringidas (tipo 0) \mathcal{G}_0 .	Lenguajes no restringidos $\mathcal{L}(\mathcal{G}_0)$.	Máquina de Turing.
Gramáticas sensibles al contexto (tipo 1) \mathcal{G}_1 .	Lenguajes sensibles al contexto $\mathcal{L}(\mathcal{G}_1)$.	Autómatas linealmente independientes.
Gramáticas independientes del contexto (tipo 2) \mathcal{G}_2 .	Lenguajes independientes del contexto $\mathcal{L}(\mathcal{G}_2)$.	Autómatas de pila (<i>stack</i>).
Gramáticas regulares (tipo 3) \mathcal{G}_3 .	Lenguajes regulares $\mathcal{L}(\mathcal{G}_3)$.	Autómatas de estado finito.

Cuadro 3.2: Gramáticas, lenguajes y reconocedores.

Ejemplo 3.24. La gramática del ejemplo 3.20 (pág. 108) es una gramática tipo 3; la del ejemplo 3.21 (pág. 109) es una tipo 2; la del ejemplo 3.22 (pág. 109) es una tipo 1 y la del ejemplo 3.23 (pág. 109) es una tipo 0.

3.3.4. Notación alternativa para las gramáticas

Definición 3.16 (Notación BNF). Un tipo de notación muy utilizada en informática para especificar gramáticas es la notación BNF (Backus - Nour Form):

BNF-1 Los símbolos no terminales se encierran entre <>.

BNF-2 Las producciones tienen la forma $\alpha ::= \beta$.

BNF-3 Si existen varias derivaciones de un mismo “símbolo”, éstas se representan por $\alpha ::= \beta_1 | \dots | \beta_n$.

Ejemplo 3.25.

```

I ::= < lista >
< lista > ::= < lista > + < dígito >
          | < lista > - < dígito >
          | < dígito >
< dígito > ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Algunas palabras son: $\alpha_1 \equiv 3 - 3 + 2$, $\alpha_2 \equiv 1 + 2 - 3$, etc.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Ejemplo 3.26.

$$\begin{aligned}
 I ::= & <\text{entero}> \\
 <\text{entero}> ::= & <\text{entero-con-signo}> \\
 & | <\text{entero-sin-signo}> \\
 <\text{entero-con-signo}> ::= & + <\text{entero-sin-signo}> \\
 & | - <\text{entero-sin-signo}> \\
 <\text{entero-sin-signo}> ::= & <\text{dígito}> \\
 & | <\text{dígito}><\text{entero-sin-signo}> \\
 <\text{dígito}> ::= & 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 \end{aligned}$$

Algunas palabras son: -344, 567, +9784, 8, 0000, etc.

3.3.5. Algunos aspectos sobre las gramáticas

Definición 3.17 (Árbol de análisis sintáctico). El objetivo del árbol de análisis sintáctico es ilustrar la derivación de una cadena del lenguaje a partir del símbolo inicial de la gramática. Este árbol también es llamado árbol de *parsing* o árbol de derivación. La construcción del árbol de análisis sintáctico está dirigida por las siguientes reglas:

1. La raíz es el símbolo inicial.
2. Las hojas son símbolos terminales.
3. Los nodos interiores son símbolos no terminales.
4. Si α es un nodo interior y $\beta_1, \beta_2, \dots, \beta_n$ son sus hijos de izquierda a derecha, entonces $\alpha \rightarrow \beta_1\beta_2\dots\beta_n$ es una producción.

Ejemplo 3.27. Sea \mathcal{G} una gramática definida por las siguientes producciones:

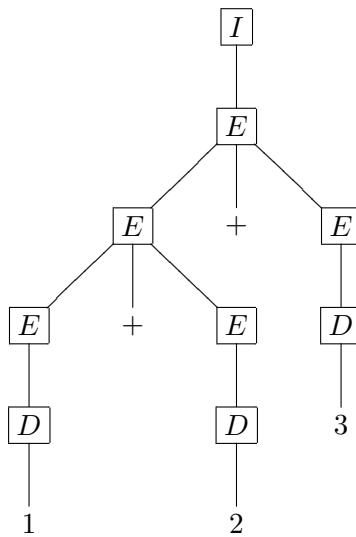
$$\begin{aligned}
 I \rightarrow & E \\
 E \rightarrow & E + E \mid E - E \mid D \\
 D \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
 \end{aligned}$$

La figura 3.2 representa el árbol de análisis sintáctico para la palabra $\alpha \equiv 1 + 2 + 3$.

La definición que realizamos del árbol de análisis sintáctico es válida únicamente para gramáticas tipo II o gramáticas tipo III, es decir, gramáticas para las cuales el contexto del lado izquierdo de cualquier producción es vacío.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Figura 3.2: Árbol de análisis sintáctico para $\alpha \equiv 1 + 2 + 3$.

Definición 3.18 (Derivación: izquierda y derecha). La derivación por izquierda consiste en sustituir en cada paso de la derivación de una cadena el símbolo no terminal más a la izquierda perteneciente a ella. Similarmente, la derivación por derecha consiste en sustituir en cada paso de la derivación de una cadena el símbolo no terminal más a la derecha perteneciente a ella.

Ejemplo 3.28. Sea \mathcal{G} una gramática definida por las siguientes producciones:

$$\begin{aligned}
 I &\rightarrow E \\
 E &\rightarrow E + E \\
 &\quad | \ (E) \\
 &\quad | \ -E \\
 &\quad | \ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
 \end{aligned}$$

Sea $\alpha \equiv -(3 + 1)$.

Derivación por la izquierda:

$$\begin{aligned}
 I &\rightarrow E \\
 &\rightarrow -E \\
 &\rightarrow -(E) \\
 &\rightarrow -(E+E) \\
 &\rightarrow -(3+E) \text{ se sustituyó el símbolo no terminal más a la izquierda} \\
 &\rightarrow -(3+1)
 \end{aligned}$$

Derivación por la derecha:

$$\begin{aligned}
 I &\rightarrow E \\
 &\rightarrow -E \\
 &\rightarrow -(E) \\
 &\rightarrow -(E+E) \\
 &\rightarrow -(E+1) \text{ se sustituyó el símbolo no terminal más a la derecha} \\
 &\rightarrow -(3+1)
 \end{aligned}$$

La figura 3.3 representa los árboles de derivación para la derivación por la izquierda y para la derivación por la derecha que hemos realizado. Es decir, sin importar qué derivación realizamos, obtenemos para la palabra $\alpha \equiv -(3+1)$ el mismo árbol de análisis sintáctico.

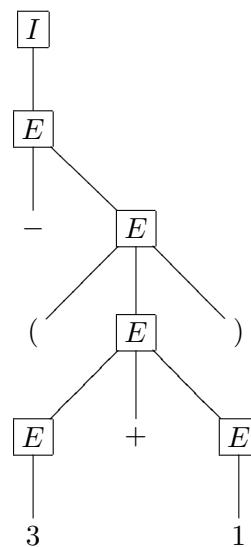


Figura 3.3: Árbol de derivación por la derecha y por la izquierda para $\alpha \equiv 3 + 1$.

Definición 3.19 (Ambigüedad). Sea $\mathcal{G} = \langle N, T, P, I \rangle$ una gramática. Se dice que \mathcal{G} es ambigua si (las siguientes condiciones son equivalentes):

1. Existe $\alpha \in \mathcal{L}(\mathcal{G})$ tal que $I \xrightarrow{*} \alpha$ y $I \xrightarrow{*} \alpha'$, son dos derivaciones diferentes para α . Es decir existen $\beta_1, \beta_2, \dots, \beta_n$ y $\gamma_1, \gamma_2, \dots, \gamma_m$ tales que:

$$I \rightarrow \alpha_1 \wedge \alpha_1 \rightarrow \beta_2 \wedge \dots \wedge \beta_{n-1} \rightarrow \beta_n \wedge \beta_n \rightarrow \alpha$$

$$I \rightarrow \gamma_1 \wedge \gamma_1 \rightarrow \gamma_2 \wedge \dots \wedge \gamma_{m-1} \rightarrow \gamma_m \wedge \gamma_m \rightarrow \alpha.$$
2. Existe más de una derivación por el mismo sentido para una palabra $\alpha \in \mathcal{L}(\mathcal{G})$.
3. Una palabra $\alpha \in \mathcal{L}(\mathcal{G})$ tiene más de un árbol de análisis sintáctico.

Ejemplo 3.29. Sea \mathcal{G} una gramática definida por las siguientes producciones:

$$\begin{aligned} I &\rightarrow E \\ E &\rightarrow E + E \mid E - E \\ &\quad | \quad 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

La figura 3.4 representa dos árboles de análisis sintáctico para la palabra $\alpha \equiv 1 - 2 + 3$. Luego la gramática \mathcal{G} es ambigua.

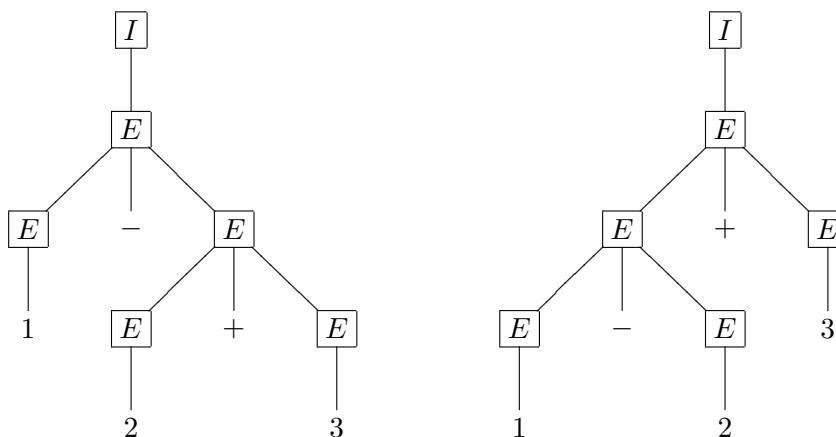


Figura 3.4: Árboles de análisis sintáctico para $\alpha \equiv 1 - 2 + 3$.

Ejemplo 3.30. Sea \mathcal{G} la gramática para el **if-else** definida por:

$$\begin{aligned}
 I &\rightarrow PROP \\
 PROP &\rightarrow if\ EXP\ then\ PROP \\
 PROP &\rightarrow if\ EXP\ then\ PROP\ else\ PROP \\
 PROP &\rightarrow OTRA \\
 PROP &\rightarrow p_1\mid p_2 \\
 EXP &\rightarrow e_1\mid e_2 \\
 OTRA &\rightarrow PROP
 \end{aligned}$$

Sea $\alpha \equiv if\ e_1\ then\ if\ e_2\ then\ p_1\ else\ p_2$. Las figuras 3.5 y 3.6 representan dos árboles de derivación para α .

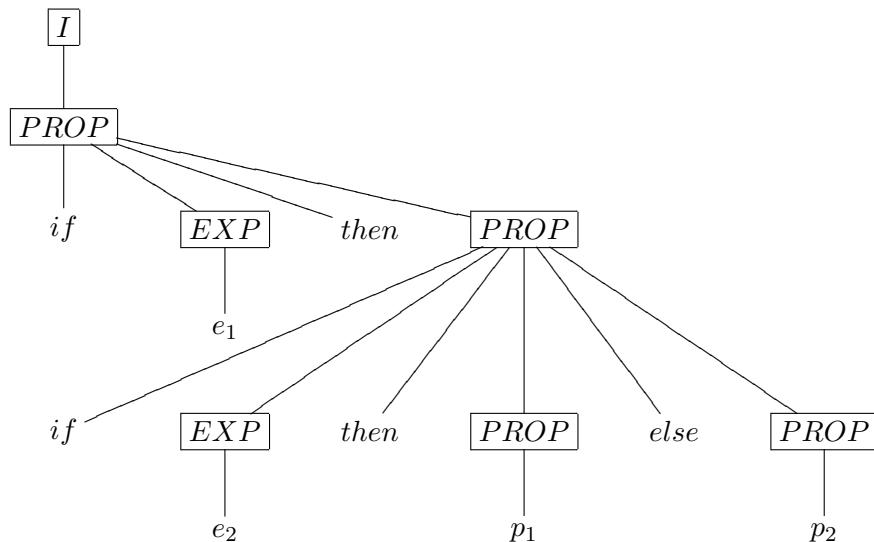


Figura 3.5: Primer árbol de análisis sintáctico con base en una gramática ambigua para $\alpha \equiv if\ e_1\ then\ if\ e_2\ then\ p_1\ else\ p_2$.

De acuerdo con lo anterior, la gramática \mathcal{G} para el **if-else** es ambigua. El problema consiste en el emparejamiento del **else**. La solución es formalizar en la gramática la siguiente regla: emparejar cada **else** con el **if** más cercano (esta es la regla usual que implementan los lenguajes de programación).

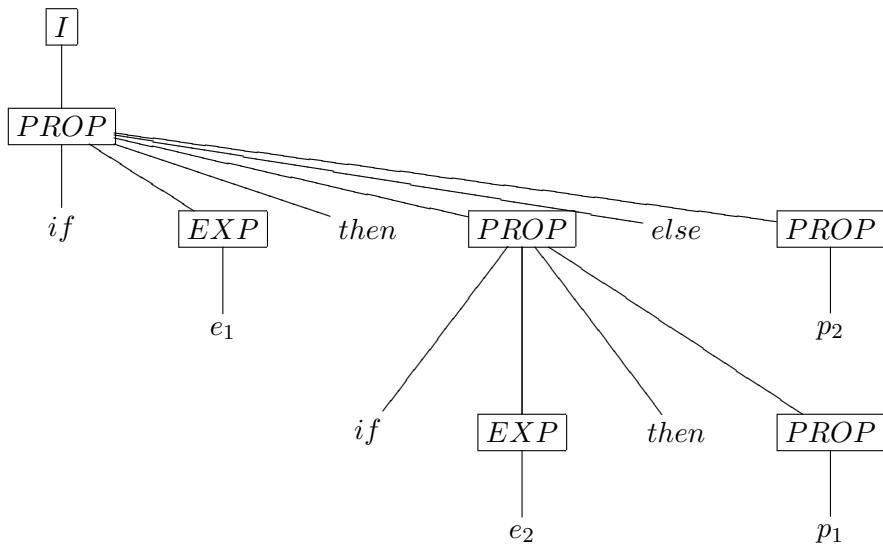


Figura 3.6: Segundo árbol de análisis sintáctico con base en una gramática ambigua para $\alpha \equiv \text{if } e_1 \text{ then if } e_2 \text{ then } p_1 \text{ else } p_2$.

Entonces, la gramática \mathcal{G} para el **if-else** no ambigua está definida por:

$$\begin{aligned}
 I &\rightarrow PROP \\
 PROP &\rightarrow IF \\
 PROP &\rightarrow IFE \\
 IFE &\rightarrow \text{if EXP then IFE else IFE} \\
 IFE &\rightarrow OTRA \\
 IF &\rightarrow \text{if EXP then PROP} \\
 IF &\rightarrow \text{if EXP then IFE else IFE} \\
 EXP &\rightarrow e_1 \mid e_2 \\
 OTRA &\rightarrow p_1 \mid p_2
 \end{aligned}$$

La figura 3.7 representa el árbol de *parsing* para la palabra:
 $\alpha \equiv \text{if } e_1 \text{ then if } e_2 \text{ then } p_1 \text{ else } p_2$.

Definición 3.20 (Recursividad). La recursividad de un gramática es el sentido en el cual crece el árbol de análisis sintáctico.

Ejemplo 3.31. Sea $\mathcal{L} = \{\alpha\beta \mid \alpha \equiv 1 \text{ y } \beta \text{ es una secuencia de cero o más } 01\}$ un lenguaje. Para \mathcal{L} vamos a construir una gramática recursiva por la derecha y una gramática recursiva por la izquierda:

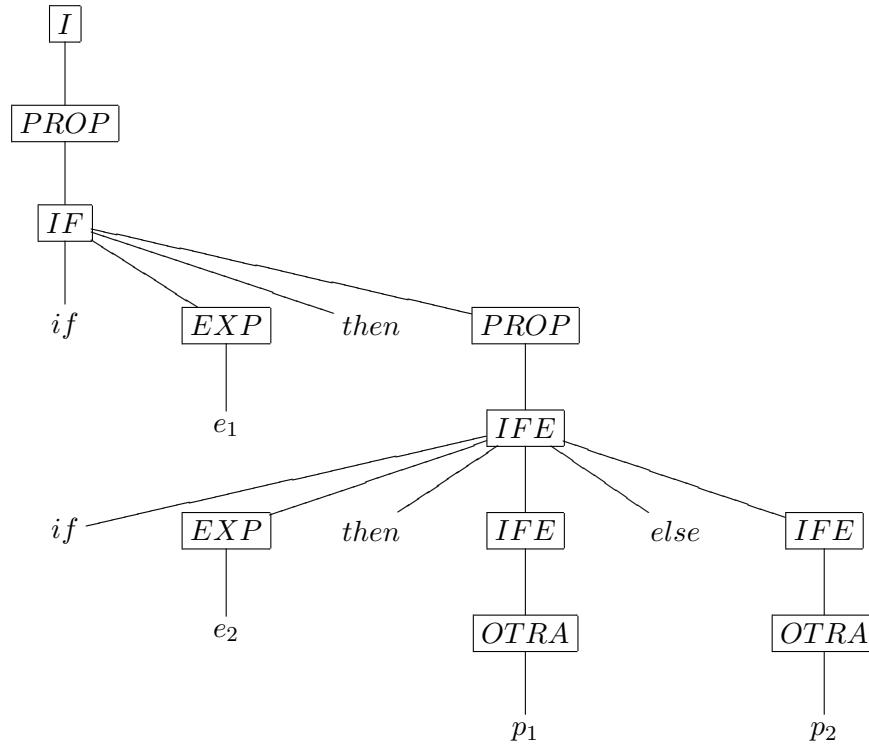


Figura 3.7: Árbol de análisis sintáctico con base en una gramática no ambigua para $\alpha \equiv \text{if } e_1 \text{ then if } e_2 \text{ then } p_1 \text{ else } p_2$.

1. Gramática recursiva por la derecha:

$$\begin{aligned} I &\rightarrow 1B \mid 1 \\ A &\rightarrow 1B \mid 1 \\ B &\rightarrow 0A \end{aligned}$$

Analizando estas producciones, podemos realizar la siguiente simplificación:

$$\begin{aligned} I &\rightarrow A \\ A &\rightarrow 1B \mid 1 \\ B &\rightarrow 0A \end{aligned}$$

Y analizando de nuevo las producciones obtenidas, podemos realizar la siguiente simplificación:

$$\begin{aligned} I &\rightarrow A \\ A &\rightarrow 10A \mid 1 \end{aligned}$$

Con lo cual podemos observar que la gramática crece por la derecha. En este caso $\mathcal{L} = (10)^*1$. Para la palabra $\alpha \equiv 10101$, el árbol de análisis sintáctico, el cual refleja la recursividad por la derecha, está representado por la figura 3.8.

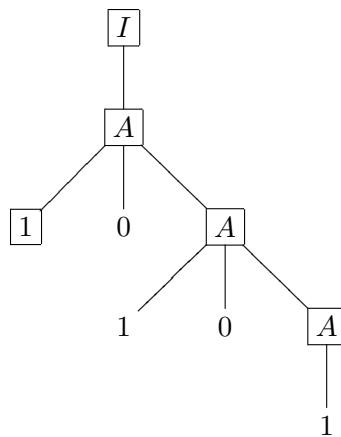


Figura 3.8: Recursividad por la derecha para $\alpha \equiv 10101$.

2. Gramática recursiva por la izquierda:

$$\begin{aligned} I &\rightarrow B1 \mid 1 \\ A &\rightarrow B1 \mid 1 \\ B &\rightarrow A0 \end{aligned}$$

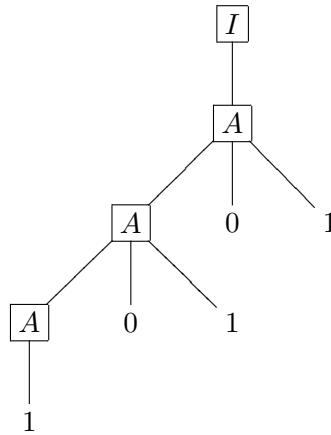
Analizando estas producciones podemos realizar la siguiente simplificación:

$$\begin{aligned} I &\rightarrow A \\ A &\rightarrow B1 \mid 1 \\ B &\rightarrow A0 \end{aligned}$$

Y analizando de nuevo las producciones obtenidas, podemos realizar la siguiente simplificación:

$$\begin{aligned} I &\rightarrow A \\ A &\rightarrow A01 \mid 1 \end{aligned}$$

Así podemos observar que la gramática crece por la izquierda. En este caso, $\mathcal{L} = 1(01)^*$. Para la palabra $\alpha \equiv 10101$, el árbol de análisis sintáctico, que refleja la recursividad por la izquierda, está representado por la figura 3.9.

Figura 3.9: Recursividad por la izquierda para $\alpha \equiv 10101$.

3.3.6. Algunos teoremas sobre gramáticas

Teorema 3.4. Una gramática es sensible al contexto (tipo 1), si y sólo si para las producciones de la forma $\alpha \rightarrow \beta$, se tiene que $l(\alpha) \leq l(\beta)$.

Demostración. Si \mathcal{G} es de tipo 1, las producciones son de la forma: $\alpha A \beta \rightarrow \alpha \gamma \beta (\gamma \neq \varepsilon)$; donde $\alpha, \beta, \gamma \in (\mathbf{N} \cup \mathbf{T})^*$, $A \in \mathbf{N} \cup \mathbf{I}$, y ε es la palabra vacía, entonces:

$$l(\alpha A \beta) = l(\alpha) + l(A) + l(\beta);$$

como $l(A) = 1$ y $l(\gamma) \geq 1$, porque $\gamma \neq \varepsilon$,

$$\begin{aligned} l(\alpha A \beta) &\leq l(\alpha) + l(\gamma) + l(\beta) \\ &\leq l(\alpha \gamma \beta). \end{aligned}$$

□

Teorema 3.5. Si \mathcal{G} es una gramática lineal derecha, entonces existe una gramática lineal derecha \mathcal{G}' , tal que \mathcal{G}' es equivalente a \mathcal{G} y \mathcal{G}' no contiene producciones de la forma $A \rightarrow b\mathbf{I}$, donde \mathbf{I} es el símbolo inicial, ni contiene producciones de la forma $A \rightarrow \varepsilon$, donde ε es la palabra vacía.

Demostración.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

1. Eliminación de las producciones $A \rightarrow \varepsilon$

Sea \mathcal{G} con las siguientes producciones:

$$\begin{aligned} I &\rightarrow bB \\ B &\rightarrow aB \\ B &\rightarrow \varepsilon \end{aligned}$$

Entonces \mathcal{G}' queda así:

$$\begin{aligned} I &\rightarrow bB \\ I &\rightarrow b \\ B &\rightarrow aB \\ B &\rightarrow a \end{aligned}$$

2. Eliminación de las producciones $A \rightarrow bI$

Sea \mathcal{G} con las siguientes producciones:

$$\begin{aligned} I &\rightarrow bA \\ A &\rightarrow aI \\ A &\rightarrow a \end{aligned}$$

Entonces \mathcal{G}' queda así:

$$\begin{aligned} I &\rightarrow bA \\ B &\rightarrow bA \\ A &\rightarrow aB \\ A &\rightarrow a \end{aligned}$$

□

Teorema 3.6. *Para toda gramática lineal derecha (tipo 3) existe una gramática lineal izquierda (tipo 3) equivalente.*

Demostración. El teorema se demuestra con base en la construcción de grafos asociados con las gramáticas. La gramática lineal derecha no debe contener producciones de la forma $A \rightarrow bI$, ni producciones de la forma $A \rightarrow \varepsilon$; el cumplimiento de estas restricciones se puede garantizar por el resultado del teorema 3.5 (pág. 120).

Inicialmente indicamos la construcción de un digrafo $\vec{\mathcal{G}}$ para una gramática lineal derecha \mathcal{G} :

1. Nodos: Símbolos no terminales, más I y ε .

2. Arcos: Existe un arco del nodo v_1 al nodo v_2 , etiquetado con c , si y sólo si, $v_1 \rightarrow cv_2$ es una producción de \mathcal{G} .

En este caso la producción de las palabras es de izquierda a derecha. Para el digrafo $\vec{\mathcal{G}}$, cambiamos I por ε e invertimos los arcos. Así obtenemos un nuevo digrafo $\vec{\mathcal{G}}'$. El digrafo $\vec{\mathcal{G}}'$ corresponde al digrafo de una gramática lineal izquierda \mathcal{G}' equivalente a la gramática lineal derecha \mathcal{G} . La lectura del digrafo $\vec{\mathcal{G}}'$ se realiza bajo las siguientes convenciones:

1. Nodos: Símbolos no terminales, más I y ε .
2. Arcos: Existe un arco del nodo v_1 al nodo v_2 , etiquetado con c , si y sólo si, $v_1 \rightarrow v_2c$ es una producción de \mathcal{G}' .

En este caso la producción de las palabras es de derecha a izquierda. Entonces, a partir del digrafo $\vec{\mathcal{G}}'$ se puede obtener la gramática lineal izquierda \mathcal{G}' equivalente a la gramática lineal derecha \mathcal{G} . \square

Ejemplo 3.32. Sea \mathcal{G} una gramática lineal derecha:

$$\begin{aligned} I &\rightarrow 1B \mid 1 \\ A &\rightarrow 1B \mid 1 \\ B &\rightarrow 0A, \end{aligned}$$

donde $\mathcal{L} = (10)^*1$.

La figura 3.10 representa el digrafo $\vec{\mathcal{G}}$ para \mathcal{G} . Entonces cambiamos I por ε e invertimos los arcos, con lo cual obtenemos un nuevo digrafo $\vec{\mathcal{G}}'$, representado por la figura 3.11.

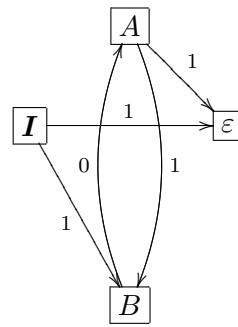
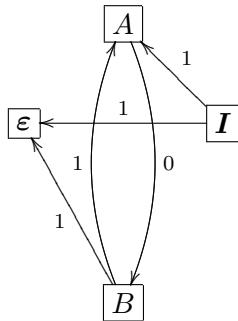


Figura 3.10: Digrafo para la gramática lineal derecha \mathcal{G} .

De acuerdo con la figura 3.11, tenemos que la gramática lineal izquierda \mathcal{G}' está formada por las siguientes producciones:

$$\begin{aligned} I &\rightarrow A1 \mid 1 \\ A &\rightarrow B0 \\ B &\rightarrow 1 \mid A1, \end{aligned}$$

Figura 3.11: Digrafo para la gramática lineal izquierda \mathcal{G}' .

donde $\mathcal{L} = 1(01)^*$.

Entonces \mathcal{G} es un gramática lineal derecha y \mathcal{G}' es la gramática lineal izquierda equivalente a \mathcal{G} , obtenida por el método expuesto en el teorema 3.6 (pág. 121).

3.4. Expresiones Regulares

Las expresiones regulares introducidas por Stephen Kleene para denotar conjuntos regulares (lenguajes regulares) y son de uso frecuente en la descripción de la sintaxis de los lenguajes de programación.

Definición 3.21 (Expresión regular). Presentamos una definición recursiva para las expresiones regulares. Sea Σ un alfabeto, entonces:

1. ϵ (palabra vacía) es un expresión regular de Σ .
2. Si $r \in \Sigma$, entonces r es un expresión regular de Σ .
3. Si r y s son expresiones regulares que denotan los lenguajes $\mathcal{L}(r)$ y $\mathcal{L}(s)$ respectivamente, entonces:
 - a) $r \mid s$ es un expresión regular, que denota la unión de $\mathcal{L}(r)$ y $\mathcal{L}(s)$.
 - b) $(r)(s)$ es un expresión regular, que denota la concatenación de $\mathcal{L}(r)$ y $\mathcal{L}(s)$.
 - c) $(r)^*$ es un expresión regular, que denota la clausura de Kleene de $\mathcal{L}(r)$.

La precedencia de los operadores de mayor a menor es $^*, (), |$. Además todos los operadores son asociativos por la izquierda, por ejemplo $(a) \mid ((b^*)(c)) \equiv a \mid b^*c$.

Ejemplo 3.33. Sea $\Sigma = \{a, b\}$, entonces:

1. Si $r \equiv a \mid b$ entonces $\mathcal{L}(r) = \{a, b\}$.

2. Si $r \equiv (a \mid b)(a \mid b)$ entonces $\mathcal{L}(r) = \{aa, ab, ba, bb\}$.
3. Si $r \equiv a^*$ entonces $\mathcal{L}(r) = \{\varepsilon, a, aa, aaa, \dots\}$.
4. Si $r \equiv (a|b)^*$ entonces $\mathcal{L}(r) = \{\text{todas las cadenas de } a \text{ y } b \text{ incluyendo } \varepsilon\}$.
5. Si $r \equiv a|a^*b$ entonces $\mathcal{L}(r) = \{a, b, ab, aab, aaab, \dots\}$.

Definición 3.22 (Definición regular). Una definición regular tiene el formato:

$$\text{nombre} \rightarrow \text{expresión regular}$$

Ejemplo 3.34. La definición regular para un identificador viene dada por:

$$\begin{aligned} \text{identificador} &\rightarrow \text{letra}(\text{letra} \mid \text{dígito})^* \\ \text{letra} &\rightarrow A \mid \dots \mid Z \mid a \mid \dots \mid z \\ \text{dígito} &\rightarrow 0 \mid \dots \mid 9. \end{aligned}$$

Ejemplo 3.35. La definición regular para un número real sin signo viene dada por:

$$\begin{aligned} \text{dígito} &\rightarrow 0 \mid \dots \mid 9 \\ \text{dígitos} &\rightarrow \text{dígito} \text{ dígitos}^* \\ \text{decimal} &\rightarrow .\text{dígitos} \mid \varepsilon \\ \text{exponente} &\rightarrow (E(+ \mid - \mid \varepsilon) \text{ dígitos}) \mid \varepsilon \\ \text{número} &\rightarrow \text{dígitos decimal exponente}. \end{aligned}$$

Abreviación en la notación:

1. Uno o más casos: +.
2. Cero o más casos: *.
3. Cero o un caso: ?.
4. Clase de caracteres: $[abc] \equiv (a \mid b \mid c \text{ y } [a-z] \equiv (a \mid \dots \mid z)$.

Ejemplo 3.36. La definición regular para un número real sin signo, usando abreviaciones en la notación, viene dada por:

$$\begin{aligned} \text{dígito} &\rightarrow [0-9] \\ \text{dígitos} &\rightarrow \text{dígito}+ \\ \text{decimal} &\rightarrow .\text{dígitos}? \\ \text{exponente} &\rightarrow (E(+ \mid -)? \text{ dígitos})? \\ \text{número} &\rightarrow \text{dígitos decimal exponente}. \end{aligned}$$

3.5. Ejercicios

Ejercicio 3.1. Sea $\Sigma = \{a, b, c, d, e\}$:

1. ¿Cuál es el cardinal de Σ^2 y de Σ^3 ?
2. ¿Cuántas palabras de Σ^* tienen una longitud de al menos cinco?

Ejercicio 3.2. Para $\Sigma = \{w, x, y, z\}$, determine el número de palabras de Σ^* de longitud cinco tal que:

1. Que comiencen por w .
2. Con precisamente dos w .
3. Sin w .
4. Con un número par de w .

Ejercicio 3.3. Si $\alpha \in \Sigma^*$ y $l(\alpha^4) = 36$ (longitud), ¿cuánto es $l(\alpha)$?

Ejercicio 3.4. Sea $\Sigma = \{\beta, x, y, z\}$, donde β denota un espacio en blanco, de modo que $x\beta \neq x$, $\beta\beta \neq \beta$ y $x\beta y \neq xy$, pero $x\varepsilon y = xy$. Calcule lo siguiente:

1. $l(\varepsilon)$.
2. $l(\varepsilon\varepsilon)$.
3. $l(\beta)$.
4. $l(\beta\beta)$.
5. $l(\beta^3)$.
6. $l(x\beta\beta y)$.
7. $l(\beta\varepsilon)$.
8. $l(\varepsilon^{10})$.

Ejercicio 3.5. Para el alfabeto $\Sigma = \{0, 1\}$, sean $A, B, C \in \Sigma^*$ los siguientes lenguajes:

$$A = \{0, 1, 00, 11, 000, 111, 0000, 1111\},$$

$$B = \{\alpha \in \Sigma^* \mid 2 \leq l(\alpha)\},$$

$$C = \{\alpha \in \Sigma^* \mid 2 \geq l(\alpha)\}.$$

Determine los siguientes lenguajes de Σ^* :

1. $A \cup B$.

2. $A - B$.
3. $A \Delta B$.
4. $A \cap B$.
5. $B \cap C$.
6. $B \cup C$.
7. $(A \cap C)^*$.
8. $A^* \cap C^*$.
9. $A^* \cap B^*$.

Ejercicio 3.6. Sean $A = \{10, 11\}$, $B = \{00, 1\}$ lenguajes para el alfabeto $\Sigma = \{0, 1\}$. Determine los siguientes lenguajes:

1. AB .
2. BA .
3. A^2 .
4. B^2 .

Ejercicio 3.7. Demostrar el teorema 3.1 (pág. 102).

Ejercicio 3.8. Considere las siguientes gramáticas:

1. Gramática \mathcal{G}_1 :

$$\begin{aligned} I &\rightarrow \varepsilon \\ I &\rightarrow S \\ S &\rightarrow SS \\ S &\rightarrow c \end{aligned}$$

2. Gramática \mathcal{G}_2 :

$$\begin{aligned} I &\rightarrow \varepsilon \\ I &\rightarrow S \\ S &\rightarrow cSd \\ S &\rightarrow cd \end{aligned}$$

3. Gramática \mathcal{G}_3 :

$$\begin{aligned} I &\rightarrow \varepsilon \\ I &\rightarrow S \\ S &\rightarrow Sd \\ S &\rightarrow cS \\ S &\rightarrow c \\ S &\rightarrow d \end{aligned}$$

4. Gramática \mathcal{G}_4 :

$$\begin{aligned} I &\rightarrow cS \\ S &\rightarrow d \\ S &\rightarrow cS \\ S &\rightarrow Td \\ T &\rightarrow Td \\ T &\rightarrow d \end{aligned}$$

5. Gramática \mathcal{G}_5 :

$$\begin{aligned} I &\rightarrow \varepsilon \\ I &\rightarrow S \\ S &\rightarrow ScS \\ S &\rightarrow c \end{aligned}$$

- a. Describir $\mathcal{L}(\mathcal{G}_i)$ para $i = 1, 2, 3, 4, 5$.
- b. Indicar cualquier inclusión $\mathcal{L}(\mathcal{G}_i)$.
- c. Para cada lenguaje $\mathcal{L}(\mathcal{G}_i)$, dar una derivación de una palabra de longitud 4.

Ejercicio 3.9. Construya una gramática que genere cada uno de los siguientes lenguajes:

1. $\{0^m 1^n \mid m \geq n \geq 0\}$.
2. $\{0^m 1^n \mid m \text{ impar y } n \text{ par o } n \text{ impar y } m \text{ par}\}$.
3. $\{0^k 1^m 0^n \mid n = k + m\}$.
4. $\{wcw \mid w \in \{0, 1\}^*\}$.

Ejercicio 3.10. ¿Es posible construir el árbol de análisis sintáctico para la gramática \mathfrak{G} que genera el lenguaje $\mathcal{L}(\mathfrak{G}) = \{a^n b^n c^n; n \geq 1\}$ presentada en el ejemplo 3.22 (pág. 109)? Por qué?

Ejercicio 3.11. Demuestre que las siguientes gramáticas son ambiguas:

1. Gramática \mathcal{G}_1 :

$$\begin{aligned}\mathbf{I} &\rightarrow A \\ A &\rightarrow A0A \\ A &\rightarrow 1\end{aligned}$$

2. Gramática \mathcal{G}_2 :

$$\begin{aligned}\mathbf{I} &\rightarrow A \\ A &\rightarrow B0 \\ A &\rightarrow A0 \\ B &\rightarrow B0 \\ A &\rightarrow 1 \\ B &\rightarrow 1\end{aligned}$$

3. Gramática \mathcal{G}_3 :

$$\begin{aligned}\mathbf{I} &\rightarrow S \\ S &\rightarrow bA \mid aB \\ A &\rightarrow a \mid aS \mid bAA \\ B &\rightarrow b \mid bS \mid aBB\end{aligned}$$

Ejercicio 3.12. Para la gramática \mathcal{G}_3 del ejercicio 3.11, construir una gramática no ambigua.

Ejercicio 3.13. Obtener una derivación por la izquierda para la palabra $\alpha \equiv abaca$, que sea distinta a la derivación por la derecha siguiente:

$$\mathbf{I} \rightarrow S \rightarrow ScS \rightarrow Sca \rightarrow SbSca \rightarrow Sbaca \rightarrow abaca;$$

para la gramática

$$\begin{aligned}\mathbf{I} &\rightarrow S \\ S &\rightarrow SbS \mid ScS \mid a\end{aligned}$$

Ejercicio 3.14. ¿Es posible obtener una gramática regular que sea ambigua? Si se puede, dar un ejemplo. Si no, justifique su respuesta.

Ejercicio 3.15. Una producción regular por la izquierda es una producción de la forma $A \rightarrow Bw$ donde A y B son no terminales y w es terminal. Una producción regular por la derecha es una producción de la forma $A \rightarrow wB$. Por tanto, las gramáticas regulares por la izquierda y las gramáticas regulares por la derecha contienen solamente producciones regulares por la izquierda y producciones regulares por la derecha, respectivamente. Demuestre que una gramática regular no puede contener ambos tipos de producciones.

Ejercicio 3.16. Determine el tipo de las siguientes gramáticas. Presente todas las caracterizaciones que sean posibles.

1. Gramática \mathcal{G}_1 :

$$\begin{aligned} I &\rightarrow B \\ B &\rightarrow bB \\ B &\rightarrow b \\ B &\rightarrow aA \\ A &\rightarrow aB \\ A &\rightarrow bA \\ A &\rightarrow a \end{aligned}$$

2. Gramática \mathcal{G}_2 :

$$\begin{aligned} I &\rightarrow AB \\ AB &\rightarrow BA \\ A &\rightarrow aA \\ B &\rightarrow Bb \\ A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

3. Gramática \mathcal{G}_3 :

$$\begin{aligned} I &\rightarrow A \\ I &\rightarrow AAB \\ Aa &\rightarrow ABa \\ B &\rightarrow BA \\ A &\rightarrow aa \\ Bb &\rightarrow ABb \\ AB &\rightarrow ABB \\ B &\rightarrow b \end{aligned}$$

4. Gramática \mathcal{G}_4 :

$$\begin{aligned} I &\rightarrow BAB \\ I &\rightarrow ABA \\ A &\rightarrow AB \\ B &\rightarrow BA \\ A &\rightarrow aA \\ A &\rightarrow ab \\ B &\rightarrow b \end{aligned}$$

5. Gramática \mathcal{G}_5 :

$$\begin{aligned} I &\rightarrow C \\ C &\rightarrow AAC \\ AA &\rightarrow B \\ B &\rightarrow bB \\ A &\rightarrow a \end{aligned}$$

Ejercicio 3.17. Construya la gramática que genere las palabras no nulas que tengan la propiedad dada:

1. Palabras definidas sobre $\{a, b\}$ que empiecen por a .
2. Palabras definidas sobre $\{a, b\}$ que terminen en ba .
3. Palabras definidas sobre $\{a, b\}$ que contengan ba .
4. Números con punto flotante (como $0,294, 89,0, 67,284$).

5. Números exponenciales (que incluyen a los números con punto flotante y a otros tales como $6,9E3, 8E12, 9, 6E - 4, 9E - 10$).

Ejercicio 3.18. Una palabra α es un palíndromo si es su propio reverso, esto es $\alpha = \alpha^{-1}$. Un lenguaje \mathcal{L} es palíndromo si cada una de sus palabras es un palíndromo. Sean las gramáticas:

1. Gramática \mathcal{G}_1 :

$$\begin{aligned} I &\rightarrow S \\ S &\rightarrow aSa \\ S &\rightarrow aSb \\ S &\rightarrow bSb \\ S &\rightarrow bSa \\ S &\rightarrow aa \\ S &\rightarrow bb \end{aligned}$$

2. Gramática \mathcal{G}_2 :

$$\begin{aligned} I &\rightarrow S \\ S &\rightarrow aS \\ S &\rightarrow Sa \\ S &\rightarrow bS \\ S &\rightarrow Sb \\ S &\rightarrow a \\ S &\rightarrow b \end{aligned}$$

- a. Describir informalmente $\mathcal{L}(\mathcal{G}_1)$ y $\mathcal{L}(\mathcal{G}_2)$.
- b. ¿Es alguno de ellos un lenguaje palíndromo?

Ejercicio 3.19. Obtener una gramática regular para los siguientes lenguajes:

1. $a^*b \mid a$.
2. $a^*b \mid b^*a$.
3. $(a^*b \mid b^*a)^*$.

Ejercicio 3.20. La gramática dada por

$$\begin{aligned} I &\rightarrow S \\ S &\rightarrow bA \mid aB \mid \epsilon \\ A &\rightarrow abaS \\ B &\rightarrow babS \end{aligned}$$

genera un lenguaje regular. Obtener una expresión regular para este lenguaje.

Ejercicio 3.21. Sean r , s y t expresiones regulares sobre el mismo alfabeto Σ . Demuestre:

1. $r \mid s = s \mid r$.
2. $r \mid \emptyset = r = \emptyset \mid r$.
3. $r \mid r = r$.
4. $r\emptyset = \emptyset r = r$.
5. $(rs)t = r(st)$.
6. $r^* = r^{**} = r^*r^*$.

3.6. Notas Bibliográficas

La definición y operaciones sobre las palabras y/o lenguajes son presentadas por [1, 6, 12, 14, 20]. Las definiciones de semigrupo y monoide fueron tomadas de [24]. La relación entre monoide y lenguaje es presentada por [22]. Los sistemas formales son presentados por [12, 23]. El ejemplo 3.18 (pág. 106) de un sistema combinatorio fue tomado de [12]. Varios textos presentan los elementos relacionados con las gramáticas (definición, derivaciones por la izquierda y por derecha, recursividad, ambigüedad); algunos de ellos son [1, 6, 19, 20]. La clasificación de las gramáticas es presentada por [6]. La notación BNF es presentada por [1, 19]. Las expresiones regulares son presentadas por [1, 6, 20].

Capítulo 4

Autómatas de Estado Finito

Siguiendo el contexto de la teoría general de sistemas representamos un sistema \mathcal{S} por la figura 4.1.



Figura 4.1: Representación de un sistema (1).

1. Elementos del sistema

E-1 $x(t)$: Vector de entradas al sistema (en un tiempo discreto), es decir,

$$x(t) = \begin{pmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{pmatrix}.$$

E-2 $s(t)$: Reacciones exógenas del sistema (en un tiempo discreto), es decir,

$$s(t) = \begin{pmatrix} s_1(t) \\ s_2(t) \\ \vdots \\ s_m(t) \end{pmatrix}.$$

E-3 Definimos Σ como el espacio de entradas al sistema, es decir,

$$\Sigma = \{x_i(t), \quad i = 1, \dots, n\}.$$

E-4 Definimos Γ como el espacio de configuraciones internas (reacciones endógenas frente a $x(t)$).

E-5 Definimos Δ como el espacio de reacciones exógenas del sistema, es decir,

$$\Delta = \{s_i(t), \quad i = 1, \dots, m\}.$$

2. Comportamiento del sistema

C-1 La función endógena δ representa el comportamiento interno del sistema y está definida por:

$$\delta: \Gamma \times \Sigma \rightarrow \Gamma.$$

C-2 La función exógena λ representa la salida del sistema y está definida por:

$$\lambda: \Gamma \times \Sigma \rightarrow \Delta.$$

Entonces nuestro sistema \mathcal{S} queda representado por la figura 4.2.



Figura 4.2: Representación de un sistema (2).

4.1. Máquinas de Estado Finito

Definición 4.1 (Máquina de estado finito). Una máquina de estado finito está definida por la estructura matemática $\mathcal{MFE} = < \Sigma, \Delta, \Gamma, \delta, \lambda, k_0 >$, donde:

Σ : Alfabeto de entrada (finito y diferente de vacío).

Δ : Alfabeto de salida (finito y diferente de vacío).

Γ : Conjunto de estados (finito y diferente de vacío).

δ : Función de estado siguiente, definida por: $\delta: \Gamma \times \Sigma \rightarrow \Gamma$.

λ : Función de salida, definida por: $\lambda: \Gamma \times \Sigma \rightarrow \Delta$.

k_0 : Estado inicial ($k_0 \in \Gamma$).

Para concretizar las diferentes representaciones de una máquina de estado finito, utilizaremos un ejemplo clásico de una máquina de estado finito, el cual corresponde a un sumador binario.

Ejemplo 4.1. Construyamos una máquina de estado finito que modele el comportamiento de un sumador binario con dos entradas, esquematizado por la figura 4.3. Sea $\mathcal{MFE} = < \Sigma, \Delta, \Gamma, \delta, \lambda, k_0 >$, donde: $\Sigma = \{(00), (01)(10), (11)\}$, $\Delta = \{0, 1\}$, $\Gamma = \{N, A\}$ (N : No acarreo; A : Acarreo) y $k_0 = N$.

Las funciones δ y λ dependen de la representación que se escoja para la máquina de estado finito. Estas representaciones serán descritas a continuación.



Figura 4.3: Sumador binario.

Definición 4.2 (Diagrama de transición). Una máquina de estado finito se puede representar por medio de una digrafo, llamado digrafo de transición, siguiendo las siguientes convenciones:

1. Nodos: $k_i \in \Gamma$.
2. Arcos: Existe una arco del nodo q_i al nodo q_j , etiquetado con e/s , si y sólo si, $\delta(q_i, e) = q_j$ y $\lambda(q_i, e) = s$.
3. Se coloca un arco no etiquetado para indicar el estado inicial k_0 .

Ejemplo 4.2. La figura 4.4 representa el diagrama de transición para el sumador binario.

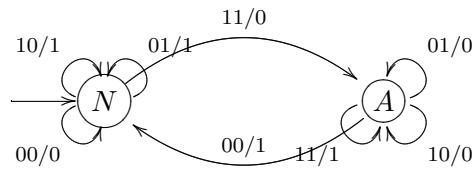


Figura 4.4: Diagrama de transición para un sumador binario (1).

Observación. Si existen varios arcos del nodo q_i al nodo q_j , para simplificar el diagrama, éstos se pueden representar mediante uno sólo arco con varias etiquetas (tal como lo indica la figura 4.5).

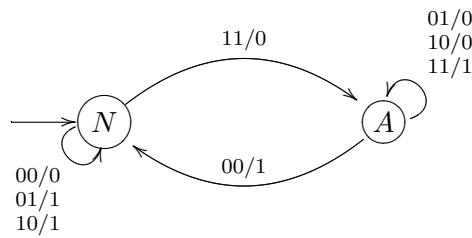


Figura 4.5: Diagrama de transición para un sumador binario (2).

Definición 4.3 (Tabla de transición). Una máquina de estado finito se puede representar por medio de una tabla T de transición, siguiendo las siguientes convenciones:

1. Filas: Estados.
2. Columnas: Símbolos del alfabeto de entrada Σ .
3. $T_{i,j} = (q', s')$, si y sólo si, $(\delta(q_i, e_j) = q' \wedge \lambda(q_i, e_j) = s')$.
4. Se denota el estado inicial k_o , subrayando éste en la fila correspondiente a los estados.

Ejemplo 4.3. La tabla 4.1 representa la tabla de transición para el sumador binario.

Γ/Σ	00	01	10	11
<u>N</u>	$N, 0$	$N, 1$	$N, 1$	$A, 0$
<u>A</u>	$N, 1$	$A, 0$	$A, 0$	$A, 1$

Cuadro 4.1: Tabla de transición para un sumador binario.

Definición 4.4 (Representación explícita). Finalmente, una máquina de estado finito se puede representar listando explícitamente todos sus componentes. Esta forma de representación recibe el nombre de representación explícita.

Ejemplo 4.4. Para el sumador binario tenemos la siguiente representación explícita: $\Sigma = \{(00), (01)(10), (11)\}$, $\Delta = \{0, 1\}$, $\Gamma = \{N, A\}$ (N : No acarreo; A : Acarreo), $k_0 = N$ y

$\delta(N, 00) = N$	$\lambda(N, 00) = 0$
$\delta(N, 01) = N$	$\lambda(N, 01) = 1$
$\delta(N, 10) = N$	$\lambda(N, 10) = 1$
$\delta(N, 11) = A$	$\lambda(N, 11) = 0$
$\delta(A, 00) = N$	$\lambda(A, 00) = 1$
$\delta(A, 01) = A$	$\lambda(A, 01) = 0$
$\delta(A, 10) = A$	$\lambda(A, 10) = 0$
$\delta(A, 11) = A$	$\lambda(A, 11) = 1$

Definición 4.5 (Máquina de Mealy). Una máquina de Mealy está definida por la estructura matemática $\mathcal{MMA}LY = < \Sigma, \Delta, \Gamma, \delta, \lambda >$, donde:

Σ : Alfabeto de entrada (finito y diferente de vacío).

Δ : Alfabeto de salida (finito y diferente de vacío).

Γ : Conjunto de estados (finito y diferente de vacío).

δ : Función de estado siguiente, definida por: $\delta: \Gamma \times \Sigma \rightarrow \Gamma$.

λ : Función de salida, definida por: $\lambda: \Gamma \times \Sigma \rightarrow \Delta$.

De acuerdo con la definición anterior, una máquina de Mealy es una máquina de estado finito. En ésta no hemos incluido el estado inicial k_0 , para simplificar el desarrollo formal.

¿Analizamos qué ocurre si desearamos adicionar la palabra vacía (representada por ε) a nuestro alfabeto de entrada? La idea es utilizar la palabra vacía como el elemento neutro de un monoide, lo cual será desarrollado en la próxima sección.

Veamos primero qué ocurre con la función de estado siguiente δ . Es claro que es necesario ampliar el dominio de la función a: $\delta: \Gamma \times \Sigma \cup \{\varepsilon\} \rightarrow \Gamma$. Además, necesitamos utilizar la convención de que $\delta(k, \varepsilon) = k$, para todo $k \in \Gamma$.

Veamos ahora qué sucede con la función de salida λ . Es claro que es necesario ampliar el dominio de la función a: $\lambda: \Gamma \times \Sigma \cup \{\varepsilon\} \rightarrow \Delta$.

¿Pero qué sucede con $\lambda(k, \varepsilon)$? Puede ocurrir que por lo menos para algún $k \in \Gamma$ suceda que $\lambda(k, \varepsilon) = e_1 \wedge \lambda(k, \varepsilon) = e_2 \wedge \dots \lambda(k, \varepsilon) = e_n$. Esto es factible debido a que pueden existir diferentes (finitas) salidas asociadas con la llegada al estado k . Lo anterior nos imposibilita definir la función λ para el nuevo alfabeto de entrada $\Sigma \cup \{\varepsilon\}$.

Sólo es posible definir la función $\lambda(k, \varepsilon)$ si la máquina de Mealy satisface el siguiente enunciado:

$$\forall k, k', k'', e', e'' \quad (((k, k', k'' \in \Gamma) \wedge (e', e'' \in \Sigma) \wedge (k = \delta(k', e') = \delta(k'', e''))) \Rightarrow (\lambda(k', e') = \lambda(k'', e'))),$$

es decir, la salida sólo depende del estado que se alcanza. La satisfacción de este enunciado genera un clase de máquina abstracta llamada máquina de Moore.

Definición 4.6 (Máquina de Moore). Una máquina de Moore está definida por la estructura matemática $\mathcal{MMORE} = < \Sigma, \Delta, \Gamma, \delta, \lambda >$, donde:

Σ : Alfabeto de entrada (finito y diferente de vacío).

Δ : Alfabeto de salida (finito y diferente de vacío).

Γ : Conjunto de estados (finito y diferente de vacío).

δ : Función de estado siguiente, definida por: $\delta: \Gamma \times \Sigma \rightarrow \Gamma$.

λ : Función de salida, definida por: $\lambda: \Gamma \times \Sigma \rightarrow \Delta$.

Además, una máquina de Moore debe satisfacer el siguiente enunciado:

$$\forall k, k', k'', e', e'' \quad (((k, k', k'' \in \Gamma) \wedge (e', e'' \in \Sigma) \wedge (k = \delta(k', e') = \delta(k'', e''))) \Rightarrow (\lambda(k', e') = \lambda(k'', e'))).$$

Es importante, entonces, que tengamos en cuenta que en una máquina de Mealy las salidas están asociadas con las transiciones; en cambio, en una máquina de Moore las salidas están asociadas con los estados, esto es, todas las transiciones que están asociadas con un mismo estado tienen la misma salida. Las máquinas de Moore, desde la perspectiva de las máquinas de estado, serán los autómatas de estado finito.

El teorema que presentamos a continuación establece la equivalencia entre las máquinas de Mealy y las máquinas de Moore.

Teorema 4.1.

1. Toda máquina de Moore es equivalente a una máquina de Mealy.

2. Toda máquina de Mealy es equivalente a una máquina de Moore.

Demostración.

1. Toda máquina de Moore es equivalente a una máquina de Mealy.

De acuerdo con su definición, una máquina de Moore es una máquina de Mealy que satisface el enunciado especificado en su definición. Es decir, una máquina de Moore es un caso particular de una máquina de Mealy.

2. Toda máquina de Mealy es equivalente a una máquina de Moore.

Sea $\mathcal{ME} = < \Sigma, \Delta, \Gamma, \delta, \lambda >$ una máquina de Mealy. A partir de \mathcal{ME} construimos una máquina de Moore $\mathcal{MO} = < \Sigma', \Delta', \Gamma', \delta', \lambda' >$, definida por:

- a) $\Sigma' = \Sigma$.
- b) $\Delta' = \Delta$.
- c) Γ' se obtiene dividiendo cada $k \in \Gamma$ en tantos estados k^s como salidas s se puedan asociar con k es decir:

$$k^s \in \Gamma' \iff \exists k' \exists e \exists s ((k \in \Gamma) \wedge (e \in \Sigma) \wedge (s \in \Delta) \wedge (\delta(k', e) = k) \wedge (\lambda(k', e) = s)).$$

- d) $\delta'(k^s, e) = \delta(k, e)^{\lambda(k, e)}$.
- e) $\lambda'(k^s, e) = \lambda(k, e)$.

□

Ejemplo 4.5. Construir una máquina de Moore equivalente a la máquina de Mealy correspondiente al sumador binario.

La máquina de Mealy, para el sumador binario, está representada por la figura 4.6.

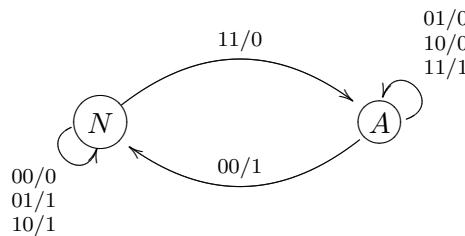


Figura 4.6: Máquina de Mealy para un sumador binario.

De acuerdo con el teorema 4.1 (pág. 137), vamos a construir una máquina de Moore, $\mathcal{MMORE} = < \Sigma', \Delta', \Gamma', \delta', \lambda' >$, donde: $\Sigma' = \{(00), (01), (10), (11)\}$, $\Delta' = \{0, 1\}$, $\Gamma' = \{00/0, 01/1, 10/1, 11/0, 01/0, 10/0, 11/1\}$. El

estado N debe ser dividido en dos estados N^0 y N^1 ; y el estado A también debe ser dividido en dos estados A^0 y A^1 , con lo que $\Gamma' = \{N^0, N^1, A^0, A^1\}$,

δ' está definida por:

$$\begin{aligned}\delta'(N^0, 00) &= \delta'(N^1, 00) = \delta(N, 00)^{\lambda(N, 00)} = N^0 \\ \delta'(N^0, 01) &= \delta'(N^1, 01) = \delta(N, 01)^{\lambda(N, 01)} = N^1 \\ \delta'(N^0, 10) &= \delta'(N^1, 10) = \delta(N, 10)^{\lambda(N, 10)} = N^1 \\ \delta'(N^0, 11) &= \delta'(N^1, 11) = \delta(N, 11)^{\lambda(N, 11)} = A^0 \\ \delta'(A^0, 00) &= \delta'(A^1, 00) = \delta(A, 00)^{\lambda(A, 00)} = N^1 \\ \delta'(A^0, 01) &= \delta'(A^1, 01) = \delta(A, 01)^{\lambda(A, 01)} = A^0 \\ \delta'(A^0, 10) &= \delta'(A^1, 10) = \delta(A, 10)^{\lambda(A, 10)} = A^0 \\ \delta'(A^0, 11) &= \delta'(A^1, 11) = \delta(A, 11)^{\lambda(A, 11)} = A^1\end{aligned}$$

y λ' está definida por:

$$\begin{aligned}\lambda'(N^0, 00) &= \lambda'(N^1, 00) = \lambda(N, 00) = 0 \\ \lambda'(N^0, 01) &= \lambda'(N^1, 01) = \lambda(N, 01) = 1 \\ \lambda'(N^0, 10) &= \lambda'(N^1, 10) = \lambda(N, 10) = 1 \\ \lambda'(N^0, 11) &= \lambda'(N^1, 11) = \lambda(N, 11) = 0 \\ \lambda'(A^0, 00) &= \lambda'(A^1, 00) = \lambda(A, 00) = 1 \\ \lambda'(A^0, 01) &= \lambda'(A^1, 01) = \lambda(A, 01) = 0 \\ \lambda'(A^0, 10) &= \lambda'(A^1, 10) = \lambda(A, 10) = 0 \\ \lambda'(A^0, 11) &= \lambda'(A^1, 11) = \lambda(A, 11) = 1\end{aligned}$$

La figura 4.7 representa la máquina de Moore para el sumador binario. Allí, cada estado (nodo) se marca con su nombre y con la salida asociada con él, por medio de una etiqueta de la forma $NombreEstadoSalidaAsociada$.

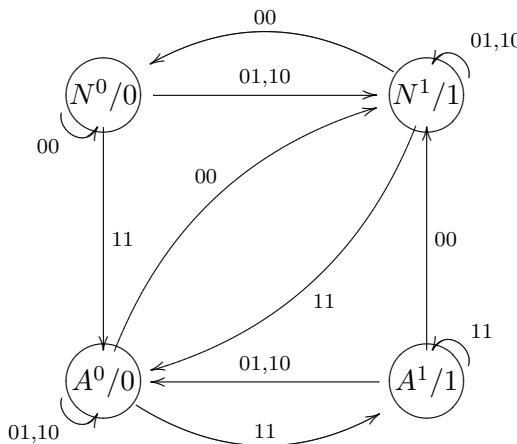


Figura 4.7: Máquina de Moore para un sumador binario.

4.2. Autómatas de Estado Finito

En esta sección y en las próximas nos interesaremos en las relaciones existentes entre autómatas, gramáticas y lenguajes. Desde este punto de vista, señalemos de una vez que los autómatas desempeñan los siguientes papeles: como aceptadores-reconocedores (figura 4.8) o como generadores-traductores (figura 4.9).

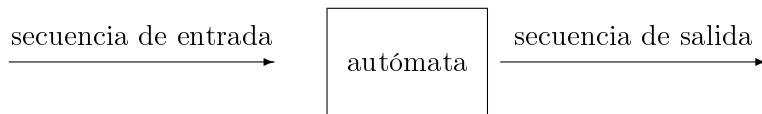


Figura 4.8: Autómata como generador.

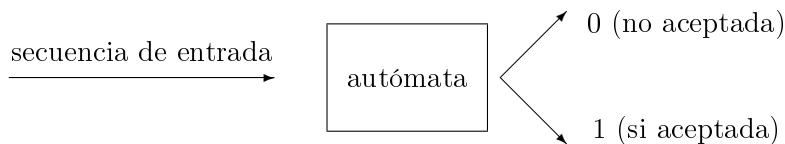


Figura 4.9: Autómata como reconocedor.

En este contexto nos interesa esencialmente el problema del reconocimiento. Este problema está dividido en los siguientes problemas:

P-I Problema de síntesis: dado un lenguaje $\mathcal{L}(\Sigma)$; ¿qué autómata lo reconoce?

P-II Problema de análisis: dado un autómata \mathcal{A} ; ¿qué lenguaje lo reconoce?

Nuestra presentación estará entonces dirigida a ofrecer algunos elementos para la solución al problema del reconocimiento (tanto el problema de síntesis, como el problema de análisis) para el caso de los autómatas de estado finito y los lenguajes regulares (tipo III).

Recordemos que en una máquina de Moore los estados están unívocamente asociados con las salidas. Si operamos sobre una máquina de Moore con únicamente dos salidas, podemos pensar sólo en dos tipos de estados: de aceptación y de no aceptación. La noción de un autómata de estado finito nos permite concretizar esta posibilidad.

Definición 4.7 (Autómata de estado finito). Un autómata de estado finito está definido por la estructura matemática $\mathcal{AF} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$, donde:

Σ : Alfabeto de entrada (finito y diferente de vacío).

Γ : Conjunto de estados (finito y diferente de vacío).

Δ : Conjunto de estados de aceptación ($\Delta \subset \Gamma$).

δ : Función de estado siguiente, definida por: $\delta: \Gamma \times \Sigma \rightarrow \Gamma$.

k_0 : Estado inicial ($k_0 \in \Gamma$).

De manera similar a las máquinas de estado finito, los autómatas de estado finito tienen tres formas de representación.

Definición 4.8 (Diagrama de transición). Un autómata de estado finito se puede representar por medio de una digrafo, llamado digrafo de transición, de acuerdo con las siguientes convenciones:

1. Nodos simples: $k_i \in \Gamma - \Delta$, o estados de no aceptación.
2. Nodos dobles: $k_i \in \Delta$, o estados de aceptación.
3. Arcos: Existe una arco del nodo k_i al nodo k_j , etiquetado con e , si y sólo si, $\delta(k_i, e) = k_j$.
4. Se coloca un arco no etiquetado para indicar el estado inicial k_0 .

Ejemplo 4.6. El diagrama de transición de la figura 4.10 representa un autómata de estado finito.

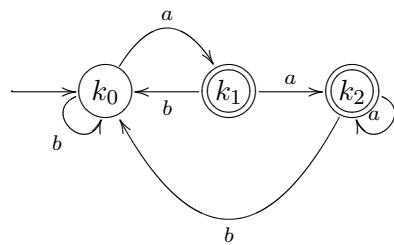


Figura 4.10: Diagrama de transición para un autómata de estado finito.

Definición 4.9 (Tabla de transición). Un autómata de estado finito se puede representar por medio de una tabla T de transición, de acuerdo con las siguientes convenciones:

1. Filas: Estados.
2. Para los estados de aceptación se antepone un asterisco en la fila correspondiente.
3. Columnas: símbolos del alfabeto de entrada Σ .
4. $T_{i,j} = k'$ si $\delta(k_i, e_j) = k'$.
5. Se denota el estado inicial k_0 , subrayando éste en la fila correspondiente a los estados.

Ejemplo 4.7. El autómata de estado finito representado por la figura 4.10 tiene su tabla de transición representada por la tabla 4.2.

Γ/Σ	a	b
k_0	k_1	k_0
$*k_1$	k_2	k_0
$*k_2$	k_2	k_0

Cuadro 4.2: Tabla de transición para un autómata de estado finito.

Definición 4.10 (Representación explícita). Similarmente a las máquinas de estado finito, un autómata de estado finito se puede representar listando explícitamente todos sus componentes. Esta forma de representación recibe el nombre de representación explícita del autómata.

Ejemplo 4.8. El autómata de estado finito representado por la figura 4.10, puede representarse explícitamente como sigue, $\mathcal{A} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$ donde: $\Sigma = \{a, b\}$, $\Gamma = \{k_0, k_1, k_2\}$, $\Delta = \{k_1, k_2\}$, k_0 : Estado inicial y

$$\begin{array}{ll} \delta(k_0, a) = k_1 & \delta(k_0, b) = k_0 \\ \delta(k_1, a) = k_2 & \delta(k_1, b) = k_0 \\ \delta(k_2, a) = k_2 & \delta(k_2, b) = k_0. \end{array}$$

4.3. Reconocedor Finito

Un reconocedor finito de un lenguaje \mathcal{L} , es un autómata de estado finito que sólo acepta las palabras de dicho lenguaje. Esto es, inicializando el autómata en un cierto estado e introduciendo una palabra de entrada perteneciente a \mathcal{L} , finaliza en un estado de aceptación; y al introducir una palabra no perteneciente a \mathcal{L} finaliza en un estado de no aceptación.

Definición 4.11 (Reconocedor finito). Un reconocedor finito es un autómata de estado finito $\mathcal{RF} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$.

Para pensar en el reconocimiento de palabras por parte del reconocedor finito, es necesario expandir la función δ para permitir el procesamiento de las palabras.

Definición 4.12 (Expansión de la función δ). Sea $\mathcal{RF} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$ un reconocedor finito. Definimos una nueva función δ^* denominada la expansión de la función δ , como sigue:

Sea $\alpha \equiv a_1a_2\dots a_n \in \Sigma^*$, entonces, la función $\delta^*: \Gamma \times \Sigma^* \rightarrow \Gamma$ está definida por: $\delta(k, \alpha) = \delta(\dots(\delta(\delta(k, a_1), a_2), \dots), a_n)$.

Definición 4.13 (Palabra aceptada por un reconocedor finito). Sea $\mathcal{RF} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$ un reconocedor finito y sea $\alpha \equiv a_1a_2\dots a_n \in \Sigma^*$. Decimos que \mathcal{RF} reconoce la palabra α , si y sólo si, existe una secuencia de estados k_0, k_1, \dots, k_n tales que:

1. k_0 es el estado inicial.

$$2. \delta(k_{i-1}, a_i) = k_i, \text{ para } 0 < i \leq n.$$

$$3. k_n \in \Delta.$$

Es decir, \mathcal{RF} reconoce la palabra α si y sólo si $\delta^*(k_0, \alpha) \in \Delta$.

Definición 4.14 (Lenguaje aceptado por un reconocedor finito). Sea \mathcal{RF} un reconocedor finito $\mathcal{RF} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$. El lenguaje aceptado por \mathcal{RF} , denotado por, $\mathcal{L}(\mathcal{RF})$ está definido por: $\mathcal{L}(\mathcal{RF}) = \{\alpha \in \Sigma^* \mid \delta^*(k_0, \alpha) \in \Delta\}$.

Ejemplo 4.9. El reconocedor finito \mathcal{RF} representado por la figura 4.11 reconoce el lenguaje $\mathcal{L}(\mathcal{RF}) = \{ab, aab, abb, aaa \dots b, abbb \dots b, \dots\} = \{a^n b^m; n, m \geq 1\}$.

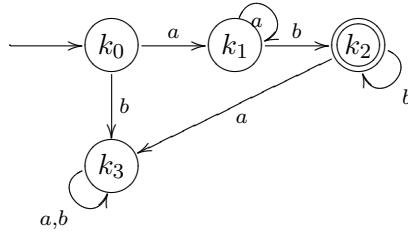


Figura 4.11: Reconocedor finito para $\mathcal{L} = \{a^n b^m; n, m \geq 1\}$.

Ejemplo 4.10. El reconocedor finito representado por la figura 4.12 reconoce el lenguaje $\mathcal{L}(\mathcal{RF}) = \{1(01)^n; n \geq 0\}$. Para $\alpha \equiv 1011$, tenemos que:

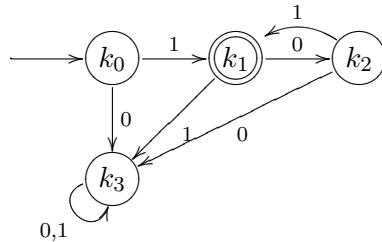
$$\begin{aligned} \delta^*(k_0, \alpha) &= \delta(\delta(\delta(\delta(k_0, 1), 0), 1), 1) \\ &= \delta(\delta(\delta(k_1, 0), 1), 1) \\ &= \delta(\delta(k_2, 1), 1) \\ &= \delta(k_1, 1) \\ &= k_3 \notin \Delta, \end{aligned}$$

entonces $\alpha \notin \mathcal{L}(\mathcal{RF})$.

Para $\beta \equiv 101$, tenemos que:

$$\begin{aligned} \delta^*(k_0, \alpha) &= \delta(\delta(\delta(k_0, 1), 0), 1) \\ &= \delta(\delta(k_1, 0), 1) \\ &= \delta(k_2, 1)) \\ &= k_1 \in \Delta, \end{aligned}$$

entonces $\beta \in \mathcal{L}(\mathcal{RF})$.

Figura 4.12: Reconocedor finito para $\mathcal{L} = \{1(01)^n; n \geq 0\}$.

Hagamos una pausa en nuestro trabajo y formulemos la siguiente pregunta. Dado un lenguaje cualquiera, digamos $\mathcal{L} \subseteq \Sigma^*$, ¿siempre es posible encontrar o construir un reconocedor finito para \mathcal{L} ?; ¿qué significa que tal pregunta hallase una respuesta afirmativa?

Dejamos al lector la segunda pregunta. La respuesta a la primera pregunta (en consonancia con la segunda) no puede ser otra que negativa. Para lograr una satisfacción que corrobore nuestra negación, sabemos que es suficiente hallar un contraejemplo. Esto es, hallar un lenguaje \mathcal{L} para el cual se pueda probar la inexistencia de un reconocedor finito \mathcal{RF} , tal que \mathcal{RF} sea un reconocedor para \mathcal{L} .

Ejemplo 4.11. Sea $\Sigma = \{0, 1\}$ un alfabeto y consideremos el lenguaje $\mathcal{L}(\Sigma) = \{1^{n^2} \mid n \geq 1\}$. Para probar que tal lenguaje no puede ser reconocido por un reconocedor finito, razonemos por reducción al absurdo.

1. Supongamos que existe un reconocedor finito \mathcal{RF} para \mathcal{L} , esto es, $\mathcal{L}(\mathcal{RF}) = \mathcal{L}$ y $\overline{\Gamma} = p$ (p es el cardinal del conjunto de estados).
2. Sea $k \in \mathbb{N}$. Podemos garantizar que $\delta^*(q_0, 1^{k^2}) \in \Delta$.
3. Sean, $\alpha_0 = 1^0, \alpha_1 = 1^1, \dots, \alpha_p = 1^p$; $p + 1$ palabras de Σ^* , donde $\overline{\Gamma} = p$
4. La sucesión de estados $\delta^*(q_0, \alpha_0), \delta^*(q_0, \alpha_1), \dots, \delta^*(q_0, \alpha_p)$ debe necesariamente tener alguna repetición (por 4).
5. Para algún i, j ; tales que $0 < i, j < p$ y $i \neq j$ se tiene que $\delta^*(q_0, 1^i) = \delta^*(q_0, 1^j)$. Supongamos que $j > i$, entonces $0 < j - i < p$.
6. $\delta^*(q_0, 1^{k^2}) \in \Delta \implies \delta^*(q_0, 1^{k^2+(j-i)}) \in \Delta$. Ya que:

$$\begin{aligned}
\delta^*(q_0, 1^{k^2}) &= \delta^*(q_0, 1^i 1^{k^2-i}) \\
&= \delta^*(\delta^*(q_0, 1^i), 1^{k^2-i}) \\
&= \delta^*(\delta^*(q_0, 1^j), 1^{k^2-i}) \\
&= \delta^*(q_0, 1^j 1^{k^2-i}) \\
&= \delta^*(q_0, 1^{k^2+(j-i)}).
\end{aligned}$$

7. Sabemos que, para p existe k tal que $(k+1)^2 - k^2 > p$.
8. Luego, $(k+1)^2 - k^2 > j - i$ (por 6 y 7).
9. Luego, $k^2 < k^2 + j - i < (k+1)^2$. Entonces $k^2 + j - i$ no es cuadrado perfecto.
10. $1^{k^2+(j-i)} \notin \mathcal{L}(\Sigma)$ y $\delta^*(q_0, 1^{k^2+(j-i)}) \in \Delta$.
11. Luego, $\mathcal{L}(\mathcal{RF}) \neq \mathcal{L}(\Sigma)$. Contradicción con (1).
12. Luego, no existe un reconocedor finito para $\mathcal{L}(\Sigma)$.

4.4. Algunas Clases de Autómatas

4.4.1. Autómatas de estado finito deterministas

Presentamos en esta sección dos distinciones entre los autómatas de estado finito.

Definición 4.15 (AFD). Sea $\mathcal{AFD} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$ un autómata de estado finito. El autómata \mathcal{AFD} es un autómata de estado finito determinista (AFD) si la función de estado siguiente ($\delta : \Gamma \times \Sigma \rightarrow \Gamma$) determina un y sólo un estado siguiente.

Ejemplo 4.12. Observemos la figura 4.13. En este caso $\delta(k_0, a) = k_0 \wedge \delta(k_0, a) = k_1$, es decir, el comportamiento del estado k_0 para la entrada a , es no determinista. Además, de acuerdo con nuestra definición formal de un autómata de estado finito, δ debe ser una función y por ende no puede tener este comportamiento. Adicionalmente tenemos que $\delta(k_1, a)$ y $\delta(k_2, b)$ no están definidas.

Para formalizar el tipo de situaciones presentadas por el ejemplo anterior, es necesario modificar la definición de la función δ , de manera que se obtenga una máquina abstracta, la cual llamaremos autómata de estado finito no determinista (AFN).

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

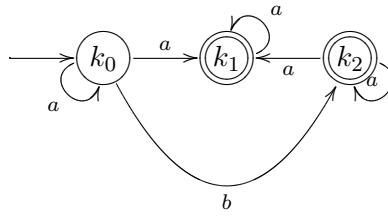


Figura 4.13: Autómata de estado finito no determinista.

4.4.2. Autómatas de estado finito no deterministas

Definición 4.16 (AFN). Un autómata de estado finito no determinista (AFN) está definido por la estructura matemática $\mathcal{AFN} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$, donde:

Σ : Alfabeto de entrada (finito y diferente de vacío).

Γ : Conjunto de estados (finito y diferente de vacío).

Δ : Conjunto de estados de aceptación ($\Delta \subset \Gamma$).

k_0 : Estado inicial ($k_0 \in \Gamma$).

Es decir; Σ, Γ, Δ y k_0 tienen el mismo significado que en un AFD. Esto es, la diferencia entre un AFD y un AFN está determinada por la función δ .

δ : Función de estado siguiente, definida por: $\delta : \Gamma \times \Sigma \rightarrow P(\Gamma)$ (donde $P(\Gamma)$ denota las partes de Γ).

Ejemplo 4.13. De acuerdo con la definición anterior, la figura 4.13 corresponde a un autómata finito no determinista. Tal autómata está dado por $\mathcal{AFN} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$, donde: $\Sigma = \{a, b\}$, $\Gamma = \{k_0, k_1, k_2\}$, $\Delta = \{k_1, k_2\}$, k_0 : Estado inicial y

$$\delta(k_0, a) = \{k_0, k_1\}$$

$$\delta(k_0, b) = \{k_2\}$$

$$\delta(k_1, a) = \{k_1\}$$

$$\delta(k_1, b) = \emptyset$$

$$\delta(k_2, a) = \{k_1, k_2\}$$

$$\delta(k_2, b) = \emptyset.$$

El siguiente teorema establece la equivalencia entre un AFN y un AFD.

Teorema 4.2. Para todo \mathcal{AFN} existe un \mathcal{AFD} equivalente.

Demostración. Sea $\mathcal{AFN} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$ un autómata de estado finito no determinista. A partir de \mathcal{AFN} es posible construir un autómata de estado finito determinista $\mathcal{AFD} = < \Sigma', \Gamma', \Delta', \delta', k'_0 >$, definido por:

$$1. \Sigma' = \Sigma.$$

$$2. k'_0 = \{k_0\}.$$

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

3. $\Gamma' = P(\Gamma)$. Esto significa que hay tantos estados como subconjuntos tenga el conjunto Γ , es decir, $\overline{\overline{\Gamma}} = 2^n$, donde $n = |\Gamma|$. Observe que Γ' continúa siendo un conjunto finito.

4. δ' es una función definida por:

$\delta : \Gamma' \times \Sigma \rightarrow \Gamma'$, es decir, $\delta : P(\Gamma) \times \Sigma \rightarrow P(\Gamma)$, donde:

$$\delta'(X, e) = \begin{cases} \emptyset & \text{sii } X = \emptyset, \\ \bigcup_{k \in X} \{\delta(k, e)\} & \text{sii } X \neq \emptyset. \end{cases}$$

5. $\Delta' = \{X \in P(\Gamma) \mid X \text{ contiene un estado de aceptación perteneciente a } \Delta\}$.

□

Ejemplo 4.14. Construir para el AFN representado por la figura 4.14 un AFD equivalente.

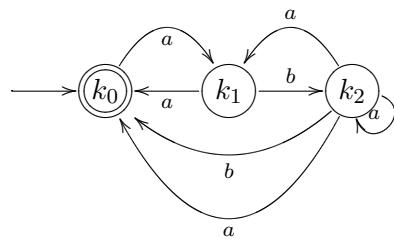


Figura 4.14: Ejemplo AFN.

De la figura 4.14 obtenemos los siguientes elementos del AFN: $\Sigma = \{a, b\}$; $\Gamma = \{k_0, k_1, k_2\}$; $\Delta = \{k_0\}$; el estado inicial es k_0 y finalmente la función δ se "lee" de la figura. De acuerdo con el teorema 4.2 (pág. 146) construimos un AFD $= \langle \Sigma', \Gamma', \Delta', \delta', k'_0 \rangle$ como sigue: $\Sigma' = \Sigma = \{a, b\}$, $k'_0 = k_0$, $\Gamma' = P(\Gamma) = \{\emptyset, \{k_0\}, \{k_1\}, \{k_2\}, \{k_0, k_1\}, \{k_0, k_2\}, \{k_1, k_2\}, \{k_0, k_1, k_2\}\}$. La función δ' está definida por:

$$\begin{aligned} \delta'(\{k_0, k_1, k_2\}, a) &= \delta(k_0, a) \bigcup \delta(k_1, a) \bigcup \delta(k_2, a) \\ &= \{k_1\} \bigcup \{k_0\} \bigcup \{k_0, k_1, k_2\} \\ &= \{k_0, k_1, k_2\}, \end{aligned}$$

por lo tanto:

$$\begin{aligned}
 \delta'(\{k_0, k_1, k_2\}, b) &= \delta(k_0, b) \bigcup \delta(k_1, b) \bigcup \delta(k_2, b) = \emptyset \bigcup \{k_2\} \bigcup \{k_0\} = \{k_0, k_2\}, \\
 \delta'(\{k_0, k_1\}, a) &= \delta(k_0, a) \bigcup \delta(k_1, a) = \{k_1\} \bigcup \{k_0\} = \{k_0, k_1\}, \\
 \delta'(\{k_0, k_1\}, b) &= \delta(k_0, b) \bigcup \delta(k_1, b) = \emptyset \bigcup \{k_2\} = \{k_2\}, \\
 \delta'(\{k_0, k_2\}, a) &= \delta(k_0, a) \bigcup \delta(k_2, a) = \{k_1\} \bigcup \{k_0, k_1, k_2\} = \{k_0, k_1, k_2\}, \\
 \delta'(\{k_0, k_2\}, b) &= \delta(k_0, b) \bigcup \delta(k_2, b) = \emptyset \bigcup \{k_0\} = \{k_0\}, \\
 \delta'(\{k_1, k_2\}, a) &= \delta(k_1, a) \bigcup \delta(k_2, a) = \{k_0\} \bigcup \{k_0, k_1, k_2\} = \{k_0, k_1, k_2\}, \\
 \delta'(\{k_1, k_2\}, b) &= \delta(k_1, b) \bigcup \delta(k_2, b) = \{k_2\} \bigcup \{k_0\} = \{k_0, k_2\}, \\
 \delta'(\{k_0\}, a) &= \delta(k_0, a) = \{k_1\}, \\
 \delta'(\{k_0\}, b) &= \delta(k_0, b) = \emptyset, \\
 \delta'(\{k_1\}, a) &= \delta(k_1, a) = \{k_0\}, \\
 \delta'(\{k_1\}, b) &= \delta(k_1, b) = \{k_2\}, \\
 \delta'(\{k_2\}, a) &= \delta(k_2, a) = \{k_0, k_1, k_2\}, \\
 \delta'(\{k_2\}, b) &= \delta(k_2, b) = \{k_0\}, \\
 \delta'(\emptyset, a) &= \emptyset, \\
 \delta'(\emptyset, b) &= \emptyset.
 \end{aligned}$$

Además, como el conjunto de estados de aceptación es $\Delta = \{k_0\}$, entonces: $\Delta' = \{\{k_0\}, \{k_0, k_1\}, \{k_0, k_2\}, \{k_0, k_1, k_2\}\}$.

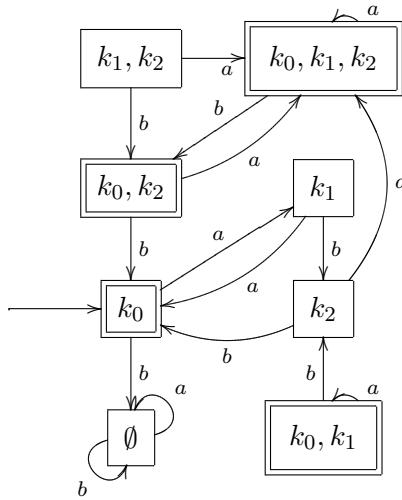
La figura 4.15 representa el \mathcal{AFD} que hemos construido. Observe el lector que los estados $\{k_0, k_1\}$ y $\{k_1, k_2\}$ nunca se alcanzan, por lo cual se pueden eliminar del diagrama, obteniendo así la figura 4.16. Además, observe el lector que el estado \emptyset es un estado absorbente, es decir, una vez se llega a él no es posible salir de ahí.

4.5. Álgebra y Autómatas

4.5.1. Monoides asociados con un autómata

Sea $\mathcal{A} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$ un autómata de estado finito. Podemos probar que el autómata \mathcal{A} tiene asociados dos monoides. Veámoslo.

1. Monoide generado por Σ : $< \Sigma^*, \bullet, \varepsilon >$ donde:
 - Σ^* : Lenguaje universal para el alfabeto Σ .
 - \bullet : Concatenación de palabras.
 - ε : Palabra vacía.

Figura 4.15: Construcción de un \mathcal{AFD} a partir de un \mathcal{AFN} (1).

Demostración. Es necesario demostrar que $\langle \Sigma^*, \bullet, \varepsilon \rangle$ satisface las propiedades de un monoide, es decir, es necesario demostrar que: la operación \bullet es cerrada en Σ^* , es asociativa y por último probar la existencia de un elemento neutro.

- a) La concatenación de palabras es una operación cerrada
Sean $\alpha, \beta \in \Sigma^*$ entonces $\alpha \bullet \beta \in \Sigma^*$.
- b) Asociatividad
 $\forall \alpha \forall \beta \forall \gamma \in \Sigma^* ((\alpha \bullet (\beta \bullet \gamma)) = ((\alpha \bullet \beta) \bullet \gamma) = \alpha \beta \gamma)$.
- c) Existencia elemento neutro
 $\forall \alpha \in \Sigma^* (\alpha \bullet \varepsilon = \varepsilon \bullet \alpha = \alpha)$.

□

2. Monoide de transformaciones de Γ : $\langle \Gamma^\Gamma, \circ, f_\varepsilon \rangle$ donde:

Γ^Γ : Conjunto de funciones de Γ en Γ , definidas para cada uno de los símbolos pertenecientes a Σ , es decir, $\Gamma^\Gamma = \{f_a : \Gamma \rightarrow \Gamma \mid a \in \Sigma\}$.

\circ : Composición de funciones.

f_ε : Función de la palabra vacía. $\forall k \in \Gamma (f_\varepsilon(k) = k)$.

Demostración. Es necesario demostrar que $\langle \Gamma^\Gamma, \circ, f_\varepsilon \rangle$ satisface las propiedades de un monoide, es decir, es necesario demostrar que la operación \circ , es cerrada en Γ^Γ , es asociativa y que existe un elemento neutro.

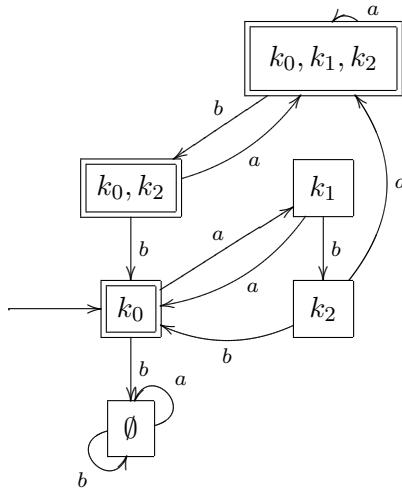


Figura 4.16: Construcción de un \mathcal{AFD} a partir de un \mathcal{AFN} (2).

- a) La composición de funciones es una operación cerrada
Sean $f_a, f_b \in \Gamma^\Gamma$ entonces $f_a \circ f_b \in \Gamma^\Gamma$.
 - b) Asociatividad
 $\forall k \in \Gamma, \forall a, b, c \in S((f_a \circ (f_b \circ f_c))(k)) = ((f_a \circ f_b) \circ f_c)(k)$.
 - c) Existencia de elemento neutro
 $\forall a \in \Sigma(f_a \circ f_\varepsilon = f_\varepsilon \circ f_a = f_a)$.

□

Una vez definidos los monoides para el autómata \mathcal{A} , es necesario extender las funciones f_a , de manera que operen sobre cadenas de palabras y no únicamente sobre símbolos del alfabeto Σ . La función f_a está definida por: $f_a: \Gamma \rightarrow \Gamma \mid a \in \Sigma$. Ahora necesitamos una función f_α definida por: $f_\alpha: \Gamma \rightarrow \Gamma \mid \alpha \in \Sigma^*$. Sea $\alpha \equiv a_1a_2 \dots a_n$, donde $a_i \in \Sigma$, entonces:

$$f_\alpha(k) = (f_{a_n} \circ f_{a_{n-1}} \circ \cdots \circ f_{a_1})(k) = f_{a_n}(f_{a_{n-1}}(\dots(f_{a_1}(k))\dots)).$$

Ejemplo 4.15. Para el autómata representado por la figura 4.17 tenemos que:

$$\Gamma^{\bar{\Gamma}} = \{f_a : \Gamma \rightarrow \Gamma \mid a \in \Sigma\}.$$

$f_a: \Gamma \rightarrow \Gamma$, donde $f_a(k_1) = k_1$ y $f_a(k_2) = k_1$.

$f_b: \Gamma \rightarrow \Gamma$, donde $f_b(k_1) = k_2$ y $f_b(k_2) = k_2$.

Luego, $\Gamma^\Gamma = \{f_a, f_b, f_\varepsilon\}$.

Sea $\alpha \equiv abb$, entonces:

$$\begin{aligned} f_\alpha(k_1) &= (f_b \circ f_b \circ f_a)(k_1) \\ &= f_b(f_b(f_a(k_1))) \\ &= f_b(f_b(k_1)) \\ &= f_b(k_2) \\ &= k_2; \end{aligned}$$

$$\begin{aligned} f_\alpha(k_2) &= (f_b \circ f_b \circ f_a)(k_2) \\ &= f_b(f_b(f_a(k_2))) \\ &= f_b(f_b(k_1)) \\ &= f_b(k_2) \\ &= k_2 \end{aligned}$$

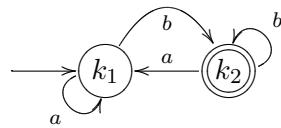


Figura 4.17: Expansión de la función $f_a, a \in \Sigma$ en $f_\alpha, \alpha \in \Sigma^*$.

El propósito que buscamos es poder expresar el comportamiento del autómata \mathcal{A} por medio de una función $t: \Sigma^* \rightarrow \Gamma^\Gamma$, donde $t(\alpha) = f_\alpha$. El objetivo es asignar a cada cadena de entrada de Σ^* una función de Γ en Γ .

4.5.2. Comportamiento entrada-estados de un autómata

Definición 4.17 (Homomorfismo monoides). Sean $\langle A, *, e \rangle$ y $\langle A', *, e' \rangle$ dos monoides y sea $f: A \rightarrow A'$ una función. Se dice que f es un homomorfismo entre $\langle A, *, e \rangle$ y $\langle A', *, e' \rangle$ si f preserva el símbolo de función y f preserva el símbolo de constante, es decir, $\forall a \forall b \in A (f(a * b) = (f(a) *' f(b)))$, además $f(e) = e'$.

El siguiente teorema presenta un homomorfismo entre los monoides asociados con un autómata.

Teorema 4.3. Si $\mathcal{A} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$ es un autómata de estado finito, $\langle \Sigma^*, \bullet, \varepsilon \rangle$ es el monoide generado por Σ y $\langle \Gamma^\Gamma, \circ, f_\varepsilon \rangle$ es el monoide de transformaciones de Γ , entonces la función $t: \Sigma^* \rightarrow \Gamma^\Gamma$, tal que $t(\alpha) = f_\alpha$ es un homomorfismo entre los monoides $\langle \Sigma^*, \bullet, \varepsilon \rangle$ y $\langle \Gamma^\Gamma, \circ, f_\varepsilon \rangle$.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a

EDICT._GUIA_MEX - Editorial Guias Mexico

Demostración. Ejercicio 4.19. □

La figura 4.18 representa el homomorfismo entre los dos monoides.

$$\begin{array}{ccc} <\Sigma^*, \bullet, \varepsilon> & \xrightarrow{t} & <\Gamma^\Gamma, \circ, f_\varepsilon> \\ \bullet \downarrow & & \downarrow \circ \\ <\Sigma^*, \bullet, \varepsilon> & \xrightarrow{t} & <\Gamma^\Gamma, \circ, f_\varepsilon> \end{array}$$

Figura 4.18: Homomorfismo entre los monoides $\langle \Sigma^*, \bullet, \varepsilon \rangle$ y $\langle \Gamma^\Gamma, \circ, f_\varepsilon \rangle$.

Definición 4.18 (Comportamiento entrada-estados). Sea un autómata de estado finito $\mathcal{A} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$, sea $\langle \Sigma^*, \bullet, \varepsilon \rangle$ el monoide generado por Σ , sea $\langle \Gamma^\Gamma, \circ, f_\varepsilon \rangle$ el monoide de transformaciones de Γ y sea $t: \Sigma^* \rightarrow \Gamma^\Gamma$ (donde $t(\alpha) = f_\alpha$), el homomorfismo entre los monoides $\langle \Sigma^*, \bullet, \varepsilon \rangle$ y $\langle \Gamma^\Gamma, \circ, f_\varepsilon \rangle$. La función t es denominada el comportamiento de entrada-estados del autómata \mathcal{A} .

4.5.3. Relación de equirrespuesta de un autómata

Definición 4.19 (Relación de equirrespuesta). Sea un autómata de estado finito $\mathcal{A} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$ y sea $t: \Sigma^* \rightarrow \Gamma^\Gamma$ el comportamiento de entrada-estados del autómata \mathcal{A} . La relación de equirrespuesta de \mathcal{A} , representada por \approx_A , está definida por:

$$\begin{aligned} \alpha \approx_A \beta &\stackrel{\text{def}}{=} t(\alpha) = t(\beta) \\ &\stackrel{\text{def}}{=} f_\alpha = f_\beta \\ &\stackrel{\text{def}}{=} \forall k \in \Gamma (\delta^*(k, \alpha) = \delta^*(k, \beta) \text{ para } \alpha, \beta \in \Sigma^*). \end{aligned}$$

Para efectos de simplificación en la notación, ‘ $\delta^*(k, \alpha)$ ’ se representará, de ahora en adelante, por ‘ $\delta(k, \alpha)$ ’.

Teorema 4.4. La relación de equirrespuesta \approx_A es una relación de equivalencia.

Demostración. Es necesario demostrar que \approx_A es una relación reflexiva, simétrica y transitiva.

1. Reflexiva

$$\forall k \in \Gamma (\delta(k, \alpha) = \delta(k, \alpha)) \Rightarrow \alpha \approx_A \alpha.$$

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

2. Simétrica

$$\begin{aligned}\alpha \approx_A \beta &\Rightarrow \forall k \in \Gamma(\delta(k, \alpha) = \delta(k, \beta)) \\ &\Rightarrow \forall k \in \Gamma(\delta(k, \beta) = \delta(k, \alpha)) \\ &\Rightarrow \beta \approx_A \alpha.\end{aligned}$$

3. Transitiva

$$\begin{aligned}\alpha \approx_A \beta \wedge \beta \approx_A \gamma &\Rightarrow \forall k \in \Gamma(\delta(k, \alpha) = \delta(k, \beta)) \wedge \forall k \in \Gamma(\delta(k, \beta) = \delta(k, \gamma)) \\ &\Rightarrow \forall k \in \Gamma(\delta(k, \alpha) = \delta(k, \gamma)) \\ &\Rightarrow \alpha \approx_A \gamma.\end{aligned}$$

□

Del teorema anterior podemos garantizar que la relación de equirrespuesta \approx_A induce una partición sobre el conjunto Σ^* (en clases de equivalencia) definida por: $\Sigma^*/\approx_A = \{[\alpha] \mid \alpha \in \Sigma^*\}$ donde, $[\alpha] = \{\beta \in \Sigma^* \mid \alpha \approx_A \beta\}$.

Definición 4.20 (Partición de índice finito). Una partición P tiene índice finito si el cardinal de P , denotado por $\overline{\overline{P}}$, es finito.

Teorema 4.5. *La partición Σ^*/\approx_A tiene índice finito.*

Demostración. Para el homomorfismo de comportamiento de entrada-estados $t: \Sigma^* \rightarrow \Gamma^\Gamma$, donde $t(\alpha) = f_\alpha$; se tiene que $\overline{\overline{\Gamma^\Gamma}} \leq n^n$, donde n es el número de estados del autómata de estado finito; luego $\overline{\overline{\Gamma^\Gamma}}$ tiene cardinal finito. Si el homomorfismo $t: \Sigma^* \rightarrow \Gamma^\Gamma$ es sobreyectivo, entonces $\overline{\overline{\Sigma^*/\approx_A}} = \overline{\overline{\Gamma^\Gamma}}$; de lo contrario $\overline{\overline{\Sigma^*/\approx_A}} \leq \overline{\overline{\Gamma^\Gamma}}$. En cualquier caso, Σ^*/\approx_A tiene cardinal finito, luego es de índice finito. □

4.5.4. Relaciones de congruencia

Definición 4.21 (Relación de congruencia derecha). Sea $\langle A, *, e \rangle$ un monoide y R una relación de equivalencia definida en A . R es una relación de congruencia derecha en $\langle A, *, e \rangle$, si y sólo si, $\forall a \forall b \forall c \in A \quad ((aRb) \Rightarrow ((a * c)R(b * c)))$.

Definición 4.22 (Relación de congruencia izquierda). Sea $\langle A, *, e \rangle$ un monoide y R una relación de equivalencia definida en A . R es una relación de congruencia izquierda en $\langle A, *, e \rangle$, si y sólo si, $\forall a \forall b \forall c \in A \quad ((aRb) \Rightarrow ((c * a)R(c * b)))$.

Definición 4.23 (Relación de congruencia). Sea $\langle A, *, e \rangle$ un monoide y R una relación de equivalencia definida en A . R es una relación de congruencia en $\langle A, *, e \rangle$, si y sólo si, $\forall a \forall b \forall c \in A \quad ((aRb) \Rightarrow ((a * c)R(b * c) \wedge (c * a)R(c * b)))$.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Teorema 4.6. La relación de equirrespuesta \approx_A definida en Σ^* (Σ^* es el dominio del monoide $\langle \Sigma^*, \bullet, \varepsilon \rangle$), es una relación de congruencia.

Demostración. Es necesario demostrar que \approx_A es una relación de congruencia derecha e izquierda sobre $\langle \Sigma^*, \bullet, \varepsilon \rangle$.

1. \approx_A es una relación de congruencia derecha sobre $\langle \Sigma^*, \bullet, \varepsilon \rangle$

$$\begin{aligned}\alpha \approx_A \beta &\Leftrightarrow t(\alpha) = t(\beta) \\ &\Leftrightarrow t(\alpha\eta) = t(\alpha) \circ t(\eta) = t(\beta) \circ t(\eta) = t(\beta\eta) \\ &\Leftrightarrow \alpha\eta \approx_A \beta\eta, \quad \text{para toda } \eta \in \Sigma^*.\end{aligned}$$

2. \approx_A es una relación de izquierda sobre $\langle \Sigma^*, \bullet, \varepsilon \rangle$

Se demuestra de forma similar a la anterior.

Luego, \approx_A es una relación de congruencia sobre $\langle \Sigma^*, \bullet, \varepsilon \rangle$. □

4.5.5. Relación equirrespuesta de un reconocedor finito

Definición 4.24 (Relación equirrespuesta de un reconocedor finito). Sea un reconocedor finito (autómata de estado finito) $\mathcal{R} = \langle \Sigma, \Gamma, \Delta, \delta, k_0 \rangle$. La relación de equirrespuesta de \mathcal{R} , representada por \approx_R , está definida por:

$$\alpha \approx_R \beta \stackrel{\text{def}}{\equiv} \delta(k_0, \alpha) = \delta(k_0, \beta) \text{ para todo } \alpha, \beta \in \Sigma^*.$$

Teorema 4.7. La relación de equirrespuesta \approx_R es una relación de equivalencia.

Demostración. La demostración se deja como ejercicio. □

Del teorema anterior podemos garantizar que la relación de equirrespuesta \approx_R induce una partición sobre el conjunto Σ^* en clases de equivalencia definida por: $\Sigma^*/\approx_R = \{[\alpha] \mid \alpha \in \Sigma^*\}$, donde, $[\alpha] = \{\beta \in \Sigma^* \mid \alpha \approx_R \beta\}$.

Teorema 4.8. La relación de equirrespuesta \approx_A de un autómata de estado finito implica la relación de equirrespuesta \approx_R de un reconocedor finito.

Demostración.

$$\begin{aligned}\alpha \approx_A \beta &\Rightarrow \forall k \in \Gamma (\delta(k, \alpha) = \delta(k, \beta)) \\ &\Rightarrow \delta(k_0, \alpha) = \delta(k_0, \beta) \\ &\Rightarrow \alpha \approx_R \beta.\end{aligned}$$

□

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados

Teorema 4.9. La relación de equirrespuesta \approx_R de un reconocedor finito no implica la relación de equirrespuesta \approx_A de un autómata de estado finito.

Demostración.

$$\begin{aligned}\alpha \approx_R \beta &\Rightarrow \delta(k_0, \alpha) = \delta(k_0, \beta) \\ &\Rightarrow \forall k \in \Gamma(\delta(k, \alpha) = \delta(k, \beta)) \\ &\Rightarrow \alpha \approx_A \beta.\end{aligned}$$

□

Definición 4.25 (Refinamiento de una partición). Se dice que una partición P refina una partición Q , si y sólo si, $\overline{\overline{P}} \geq \overline{\overline{Q}}$.

Teorema 4.10. La partición Σ^*/\approx_A refina la partición Σ^*/\approx_R .

Demostración.

1. Como $\approx_A \Rightarrow \approx_R$ (por el teorema 4.8 (pág. 154)), entonces se observa la posible igualdad entre $\overline{\Sigma^*/\approx_A}$ y $\overline{\Sigma^*/\approx_R}$.
2. Como $\approx_R \not\Rightarrow \approx_A$ (por el teorema 4.9 (pág. 155)), entonces se observa la posibilidad de que $\overline{\overline{\Sigma^*/\approx_A}} > \overline{\overline{\Sigma^*/\approx_R}}$.
3. Entonces $\overline{\overline{\Sigma^*/\approx_A}} \geq \overline{\overline{\Sigma^*/\approx_R}}$, luego Σ^*/\approx_A refina a Σ^*/\approx_R .

□

Teorema 4.11. La partición Σ^*/\approx_R tiene índice finito.

Demostración. Como Σ^*/\approx_A tiene índice finito (teorema 4.5 (pág. 153)) y Σ^*/\approx_A refina a Σ^*/\approx_R (teorema 4.10 (pág. 155)), entonces Σ^*/\approx_A tiene índice finito. □

Teorema 4.12. La relación de equirrespuesta \approx_R definida en Σ^* (Σ^* que es el dominio del monoide $< \Sigma^*, \bullet, \epsilon >$), es una relación de congruencia derecha, es decir, $\forall \alpha, \beta, \eta \in \Sigma^* ((\alpha \approx_R \beta) \Rightarrow ((\alpha\eta) \approx_R (\beta\eta)))$.

Demostración.

$$\begin{aligned}\alpha \approx_A \beta &\Rightarrow (\delta(k_0, \alpha) = \delta(k_0, \beta)) \\ &\Rightarrow (\delta(k_0, \alpha\eta) = \delta(\delta(k_0, \alpha), \eta) = \delta(\delta(k_0, \beta), \eta) = \delta(k_0, \beta\eta)) \\ &\Rightarrow \alpha\eta \approx_A \beta\eta \quad \text{para toda } \eta \in \Sigma^*.\end{aligned}$$

□

4.6. Álgebra y Lenguajes

4.6.1. Relación de congruencia derecha inducida por un lenguaje

Definición 4.26 (Relación inducida por un lenguaje). Sea Σ un alfabeto y $\mathcal{L} \subseteq \Sigma^*$ un lenguaje. La relación inducida por \mathcal{L} , representada por \approx_L , está definida de Σ^* en Σ^* por: sean $\alpha, \beta, \delta \in \Sigma^*$, entonces:

$$\alpha \approx_L \beta \stackrel{\text{def}}{\equiv} (\alpha\delta \in \mathcal{L} \leftrightarrow \beta\delta \in \mathcal{L}).$$

Teorema 4.13. *La relación inducida por \mathcal{L} (\approx_L) es una relación de equivalencia.*

Demostración.

1. Reflexiva

$$(\alpha\delta \in \mathcal{L} \leftrightarrow \alpha\delta \in \mathcal{L}) \Rightarrow \alpha \approx_L \alpha.$$

2. Simétrica

$$\begin{aligned} \alpha \approx_L \beta &\Rightarrow (\alpha\delta \in \mathcal{L} \leftrightarrow \beta\delta \in \mathcal{L}) \\ &\Rightarrow \beta \approx_L \alpha. \end{aligned}$$

3. Transitiva

$$\begin{aligned} \alpha \approx_L \beta \wedge \beta \approx_L \gamma &\Rightarrow (\alpha\delta \in \mathcal{L} \leftrightarrow \beta\delta \in \mathcal{L}) \wedge (\beta\delta \in \mathcal{L} \leftrightarrow \gamma\delta \in \mathcal{L}) \\ &\Rightarrow (\alpha\delta \in \mathcal{L} \leftrightarrow \gamma\delta \in \mathcal{L}) \\ &\Rightarrow \alpha \approx_L \gamma. \end{aligned}$$

□

El teorema anterior nos garantiza que la relación \approx_L induce una partición sobre el conjunto Σ^* (en clases de equivalencia) definida por: $\Sigma^*/\approx_L = \{[\alpha] \mid \alpha \in \Sigma^*\}$; donde, $[a] = \{\beta \in \Sigma^* \mid \alpha \approx_L \beta\}$.

Teorema 4.14. *La relación inducida por \mathcal{L} (\approx_L) es una relación de congruencia derecha, es decir: $\forall \alpha, \beta, \delta \in \Sigma^* \quad ((\alpha \approx_L \beta) \Rightarrow (\alpha\delta \approx_L \beta\delta))$.*

Demostración. Sea $\alpha, \beta, \delta \in \Sigma^*$; $\delta \equiv \eta\theta$

$$\begin{aligned} \alpha \approx_L \beta &\Rightarrow (\alpha\delta \in \mathcal{L} \leftrightarrow \beta\delta \in \mathcal{L}) \\ &\Rightarrow (\alpha\eta\theta \in \mathcal{L} \leftrightarrow \beta\eta\theta \in \mathcal{L}) \\ &\Rightarrow \alpha\eta \approx_L \beta\eta. \end{aligned}$$

□

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

4.6.2. Condición para que un lenguaje sea aceptado por un reconocedor finito

Teorema 4.15. Sea Σ un alfabeto y $\mathcal{L} \subseteq \Sigma^*$ un lenguaje. \mathcal{L} es un lenguaje aceptado por un reconocedor finito, si y sólo si la relación de congruencia derecha inducida por \mathcal{L} tiene índice finito.

Demostración. (primera parte)

Si \mathcal{L} es un lenguaje aceptado por un reconocedor finito (hipótesis), entonces la relación de congruencia derecha inducida por \mathcal{L} tiene índice finito (tesis).

Hipótesis:

Sean $\mathcal{R} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$ un reconocedor finito, $\mathcal{L} \subseteq \Sigma^*$ un lenguaje, $\mathcal{L}(\mathcal{R})$ es el conjunto de cadenas aceptadas por \mathcal{R} y $\mathcal{L} = \mathcal{L}(\mathcal{R})$.

Tesis:

Sea $\alpha, \beta \in \Sigma^*$; $\alpha \approx_R \beta \Rightarrow \alpha \approx_L \beta$.

Entonces,

$$\begin{aligned}\alpha \approx_R \beta &\Leftrightarrow \delta(k_o, \alpha) = \delta(k_o, \beta) \quad (\text{definición de } \approx_R) \\ &\Rightarrow \delta(k_o, \alpha\eta) = \delta(k_o, \beta\eta) \\ &\Rightarrow \delta(k_o, \alpha\eta) \in \Delta \Leftrightarrow \delta(k_o, \beta\eta) \in \Delta \\ &\Rightarrow \alpha\eta \in \mathcal{L}(\mathcal{R}) \Leftrightarrow \beta\eta \in \mathcal{L}(\mathcal{R}) \\ &\Rightarrow \alpha\eta \in \mathcal{L} \Leftrightarrow \beta\eta \in \mathcal{L} \\ &\Rightarrow \alpha \approx_L \beta.\end{aligned}$$

Luego, Σ^*/\approx_R refina a Σ^*/\approx_L y, como Σ^*/\approx_R tiene índice finito (teorema 4.11 (pág. 155)), entonces Σ^*/\approx_L tiene índice finito. \square

Demostración. (segunda parte)

Si la relación de congruencia derecha inducida por \mathcal{L} tiene índice finito (hipótesis), entonces \mathcal{L} es un lenguaje aceptado por un reconocedor finito (tesis).

Idea: Construir un reconocedor finito que acepta \mathcal{L} .

Sea $\mathcal{R} = < \Sigma, \Gamma, \Delta, \delta, k_0 >$ un reconocedor finito donde:

Σ : Alfabeto del lenguaje \mathcal{L} .

Γ : Σ^*/\approx_L (finito porque Σ^*/\approx_L tiene índice finito por hipótesis). Los elementos de Γ son clases de equivalencia $[a] = \{\beta \in \Sigma^* \mid \alpha \approx_L \beta\}$.

δ : $\Gamma \times \Sigma \rightarrow \Gamma$, donde $\delta([\alpha], a) = [\alpha a]$ para toda $a \in \Sigma$.

k_o : $[\varepsilon]$ (ε es la palabra vacía).

Δ : $\{[\beta] \mid \beta \in \mathcal{L}\}$.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

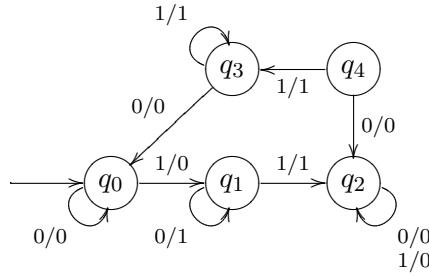
Veamos que $\mathcal{L} = \mathcal{L}(\mathcal{R})$.

$$\begin{aligned}\alpha \in \mathcal{L}(\mathcal{R}) &\Leftrightarrow \delta(k_o, \alpha) \in \Delta \\ &\Leftrightarrow \delta([\varepsilon], \alpha) \in \Delta \\ &\Leftrightarrow [\varepsilon\alpha] \in \Delta \\ &\Leftrightarrow [\alpha] \in \Delta \\ &\Leftrightarrow \alpha \in \mathcal{L} \quad (\text{por definición de } \Delta).\end{aligned}$$

□

4.7. Ejercicios

Ejercicio 4.1. Para la máquina de estado finito representada por la figura:



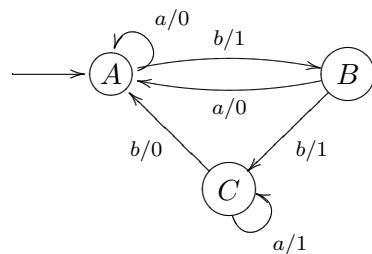
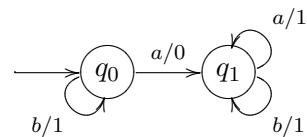
- Determine la palabra de salida para la entrada 110111, comenzando en q_0 . ¿Cuál es el último estado de transición?
- Determine la palabra de salida para la entrada 110111, comenzando en q_1 . ¿Qué sucede cuando q_2 y q_3 son los estados iniciales?
- Encuentre la tabla de transición para esta máquina.
- ¿Desde qué estado se debe comenzar para que la palabra de entrada 10010 produzca la salida 10000?

Ejercicio 4.2. Dibuje el diagrama de transición correspondiente a las máquinas de estado finito, indicadas por las siguientes tablas de transición:

Γ/Σ	a	b
q_0	$q_1, 1$	$q_1, 1$
q_1	$q_0, 0$	$q_1, 1$

Γ/Σ	a	b	c
q_0	$q_1, 1$	$q_0, 1$	$q_2, 2$
q_1	$q_0, 2$	$q_2, 0$	$q_2, 0$
q_2	$q_3, 1$	$q_3, 0$	$q_0, 1$
q_3	$q_1, 2$	$q_1, 0$	$q_0, 2$

Ejercicio 4.3. Para cada una de las máquinas de estado finito indicadas por el diagrama de transición, construya su representación explícita.



Ejercicio 4.4. Tal como está representado el sumador binario por el diagrama de la figura 4.5, es necesario sumar $01 + 01$ si se desea realizar la suma de $1 + 1$. ‘Analice por qué.

Ejercicio 4.5. Un modelo simplificado del comportamiento de un estudiante puede ser descrito por una máquina de estado finito. Sea:

$$\Sigma = \{\text{tarea, fiesta, examen}\}.$$

$$\Delta = \{\text{cantar, maldecir, dormir}\}.$$

$$\Gamma = \{\text{feliz, enojado, deprimido}\}.$$

$$k_0 = \text{feliz}.$$

Construya una máquina de estado finito que modele un posible comportamiento del estudiante.

Ejercicio 4.6. Diseñe una máquina de estado finito que modele el comportamiento de una máquina expendedora de Coca-Cola y Malta. La máquina acepta monedas de 5, 10 y 25 pesos y dispone de dos botones; C para Coca-Cola y M para Malta. Cada producto cuesta 20 pesos y se espera por supuesto que la máquina entregue el producto solicitado y la devuelva si ésta existe.

Ejercicio 4.7. Un procedimiento sencillo y muy utilizado para detectar errores en una transmisión digital consiste en enviar un bit de paridad. Este bit puede ser tal, que haga par el número total de “unos” enviados, o haga par el número total de “ceros” enviados (en este caso se habla de paridad par), o también puede ser que este bit haga impar el número total de “unos” o el número total de ceros enviados.

Para el caso de paridad par de “unos”, el generador de paridad actúa de la siguiente forma (supongamos la longitud del mensaje de 3 bits): si el mensaje a enviar tiene un número impar de “unos” (por ejemplo “100”), el generador de paridad adiciona un “uno” (“1100”) y envía el mensaje; si por el contrario el mensaje que se enviará tiene un número par de “unos” (por ejemplo “101”) el generador de paridad adiciona un “cero” (“0101”) y envía el mensaje.

El detector de paridad (para el caso de paridad par de “unos”) trabaja de la siguiente forma: si el mensaje recibido tiene un número par de “unos”, la transmisión del mensaje NO tuvo errores; pero si el mensaje recibido tiene un número impar de “unos”, entonces la transmisión del mensaje SÍ tuvo errores y debe ser retransmitido.

Construya una máquina de estado finito que se comporte como un detector de paridad par de “unos”. Represente la máquina por medio de su diagrama de transiciones.

Ejercicio 4.8. Obtener la tabla de transiciones y el diagrama de transiciones de la máquina de Moore equivalente a la máquina de Mealy, descrita por la siguiente tabla:

Γ/Σ	e_1	e_2
q_0	q_3, s_1	q_2, s_2
q_1	q_4, s_2	q_3, s_2
q_3	q_4, s_1	q_2, s_2
q_4	q_2, s_2	q_4, s_1

Ejercicio 4.9. Definir las máquinas de Mealy y de Moore para un restador binario.

Ejercicio 4.10. Construya un autómata de estado finito para solucionar el problema expuesto en la siguiente carta:

Querido amigo:

Al poco tiempo de comprar esta vieja mansión tuve la desagradable sorpresa de comprobar que está hechizada con dos sonidos de ultratumba que la hacen prácticamente inhabitable: un canto picaresco y una risa sardónica. Aún conservo, sin embargo, cierta esperanza, pues la experiencia me ha demostrado que su comportamiento obedece a ciertas leyes, oscuras pero infalibles, y que puede modificarse tocando el órgano y quemando incienso.

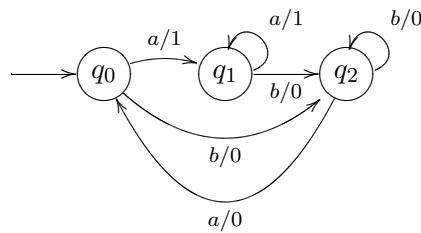
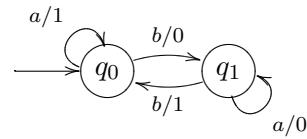
En cada minuto, cada sonido está presente o ausente. Lo que cada uno de ellos hará en el minuto siguiente depende de que lo pasa en el minuto actual, de la siguiente manera:

El canto conservará el mismo estado (presente o ausente), salvo si durante el minuto actual no se oye la risa y toca el órgano, en cuyo caso el canto toma el estado opuesto.

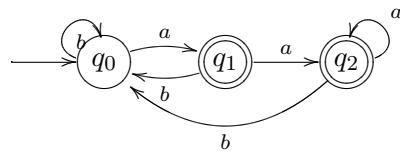
En cuanto a la risa, si no quemo incienso, se oirá o no según que el canto esté presente o ausente (de modo que la risa imita al canto con un minuto de retardo). Ahora bien, si quemo incienso la risa hará justamente lo contrario de lo que hacía el canto.

En el momento en que le escribo estoy oyendo a la vez la risa y el canto. Le quedará muy agradecido si me dice qué manipulaciones de órgano e incienso debo seguir para restablecer definitivamente la calma.

Ejercicio 4.11. Demuestre que cada una de las máquinas de estado finito siguientes es un autómata de estado finito y trace de nuevo el diagrama de transición como uno de un autómata de estado finito.



Ejercicio 4.12. ¿Son las palabras *abaa* y *abbaa* aceptadas por el reconocedor finito indicado por la figura?



Ejercicio 4.13. Diseñe un reconocedor finito que acepte únicamente las palabras no nulas sobre $\{a, b\}$ que satisfagan las siguientes condiciones:

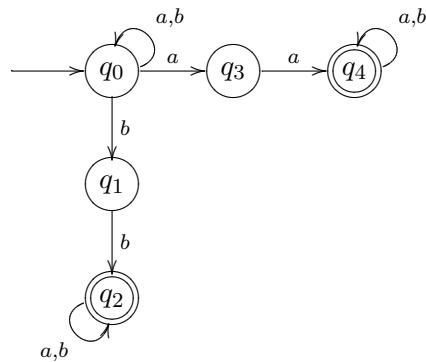
1. No contengan letras *a*.
2. Tienen un número par de letras *aes*.
3. Exactamente una letra *b*.
4. Exactamente dos letras *a*.
5. $\{\alpha \mid \text{toda } a \text{ de } \alpha \text{ está entre dos } bes\}$.
6. $\{\alpha \mid \alpha \text{ contiene la subcadena } abab\}$.
7. $\{\alpha \mid \alpha \text{ no contiene ninguna de las subcadenas } aa \text{ o } bb\}$.
8. $\{\alpha \mid \alpha \text{ tiene un número impar de } aes \text{ y un número par de } bes\}$.

9. $\{\alpha \mid \alpha \text{ tiene } ab \text{ y } ba \text{ como subcadenas}\}.$

Ejercicio 4.14. Dos reconocedores (automátas) finitos \mathcal{RF}_1 y \mathcal{RF}_2 se dicen equivalentes si y sólo si aceptan el mismo lenguaje, es decir, $\mathcal{L}(\mathcal{RF}_1) = \mathcal{L}(\mathcal{RF}_2)$.

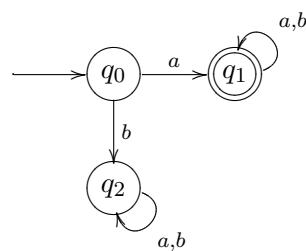
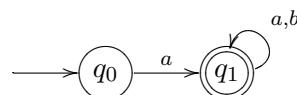
Sea RF el conjunto de todos los reconocedor finitos sobre un alfabeto Σ . Sea $R \subseteq RF \times RF$ la relación definida por: la pareja $(\mathcal{RF}_1, \mathcal{RF}_2)$ está en R si y sólo si \mathcal{RF}_1 es equivalente a \mathcal{RF}_2 . Demuestre que la relación R es una relación de equivalencia en RF (y por tanto, que la definición de equivalencia de reconocedores finitos es consistente con el uso matemático habitual de los términos).

Ejercicio 4.15. Para el autómata de estado finito representado por la figura:



1. ¿Qué tipo de autómata es?
2. ¿Qué lenguaje reconoce?

Ejercicio 4.16. Para los autómatas de estado finito representados por la figuras:



1. ¿Cuál es un AFD?
2. ¿Cuál es un AFN?
3. Pruebe que reconocen el lenguaje dado por $a(a \mid b)^*$.
4. Justifique por qué son equivalentes.

Ejercicio 4.17. Diseñe un AFN que acepte únicamente los conjuntos de palabras dados por:

1. $\{a\}$.
2. $\{b\}$.
3. $\{a, b\}$.
4. $(a \mid b)^* \mid (aba)^+$.
5. Palabras de la forma $bowwow, bowwowwow, bowwowwowow, \dots$
6. Palabras de la forma $ohmy, ohmyohmy, ohmyohmyohmy, \dots$
7. La unión de los dos lenguajes anteriores.

Ejercicio 4.18. Obtener los monoides del sumador y del restador binario y compararlos.

Ejercicio 4.19. Demuestre el teorema 4.3 (pág. 151).

4.8. Notas Bibliográficas

La presentación de un sistema desde la teoría de sistemas fue tomada de [6]. Las máquinas de estado finito son presentadas por [6, 10, 19, 22, 27], entre otros. Las máquinas de Moore y las máquinas de Mealy son presentadas por [6, 10]. Los autómatas de estado finito, tanto los deterministas como los no deterministas, son presentados por [6, 10, 19, 20]; el término de reconocedor finito es introducido por [10]. Las secciones que establecen la relación entre álgebra, autómatas y lenguajes fueron adaptadas de [10].

Capítulo 5

Autómatas de Pila

En lo concerniente a nuestros desarrollos realizados en el contexto de los autómatas de estado finito es posible que hayamos notado, intuitivamente, que éstos tienen una memoria limitada, esto es, sólo tienen capacidad para una “memoria” finita. El problema se puede observar claramente en el contexto del reconocimiento de algunos lenguajes donde se requiere de un autómata que almacene o guarde una gran cantidad de información. Tómese por ejemplo, el caso de lenguajes de la forma $\{a^m c^m \mid m \geq 0\}$ (independiente del contexto); para casos como éste, el autómata debería realizar diversos procesos como: debe verificar no sólo que toda ‘a’ preceda a toda ‘c’, sino que, además, tiene que contar el número de símbolos ‘a’ y de símbolos ‘c’. Tendríamos entonces que limitar el número de símbolos ‘a’ que el autómata debe contar.

Necesitamos entonces, contar con autómatas que estén dotados de un dispositivo que les permita un almacenamiento no limitado de información y, además, con capacidad para guardar y comparar información. Este sería, por ejemplo, el caso para un autómata reconocedor del siguiente lenguaje independiente del contexto: $\{c\alpha c\alpha^3 \mid \alpha \in \Sigma^*, \text{ y } \Sigma = \{a, b, c\}\}$.

Buscaremos, pues, construir formalmente un tipo de autómata que tenga las buenas propiedades antes mencionadas, de forma tal que lleguemos a solucionar el problema, al menos para cierto tipo de lenguajes; pues, como sabemos, existen lenguajes que no son reconocibles por ningún tipo de autómata finito de las clases que hemos venido considerando. El autómata que se requiere es conocido y se ha designado como autómata de pila o autómata de *stack*.

En general, podemos decir que el comportamiento de un autómata de pila es muy similar a los autómatas de estado finito. La diferencia entre un autómata de pila y un autómata de estado finito estriba en que para el autómata de pila, además del estado actual y del símbolo de entrada, se debe considerar el símbolo de cierto alfabeto, que está (en el momento o tiempo considerado) en la cima de una pila, también llamada *stack*. La pila es, justamente, el nuevo tipo de dispositivo requerido para poder diseñar reconocedores para el tipo de lenguajes que hemos mencionado en esta introducción. Otro aspecto a resaltar en el

comportamiento o funcionamiento de un autómata de pila consiste en el siguiente hecho: un autómata de pila, además de requerir cambiar el estado actual cuando se tiene determinada configuración, también requiere cambiar la información que se halla en ese momento dado en la cima de la pila.

Para simplificar la expresión “Autómata de pila no determinista” (que será básicamente el que estudiaremos en este capítulo) introducimos la sigla APND.

5.1. Autómata de pila no determinista

Desde un enfoque formal diremos que un APND es un estructura formal constituida como una héptupla, mediante la interrelación de un conjunto de estados, un alfabeto de entradas, un alfabeto de pila, una relación de transición, un conjunto de estados de aceptación y dos símbolos distinguidos (estado inicial y símbolo inicial de la pila).

Definición 5.1 (APND). Un autómata de pila (o autómata de *stack*) no determinista está definido por la estructura matemática $\mathcal{APND} = < Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 >$, donde:

Q : Conjunto de estados (finito y diferente de vacío).

Σ : Alfabeto de entrada (finito y diferente de vacío).

Γ : Alfabeto auxiliar o alfabeto de la pila (finito y diferente de vacío).

δ : Relación de transición.

q_0 : Estado inicial del autómata ($q_0 \in Q$).

A : Conjunto de estados de aceptación ($A \subset Q$).

z_0 : Símbolo inicial de la pila ($z_0 \in \Gamma$).

Es importante que anotemos que en realidad δ sólo es una función para un autómata de pila determinista. Para el caso no determinista δ es una relación. De acuerdo con las precisiones hechas sobre el comportamiento de los autómatas no deterministas, diremos que δ es una relación tal que a cada terna $(q, s, z) \in Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$, asocia una o más parejas de la forma (q', β) donde $q' \in Q$ y $\beta \in \Gamma^*$.

Por consiguiente, para definir la regla de transición (δ), debemos considerar el estado actual q , el símbolo de entrada s en el momento considerado y la información que se halla en ese momento en la cima de la pila z ; es decir, determinar la terna (q, s, z) . Luego tenemos que considerar cuál debe ser la reacción del autómata (q') y cuál la información (β) que habrá de ser ubicada en la cima de la pila; es decir, determinar la pareja (q', β) , donde la información a empilar β se ubica en el lugar del símbolo que se hallaba antes en la cima de la pila. Observemos igualmente que la naturaleza de la relación δ , es decir $\delta(q, s, z)$, obliga a que se tenga siempre un símbolo en la cima de la pila, esto es, el autómata no podría efectuar ninguna transición si la pila está vacía.

Precisemos ahora algunas características fundamentales de un APND.

1. Para que un APND pueda efectuar algún movimiento es necesario que exista algún símbolo $z \in \Gamma$ en la cima de la pila. Ello se desprende de la definición de δ , ya que

$$\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*.$$

2. Para cumplir la condición anterior es necesario precisar un símbolo inicial para la pila. $z_0 \in \Gamma$.
3. Dado que ε es la palabra vacía y que se puede tener que $\delta(q, \varepsilon, z) = \{(q', zz)\}$, entonces es posible, por ejemplo, que un APND cambie de estado y apile un símbolo $z \in \Gamma$ sin que ocurra ninguna entrada.
4. Como δ no necesariamente es una función, entonces puede ocurrir que existan ternas (q, a, z) tales que:
 - a) $\delta(q, a, z)$ no existe y el APND se detiene.
 - b) $\delta(q, a, z) \in Q \times \Gamma^*$ y $\overline{\delta(q, a, z)} > 1$, es decir, $\delta(q, a, z)$ tiene más de una imagen, por lo tanto, la transición la realizará el APND de un modo no determinista.
5. Si $q_j \in A$, esto es, q_j es un estado de aceptación y $\delta(q, a, z) = \{(q_j, \beta)\}$, entonces el autómata se detiene, es decir, un APND se detiene toda vez que realice una transición a un estado de aceptación.

Ejemplo 5.1. Sea $\mathcal{A}_{PND} = < Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 >$ un APND, donde $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{c, d\}$, $\Gamma = \{A, B\}$, q_0 es el estado inicial, $A = \{q_2\}$, $z_0 = B$ y la relación de transición δ la definimos mediante la tabla de transiciones indicada por la tabla 5.1.

δ	(c, A)	(d, A)	(c, B)	(d, B)	(ε, A)	(ε, B)
q_0	(q_1, BA)	(q_2, AA)		(q_1, BB)	(q_2, A)	(q_2, B)
q_1	$\{(q_0, \varepsilon), (q_2, BA)\}$		(q_0, ε)	(q_1, BB)	(q_0, AA)	(q_0, B)
q_2				(q_1, BB)		(q_2, ε)

Cuadro 5.1: Tabla de transición δ para el ejemplo 5.1.

A partir de la tabla de transición 5.1, podemos inferir los siguientes aspectos:

1. δ depende del estado actual (las filas), del símbolo de entrada y del símbolo actual en la cima de la pila (etiquetas de columnas).
2. En el cruce de filas y columnas se especifica un estado siguiente y una acción de la pila para los símbolos actuales de la entrada y de la cima de la pila. Así, estando en el estado q_1 , con entrada c y con B como símbolo actual en la cima de la pila, el autómata hace transición al estado q_0 y desempila a B .
3. Igualmente, puede verse de la tabla que no necesariamente hay transición del autómata para todas las posibles ternas de $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$. De allí que si el autómata pasa

a un estado q_i para el cual no se programó su transición, entonces el autómata se detiene. Por ejemplo, cuando el autómata pasa al estado q_2 con A en la cima de la pila, observamos que no hay transición programada.

4. $\delta(q_1, c, B) = \{(q_0, \varepsilon)\}$, indica que el autómata estando en el estado q_1 , con c como entrada y B en la cima de la pila, desempila la B y pasa al estado q_0 .
5. $\delta(q_1, c, A) = \{(q_0, \varepsilon), (q_2, BA)\}$, indica que hay dos posibles respuestas, y que una de ellas será seleccionada de modo no determinista: estando en q_1 y con (c, A) , el autómata, o bien desempila a A y pasa al estado q_0 , o bien empila a B y pasa el estado q_2 (estado de aceptación).
6. Observemos que cuando el autómata está en el estado q_1 con (d, B) , éste permanece en ese estado empilando signos B ; y cuando está en q_1 , con (c, B) pasa a q_0 y desempila a B .
7. Sea la palabra $dcdc \in \Sigma^*$; entonces, dado que $\delta(q_0, d, B) = \{(q_1, BB)\}$ tenemos que

$$\begin{aligned}\delta(q_0, dcdc, B) &= \delta(q_1, cdc, BB) \\ &= \delta(q_0, dc, B) \\ &= \delta(q_1, c, BB) \\ &= \delta(q_0, \varepsilon, B) \\ &= \delta(q_2, \varepsilon, B) \\ &= (q_2, \varepsilon).\end{aligned}$$

Luego, el autómata se detiene en el estado de aceptación q_2 , con la pila vacía. No es posible ningún movimiento.

Eso significa que el autómata para computar sobre una palabra $\alpha \in \Sigma^*$ parte de la configuración inicial $\delta(q_0, s_1, B)$, donde q_0 es el estado inicial, B es el símbolo inicial de la pila y s_1 es el símbolo más a la izquierda de la palabra α .

5.2. Autómatas de pila y reconocedores

De modo similar a los autómatas que hemos estudiado en capítulos anteriores, un APND genera cierto lenguaje y puede servir de reconocedor de algunos lenguajes.

Definición 5.2 (Lenguaje generado por un APND). Sea un autómata de pila no determinista $\mathcal{A}_{PND} = < Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 >$. El lenguaje generado por \mathcal{A}_{PND} , denotado por $\mathcal{L}(\mathcal{A}_{PND})$, está definido por el conjunto:

$$\mathcal{L}(\mathcal{A}_{PND}) = \{\alpha \in \Sigma^* \mid (q_0, \alpha, z_0) \vdash_{\delta} (q, \varepsilon, \theta); q \in A \wedge \theta \in \Gamma^*\}.$$

La expresión $(q_0, \alpha, z_0) \vdash_{\delta} (q, \varepsilon, \theta)$ significa que mediante la relación δ , partiendo de la configuración inicial (q_0, α, z_0) , el autómata realiza sucesivas computaciones hasta que α se agote y termina en la configuración final (q, ε, θ) ; donde, q es un estado de aceptación y $\theta \in \Gamma^*$ puede o no ser vacía, esto es, el autómata puede o no terminar con la pila vacía. Si la pila está vacía el autómata necesariamente se detiene.

Definición 5.3 (APND como reconocedor de lenguajes). Sea \mathcal{L} un lenguaje. Decimos que un APND \mathcal{A}_{PND} es un reconocedor de \mathcal{L} , si y sólo si, $\mathcal{L} \subseteq \mathcal{L}(\mathcal{A}_{PND})$.

Observación. Con el símbolo \vdash_1 , indicamos la configuración inmediata.

Ejemplo 5.2. En el ejemplo 5.1 (pág. 167), el autómata:

1. Reconoce la palabra d :

$$\begin{aligned} (q_0, d, B) &\vdash_1 (q_1, \varepsilon, BB) \\ &\vdash_1 (q_0, \varepsilon, BB) \\ &\vdash_1 (q_2, \varepsilon, BB) \\ &\vdash_1 (q_2, \varepsilon, B) \\ &\vdash_1 (q_2, \varepsilon, \varepsilon), \end{aligned}$$

donde q_2 es un estado de aceptación.

2. No reconoce la palabra c , es decir, $(q_0, c, B) \vdash_1 ?$
3. Reconoce la palabra dc , esto es, $(q_0, dc, B) \vdash_{\delta} (q_2, \varepsilon, \varepsilon) :$

$$\begin{aligned} (q_0, dc, B) &\vdash_1 (q_1, c, BB) \\ &\vdash_1 (q_0, \varepsilon, B) \\ &\vdash_1 (q_2, \varepsilon, B) \\ &\vdash_1 (q_2, \varepsilon, \varepsilon). \end{aligned}$$

4. De (2) y (3) se observa que la palabra dcc no es reconocida, pero la palabra dcd sí lo es.
5. El autómata reconoce el lenguaje $\mathcal{L}_0 = \{d^n \mid n \geq 0\}$.
 - a) $(q_0, d, B) \vdash (q_2, \varepsilon, \varepsilon)$. Reconoce a d^1 por (1).

b) Reconoce a d^2 :

$$\begin{aligned}
 (q_0, dd, B) &\vdash_1 (q_1, d, BB) \\
 &\vdash_1 (q_1, \varepsilon, BBB) \\
 &\vdash_1 (q_0, \varepsilon, BBB) \\
 &\vdash_1 (q_2, \varepsilon, BBB) \\
 &\vdash_1 (q_2, \varepsilon, BB) \\
 &\vdash_1 (q_2, \varepsilon, B) \\
 &\vdash_1 (q_2, \varepsilon, \varepsilon).
 \end{aligned}$$

c) De (5) se puede ver que reconoce las palabras d^3, d^4, \dots

d) Luego $\mathcal{L}_0 \subseteq \mathcal{L}(\mathcal{A}_{PND})$.

6. El autómata reconoce a: $d, dc, dcd, dc当地, dcdcd, dc当地dc当地, dc当地cdcd, \dots$

7. $\mathcal{L}(\mathcal{A}_{PND}) = \{(dc)^*d^n \mid n \geq 0\}$.

Ejemplo 5.3. Sea \mathcal{L}_1 un lenguaje sobre $\Sigma = \{a, b\}$, tal que \mathcal{L}_1 contiene igual cantidad de símbolos a y de símbolos b .

Nos interesa construir un APND que acepte o reconozca a \mathcal{L}_1 . Para tal efecto debemos poder contar las ocurrencias de los símbolos a y b en una palabra dada de Σ^* . Aquí podemos usar la idea de pila, simplemente empilando cuando leamos a y desempilando cuando leamos b , o viceversa.

Sea \mathcal{A}_{PND} un APND definido por la tabla de transiciones indicada por la tabla 5.2.

δ	(a, A)	(a, B)	(a, C)	(b, A)	(b, B)	(b, C)
q_0	(q_0, AA)	(q_0, ε)	(q_0, AC)	(q_0, ε)	(q_0, BB)	(q_0, BC)
q_1						

δ	(ε , A)	(ε , B)	(ε , C)
q_0			(q_1, C)
q_1			

Cuadro 5.2: Tabla de transición δ para el ejemplo 5.3.

Además, $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $\Gamma = \{A, B, C\}$, $A = \{q_1\}$ y $z_0 = C$.

Para computar sobre la palabra $\alpha \equiv bababaab$, el autómata \mathcal{A}_{PND} realiza el siguiente

proceso:

$$\begin{aligned}
 (q_0, bababaab, C) &\vdash_1 (q_0, ababaab, BC) \\
 &\vdash_1 (q_0, babaab, C), \text{ desempila} \\
 &\vdash_1 (q_0, abaab, BC), \text{ empila} \\
 &\vdash_1 (q_0, baab, C), \text{ desempila} \\
 &\vdash_1 (q_0, aab, BC), \text{ empila} \\
 &\vdash_1 (q_0, ab, C), \text{ desempila} \\
 &\vdash_1 (q_0, b, AC), \text{ empila} \\
 &\vdash_1 (q_0, \varepsilon, C), \text{ desempila} \\
 &\vdash_1 (q_1, \varepsilon, C), \text{ estado de aceptación,}
 \end{aligned}$$

debido a que no hay programada transición para la configuración (q_1, ε, C) , el autómata se detiene y $\alpha \in \mathcal{L}(\mathcal{A}_{PND})$. Luego, $\mathcal{L}(\mathcal{A}_{PND}) = \mathcal{L}_1$, por ende, \mathcal{A}_{PND} es un reconocedor de \mathcal{L}_1 . ¿Qué ocurriría en el proceso cuando la palabra no es reconocida por el autómata?

5.3. Lenguajes independientes del contexto y autómatas de pila

En esta sección nos ocuparemos de la tarea de establecer algunas conexiones entre los APND y los lenguajes independientes del contexto. Recordemos que un lenguaje independiente del contexto se corresponde con el lenguaje generado por una grámatica independiente del contexto, grámativas cuyas reglas de producción son de la forma $A \rightarrow \gamma; \gamma \neq \varepsilon$, donde $A \in \mathbf{N}$ (no terminal) y $\gamma \in (\mathbf{N} \cup \mathbf{T})^*$.

La primera conexión que nos interesa establecer está dada por el siguiente teorema, el cual establece que para todo lenguaje independiente del contexto siempre es posible construir un reconocedor que sea APND.

Teorema 5.1. *Si \mathcal{L} es un lenguaje independiente del contexto, entonces existe al menos un APND \mathcal{A}_{PND} tal que, $\mathcal{L}(\mathcal{A}_{PND}) = \mathcal{L}$.*

Demostración. La demostración de este teorema es esencialmente constructiva. Supongamos que tenemos una grámatica $\mathcal{G} = < \mathbf{N}, \mathbf{T}, \mathbf{P}, \mathbf{I} >$ independiente del contexto. Asumimos que $\mathcal{L} = \mathcal{L}(\mathcal{G})$, para alguna grámatica \mathcal{G} independiente del contexto. Se trata entonces de construir un APND \mathcal{A}_{PND} tal que $\mathcal{L}(\mathcal{A}_{PND}) = \mathcal{L}(\mathcal{G})$; esto es, el APND nos debe permitir validar el hecho de que todas las palabras de $\mathcal{L}(\mathcal{G})$ son reconocidas por el autómata. Sin perder generalidad, asumiremos que las palabras generadas por el autómata son derivaciones por la izquierda.

Pasemos ahora a construir dicho autómata. Supongamos que sólo necesitamos tres estados. Sea entonces $\mathcal{A}_{PND} = < Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 >$, donde:

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

$$Q = \{q_0, q_1, q_2\},$$

$$\Sigma = T,$$

$$\Gamma = N \cup T \cup I,$$

$$A = \{q_2\}.$$

La relación de transición interna δ , estará determinada por el siguiente tipo de reglas de transición:

R-1 Introducimos el símbolo inicial I en la pila, esto es $\delta(q_0, \varepsilon, z_0) = \{(q_1, Iz_0)\}$.

R-2 $\delta(q_1, \varepsilon, A) = \{(q_1, \alpha) \mid A \rightarrow \alpha\}$, para todo $A \in N$ (símbolos no terminales). Esto es, si el símbolo que está en la cima de la pila es un símbolo no terminal, entonces empilamos α (lado derecho de la producción $A \rightarrow \alpha$), conservando el mismo estado.

R-3 $\delta(q_1, x, x) = \{(q_1, \varepsilon)\}$ para todo $x \in \Sigma = T$ (símbolos terminales). Esto es, si el símbolo de entrada (x) y el símbolo de la cima de la pila son idénticos, entonces lo desempilamos, conservando el mismo estado.

R-4 $\delta(q_1, \varepsilon, z_0) = \{(q_2, z_0)\}$. Esto es, en la cima de la pila queda el símbolo inicial de la pila, el autómata en el estado de aceptación q_2 , y entonces, el autómata de detiene y acepta la palabra de $\mathcal{L}(\mathcal{G})$.

Ahora, es posible probar que si $\alpha \equiv x_1 \dots x_n$ es aceptada por este APND, entonces α es derivable de la gramática \mathcal{G} , esto es, $\alpha \in \mathcal{L}(\mathcal{G})$.

Si en el APND (\mathcal{APND}) antes construido tenemos que $(q_1, x, A\beta) \vdash_1 (q_1, x, \theta\beta)$, entonces podemos tener en \mathcal{G} que, $A \rightarrow \theta$; de allí que tengamos:

$$\begin{aligned}
 (q_1, x_1 \dots x_n, Iz_0) &\vdash_1 (q_1, x_1 \dots x_n, x_1\theta z_0) \\
 &\vdash_1 (q_1, x_2 \dots x_n, \theta z_0) \\
 &\vdots \\
 &\vdash_1 (q_1, x_n, x_n z_0) \\
 &\vdash_1 (q_1, \varepsilon, z_0) \\
 &\vdash_1 (q_2, \varepsilon, z_0).
 \end{aligned}$$

Luego, aplicando las reglas de \mathcal{G} a lo anterior, tenemos que:

$$I \rightarrow x_1\theta \wedge x_1\theta \rightarrow x_1x_2\theta \wedge \dots \wedge x_1 \dots x_{n-1}\theta \rightarrow x_1 \dots x_n,$$

luego $I \rightarrow^* x_1 \dots x_n$, de allí que $\alpha \equiv x_1 \dots x_n$ que fue aceptada por el APND es también derivable de las reglas de producción de \mathcal{G} .

Recíprocamente, si $I \rightarrow^* x_1 \dots x_n$, entonces se tiene en \mathcal{G} :

$$I \rightarrow x_1\theta \wedge x_1\theta \rightarrow x_1x_2\theta \wedge \dots \wedge x_1 \dots x_{n-1}\theta \rightarrow x_1 \dots x_n,$$

de allí, se sigue que del APND aquí construido:

$$\begin{aligned}
 (q_1, x_1 \dots x_n, \mathbf{I}z_0) &\vdash_1 (q_1, x_1 \dots x_n, x_1\theta z_0) \\
 &\vdash_1 (q_1, x_2 \dots x_n, \theta z_0) \\
 &\vdots \\
 &\vdash_1 (q_1, x_n, x_n z_0) \\
 &\vdash_1 (q_1, \varepsilon, z_0) \\
 &\vdash_1 (q_2, \varepsilon, z_0);
 \end{aligned}$$

luego, el APND reconoce α , esto es, $\alpha \in \mathcal{L}(\mathcal{A}_{PND})$.

□

Ejemplo 5.4. El siguiente ejemplo nos ilustra el proceso de construcción de un APND para la gramática $\mathcal{G} = \langle N, T, P, I \rangle$, donde, $N = \emptyset$, $T = \{a, b, c\}$, $P = \{\mathbf{I} \rightarrow a\mathbf{I}a, \mathbf{I} \rightarrow b\mathbf{I}b, \mathbf{I} \rightarrow c\}$. El lenguaje independiente del contexto a reconocer es $\mathcal{L}(\mathcal{G}) = \{\alpha c \alpha^{-n} \mid \alpha \in \{a, b\}^*\}$ (α^{-n} es la palabra inversa de α).

Construyamos el APND que corresponde a $\mathcal{L}(\mathcal{G})$ aplicando las reglas de construcción señaladas en la demostración del teorema 5.1 (pág. 171), así:

$$\delta(q_0, \varepsilon, z_0) = \{(q_1, \mathbf{I}z_0)\}, \quad (\text{R-1})$$

$$\delta(q_1, \varepsilon, \mathbf{I}) = \{(q_1, a\mathbf{I}a), (q_1, b\mathbf{I}b), (q_1, c)\}, \quad (\text{R-2})$$

$$\delta(q_1, a, a) = \delta(q_1, b, b) = \delta(q_1, c, c) = \{(q_1, \varepsilon)\}, \quad (\text{R-3})$$

$$\delta(q_1, \varepsilon, z_0) = \{(q_2, z_0)\}.$$

Mostremos que, por ejemplo, la palabra $abcba \in \mathcal{L}(\mathcal{G})$ es reconocida por el autómata, así:

$$\begin{aligned}
 (q_0, abcba, z_0) &\vdash_1 (q_1, abcba, \mathbf{I}z_0) \\
 &\vdash_1 (q_1, abcba, a\mathbf{I}az_0) \\
 &\vdash_1 (q_1, bcba, \mathbf{I}az_0), \quad \text{desempila } a \\
 &\vdash_1 (q_1, bcba, b\mathbf{I}baz_0) \\
 &\vdash_1 (q_1, cba, \mathbf{I}baz_0), \quad \text{desempila } b \\
 &\vdash_1 (q_1, cba, cbaz_0) \\
 &\vdash_1 (q_1, ba, baz_0), \quad \text{desempila } c \\
 &\vdash_1 (q_1, a, az_0), \quad \text{desempila } b \\
 &\vdash_1 (q_1, \varepsilon, z_0), \quad \text{desempila } a \\
 &\vdash_1 (q_2, \varepsilon, z_0),
 \end{aligned}$$

luego, se reconoce $abcb$.

La segunda conexión que queremos esbozar hace relación con la pregunta de si todo lenguaje que sea reconocido por APND es un lenguaje independiente del contexto. Antes de presentar el teorema que establece esta conexión, introducimos primero una definición y un teorema cuya demostración dejamos como ejercicio.

Definición 5.4 (Lenguaje reconocido con una pila vacía). Sea \mathcal{A}_{PND} un autómata de pila no determinista. El siguiente conjunto define el lenguaje reconocido por la pila vacía del autómata:

$$\mathcal{P}(\mathcal{A}_{PND}) = \{\alpha \in \Sigma^* \mid (q_0, \alpha, z_0) \vdash_{\delta} (q, \varepsilon, \varepsilon)\},$$

esto es, $\mathcal{P}(\mathcal{A}_{PND})$ es el conjunto de entradas α que generan una sucesión de computaciones sucesivas que se inician en el estado inicial y culminan en un estado cualquiera q con la pila vacía.

Teorema 5.2. Si \mathcal{A}_{PND} es un APND tal que $\mathcal{P}(\mathcal{A}_{PND}) \neq \mathcal{L}(\mathcal{A}_{PND})$, entonces existe un APND \mathcal{A}_0 tal que $\mathcal{L}(\mathcal{A}_{PND}) = \mathcal{P}(\mathcal{A}_0)$.

Demostración. Ejercicio 5.12. □

Lo que afirma el teorema anterior es que podemos hallar un APND \mathcal{A}_0 tal que el lenguaje reconocido por la pila vacía de \mathcal{A}_0 , coincida con el lenguaje reconocido por un APND \mathcal{A}_{PND} . Este teorema tiene sentido pues estos dos lenguajes no siempre son iguales para un autómata dado, como lo indica el siguiente ejemplo.

Ejemplo 5.5. Considérese el caso de un APND \mathcal{A}_{PND} cuya relación de transición sea: $\delta(q_0, c, z_0) = (q_0, cz_0)$, $\delta(q_0, a, z_0) = (q_1, \varepsilon)$, $\delta(q_0, c, c) = (q_2, \varepsilon)$. Aquí, $\mathcal{P}(\mathcal{A}_{PND}) = \{a\}$ y $\mathcal{L}(\mathcal{A}_{PND}) = \{cc\}$; donde el conjunto de estados de aceptación es $A = \{q_2\}$.

El hecho básico del teorema 5.2 (pág. 174) consiste en construir un APND \mathcal{A}_0 de modo tal que el conjunto de estados de aceptación sea un conjunto unitario, estado al cual \mathcal{A}_0 alcanzaría sólo con la pila vacía.

Presentamos ahora el teorema que establece la segunda conexión entre los APND y los lenguajes independientes al contexto, mencionada anteriormente.

Teorema 5.3. Si \mathcal{L} es un lenguaje reconocido por APND, entonces \mathcal{L} es un lenguaje independiente del contexto.

Demostración. La demostración de este teorema es esencialmente constructiva. Se trata de construir, a partir de una APND dado, una gramática independiente del contexto que le sea correspondiente, y de modo tal que una derivación en esta gramática simule los movimientos que el autómata haría para reconocer la palabra derivada.

Sea $\mathcal{A}_{PND} = \langle Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 \rangle$, un APND. Para hallar una gramática independiente del contexto cuyas reglas interpreten o simulen el comportamiento de \mathcal{A}_{PND} , se realiza el siguiente proceso constructivo:

1. Hallamos un APND \mathcal{A}_0 equivalente al autómata dado, tal que \mathcal{A}_0 sólo contenga un estado de aceptación q_f y llegue a él con la pila vacía (teorema 5.2 (pág. 174)).
2. Elegimos como símbolo inicial de la gramática $I = [q_0 z_0 q_f]$. La intepretación se realiza en el paso siguiente.
3. Los elementos de N o símbolos no terminales se generan como expresiones de la forma $[qAq']$, donde $A \in \Gamma$ y $q, q' \in Q$. Interpretamos que $[qAq'] \rightarrow \alpha$, simula la acción del autómata, consistente en (estando en el estado actual q) desempilar a A y moverse al estado q' mientras consume la entrada α . Veamos, entonces, cómo se generan las reglas de producción para la gramática correspondiente al APND dado.
 - a) Si $(q_j, \varepsilon) \in \delta(q_i, x, A)$ entonces se tiene la regla $[q_i A q_j] \rightarrow x$.
 - b) Si $(q_j, BC) \in \delta(q_i, x, A)$ (la entrada x desempila a A , pero el autómata, en su proceso, debe desempilar a B y a C , para poder llegar a q_f con la pila vacía), entonces, se incluyen todas las producciones de la forma $[q_i A q_m] \rightarrow x[q_i B q_n] [q_n C q_m]$, donde q_m y q_n son cualquier estado del autómata.

□

Ejemplo 5.6. Consideremos el autómata $\mathcal{A}_{PND} = \langle Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 \rangle$ tal que $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $\Gamma = \{A, z_0\}$, $A = \{q_2\}$ y la relación de transición δ definida por:

$$\delta(q_0, a, z_0) = \{(q_0, Az_0)\}, \quad (5.1a)$$

$$\delta(q_0, a, A) = \{(q_0, AA)\}, \quad (5.1b)$$

$$\delta(q_0, b, A) = \{(q_1, \varepsilon)\}, \quad (5.1c)$$

$$\delta(q_1, b, A) = \{(q_1, \varepsilon)\}, \quad (5.1d)$$

$$\delta(q_1, \varepsilon, A) = \{(q_1, \varepsilon)\}, \quad (5.1e)$$

$$\delta(q_1, \varepsilon, z_0) = \{(q_2, \varepsilon)\}. \quad (5.1f)$$

Este autómata satisface la condición de que sólo hace transición al estado de aceptación q_2 cuando la pila esté vacía. Además, en la demostración del teorema 5.2 (pág. 174) se verifica que las transiciones del autómata equivalente son de la forma $\delta(q_i, x, A) = \{c_1, c_2, \dots, c_k\}$, donde las c_i son de la forma $c_i = (q, \varepsilon)$ ó $c_i = (q, BC)$ (sólo se empila o desempila con un único símbolo de Γ).

1. Símbolo inicial de la gramática: $I = [q_0 z_0 q_2]$.

2. Reglas de producción de la gramática correspondientes a \mathcal{A}_{PND} :

- a) Las reglas correspondientes a las transiciones (5.1c), (5.1d), (5.1e) y (5.1f) son del tipo (3a) (teorema 5.3 (pág. 174)). Luego, tenemos de ellas:

Para (5.1c): $[q_1 A q_1] \rightarrow b$,
 para (5.1d): $[q_1 A q_1] \rightarrow b$,
 para (5.1e): $[q_1 A q_1] \rightarrow \varepsilon$,
 para (5.1f): $[q_1 z_0 q_2] \rightarrow \varepsilon$.

- b) Las reglas correspondientes a las transiciones (5.1a) y (5.1b) son del tipo (3b) (teorema 5.3 (pág. 174)). Luego, tenemos de ellas:

Para (5.1a):

$$\begin{aligned} [q_0 z_0 q_0] &\rightarrow a [q_0 A q_0] [q_0 z_0 q_0] \\ &\rightarrow a [q_0 A q_1] [q_1 z_0 q_0] \\ &\rightarrow a [q_0 A q_2] [q_2 z_0 q_0], \\ [q_0 z_0 q_1] &\rightarrow a [q_0 A q_0] [q_0 z_0 q_1] \\ &\rightarrow a [q_0 A q_1] [q_1 z_0 q_1] \\ &\rightarrow a [q_0 A q_2] [q_2 z_0 q_1], \\ [q_0 z_0 q_2] &\rightarrow a [q_0 A q_0] [q_0 z_0 q_2] \\ &\rightarrow a [q_0 A q_1] [q_1 z_0 q_2] \\ &\rightarrow a [q_0 A q_2] [q_2 z_0 q_2]. \end{aligned}$$

Para (5.1b):

$$\begin{aligned} [q_0 A q_0] &\rightarrow a [q_0 A q_0] [q_0 A q_0] \\ &\rightarrow a [q_0 A q_1] [q_1 A q_0] \\ &\rightarrow a [q_0 A q_2] [q_2 A q_0], \\ [q_0 A q_1] &\rightarrow a [q_0 A q_0] [q_0 A q_1] \\ &\rightarrow a [q_0 A q_1] [q_1 A q_1] \\ &\rightarrow a [q_0 A q_2] [q_2 A q_1], \\ [q_0 A q_2] &\rightarrow a [q_0 A q_0] [q_0 A q_2] \\ &\rightarrow a [q_0 A q_1] [q_1 A q_2] \\ &\rightarrow a [q_0 A q_2] [q_2 A q_2]. \end{aligned}$$

Ejemplo 5.7. Indiquemos el modo como la derivación en la gramática simula el comportamiento del reconocedor.

Por ejemplo, la cadena $aaabbb$ es reconocida por el autómata anterior. Veamos la secuencia de configuraciones correspondientes.

$$\begin{aligned}
 (q_0, aaabbb, z_0) &\vdash_1 (q_0, aabbb, Az_0), & \text{de regla (5.1a)} \\
 &\vdash_1 (q_0, abbb, AAz_0), & \text{de regla (5.1b)} \\
 &\vdash_1 (q_0, bbb, AAAz_0), & \text{de regla (5.1b)} \\
 &\vdash_1 (q_0, bb, AAz_0), & \text{de regla (5.1c)} \\
 &\vdash_1 (q_0, b, Az_0), & \text{de regla (5.1c)} \\
 &\vdash_1 (q_0, \varepsilon, z_0), & \text{de regla (5.1d)} \\
 &\vdash_1 (q_0, \varepsilon, \varepsilon), & \text{de regla (5.1f).}
 \end{aligned}$$

Luego, termina en el estado de aceptación q_2 con la pila vacía.

Ahora, la derivación en la gramática que construimos en el ejemplo 5.6 (pág. 175) para la palabra $aaabbb$, viene dada por:

$$\begin{aligned}
 [q_0z_0q_0] &\rightarrow a [q_0Aq_0] [q_0z_0q_0] \\
 &\rightarrow aa [q_0Aq_1] [q_1Aq_1] [q_1z_0q_2] \\
 &\rightarrow aaa [q_0Aq_1] [q_1Aq_1] [q_1Aq_1] [q_1z_0q_2] \\
 &\rightarrow aaab [q_1Aq_1] [q_1Aq_1] [q_1z_0q_2] \\
 &\rightarrow aaabb [q_1Aq_1] [q_1z_0q_2] \\
 &\rightarrow aaabbb [q_1z_0q_2] \\
 &\rightarrow aaabb\epsilon \\
 &\rightarrow aaabbb
 \end{aligned}$$

Para cerrar este capítulo sobre autómatas de pila, consideramos importante realizar las siguientes observaciones:

1. Los autómatas de pila nos sirven entonces para probar, no obstante cierta complejidad, si un determinado lenguaje es o no independiente del contexto. Incluso se puede afirmar que, en la mayoría de los casos, es más sencillo construir el autómata que diseñar la gramática independiente del contexto.
2. Los autómatas de pila pueden ser usados como métodos más expeditos para probar o verificar algunas propiedades específicas de los lenguajes independientes del contexto (como lo ejemplifica la próxima sección de ejercicios).

5.4. Ejercicios

Ejercicio 5.1. Dada la siguiente tabla de transición para un APND que designamos por \mathcal{A}_{PND} donde $Q = \{q_0, q_1, q_2, q_3\}$, $A = \{q_3\}$ y $z_0 = A$:

δ	(a, A)	(b, A)	(ε , A)	(a, B)	(b, B)	(ε , B)
q_0	$\{(q_1, BA), (q_3, \varepsilon)\}$		(q_3, ε)			
q_1				(q_1, BB)	(q_2, ε)	
q_2						(q_2, ε)
q_3						

1. Analizar el comportamiento del autómata \mathcal{A}_{PND} , cuando de halla en los estados q_0, q_1 y q_2 .
2. Analizar el comportamiento del autómata \mathcal{A}_{PND} toda vez que entra b , con B en la cima de la pila.
3. ¿Qué significa el hecho de que $\delta(q_0, a, A) = \{(q_1, BA), (q_3, \varepsilon)\}$?
4. Verificar si la palabra $abbabb$ pertenece a $\mathcal{L}(\mathcal{A}_{PND})$.
5. Determinar $\mathcal{L}(\mathcal{A}_{PND})$.

Ejercicio 5.2. Sea $\mathcal{A}_{PND} = < Q, \Sigma \cup \{\varepsilon\}, \Gamma, \delta, q_0, A, z_0 >$ un APND, donde $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $\Gamma = \{A, B, Z\}$, $A = \{q_1\}$, $z_0 = Z$ y la relación de transición δ definida por:

$$\begin{aligned}\delta(q_0, \varepsilon, Z) &= \{(q_1, Z)\}, \\ \delta(q_0, a, Z) &= \{(q_0, AZ)\}, \\ \delta(q_0, b, Z) &= \{(q_0, BZ)\}, \\ \delta(q_0, a, A) &= \{(q_0, AA)\}, \\ \delta(q_0, b, A) &= \{(q_0, \varepsilon)\}, \\ \delta(q_0, a, B) &= \{(q_0, \varepsilon)\}, \\ \delta(q_0, b, B) &= \{(q_0, BB)\}.\end{aligned}$$

1. Elaborar una tabla de transición para \mathcal{A}_{PND} .
2. Verificar que la palabra $babaabab\varepsilon \in \mathcal{L}(\mathcal{A}_{PND})$.
3. Hallar $\mathcal{L}(\mathcal{A}_{PND})$.

Ejercicio 5.3. Para la gramática presentada en el ejemplo 5.4 (pág. 173), verificar si la palabra $aabcaab \notin \mathcal{L}(\mathcal{G})$ es reconocida por el APND presentado en ese ejemplo (sugerencia: en un momento se obtiene la configuración $(q_1, aab, baaz_0)$. ¿Existe regla de transición para seguir?)

Ejercicio 5.4. Sea $\#(\alpha, a)$ el número de símbolos a en la palabra α . Diseñar un APND tal que reconozca el lenguaje \mathcal{L} donde $\mathcal{L} = \{\beta \in \{a, b\}^* \mid \#(\beta, a) = \#(\beta, b)\}$.

Ejercicio 5.5. Diseñar un APND que sea un reconocedor para $\mathcal{L} = \{c^n b^n \mid n \geq 0\}$.

Ejercicio 5.6. Diseñar un APND que sea un reconocedor para $\mathcal{L} = \{a^m b^{2m} \mid m \geq 0\}$.

Ejercicio 5.7. Dado el autómata APND, definido por la siguiente tabla donde $A = \{q_2\}$:

δ	(a, z_0)	(a, b)	(b, a)	(b, b)
q_0	$\{(q_2, a), (q_1, \varepsilon)\}$			
q_1		(q_2, ε)	(q_1, b)	(q_1, b)
q_2				

1. Hallar el lenguaje generado por el autómata.
2. ¿Qué lenguaje será reconocido por el autómata?
3. ¿Es la palabra a^3b^4 reconocida por este autómata?
4. Si cambiamos $A = \{q_2\}$ por $A' = \{q_1, q_2\}$, ¿cuál será el lenguaje generado?

Ejercicio 5.8. Dada la gramática \mathcal{G} definida por las siguientes producciones:

$$\begin{aligned} I &\rightarrow aAA, \\ A &\rightarrow bI, \\ A &\rightarrow aI, \\ A &\rightarrow a. \end{aligned}$$

1. Hallar $\mathcal{L}(\mathcal{G})$.
2. Diseñar un APND \mathcal{A}_{PND} tal que $\mathcal{L}(\mathcal{A}_{PND}) = \mathcal{L}(\mathcal{G})$.
3. Mostrar que $a^3b^3 \in \mathcal{L}(\mathcal{A}_{PND})$.
4. Producir a^3b^3 mediante \mathcal{G} .

Ejercicio 5.9. Hallar una gramática independiente del contexto que sea correspondiente al siguiente APND donde $A = \{q_3\}$, $\Gamma = \{A, B, z_0\}$:

δ	(a, z_0)	(ε, z_0)	(a, A)	(b, z_0)	(b, A)
q_0	(q_0, Az_0)		(q_3, ε)		(q_1, ε)
q_1			(q_2, ε)		
q_2					
q_3		(q_0, Az_0)			

Ejercicio 5.10. Diseñar un APND que sea un reconocedor del lenguaje generado por la gramática indicada por las siguientes reglas:

$$\begin{aligned} I &\rightarrow aABB, \\ I &\rightarrow aAA, \\ A &\rightarrow aBB, \\ A &\rightarrow \varepsilon, \\ B &\rightarrow bBB, \\ B &\rightarrow a. \end{aligned}$$

Ejercicio 5.11. Diseñar una gramática que sea correspondiente con el lenguaje generado por el siguiente APND donde $Q = \{q_0, q_1\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, z_0\}$, $A = \{q_1\}$:

$$\begin{aligned} \delta(q_0, a, z_0) &= \{(q_0, az_0)\}, \\ \delta(q_0, b, a) &= \{(q_0, aa)\}, \\ \delta(q_0, a, a) &= \{(q_1, \varepsilon)\}. \end{aligned}$$

¿Qué tipo de gramática se obtuvo?, ¿por qué?

Ejercicio 5.12. Demostrar el teorema 5.2 (pág. 174).

Ejercicio 5.13. Dada las siguientes transiciones que definen una cierta relación de transición:

$$\begin{aligned} \delta(q_0, a, z_0) &= \{(q_0, az_0)\}, \\ \delta(q_0, a, A) &= \{(q_0, A)\}, \\ \delta(q_0, b, A) &= \{(q_1, \varepsilon)\}, \\ \delta(q_1, \varepsilon, z_0) &= \{(q_3, \varepsilon)\}. \end{aligned}$$

1. Determine el mínimo Q y el mínimo Σ .
2. ¿Cuál es el conjunto A ?, ¿por qué?
3. Hallar el lenguaje generado por este autómata.
4. ¿El autómata cumple las condiciones del teorema 5.3 (pág. 174)?
5. Investigar qué debe hacerse para hallar un autómata equivalente que las cumpla.

6. Hallar una gramática independiente del contexto que genere el mismo lenguaje que este último autómata.

Ejercicio 5.14. Probar que si \mathcal{L}_0 es un lenguaje independiente del contexto y \mathcal{L}_1 es regular, entonces $\mathcal{L}_0 \cap \mathcal{L}_1$ es un lenguaje independiente del contexto (sugerencia: considere un APND que sea un reconocedor de \mathcal{L}_0 y un autómata de estado finito que sea un reconocedor de \mathcal{L}_1 . Mediante una adecuada combinación de ellos, construya un APND que sea un reconocedor de $\mathcal{L}_0 \cap \mathcal{L}_1$).

5.5. Notas Bibliográficas

Los autómatas de pila son presentados en [17, 20]. El ejemplo 5.4 (pág. 173) y el ejercicio 5.14 (pág. 181) fueron tomados de [20]. La demostración del teorema 5.3 (pág. 174) es un reordenamiento de la demostración presentada por [20].

Capítulo 6

Complejidad algorítmica

Una vez establecido qué problemas, funciones o lenguajes son computables, la teoría de la complejidad algorítmica clasifica estos objetos de acuerdo con la cantidad de recursos necesarios para computarlos. En el contexto de la complejidad algorítmica es necesario precisar los siguientes elementos: (i) Objetos a computar, (ii) Modelo(s) de computación empleado(s), (iii) Modos de computación, (iv) Recursos a acotar y (v) Funciones de complejidad.

Iniciamos nuestro trabajo en modo de computación determinista y en un modelo de computación denominado máquinas de Turing k -cintas.

6.1. Máquinas de Turing k -cintas

La idea intuitiva de una máquina de Turing k -cintas es la de una máquina de Turing que opera simultáneamente con k cintas de trabajo, en donde existe una cabeza de lectura-escritura para cada cinta.

Definición 6.1 (Máquina de Turing k -cintas). Definimos formalmente una máquina de Turing determinista k -cintas, para $k > 1$, mediante la estructura matemática $\mathcal{MT} = \langle Q, \Sigma, M, I \rangle$, donde:

Q, Σ y M son los mismos conjuntos que para una máquina de Turing.

I : Es una función definida de $Q \times \Sigma^k$ en $\Sigma^k \times M^k \times Q \cup \{q_Y, q_N\}$.

Al definir I como un conjunto *finito* de instrucciones $I = \{i_0, i_1, i_2, \dots, i_p\}$, cada i_j es una $(3k + 2)$ -tupla de la forma: $q_m \ s_{m_i} \ \dots \ s_{m_k} \ s_{n_1} \ \dots \ s_{n_k} \ m_1 \ \dots \ m_k \ q_n$, donde $q_m \in Q$, $q_n \in Q \cup \{q_Y, q_N\}$, $s_{m_i}, s_{n_i} \in \Sigma$ y $m_i \in M$; además los estados $\{q_Y, q_N\} \notin Q$ (estos estados serán usados para la definición de lenguajes recursivos y lenguajes recursivamente enumerables presentadas a continuación).

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

6.2. Lenguajes recursivos y lenguajes recursivamente enumerables

Los objetos para los cuales se van a definir las cotas en los recursos son los lenguajes formales.

Definición 6.2 (Lenguaje recursivamente enumerable). Sea Σ una alfabeto de una máquina de Turing k-cinta \mathcal{MT}_k y sea $\mathcal{L} \subseteq (\Sigma - \{\square\})^*$ un lenguaje. Se dice que la máquina \mathcal{MT}_k acepta el lenguaje \mathcal{L} si para todo $\alpha \in (\Sigma - \{\square\})^*$ se cumple:

1. Si $\alpha \in \mathcal{L}$, entonces, la máquina \mathcal{MT}_k se detiene en el estado q_Y .
2. Si $\alpha \notin \mathcal{L}$, entonces, la máquina no se detiene en q_Y . Esto permite tres posibilidades:
 - a) \mathcal{MT}_k se detiene en el estado q_N .
 - b) \mathcal{MT}_k se detiene en un estado diferente a q_Y y a q_N .
 - c) \mathcal{MT}_k no se detiene.

Así, si existe una \mathcal{MT}_k que acepte el lenguaje \mathcal{L} , entonces se dice que el lenguaje \mathcal{L} es recursivamente enumerable.

Una clase particular de lenguajes recursivamente enumerables son los lenguajes recursivos en los que la máquina de Turing siempre se detiene.

Definición 6.3 (Lenguaje recursivo). Sea Σ una alfabeto de una máquina de Turing k-cinta \mathcal{MT}_k y sea $\mathcal{L} \subseteq (\Sigma - \{\square\})^*$ un lenguaje. Se dice que la máquina \mathcal{MT}_k decide el lenguaje \mathcal{L} si para todo $\alpha \in (\Sigma - \{\square\})^*$ se cumple:

1. Si $\alpha \in \mathcal{L}$ entonces la máquina \mathcal{MT}_k se detiene en el estado q_Y .
2. Si $\alpha \notin \mathcal{L}$ entonces la máquina \mathcal{MT}_k se detiene en el estado q_N .

Así, si existe \mathcal{MT}_k que decida el lenguaje \mathcal{L} , entonces, se dice que el lenguaje \mathcal{L} es recursivo.

Enunciamos sin demostración el siguiente teorema, debido a que es una instancia particular del teorema 2.8 (pág. 77).

Teorema 6.1. *Todo lenguaje recursivo es una lenguaje recursivamente enumerable.*

Ejemplo 6.1. Sea \mathcal{L} un lenguaje palíndromo, es decir, para toda $\alpha \in \mathcal{L}$, $\alpha \equiv \alpha^{-1}$. Se presenta el comportamiento de la máquina de Turing 2-cintas \mathcal{MT}_2 que acepta a \mathcal{L} , dejando los detalles de construcción de la máquina al lector (ejercicio 6.1).

Al inicio la máquina tiene la palabra $\beta \equiv a_1a_2 \dots a_{m-1}a_ma_ma_{m-1} \dots a_2a_1$ en la cinta₁, como lo indica la tabla 6.1.

A continuación la máquina copia la palabra β en la cinta₂, como lo indica la tabla 6.2.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

cinta ₁	a ₁	a ₂	...	a _{m-1}	a _m	a _{m-1}	...	a ₁	a ₂
cinta ₂	↑			

Cuadro 6.1: Aceptación para \mathcal{L} palíndromo por \mathcal{MT}_2 (1).

cinta ₁	a ₁	a ₂	...	a _{m-1}	a _m	a _{m-1}	...	a ₂	a ₁
cinta ₂	a ₁	a ₂	...	a _{m-1}	a _m	a _{m-1}	...	a ₂	a ₁

Cuadro 6.2: Aceptación para \mathcal{L} palíndromo por \mathcal{MT}_2 (2).

Después la máquina mueve la cabeza₁ a la posición inicial como lo indica la tabla 6.3.

Después, la máquina compara la palabra $a_1a_2\dots a_{m-1}a_ma_m a_{m-1}\dots a_2a_1$ de la cinta₁, con la palabra $a_1a_2\dots a_{m-1}a_ma_ma_{m-1}\dots a_2a_1$ de la cinta₂. Esta comparación se realiza moviendo la cabeza₁ a la derecha y la cabeza₂ a la izquierda a medida que se van leyendo los símbolos, como lo indica la tabla 6.4.

Finalmente, si la comparación es exitosa, la máquina pasa al estado q_Y y se detiene; de lo contrario, la máquina pasa al estado q_N y se detiene.

6.3. Complejidad temporal determinista

El primer recurso de computación que vamos a analizar es el tiempo.

Definición 6.4 (Complejidad temporal para \mathcal{MT}_k). Sea \mathcal{MT}_k una máquina de Turing k -cintas. El tiempo requerido para computar una entrada $\alpha \in (\Sigma - \{\square\})^*$ es el número de pasos necesarios para que la máquina \mathcal{MT}_k se detenga, cuando al inicio en la cinta₁ está la palabra α .

Sea $T: \mathbb{N} \rightarrow \mathbb{N}$ una función. Se dice que la máquina \mathcal{MT}_k opera con límite temporal $T(n)$, si para toda $\alpha \in (\Sigma - \{\square\})^*$ el tiempo requerido para procesarla está dado por $T(l(\alpha))$, donde $l(\alpha)$ es la longitud de la palabra α .

Como es de esperarse que cualquier máquina de Turing k -cintas requiera al menos $n+1$ pasos para procesar una palabra de longitud n , entonces, en realidad el límite temporal $T(n)$ significa: el máximo entre $n+1$ y $T(n)$. Es decir, nuestra convención será

$$T(n) \equiv_{def} \max(n+1, 'T(n)').$$

Ejemplo 6.2. Sea una máquina de Turing k -cintas con límite temporal $T(n) = n \log_2 n$. De acuerdo con la convención $T(n) \equiv_{def} \max(n+1, 'T(n)')$ tenemos por ejemplo que $T(1) = \max(2, 0) = 2$, $T(2) = \max(3, 2) = 3$ y $T(3) = \max(4, 4 \log_2 3) = 4 \log_2 3$. Cuando seleccionar $n+1$ ó $n \log_2 n$, está sujeto a los comportamientos de ambas funciones, tal como lo indica la figura 6.1.

cinta ₁	a ₁	a ₂	...	a _{m-1}	a _m	a _{m-1}	...	a ₂	a ₁
	↑								

cinta ₂	a ₁	a ₂	...	a _{m-1}	a _m	a _{m-1}	...	a ₂	a ₁
	↑								

Cuadro 6.3: Aceptación para \mathcal{L} palíndromo por \mathcal{MT}_2 (3).

cinta ₁	a ₁	a ₂	...	a _{m-1}	a _m	a _{m-1}	...	a ₂	a ₁
	↑								

cinta ₂	a ₁	a ₂	...	a _{m-1}	a _m	a _{m-1}	...	a ₂	a ₁
	↑								

⋮

cinta ₁	a ₁	a ₂	...	a _{m-1}	a _m	a _{m-1}	...	a ₂	a ₁
	↑								

cinta ₂	a ₁	a ₂	...	a _{m-1}	a _m	a _{m-1}	...	a ₂	a ₁
	↑								

Cuadro 6.4: Decisión para \mathcal{L} palíndromo por \mathcal{MT}_2 (4).

6.4. Notación asintótica

En el contexto de la complejidad algorítmica no estamos preocupados por las constantes multiplicativas o aditivas de las funciones de complejidad $f(n)$, por lo tanto, para representar éstas usualmente se utiliza la notación asintótica. Para efectos de completitud se presentaran las tres notaciones asintóticas más usuales; sin embargo, sólo se trabajará con una de ellas.

La notación O denota una cota superior asintótica. Sea $g(n)$ una función, con $O(g(n))$ se denota el conjunto de funciones asintóticamente acotadas superiormente por $g(n)$.

Definición 6.6 (Notación 0). $O(g(n)) = \{f(n) \mid \text{existen constantes positivas } c \text{ y } n_0 \text{ tal que } 0 \leq f(n) \leq cg(n), \text{ para todo } n \geq n_0\}$.

La convención $f(n) = O(g(n))$, indica que $f(n) \in O(g(n))$. La figura 6.2 muestra el significado de la notación O .

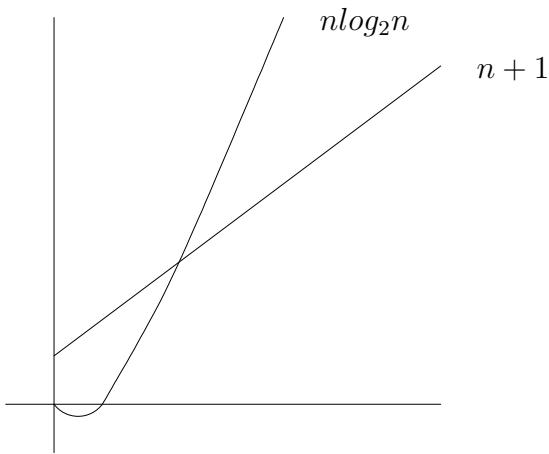


Figura 6.1: $T(n) = \max(n + 1, T(n))$.

La notación Ω denota una cota inferior asintótica. Sea $g(n)$ una función, con $\Omega(g(n))$ se denota el conjunto de funciones asintóticamente acotadas inferiormente por $g(n)$.

Definición 6.7 (Notación Ω). $\Omega(g(n)) = \{f(n) \mid \text{existen constantes positivas } c \text{ y } n_0 \text{ tal que } 0 \leq cg(n) \leq f(n), \text{ para todo } n \geq n_0\}$.

La convención $f(n) = \Omega(g(n))$, indica que $f(n) \in \Omega(g(n))$. La figura 6.3 muestra el significado de la notación Ω .

La notación Θ denota una cota asintótica. Sea $g(n)$ una función, con $\Theta(g(n))$ se denota el conjunto de funciones asintóticamente acotadas por $g(n)$.

Definición 6.8 (Notación Θ). $\Theta(g(n)) = \{f(n) \mid \text{existen constantes positivas } c_1, c_2 \text{ y } n_0 \text{ tal que } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n), \text{ para todo } n \geq n_0\}$.

La convención $f(n) = \Theta(g(n))$, indica que $f(n) \in \Theta(g(n))$. La figura 6.4 muestra el significado de la notación Θ .

A continuación, sin demostración enunciamos los siguientes teoremas relacionados con la notación O .

Teorema 6.2. Si $f: \mathbb{N} \rightarrow \mathbb{N}$ es un polinomio de grado d , es decir, $g(n) = a_dn^d + a_{d-1}n^{d-1} + \dots + a_0n^0$, entonces $f(n) = O(n^d)$.

Lo que afirma el teorema anterior, es que con respecto al acotamiento superior de un polinomio de grado d sólo es relevante el término que está elevado a este grado.

Teorema 6.3. Si $f: \mathbb{N} \rightarrow \mathbb{N}$ es un polinomio, entonces $f(n) = O(c^n)$, para $c > 1$ uniones

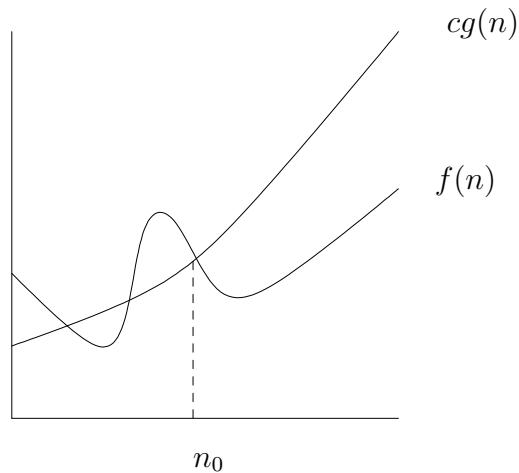


Figura 6.2: $f(n) = O(g(n))$.

Lo que afirma el teorema anterior, es que cualquier polinomio crece más despacio que cualquier función exponencial de base mayor que 1.

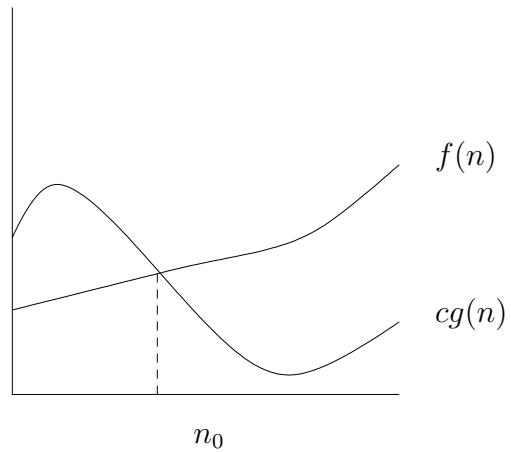
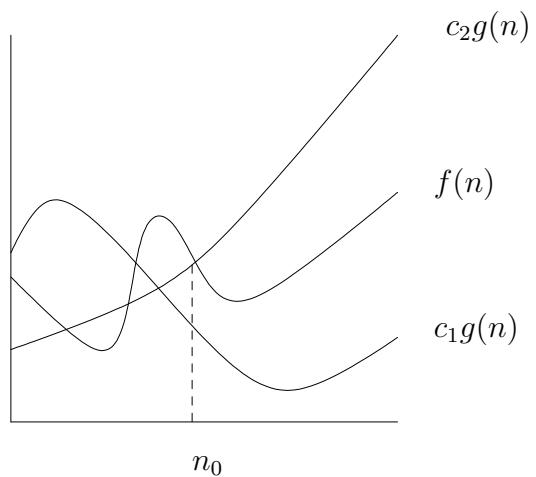
De acuerdo con el teorema 6.2 (pág. 187), la complejidad temporal del lenguaje presentado en el ejemplo 6.3 (pág. 185) es $T(n) = 3n + 3 = O(n)$. Se puede observar que las constantes multiplicativas y aditivas son irrelevantes para la clase de complejidad temporal a la cual pertenece un lenguaje.

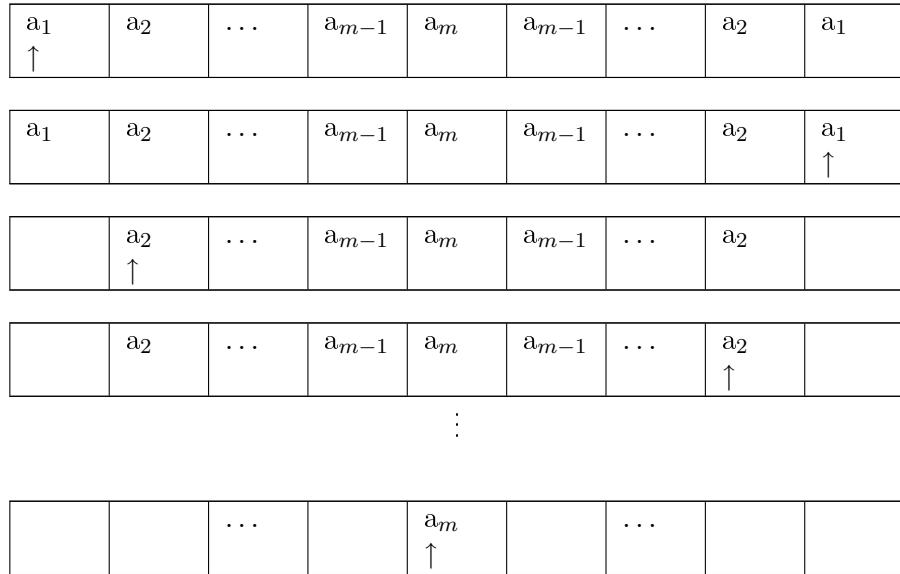
6.5. Relaciones de complejidad temporal determinista

Es bastante importante preguntarse si la elección del modelo de computación afecta o no a la clase de complejidad temporal a la cual pertenece un lenguaje. Antes de presentar un teorema que relaciona las complejidades temporales de una máquina de Turing k -cintas y una máquina de Turing, presentemos un ejemplo de un lenguaje aceptado por máquinas de Turing.

Ejemplo 6.4. Para el lenguaje presentado en el ejemplo 6.1 (pág. 184), indiquemos el comportamiento de una máquina de Turing que lo acepta, en donde dejamos los detalles de construcción de la máquina al lector (ejercicio 6.2).

Al inicio la máquina tiene la palabra $\beta \equiv a_1a_2 \dots a_{m-1}a_ma_ma_{m-1} \dots a_2a_1$ en la cinta. A continuación la máquina compara el primer y último símbolo; los elimina, y después compara el segundo símbolo con el penúltimo, y así sucesivamente, como lo indica la tabla 6.5. Finalmente, si las comparaciones fueron exitosas, la máquina pasa al estado q_Y y se detiene; de lo contrario, la máquina pasa al estado q_N y se detiene.

Figura 6.3: $f(n) = \Omega(g(n))$.Figura 6.4: $f(n) = \Theta(g(n))$.

Cuadro 6.5: Decisión para \mathcal{L} palíndromo por una \mathcal{MT} .

Sea $l(\beta) = n$ la longitud de la palabra β . Para realizar la comparación del primer y último símbolo, la máquina necesita $2n + 1$ pasos. Después la máquina tiene una palabra de longitud $n - 2$ símbolos y la comparación del primer y último símbolo requiere $2(n - 2) + 1$ pasos. Después la máquina tiene una palabra de longitud $n - 4$ y la comparación del primer y último símbolo requiere $2(n - 4)$ pasos. Y así sucesivamente. Entonces la función de complejidad temporal viene dada por

$$\begin{aligned} T(n) &= \sum_{i=0}^{\frac{n}{2}} 2(n - 2i) + 1 \\ &= \frac{(n+1)(n+2)}{2} \\ &= O(n^2). \end{aligned}$$

Es decir, si un lenguaje palíndromo \mathcal{L} se acepta con una máquina de Turing, entonces, $\mathcal{L} \in TIEMPO(0(n^2))$. Pero si el lenguaje \mathcal{L} se acepta con una máquina de Turing 2-cintas, entonces, $\mathcal{L} \in TIEMPO(0(n))$. Esta situación se generaliza con el siguiente teorema.

Teorema 6.4. *Si \mathcal{L} es un lenguaje aceptado por una máquina de Turing k -cintas \mathcal{MT}_k que opera con límite temporal $T(n)$, entonces existe una máquina de Turing \mathcal{MT} con límite temporal $O(T(n)^2)$ que acepta a \mathcal{L} .*

El teorema anterior expresa que nuestro modelo inicial de computabilidad (la máquina de Turing), también es un buen modelo de complejidad algorítmica desde el punto de vista del recurso tiempo, pues una máquina de Turing k -cintas sólo nos proporciona una ganancia de orden polinomial con respecto a una máquina de Turing.

Ahora continuemos nuestro trabajo analizando el recurso espacial. Es decir, vamos a presentar algunos aspectos de complejidades algorítmicas espaciales. Inicialmente presentaremos el modelo de computación denominado máquina de Turing $(k, 1)$ -cintas.

6.6. Máquinas de Turing $(k, 1)$ -cintas

La idea intuitiva detrás de una máquina de Turing $(k, 1)$ -cintas es la de una máquina de Turing $(k+1)$ -cintas, la cual tiene una cinta de sólo lectura y k cintas de trabajo; es decir, contiene k cintas de lectura y escritura; de ahí la nomenclatura: máquina de Turing $(k, 1)$ -cintas.

La idea de fijar una cinta de lectura, consiste en considerar que esta cinta no afectará el cálculo de la complejidad espacial de la máquina, pues, para una entrada de tamaño n , no se tendrá en cuenta el espacio ocupado por la entrada sino únicamente el espacio requerido para procesarla.

6.7. Complejidad espacial determinista

Definición 6.9 (Complejidad espacial para $\mathcal{MT}_{(k,1)}$). Sea $\mathcal{MT}_{(k,1)}$ una máquina de Turing $(k, 1)$ -cintas. El espacio requerido para computar una entrada $\alpha \in (\Sigma - \{\square\})^*$ es el máximo de celdas utilizadas por cualquiera de las k cintas de trabajo para que la máquina $\mathcal{MT}_{(k,1)}$ se detenga, cuando al inicio en la cinta de lectura está la palabra α .

Sea $S: \mathbb{N} \rightarrow \mathbb{N}$ una función. Se dice que la máquina $\mathcal{MT}_{(k,1)}$ opera con límite espacial $S(n)$, si para toda $\alpha \in (\Sigma - \{\square\})^*$ el espacio requerido para procesarla está dado por $S(l(\alpha))$, donde $l(\alpha)$ es la longitud de la palabra α .

Como es de esperarse que cualquier máquina de Turing $(k, 1)$ -cintas requiere al menos una celda de alguna de las k cintas de trabajo para procesar una palabra de longitud n , entonces, en realidad el límite espacial $S(n)$ significa el máximo entre 1 y $S(n)$. Es decir, nuestra convención será

$$S(n) \equiv_{def} \max(1, 'S(n)').$$

Ejemplo 6.5. Sea una máquina de Turing $(k, 1)$ -cintas con límite espacial $S(n) = \log_2 n$. De acuerdo a la convención $S(n) \equiv_{def} \max(1, 'S(n)')$, tenemos por ejemplo que: $S(1) = \max(1, 0) = 1$, $S(2) = \max(1, 1) = 1$ y $S(3) = \max(1, \log_2 3) = \log_2 3$. Cuando, seleccionar 1 o $\log_2 n$, está sujeto a los comportamientos de ambas funciones, tal como lo indica la figura 6.5.

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

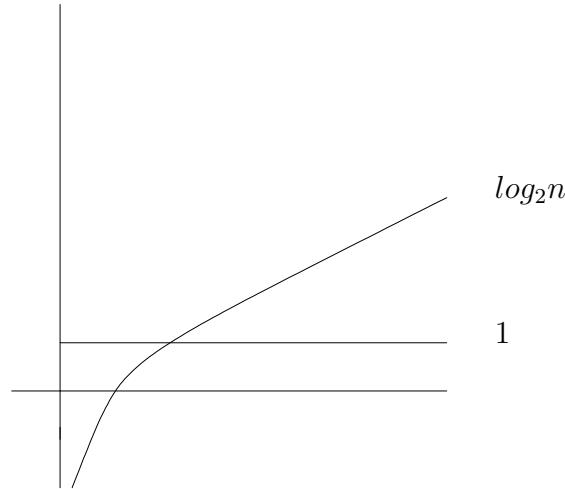


Figura 6.5: $S(n) = \max(1, 'S(n)').$

Definición 6.10 (Clase de complejidad espacial para un lenguaje). Sea \mathcal{L} un lenguaje aceptado por una máquina de Turing $(k, 1)$ -cintas con límite espacial $S(n)$. Decimos entonces que \mathcal{L} pertenece a la clase de complejidad temporal denominada $ESPACIOD(S(n))$. La clase $ESPACIOD(S(n))$ es el conjunto formado por todos los lenguajes aceptados por máquinas de Turing $(k, 1)$ -cintas con límite espacial $S(n)$.

Ejemplo 6.6. Para el lenguaje presentado en el ejemplo 6.1 (pág. 184), calculemos $S(n)$. Inicialmente supongamos que n es la longitud de la palabra inicial y que vamos a operar sobre una máquina de Turing $\mathcal{MT}_{(2,1)}$, es decir, sobre una máquina de Turing que contiene una cinta de lectura y dos cintas de trabajo.

Indiquemos el comportamiento de la máquina de Turing $(2, 1)$ -cintas $\mathcal{MT}_{(2,1)}$ que acepta a \mathcal{L} , donde dejamos los detalles de construcción de la máquina al lector (ejercicio 6.3).

La máquina va a operar con dos contadores i, j escritos en binario en las cintas de trabajo. La cinta $_i$ representará el contador i y la cinta $_j$ representará el contador j . La máquina realiza dos bucles; un bucle externo desde $i = 1$ hasta $i \leq n$, y un bucle interno desde $j = 1$ hasta $j \leq i$. En cada iteración del bucle interno la máquina compara los valores i y j . Si $i < j$, incrementa el valor de j en uno. Si $i = j$, la máquina compara el i -ésimo símbolo de izquierda a derecha con el j -ésimo símbolo de derecha a izquierda de la palabra de entrada. Si la comparación es exitosa, entonces la máquina regresa al bucle externo; de lo contrario la máquina pasa al estado q_N y se detiene. Si la máquina finaliza el bucle externo, la máquina pasa al estado q_Y y se detiene.

Esta máquina requiere como máximo escribir el número n en notación binaria en las cintas de trabajo (cinta $_i$, cinta $_j$), por lo tanto, su límite espacial $S(n)$ está dado por $S(n) =$

$\log_2 n = O(\log_2 n)$.

6.8. Relaciones de complejidad espacial entre los modelos de computación determinista

De nuevo es importante preguntarse si la elección del modelo de computación afecta o no la clase de complejidad espacial a la cual pertenece un lenguaje dado.

Teorema 6.5. *Si \mathcal{L} es un lenguaje aceptado por una máquina de Turing k -cintas \mathcal{MT}_k que opera con límite espacial $S(n)$, entonces existe una máquina de Turing $(k, 1)$ -cintas $\mathcal{MT}_{(k,1)}$ con límite espacial $O(S(n))$ que acepta a \mathcal{L} .*

Demostración. La máquina $\mathcal{MT}_{(k,1)}$ copia la palabra de entrada en la primera cinta de trabajo y entonces puede operar como la máquina \mathcal{MT}_k . \square

Teorema 6.6. *Si \mathcal{L} es un lenguaje aceptado por una máquina de Turing $(k, 1)$ -cintas $\mathcal{MT}_{(k,1)}$ que opera con límite espacial $S(n)$, entonces existe una máquina de Turing $(1, 1)$ -cintas $\mathcal{MT}_{(1,1)}$ con límite espacial $O(S(n))$ que acepta a \mathcal{L} .*

Demostración. (indicación)

Sea Σ el alfabeto de la máquina $\mathcal{MT}_{(k,1)}$. Es posible considerar los símbolos de las k cintas de trabajo de la máquina $\mathcal{MT}_{(k,1)}$ como k -tuplas de Σ^k . Estas k -tuplas pueden ser codificadas por una alfabeto Σ' de 2^k símbolos. Entonces se construye una máquina $\mathcal{MT}_{(1,1)}$ que opere con el alfabeto Σ' de forma equivalente a como opera la máquina $\mathcal{MT}_{(k,1)}$ con las k -tuplas. \square

Teorema 6.7. *Si \mathcal{L} es un lenguaje aceptado por una máquina de Turing $(1, 1)$ -cintas $\mathcal{MT}_{(1,1)}$ que opera con límite espacial $S(n)$, entonces existe una máquina de Turing \mathcal{MT} con límite espacial $O(S(n))$ que acepta a \mathcal{L} .*

Demostración. (indicación)

Para obtener la máquina \mathcal{MT} se sigue un esquema de codificación similar al empleado en la demostración del teorema 6.6 (pág. 193). \square

De nuevo lo que indican los teoremas 6.5 (pág. 193), 6.6 (pág. 193) y 6.7 (pág. 193) es que nuestro modelo inicial de computabilidad (la máquina de Turing), también es un buen modelo de complejidad algorítmica, desde el punto de vista del recurso espacial, pues una máquina de Turing $(k, 1)$ -cintas sólo proporciona una ganancia de orden constante con respecto a una máquina de Turing.

Mencionábamos al comienzo de este capítulo que uno de los aspectos importantes que debemos considerar en la teoría de complejidad es el modo de computación. Pero hasta el

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados

**Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a
EDICT._GUIA_MEX - Editorial Guias Mexico**

momento sólo hemos estudiado el modo de computación determinista. Abordamos ahora el estudio del modo de computación no determinista.

6.9. Máquina de Turing no determinista

Desde una perspectiva intuitiva podemos decir que una máquina de Turing no determinista es un máquina de Turing que puede seleccionar de un conjunto finito de posibilidades una acción o instrucción para ejecutar.

Definición 6.11 (Máquina de Turing no determinista). Definimos una máquina de Turing no determinista por la estructura matemática $\mathcal{MTN} = \langle Q, \Sigma, M, I \rangle$, donde:

Q, Σ y M son los mismos conjuntos que para una máquina de Turing.

I : Es una relación definida de $Q \times \Sigma$ en $\Sigma \times M \times Q \cup \{q_Y, q_N\}$, es decir, $I \subseteq Q \times \Sigma \times \Sigma \times M \times Q \cup \{q_Y, q_N\}$.

La diferencia esencial entre las definiciones formales de una máquina de Turing y una máquina de Turing no determinista radica en el símbolo I . En la primera, I es una función parcial, esto es, $I: Q \times \Sigma \rightarrow \Sigma \times M \times Q \cup \{q_Y, q_N\}$, mientras que en la segunda, I es una relación, esto es, $I \subseteq Q \times \Sigma \times \Sigma \times M \times Q \cup \{q_Y, q_N\}$.

Esta diferencia hace que las computaciones de una máquina de Turing no determinista presenten la forma de un árbol de computación, como se indica en la figura 6.6, mientras que las computaciones de una máquina de Turing presentan la forma de una línea de computación, como se indica en la figura 6.7.

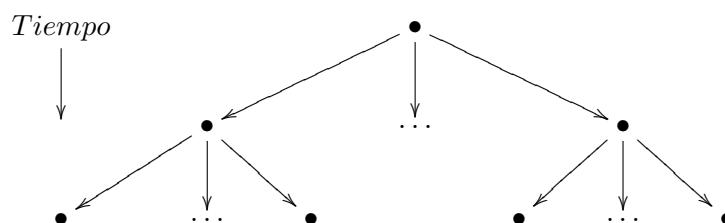


Figura 6.6: Árbol de computación: máquinas de Turing no deterministas.

6.10. Complejidad temporal y espacial no determinista

Definición 6.12 (Aceptación de lenguajes por MTN). Sea Σ una alfabeto de una máquina de Turing no determinista \mathcal{MTN} y sea $\mathcal{L} \subseteq (\Sigma - \{\square\})^*$ un lenguaje. Se dice que la máquina \mathcal{MTN} acepta el lenguaje \mathcal{L} si para todo $\alpha \in (\Sigma - \{\square\})^*$ se cumple que:

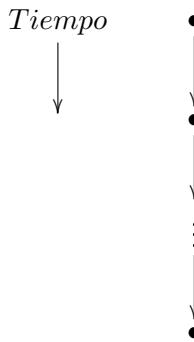


Figura 6.7: Línea de computación: máquinas de Turing.

1. Si $\alpha \in \mathcal{L}$, entonces, existe al menos una posible selección de acciones de la máquina \mathcal{MTN} tal que ésta se detiene en el estado q_Y .
2. Si $\alpha \notin \mathcal{L}$, entonces, para cualquier posible acción de la máquina \mathcal{MTN} , ésta no se detiene en el estado q_N .

Definición 6.13 (Complejidad temporal y espacial para MTN). Sea una máquina de Turing no determinista \mathcal{MTN} . El tiempo requerido para computar una entrada $\alpha \in (\Sigma - \{\square\})^*$ es el máximo de número de pasos necesarios para que la máquina \mathcal{MTN} se detenga, bajo cualquier posible selección de acciones de la máquina, cuando al inicio en la cinta está la palabra α . Es decir, el tiempo requerido para computar α es la longitud del camino más largo en el árbol de computación de la máquina.

En otras palabras, el espacio requerido para computar una entrada $\alpha \in (\Sigma - \{\square\})^*$ es el máximo de celdas utilizadas por cualquiera de las posibles acciones de la máquina, de modo tal que ésta se detenga cuando al inicio en la cinta está la palabra α .

Sea $T: \mathbb{N} \rightarrow \mathbb{N}$ una función. Se dice que la máquina \mathcal{MTN} opera con límite temporal $T(n)$ si para toda $\alpha \in (\Sigma - \{\square\})^*$ el tiempo requerido para procesarla está dado por $T(l(\alpha))$, donde $l(\alpha)$ es la longitud de la palabra α .

Sea $S: \mathbb{N} \rightarrow \mathbb{N}$ una función. Se dice que la máquina \mathcal{MTN} opera con límite espacial $S(n)$ si para toda $\alpha \in (\Sigma - \{\square\})^*$ el espacio requerido para procesarla está dado por $S(l(\alpha))$, donde $l(\alpha)$ es la longitud de la palabra α .

Definición 6.14 (Clases de complejidad no deterministas). Si \mathcal{L} es un lenguaje aceptado por una máquina de Turing no determinista con límite temporal $T(n)$, entonces decimos que \mathcal{L} pertenece a la clase de complejidad temporal denominada $TIEMPON(T(n))$. La clase $TIEMPON(T(n))$ es el conjunto formado por todos los lenguajes aceptados por máquinas de Turing no deterministas con límite temporal $T(n)$.

Si \mathcal{L} es un lenguaje aceptado por una máquina de Turing no determinista con límite espacial $S(n)$, entonces se dice que \mathcal{L} es un miembro de la clase de complejidad espacial $ESPACION(S(n))$. La clase $ESPACION(S(n))$ es el conjunto formado por todos los lenguajes aceptados por máquinas de Turing no deterministas con límite espacial $S(n)$.

6.11. Relaciones entre clases de complejidad

Existen algunas relaciones de inclusión entre las clases de complejidad temporal y espacial en los modos de computación determinista y no determinista que presentamos a continuación. Inicialmente presentamos las relaciones entre las clases de complejidad espacial determinista y no determinista.

Teorema 6.8. *Si $\mathcal{L} \in ESPACIOD(S(n))$, entonces $\mathcal{L} \in ESPACION(S(n))$. Es decir, $ESPACIOD(S(n)) \subseteq ESPACION(S(n))$.*

Demostración. Toda máquina de Turing determinista es una máquina de Turing no determinista especial en donde esta última sólo tiene una posible acción en su conjunto de posibles acciones. Luego, si una máquina de Turing determinista tiene límite espacial $S(n)$, la misma máquina considerada como una máquina de Turing no determinista tiene límite espacial $S(n)$. \square

Para poder presentar el próximo teorema necesitamos dos definiciones. Estas definiciones están pensadas para eliminar ciertas funciones “patológicas” presentes en el conjunto de posibles funciones de complejidad espacial.

Definición 6.15 (Función espacialmente construible). Sea $S: \mathbb{N} \rightarrow \mathbb{N}$ una función. Se dice que la función $S(n)$ es espacialmente construible si existe una máquina de Turing \mathcal{MT} con límite espacial $S(n)$, tal que, para todo n existe una entrada α con $l(\alpha) = n$, la máquina \mathcal{MT} utiliza $S(n)$ celdas.

Ejemplo 6.7. La función $\log_2 n$ es espacialmente construible. Sea $\mathcal{MT}_{(1,1)}$ una máquina de Turing con una cinta de lectura y una cinta de trabajo, y sea $\alpha \equiv a_1 \dots a_n$. La máquina por cada símbolo a_i escribe en la cinta de trabajo correspondiente el dígito i en binario. La máquina $\mathcal{MT}_{(1,1)}$ tiene límite espacial $\log_2 n$, donde $n = l(\alpha)$. Entonces, por el teorema 6.7 (pág. 193) existe una máquina de Turing \mathcal{MT} con límite espacial $\log_2 n$ equivalente a la máquina $\mathcal{MT}_{(1,1)}$.

Teorema 6.9. *Si $f(n)$ y $g(n)$ son funciones espacialmente construibles, entonces las funciones $f(n) + g(n)$, $f(n)g(n)$ y $2^{f(n)}$ son espacialmente construibles.*

Demostración. Ejercicio 6.4. \square

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Definición 6.16 (Construcción de manera completa). Sea $S: \mathbb{N} \rightarrow \mathbb{N}$ una función. Se dice que la función $S(n)$ es espacialmente construible de manera completa si existe una máquina de Turing \mathcal{MT} con límite espacial $S(n)$, tal que, para todo n y para toda entrada α con $l(\alpha) = n$, la máquina \mathcal{MT} utiliza $S(n)$ celdas.

Teorema 6.10. Si $f(n)$ es una función espacialmente construible y $f(n) \geq n$, entonces la función $f(n)$ es espacialmente construible de manera completa.

Demostración. Ejercicio 6.5. □

Teorema 6.11 (Teorema de Savitch). Si $\mathcal{L} \in ESPACION(S(n))$, la función $S(n)$ es espacialmente construible de manera completa y $S(n) \leq \log_2 n$, entonces $\mathcal{L} \in ESPACIOD((S(n))^2)$

Demostración. Ejercicio 6.7. □

Presentemos ahora las relaciones entre las clases de complejidad temporal determinista y no determinista.

Teorema 6.12. Si $\mathcal{L} \in TIEMPOD(T(n))$, entonces $\mathcal{L} \in TIEMPON(T(n))$. Es decir, $TIEMPOD(T(n)) \subseteq TIEMPON(T(n))$.

Demostración. De la demostración del teorema 6.8 (pág. 196), sabemos que si una máquina de Turing determinista tiene límite temporal $T(n)$, entonces la misma máquina considerada como una máquina de Turing no determinista tiene límite espacial $T(n)$. □

Teorema 6.13. Si $\mathcal{L} \in TIEMPON(f(n))$, entonces existe una constante d tal que $\mathcal{L} \in TIEMPOD(d^{f(n)})$. Es decir, $TIEMPON(f(n)) \subseteq TIEMPOD(d^{f(n)})$.

Demostración. Si se conoce el número de estados, el número de símbolos del alfabeto y la complejidad temporal de una máquina de Turing no determinista, es posible conocer el número máximo de descripciones instantáneas de la máquina, y por ende, es posible conocer el número máximo de celdas necesarias para decidir un lenguaje.

Sea \mathcal{MTN} una máquina de Turing no determinista con límite temporal $T(n)$. En $T(n)$ movimientos la máquina puede inspeccionar como máximo $T(n) + 1$ celdas. Si la máquina tiene $\bar{\Sigma}$ símbolos, entonces existen como máximo $\bar{\Sigma}^{T(n)+1}$ cadenas de longitud $T(n) + 1$ de símbolos de Σ . La cabeza de lectura-escritura puede estar en cualesquiera de las $T(n) + 1$ celdas, y para cada cadena el estado actual puede ser uno de los \bar{Q} estados. Por lo tanto, el número máximo de descripciones instantáneas para \mathcal{MTN} es $\bar{Q}(T(n) + 1)\bar{\Sigma}^{T(n)+1}$. Luego, existe c tal que $c^{T(n)} \geq \bar{Q}(T(n) + 1)\bar{\Sigma}^{T(n)+1}$; es decir, el número de descripciones instantáneas de \mathcal{MTN} está limitado por $c^{T(n)}$.

Una máquina de Turing \mathcal{MT} puede simular una \mathcal{MTN} que acepta o no un lenguaje de la siguiente manera. A partir de la descripción instantánea inicial, se genera sobre la cintas las descripciones instantáneas que se pueden alcanzar en un paso de computación

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

de \mathcal{MTN} ; a partir de éstas, genera las que se pueden alcanzar en el siguiente paso de computación de \mathcal{MTN} , y así sucesivamente hasta $T(n)$ pasos de computación de \mathcal{MTN} . Sea $T(n) + 2$ la longitud de una descripción instantánea, pues una máquina en $T(n)$ pasos puede escribir $T(n) + 1$ símbolos y añadimos el símbolo del estado actual. Para generar una descripción instantánea son necesarios $3(T(n) + 2)$ pasos; esto incluye escribir sus símbolos y buscar el estado q_Y ; si lo encuentra termina la simulación; de lo contrario, genera la siguiente descripción instantánea. De cada descripción instantánea hay como máximo $c^{T(n)}$ descripciones instantáneas siguientes y como existen $T(n)$ pasos de computación, la cantidad total de movimientos es $3(T(n) + 2)T(n)c^{T(n)}$. Por lo tanto, existe una constante d tal que $d^{T(n)} \geq 3(T(n) + 2)T(n)c^{T(n)}$. \square

Los teoremas que presentamos a continuación presentan las relaciones entre las clases de complejidad temporal y espacial deterministas.

Teorema 6.14. *Si $\mathcal{L} \in TIEMPOD(f(n))$, entonces $\mathcal{L} \in ESPACIOD(O(f(n)))$. Es decir, $TIEMPOD(f(n)) \subseteq ESPACIOD(O(f(n)))$.*

Demostración. Si una máquina de Turing realiza a lo sumo $f(n)$ pasos de computación, entonces ésta puede utilizar a lo sumo $f(n) + 1$ celdas de su cinta. \square

Teorema 6.15. *Si $\mathcal{L} \in ESPACIOD(f(n))$ y $f(n) \leq \log_2 n$, entonces existe una constante c tal que $\mathcal{L} \in TIEMPOD(c^{f(n)})$. Es decir, la clase $ESPACIOD(f(n))$ está contenida en la clase $TIEMPOD(c^{f(n)})$.*

Demostración. Si se conoce el número de estados, el número de símbolos del alfabeto y la complejidad espacial de una máquina de Turing, es posible conocer el número máximo de descripciones instantáneas de la máquina, y de allí conocer el número máximo posible de pasos que la máquina puede realizar para decidir un lenguaje.

Sea \mathcal{MT} una máquina de Turing con límite espacial $f(n)$. Esta máquina puede escribir $\overline{\Sigma}^{f(n)}$ símbolos en un espacio de $f(n)$ de celdas de $\overline{\Sigma}^{f(n)}$ formas distintas; además, para cada una de estas formas, la máquina puede estar en $\overline{Q}^{f(n)}$ estados diferentes y la cabeza de lectura-escritura puede estar en $f(n)$ posiciones diferentes. Por lo tanto el número máximo de descripciones instantáneas para \mathcal{MT} está dado por $\overline{Q}f(n)\overline{\Sigma}^{f(n)}$.

Si, $f(n) \leq \log_2 n$, entonces existe c tal que $c^{f(n)} \geq \overline{Q}f(n)\overline{\Sigma}^{f(n)}$ para toda $n > 1$. Por lo tanto el número máximo de descripciones instantáneas tiene límite superior $c^{f(n)}$; de donde se concluye que una máquina de Turing que sea capaz de ejecutar todas estas posibles descripciones instantáneas debe tener como límite temporal $TIEMPOD(c^{f(n)})$. \square

La tabla 6.6 presenta las relaciones entre las clases de complejidad indicadas por los teoremas 6.8 (pág. 196), 6.11 (pág. 197), 6.12 (pág. 197), 6.13 (pág. 197), 6.14 (pág. 198) y 6.15 (pág. 198). Las restantes relaciones entre clases de complejidad pueden ser obtenidas a partir de esta tabla.

Relación	
Espacial	$ESPACIOD(S(n)) \subseteq ESPACION(S(n))$ $ESPACION(S(n)) \subseteq ESPACIOD((S(n))^2)$ si $S(n)$ es una función espacialmente construible de manera completa y $S(n) \geq \log_2 n$
Temporal	$TIEMPOD(T(n)) \subseteq TIEMPON(T(n))$ $TIEMPON(f(n)) \subseteq TIEMPOD(d^{f(n)})$
Determinista	$TIEMPOD(f(n)) \subseteq ESPACIOD(O(f(n)))$ $ESPACIOD(f(n)) \subseteq TIEMPOD(c^{f(n)})$ si $f(n) \leq \log_2 n$

Cuadro 6.6: Relaciones entre las clases de complejidad.

6.12. Problemas intratables

La clasificación de los problemas en tratables e intratables, puede realizarse desde dos perspectivas, a saber: temporal o espacial. En esta sección sólo consideraremos la primera de ellas. Aunque todos los problemas computables sean, en teoría, tratables, desde un punto de vista práctico las cosas son diferentes. Así, en la práctica los problemas con límite temporal polinómico son considerados tratables y los que tienen límite temporal exponencial son considerados intratables.

Pasemos a realizar ciertas consideraciones importantes sobre el tema que nos ocupa, lo cual nos llevará a generar un conjunto de definiciones y teoremas importantes en el contexto de la complejidad algorítmica.

Existe en este contexto una importante clase de lenguajes que se pueden aceptar en modo determinista en tiempo polinómico, denominada la clase P .

Definición 6.17 (Clase P).

$$P = \bigcup_{i \geq 1} TIEMPOD(n^i).$$

Otra clase importante de lenguajes en este contexto, son los que se pueden aceptar en modo no determinista en tiempo polinómico, denominada la clase NP .

Definición 6.18 (Clase NP).

$$NP = \bigcup_{i \geq 1} TIEMPON(n^i).$$

Desde el punto de vista de los problemas de decisión, es decir, aquellos problemas en los cuales se espera una respuesta “SI” o una respuesta “NO”, la diferencia entre entre la clase

P y la clase NP puede ser vista como la diferencia entre encontrar la respuesta correcta (clase P) y probar que una posible respuesta es la correcta (clase NP). Intuitivamente es más fácil probar que una respuesta es la correcta que encontrar la respuesta correcta.

Aunque las clases P y NP son clases de lenguajes, hemos presentado una diferencia intuitiva entre ellas en términos de problemas de decisión. Cualquier problema de decisión puede ser visto como el problema de aceptar un cierto lenguaje construido bajo una codificación del problema de decisión en cuestión. Por esta razón hablaremos sin distinción de problemas de aceptación de lenguajes o de problemas de decisión.

Un problema importante que pertenece a la clase NP es el problema de la *satisfactibilidad*. Este problema jugará un papel muy importante en una clase de problemas que llamaremos NP -completos.

Ejemplo 6.8 (Problema de la satisfactibilidad). Sea $X = \{x_1, \dots, x_n\}$ un conjunto de variables booleanas y sea $t: X \rightarrow \{0, 1\}$ una asignación de valores de verdad para X ; donde, $t(x) = 0$ significa que x es "falsa" y $t(x) = 1$ significa que x es "verdadera". Si $x \in X$ entonces x y \bar{x} son literales sobre X . El literal \bar{x} representa el opuesto de x , es decir \bar{x} es "verdadero" bajo t si y sólo si x es falso bajo t . Una cláusula sobre X es una disyunción de literales sobre X . Una cláusula se dice satisfactible si existe una asignación de valores de verdad t para sus variables, tal que la cláusula sea "verdadera". Un conjunto de cláusulas sobre X se dice satisfactible si existe una asignación de valores de verdad t que satisface simultáneamente todas las cláusulas del conjunto.

Por ejemplo, el conjunto formado por las tres cláusulas $C = \{x_1, x_2 \vee \bar{x}_3, x_1 \vee x_3\}$ es satisfactible, pues la asignación de valores de verdad $t(x_1) = t(x_2) = 1$ y $t(x_3) = 0$, hace verdaderas cada una de las cláusulas del conjunto.

Por otra parte, el conjunto de cláusulas $C = \{x_1, \bar{x}_1 \vee \bar{x}_2, \bar{x}_1 \vee x_2\}$ no es satisfactible, pues no existe ninguna asignación de valores de verdad t que haga las tres cláusulas verdaderas simultáneamente.

El problema de la satisfactibilidad (representado como SAT) consiste en decidir si un conjunto de cláusulas arbitrario es o no es satisfactible. El problema $SAT \in NP$. Para expresar el problema de decisión SAT como el problema de aceptar un lenguaje \mathcal{L}_{SAT} , realizaremos la siguiente codificación. Cada literal de la forma x_i se codifica como $\&i_b$ donde i_b es la representación en binario del número i . Cada literal de la forma \bar{x}_i se codifica como $\neg\&i_b$ donde i_b es la representación en binario del número i . Entonces el alfabeto para \mathcal{L}_{SAT} está definido por $\Sigma = \{\vee, \neg, \&, 0, 1\} \cup \{\}, \{\}$ y el lenguaje \mathcal{L}_{SAT} , está definido por

$$\mathcal{L}_{SAT} = \{\alpha \in \Sigma^* \text{ tal que } \alpha \text{ representa un conjunto satisfactible de cláusulas }\}.$$

Dada $\alpha \in \Sigma^*$ necesitamos saber si, $\alpha \in L_{SAT}$, lo cual consiste en el problema de aceptar un lenguaje. Entonces afirmamos que $\mathcal{L}_{SAT} \in NP$.

Dado un conjunto de cláusulas codificado y una asignación de valores de verdad para sus variables, es posible evaluar su valor de verdad en tiempo polinómico. Entonces una

máquina de Turing no determinista que seleccione una posible asignación de valores de verdad para las variables de un conjunto de cláusulas codificado, puede decidir en tiempo polinómico si este conjunto de cláusulas es satisfactible, tal como lo indica la figura 6.8.

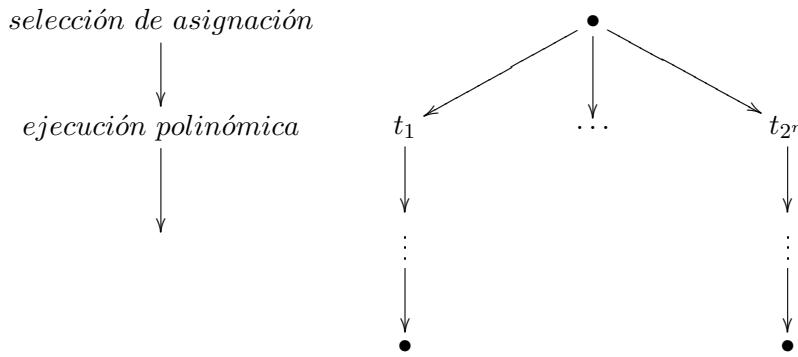


Figura 6.8: Árbol de computación para una \mathcal{MTN} para el problema SAT .

Por el teorema 6.12 (pág. 197), tenemos que $TIEMPOD(n^k) \subseteq TIEMPON(n^k)$, luego $P \subseteq NP$. Uno de los problemas más importantes en el contexto de la complejidad algorítmica es decidir si la inclusión $P \subseteq NP$, es o no, una inclusión propia, lo cual es equivalente a decidir si $P = NP$ o $P \neq NP$.

$P = NP$ equivale a decir que todo problema resuelto por una máquina de Turing no determinista con límite temporal polinómico puede ser resuelto por una máquina de Turing determinista con límite temporal polinómico. A partir del teorema 6.13 (pág. 197) podemos afirmar que todo problema resuelto por una máquina de Turing no determinista con límite temporal polinómico, puede ser resuelto por una máquina de Turing determinista con límite temporal exponencial; sin embargo, este teorema no elimina la posibilidad de encontrar la máquina de Turing determinista con límite temporal polinómico.

Por otra lado, $P \neq NP$ equivale a decir que existe por lo menos un problema resuelto por una máquina de Turing no determinista con límite temporal polinómico, que no puede ser resuelto por una máquina de Turing determinista con límite temporal polinómico. Sin embargo, hasta el momento no se conoce ningún problema con estas características.

Un concepto importante para determinar si un lenguaje pertenece a la clase P o a la clase NP es el concepto de reducibilidad en tiempo polinómico.

Definición 6.19 (Función computable en tiempo polinómico). Una función f se dice computable en tiempo polinómico si existe una máquina de Turing con límite temporal polinómico que la calcula.

Definición 6.20 (Reducción en tiempo polinómico). Un lenguaje \mathcal{L}_1 es reducible en tiempo polinómico a un lenguaje \mathcal{L}_2 , denotado por $\mathcal{L}_1 <_p \mathcal{L}_2$, si existe una función computable en tiempo polinómico f tal que, $f(\alpha) \in \mathcal{L}_2$ si y sólo si $\alpha \in \mathcal{L}_1$.

Teorema 6.16. Si $\mathcal{L}_2 \in P$ y $\mathcal{L}_1 <_p \mathcal{L}_2$, entonces $\mathcal{L}_1 \in P$.

Demostración. Sea $\mathcal{L}_2 \in P$ decidable en tiempo polinómico $p_2(n)$ y sea $\mathcal{L}_1 <_p \mathcal{L}_2$ en una reducción de tiempo polinomial f con límite temporal $p_1(n)$. Para decidir si $\alpha \in \mathcal{L}_1$ se reduce α por medio de f y se decide entonces si $f(\alpha) \in \mathcal{L}_2$. Entonces decidir si $\alpha \in \mathcal{L}_1$ tiene un límite temporal $p_2(p_1(n))$ el cual es un límite temporal polinómico; por lo tanto $\mathcal{L}_1 \in P$. \square

Teorema 6.17. Si $\mathcal{L}_2 \in NP$ y $\mathcal{L}_1 <_p \mathcal{L}_2$, entonces $\mathcal{L}_1 \in NP$.

Demostración. Ejercicio 6.8. \square

Además de permitirnos reconocer lenguajes de las clases P o NP , el concepto de reducción en tiempo polinómico nos permite hablar de unas clases importantes de lenguajes, denominados lenguajes \mathcal{C} -completos y lenguajes \mathcal{C} -difíciles.

Definición 6.21 (Lenguaje \mathcal{C} -difícil). Sea \mathcal{C} una clase de lenguajes. Un lenguaje \mathcal{L} se dice \mathcal{C} -difícil si para todo $\mathcal{L}' \in \mathcal{C}$, $\mathcal{L}' <_p \mathcal{L}$.

Definición 6.22 (Lenguaje \mathcal{C} -completo). Sea \mathcal{C} una clase de lenguajes. Sea \mathcal{L} un lenguaje \mathcal{C} -difícil. Se dice que \mathcal{L} es un lenguaje \mathcal{C} -completo si $\mathcal{L} \in \mathcal{C}$.

La clase de los problemas NP -completos es la clase de problemas NP que son equivalentes por medio de una reducción en tiempo polinómico. Por lo tanto, si se encuentra un solución determinista con límite temporal polinómico para cualquiera de los problemas en la clase NP -completos, entonces se ha encontrado una solución determinista con límite temporal polinómico para todos los problemas NP -completos.

El primer problema que se estableció como NP -completo fue el problema *SAT*.

Teorema 6.18 (Teorema de Cook). $\mathcal{L}_{SAT} \in NP$ -completos.

Demostración. (indicación)

Como *SAT* es el primer problema NP -completo, es necesario demostrar que cualquier problema NP es reducible en tiempo polinómico a *SAT*, es decir, es necesario demostrar que si $\mathcal{L} \in NP$ entonces $\mathcal{L} <_p \mathcal{L}_{SAT}$. El conjunto de problemas en NP es bastante diverso y tienen como característica común el que para cada uno de ellos existe una máquina de Turing no determinista con límite temporal polinómico que lo resuelve. Por lo tanto, para demostrar que $\mathcal{L} <_p \mathcal{L}_{SAT}$, para cualquier $\mathcal{L} \in NP$ es necesario construir una función f de tiempo polinómico tal que si $\alpha \in \mathcal{L}$, entonces $f(\alpha) \in \mathcal{L}_{SAT}$, y si $\alpha \notin \mathcal{L}$, entonces $f(\alpha) \notin \mathcal{L}_{SAT}$. Es decir, si $\alpha \in \mathcal{L}$, entonces $f(\alpha)$ es una cláusula satisfactible y si $\alpha \notin \mathcal{L}$,

entonces $f(\alpha)$ es una cláusula no satisfactible. La construcción de f se realiza con base en la máquina de Turing no determinista que decide a \mathcal{L} , por medio de una codificación de todas sus descripciones instantáneas posibles en un tiempo polinómico. \square

Una vez establecido un primer problema NP -completo, para lograr demostrar que un nuevo problema es NP -completo, basta con demostrar que el nuevo problema es NP y que existe una reducción en tiempo polinómico de un problema NP -completo al nuevo problema. Esta propiedad la formalizamos mediante los siguientes teoremas.

Teorema 6.19. *Si $\mathcal{L}_1 \in NP$ -completo y $\mathcal{L}_1 <_p \mathcal{L}_2$, entonces $\mathcal{L}_2 \in NP$ -difícil.*

Demostración. Lo que necesitamos demostrar es que si $\mathcal{L}_1 \in NP$ -completo y $\mathcal{L}_1 <_p \mathcal{L}_2$, entonces, para cualquier $\mathcal{L} \in NP$ se tiene que $\mathcal{L} <_p \mathcal{L}_2$.

Sea $\mathcal{L} \in NP$, sea $\mathcal{L}_1 \in NP$ -completo y $\mathcal{L}_1 <_p \mathcal{L}_2$. Como \mathcal{L}_1 es NP -completo, existe una reducción de tiempo polinómico f_1 de \mathcal{L} a \mathcal{L}_1 tal que $\alpha \in \mathcal{L}$ si y sólo si $f(\alpha) \in \mathcal{L}_1$. Sea $p_1(n)$ el límite temporal polinómico de la reducción f_1 . Como $\mathcal{L}_1 <_p \mathcal{L}_2$ existe una reducción de tiempo polinómico f_2 de \mathcal{L}_1 a \mathcal{L}_2 tal que $\alpha \in \mathcal{L}_1$ si y sólo si $f_2(f_1(\alpha)) \in \mathcal{L}_2$. Sea $p_2(n)$ el límite temporal polinómico de la reducción f_2 . La función $f_2(f_1(\alpha))$ es computable en tiempo polinómico $p_2(p_1(n))$ y tiene el comportamiento de una reducción en tiempo polinómico de \mathcal{L} a \mathcal{L}_2 , es decir, $\alpha \in \mathcal{L}$ si y sólo si $f_2(f_1(\alpha)) \in \mathcal{L}_2$. Entonces para cualquier $\mathcal{L} \in NP$ se tiene que $\mathcal{L} <_p \mathcal{L}_2$, de donde concluimos que $\mathcal{L}_2 \in NP$ -difícil. \square

Teorema 6.20. *Si $\mathcal{L}_1 \in NP$ -completo, $\mathcal{L}_2 \in NP$ y $\mathcal{L}_1 <_p \mathcal{L}_2$, entonces $\mathcal{L}_2 \in NP$ -completo.*

Demostración. Ejercicio 6.9. \square

Teorema 6.21. *Una variante del problema SAT es el problema 3SAT. En este caso cada cláusula debe contener tres literales. El problema 3SAT $\in NP$ -completos.*

Demostración. Ejercicio 6.10. \square

Podemos hallar problemas NP -completos en diversos campos como la lógica, la teoría de grafos, el diseño de redes, la teoría de automátas y lenguajes, entre otros. A continuación presentaremos un problema NP -completo de la teoría de grafos. Para lograr su presentación requerimos de algunas definiciones preliminares.

Definición 6.23 (Digrafo). Un digrafo \mathcal{G} , es un modelo de un lenguaje $\mathcal{L} = \{P^2\}$, donde P^2 es un símbolo de predicado de aridez dos. En otros términos, un digrafo es una estructura matemática $\mathcal{G} = < V, R >$, donde:

V : Conjunto no vacío cuyos elementos llamamos vértices.

R : Relación binaria definida sobre V ($R \subset V \times V$).

Cada digrafo $\vec{\mathcal{G}} = < V, R >$ puede ser representado por medio de un diagrama, en donde cada vértice $v \in V$ se representa por medio de un círculo etiquetado con el símbolo v , y cada $(v_i, v_j) \in R$ se representa por medio de un arco trazado del vértice v_i al vértice v_j .

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Ejemplo 6.9. $\vec{G}_2 = < \{0, 2, 5, 7, 8\}, \geq >$, representado por la figura 6.9 es un digrafo finito.

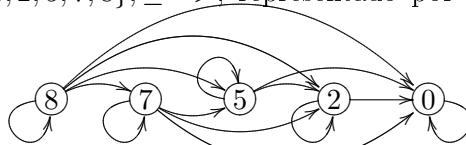


Figura 6.9: Ejemplo digrafo finito.

Aunque un grafo $\mathcal{G} = < V, R >$ también es un modelo de un lenguaje $\mathcal{L} = \{P\}$, donde P es un símbolo de predicado de aridez dos, la diferencia entre un digrafo y un grafo consiste en que este último debe satisfacer un axioma no exigido al primero.

Definición 6.24 (Grafo no dirigido). Un grafo no dirigido o simplemente un grafo $\mathcal{G} = < V, R >$ es un modelo de un lenguaje $\mathcal{L} = \{P^2\}$, tal que \mathcal{G} satisface el axioma de simetría para la relación R , es decir,

$$\mathcal{G} \models \forall x \forall y ((x, y) \in R \implies (y, x) \in R).$$

Cada grafo $\mathcal{G} = < V, R >$ puede ser representado por medio de un diagrama en donde cada vértice $v \in V$ se representa por medio de un círculo etiquetado con el símbolo v y cada par de elementos $(v_i, v_j), (v_j, v_i) \in R$ se representan por medio de una arista del vértice v_i al vértice v_j .

Ejemplo 6.10. El siguiente objeto matemático es un grafo finito.

$\mathcal{G}_2 = < \{A, B, C, D\}, R >$, donde,

$R = \{(A, A), (A, B), (B, A), (C, D), (D, C), (A, C), (C, A), (B, D), (D, B)\}$.

El grafo \mathcal{G}_2 es representado por la figura 6.10.

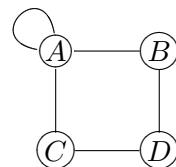


Figura 6.10: Ejemplo grafo no dirigido finito.

Nuestro siguiente problema *NP*-completo justamente es el problema del isomorfismo de grafos. Sabemos que la relación de isomorfismo es una relación entre estructuras cuya función esencial es reconocer y clasificar las que son estructuralmente idénticas. Igualmente sabemos que toda propiedad o fórmula que satisfaga una estructura debe satisfacer la otra. Tales propiedades las denominamos invariantes.

Definición 6.25 (Isomorfismo de grafos). Sean $\mathcal{G} = \langle V, R \rangle$ y $\mathcal{G}' = \langle V', R' \rangle$ dos grafos (digráficos). Decimos que \mathcal{G} es isomorfo a \mathcal{G}' , si y sólo si existe una función biyectiva $\phi: V \rightarrow V'$, tal que:

$$(v_1, v_2) \in R \Rightarrow (\phi(v_1), \phi(v_2)) \in R'; \quad \text{para todo } v_1, v_2 \in V.$$

La función $\phi: V \rightarrow V'$ se denomina isomorfismo de grafos. Escribimos $\phi: \mathcal{G} \simeq \mathcal{G}'$ para indicar que \mathcal{G} es isomorfo a \mathcal{G}' .

Ejemplo 6.11. Los grafos $\mathcal{G}_1 = \langle V_1, R_1 \rangle$ y $\mathcal{G}_2 = \langle V_2, R_2 \rangle$, representados por las figuras 6.11 y 6.12, son isomorfos.

$\mathcal{G}_1 \simeq \mathcal{G}_2$, ya que, $\bar{V}_1 = \bar{V}_2 = 4$; además podemos definir la función $\phi: V_1 \rightarrow V_2$ tal que $\phi(1) = a$, $\phi(2) = b$, $\phi(3) = c$ y $\phi(4) = d$. Entonces, $(1, 2) \in R_1$ y $(\phi(1), \phi(2)) \in R_2$; $(1, 4) \in R_1$ y $(\phi(1), \phi(4)) \in R_2$; igualmente para las parejas restantes.

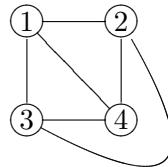


Figura 6.11: \mathcal{G}_1 isomorfo al grafo de la figura 6.12.

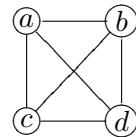


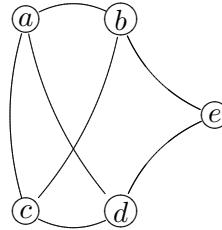
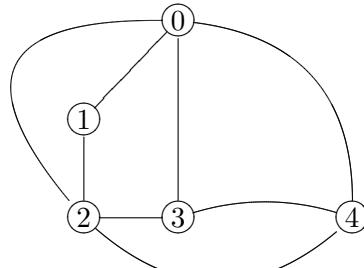
Figura 6.12: \mathcal{G}_2 isomorfo al grafo de la figura 6.11.

Un aspecto importante en el trabajo con grafos, es el concerniente a la determinación de grafos que no son isomorfos. Aunque existen algoritmos que pueden determinar en buena medida si dos pares de grafos son isomorfos, una forma de determinar que no lo son consiste en buscar una propiedad que no se preserve.

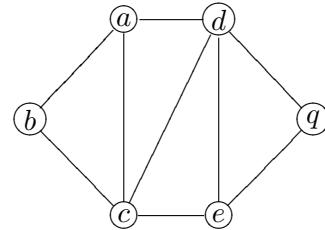
Ejemplo 6.12. Los grafos \mathcal{G}_1 y \mathcal{G}_2 , representados por las figuras 6.13 y 6.14 respectivamente, no son isomorfos puesto que \mathcal{G}_1 tiene siete lados y \mathcal{G}_2 tiene ocho lados.

Ejemplo 6.13. Los grafos \mathcal{G}_a y \mathcal{G}_b , representados por las figuras 6.15 y 6.16 respectivamente, no son isomorfos.

El vértice b en \mathcal{G}_a es adyacente a dos vértices, luego le podríamos asociar el vértice 6

Figura 6.13: \mathcal{G}_1 no isomorfo al grafo de la figura 6.14.Figura 6.14: \mathcal{G}_2 no isomorfo al grafo de la figura 6.13.

de \mathcal{G}_b . Igualmente el vértice q de \mathcal{G}_a se podría corresponder con el el vértice 1 de \mathcal{G}_b . En \mathcal{G}_a no existen más vértices de grado dos, y en \mathcal{G}_b existe todavía el vértice 3. Luego no podríamos construir una biyección que preserve las adyacencias.

Figura 6.15: \mathcal{G}_a no isomorfo al grafo de la figura 6.16.

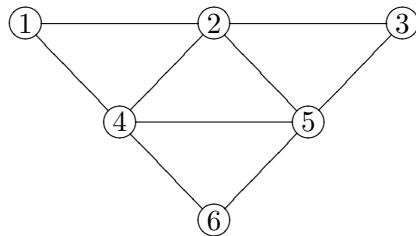
Teorema 6.22. Sean $\mathcal{G} = \langle V, R \rangle$ y $\mathcal{G}' = \langle V', R' \rangle$ dos grafos (digrafos). Determinar si existe o no un isomorfismo entre los grafos \mathcal{G} y \mathcal{G}' es un problema NP-completo.

6.13. Ejercicios

Ejercicio 6.1. Construir una máquina de Turing 2-cintas tal como es indicado en el ejemplo 6.1 (pág. 184).

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Figura 6.16: \mathcal{G}_b no isomorfo al grafo de la figura 6.15.

Ejercicio 6.2. Construir una máquina de Turing tal como es indicado en el ejemplo 6.4 (pág. 188).

Ejercicio 6.3. Construir una máquina de Turing tal como es indicado en el ejemplo ejemplo 6.6 (pág. 192).

Ejercicio 6.4. Demostrar el teorema 6.9 (pág. 196).

Ejercicio 6.5. Demostrar el teorema 6.10 (pág. 197).

Ejercicio 6.6. Demostrar que las funciones $n^2, 2^n, n!$ son funciones espacialmente construibles de manera completa

Ejercicio 6.7. Demostrar el teorema 6.11 (pág. 197).

Ejercicio 6.8. Demostrar el teorema 6.17 (pág. 202).

Ejercicio 6.9. Demostrar el teorema 6.20 (pág. 203).

Ejercicio 6.10. Presentar una indicación para demostrar el teorema 6.21 (pág. 203).

6.14. Notas Bibliográficas

Las definiciones de máquinas de Turing k -cintas y máquinas de Turing $(k, 1)$ -cintas así como las definiciones de clase de complejidad temporal y espacial (determinista y no determinista) para un lenguaje fueron tomadas de [17, 20, 28]. Las definiciones de lenguaje recursivo y lenguaje recursivamente enumerable fueron tomadas de [28]. Los ejemplos 6.1 (pág. 184), 6.3 (pág. 185) y 6.6 (pág. 192) son presentados por [17, 28]. Las convenciones $T(n) \equiv_{def} \max(n+1, 'T(n)')$ y $S(n) \equiv_{def} \max(1, 'S(n)')$ así como los ejemplos 6.3 (pág. 185) y 6.5 (pág. 191) son presentados por [17]. La notación asintótica es presentada por [5]. Los teoremas 6.2 (pág. 187) y 6.3 (pág. 187) y el ejemplo 6.4 (pág. 188) fueron tomados de [28]. El teorema 6.4 (pág. 190) es presentado por [17, 20, 28]. El teorema 6.5 (pág. 193) es presentado por [17, 28]. El teorema 6.6 (pág. 193) es presentado por [20]. Una versión un

poco diferente del teorema 6.7 (pág. 193) la ofrecen [17, 20]. Las definiciones de máquinas de Turing no deterministas y las clases de complejidad espaciales y temporales no deterministas se tomaron de [17, 28]. El teorema 6.8 (pág. 196) es presentado por [20]. Las nociones de función espacialmente construible y función espacialmente construible de manera completa son dadas por [17]. El teorema 6.9 (pág. 196) se tomó de [17, 28]. El teorema 6.10 (pág. 197) aparece en [17]. El teorema 6.11 (pág. 197) lo ofrecen [17, 20, 28]. El teorema 6.12 (pág. 197) es presentado por [20, 28]. Los teoremas 6.13 (pág. 197), 6.14 (pág. 198) y 6.15 (pág. 198) aparecen en [17, 20]. Los elementos de la sección 6.12 son dados por [11, 17, 20, 28]. El problema $P \stackrel{?}{=} NP$ fue propuesto por Steve Smale en 1.998 (medalla Field en matemáticas) como uno de los problemas matemáticos más importantes para el próximo siglo [35]; en la misma dirección, el *Clay Mathematics Institute* ofreció en el año 2.000 un premio bastante cuantioso en efectivo por la solución de siete problemas matemáticos, entre ellos el problema $P \stackrel{?}{=} NP$ [18]. Algunos textos en teoría de grafos son [2, 6, 13, 19, 22]. Los isomorfismos entre grafos son presentados por [2, 8, 14]. El teorema 6.22 (pág. 206) lo ofrece [11]. El libro [11] ofrece más de 300 problemas NP -completos. El último problema NP -completo de que los autores tienen conocimiento, es la solución general al juego “busca minas” del sistema operativo *Windows* [37].

Bibliografía

- [1] ALFRED V. AHO, RAVI SETHI Y JEFFREY D. ULLMAN. "Compiladores: principios, técnicas y herramientas". Wilmington, Delaware: Addison-Wesley Iberoamericana (1990).
- [2] KENNETH P. BOGART. "Matemáticas discretas". México D.F.: Editorial Limusa S.A. de C.V. (1996).
- [3] GEORGE BOOLOS Y RICHARD JEFFREY. "Computability and logic". Cambridge University Press, 3a. edición (1989).
- [4] XAVIER CAICEDO. "Elementos de lógica y calculabilidad". Santafé de Bogotá: Una empresa docente, U. de los Andes, 2a. edición (1990).
- [5] THOMAS CORMEN, CHARLES E. LEISERSON Y RONALD R. RIVEST. "Introduction to algorithms". The MIT Press (1995). 5a printing.
- [6] DECOROSO CRESPO. "Informática teórica. Primera parte". Madrid: Departamento de Publicaciones de la Facultad de Informática, U. Politecnica de Madrid. (1983).
- [7] MARTIN DAVIS. "Computability and unsolvability". New York: Dover Publications, Inc. (1982).
- [8] NARSING DEO. "Graph theory with application to engineering and computer science". Englewood Cliffs, N.J.: Prentice-Hall, Inc. (1974).
- [9] GÜNTER DOTZEL. The Modulator forum: letters to the editor. *The ModulaTor Technical Publication 11* (1991). Eprint: www.modulaware.com/mdl117.htm [15-Feb-1999].
- [10] GREGORIO FERNÁNDEZ Y FERNANDO SÁEZ. "Fundamentos de informática". Madrid: Alianza Editorial (1987).
- [11] MICHAEL R. GAREY Y DAVID S JOHNSON. "Computers and intractability. A guide to the theory of NP-completeness". New York: W. H. Freeman and Company (1979).
- [12] PEDRO GÓMEZ Y CRISTINA GÓMEZ. "Sistemas formales, informalmente". Santafé de Bogotá: Una empresa docente, U. de los Andes (1992). Segunda versión.
- [13] KARL GRASSMAN Y JEAN-PAUL TREMBLAY. "Matemáticas discretas y lógica: una perspectiva desde la ciencia de la computación". Madrid: Prentice Hall (1997).
- [14] RALPH P. GRIMALDI. "Matemáticas discreta y combinatoria". Wilmington, Delaware: Addison-Wesley Iberoamericana, 3a. edición (1997).
- [15] HANS HERMES. "Enumerability · Decidability · Computability". Berlin: Springer-Verlag (1969).

Esta guía de estudio, fue vendida por www.guiaconeval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaconeval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

- [16] JOHN HOPCROFT. Máquinas de Turing. *Investigación y Ciencia* **94**, 8–19 (julio 1984).
- [17] JOHN HOPCROFT Y JEFFEREY ULLMAN. “Introducción a la teoría de automátas, lenguajes y computación”. México D.F.: Compañía Editorial Continental, S.A. (CECSA) (1997).
- [18] CLAY MATHEMATICS INSTITUTE. Millennium prize problems (2000).
Eprint: www.claymath.org/prize_problems/ [10-Ago-2000].
- [19] RICHARD JOHNSONBAUG. “Matemáticas discretas”. México D.F.: Grupo Editorial Iberoamérica, S.A. (1988).
- [20] DEAN KELLEY. “Teoría de autómatas y lenguajes formales”. Prentice-Hall (UK) Ltd. (1995).
- [21] STEPHEN C. KLEENE. “Introducción a la metamatemática”. Colección: Estructura y Función. Madrid: Editorial Tecnos (1974).
- [22] BERNARD KOLMAN Y ROBERT BUSBY. “Discrete mathematical structures for computer sciences”. Englewood Cliffs, N.J.: Prentice-Hall Inc. (1984).
- [23] JEAN LADRIÈRE. “Limitaciones internas de los formalismos”. Madrid: Editorial Tecnos (1969).
- [24] LEÓN LÓPEZ LÓPEZ Y RÁUL GÓMEZ MARÍN. “Matemáticas básicas para la informática, volumen I”. Medellín: U. EAFIT (1993).
- [25] HEINER MARXEN Y JÜRGEN BUNTROCK. Attacking the Busy Beaver 5. *Bulletin of the EATCS* **000**(40), 247–251 (1990). Eprint: www.drb.insel.de/~heiner/BB/bb-mabu90.ps [08-Jun-2000].
- [26] ELLIOTT MENDELSON. “Introduction to mathematical logic”. The university series in undergraduate mathematics. New York: D. Van Nostrand Company, Inc. (1965).
- [27] MARVIN L. MINSKY. “Computation: finite and infinite machines”. Englewood Cliffs, N.J.: Prentice-Hall, Inc. (1967).
- [28] CHRISTOS PAPADIMITRIOU. “Computational complexity”. Reading: Addison-Wesley Publishing Company (1994).
- [29] ROGER PENROSE. “La nueva mente del emperador”. Colección: Libro de mano No. 38. Barcelona: Grijalbo Mondadori (1991).
- [30] ROGER PENROSE. “Las sombras de la mente”. Colección: Drakontos. Barcelona: Crítica (1996).
- [31] JEFFEREY SHALLIT. The Busy Beaver problem. Eprint: grail.cba.csuohio.edu/~somos/beaver.ps [08-Jun-2000] (1998).
- [32] ANDRÉS SICARD. Máquina universal de Turing: Algunas indicaciones para su construcción. *Rev. U. EAFIT* **108**, 61–106 (1997).
- [33] ANDRÉS SICARD RAMÍREZ. Máquinas de Turing. *Rev. U. EAFIT* **103**, 29–45 (1996).
- [34] WILFRED SIEG. Step by recursive step: Church’s analysis of effective calculability. *The Bulletin of Symbolic Logic* **3**(2), 154–180 (june 1997).
- [35] STEVE SMALE. Mathematical problems for the next century. *The Mathematical Intelligencer* **20**(2), 7–13 (1998).
- [36] ROBERT I. SOARE. Computability and recursion. *The Bulletin of Symbolic Logic* **2**(3), 284–321 (sept. 1996).

Esta guía de estudio, fue vendida por www.guiaconeval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaconeval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

- [37] IAN STEWART. Million-Dollar Minesweeper. Eprint: www.claymath.org/prize_problems/million-dollar-minesweeper.htm [07-Dic-2000] (2000).
- [38] ALAN M. TURING. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.* **42**, 230–265 (1936-7). A correction, *ibid*, vol. 43, no. 2198, p. 544–546, 1937.
- [39] ANN YASUHARA. “Recursive function theory and logic”. Computer Science and Applied Mathematics. Series of monographs and textbooks. New York: Academic Press (1971).

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Esta guía fue vendida por www.guiaceneval.com.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Índice alfabético

- AFD, *véase* autómata de estado finito determinista
AFN, *véase* autómata de estado finito no determinista
alfabeto, 99
ambigüedad, *véase* gramática
APND, *véase* autómata de pila no determinista
árbol de análisis sintáctico, *véase* gramática
autómata de *stack*, *véase* autómata de pila no determinista
autómata de estado finito
 comportamiento entrada-estados, 152
 definición, 140
 determinista, 145
 diagrama de transición, 141
 equivalencia entre, 162
 no determinista, 146
 relación de equirrespuesta, 152
 representación explícita, 142
 tabla de transición, 141
autómata de pila no determinista
 como reconocedor, 169
 definición, 166
- clase de complejidad
espacial
 determinista, 192
 no determinista, 195
temporal
 clase *NP*, 199
 clase *P*, 199
 determinista, 185
 no determinista, 195
codificación
 de Gödel, 44
 para una instrucción, 46
 para una máquina de Turing, 47
- de n-tuplas números naturales, 34
de números naturales, 34
de Turing, 40
- conjunto
decidible, 75
efectivamente enumerable, 79
enumerable, 76
primitivo recursivo, 74
recursivamente enumerable, 77
recursivo, 74
- cuantificador
 acotado, 64
- derivación, *véase* gramática
- descripción
 instantánea, 32
 cambio de, 33
 terminal, 33
- esquema
 de composición, 59
 de minimalización, 68
 de recurrencia primitiva, 59
- expresión regular, 123
- FPR, *véase* función primitiva recursiva
función
 asociada a una máquina de Turing, 35
 computable tiempo polinómico, 201
 espacialmente construible, 196
 de manera completa, 197
 numérico-teórica, 57
 parcial, 32
 Turing-computable, 35
 primitiva recursiva, 60
 recursiva, 68
 recursiva parcial, 72

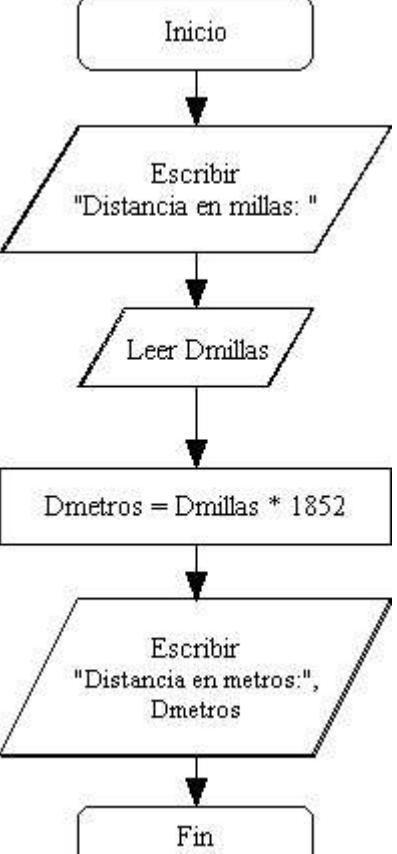
- regular, 67
- total, 32
- Turing-computable, 35
- grafo
 - digrafo, véase grafo dirigido
 - dirigido, 203
 - isomorfismo entre, 205
 - no dirigido, 204
- gramática
 - ambigüedad de, 115
 - árbol de análisis sintáctico, 112
 - definición, 107
 - derivación, 113
 - lenguaje generado por, 107
 - notación BNF, 111
 - recursividad, 117
 - taxonomía, 110
- lenguaje
 - \mathcal{C} -completo, 202
 - \mathcal{C} -difícil, 202
 - definición, 100
 - lenguaje universal, 99
 - operaciones entre
 - clausura de Kleene, 104
 - clausura positiva, 104
 - complemento, 104
 - concatenación, 104
 - intersección, 104
 - reflexión, 104
 - unión, 104
 - recursivamente enumerable, 184
 - recursivo, 184
- m-función
 - definición, 38
 - expansión, 39
- máquina de estado finito
 - de Mealy, 136
 - de Moore, 137
 - definición, 134
 - diagrama de transición, 135
 - representación explícita, 136
 - tabla de transición, 135
- máquina de Turing
 - ($k, 1$)-cintas, 191
 - k -cintas, 183
 - codificación de Gödel para una, 43
 - codificación de Turing para una, 40
 - computación de una, 33
 - definición, 27
 - descripción estándar para una, 41
 - función asociada con una, 35
 - no determinista, 194
 - número de descripción para una, 42
 - universal, 48
- monoide, 103
 - homomorfismo entre, 151
- notación asintótica
 - O , 186
 - Ω , 187
 - Θ , 187
- número
 - de Gödel, 45
- operador
 - de minimización μ , 67
- palabra
 - concatenación, 101
 - definición, 99
 - longitud de, 100
 - potencia de, 101
 - reflexión de, 102
- partición
 - índice finito, 153
 - refinamiento para, 155
- PPR, véase predicado primitivo recursivo
- predicado
 - primitivo recursivo, 63
- problema
 - de la decisión, 25
 - de la parada, 49
 - del castor afanoso, 51
- reconocedor finito
 - definición, 142
 - equivalecia entre, 162
 - lenguaje aceptado por, 143
 - palabra aceptada por, 142
 - relación de equirrespuesta, 154
- recursividad, véase gramática

- reducción
 - en tiempo polinómico, 201
- relación
 - de congruencia, 153
 - de congruencia derecha, 153
 - de congruencia izquierda, 153
 - de equirrespuesta, véase autómata de estado finito, reconocedor finito
 - inducida por un lenguaje, 156
 - numérico-teórica, 57
- semigrupo, 102
- sistema
 - combinatorio, 106
 - formal, 105
- sucesión
 - de la cinta, 33
 - definición, 32
- tesis
 - de Church-Turing, 93

Lectura 16. Metodología de la programación (Ejercicios)

Ejercicios resueltos

1. Diseñar el algoritmo (ordinograma y pseudocódigo) correspondiente a un programa que lea el valor correspondiente a una distancia en millas marinas y las escriba expresadas en metros. Sabiendo que 1 milla marina equivale a 1852 metros.

<i>Ordinograma</i>	<i>Pseudocódigo</i>								
 <pre> graph TD Inicio([Inicio]) --> P1[/Escribir "Distancia en millas:"/] P1 --> P2[/Leer Dmillas/] P2 --> R1[Dmetros = Dmillas * 1852] R1 --> P3[/Escribir "Distancia en metros:", Dmetros/] P3 --> Fin([Fin]) </pre>	<p>PROGRAMA: Millas_y_metros MÓDULO: Principal</p> <p>INICIO</p> <p>DATOS:</p> <table> <tr> <td>CONSTANTES</td> <td></td> </tr> <tr> <td>MILL_METR</td> <td>Numérico Entero = 1852</td> </tr> </table> <p>VARIABLES</p> <table> <tr> <td>Dmillas</td> <td>Numérico Entero</td> </tr> <tr> <td>Dmetros</td> <td>Numérico Entero</td> </tr> </table> <p>ALGORITMO:</p> <ul style="list-style-type: none"> Escribir "Distancia en millas: " Leer Dmillas Dmetros = Dmillas * MILL_METR Escribir "Distancia en metros:", Dmetros <p>FIN</p>	CONSTANTES		MILL_METR	Numérico Entero = 1852	Dmillas	Numérico Entero	Dmetros	Numérico Entero
CONSTANTES									
MILL_METR	Numérico Entero = 1852								
Dmillas	Numérico Entero								
Dmetros	Numérico Entero								

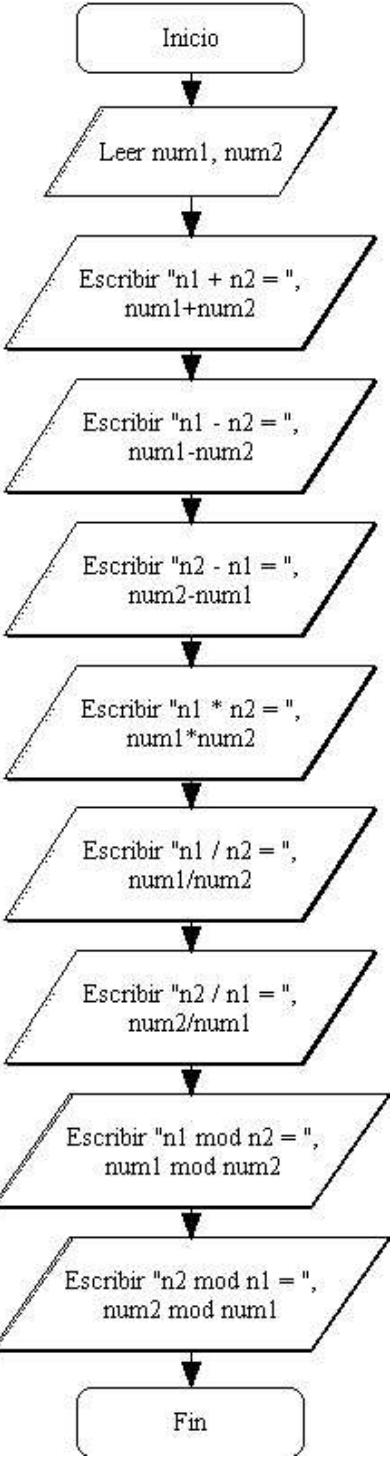
Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

2. Diseñar el algoritmo (ordinograma y pseudocódigo) correspondiente a un programa que escribe el porcentaje descontado en una compra, introduciendo por teclado el precio de la tarifa y el precio pagado.

<i>Ordinograma</i>	<i>Pseudocódigo</i>										
<pre> graph TD Inicio([Inicio]) --> LeerTarifa{Leer Tarifa} LeerTarifa --> LeerPrecio{Leer Precio} LeerPrecio --> CalculoDto[Dto = Tarifa - Precio] CalculoDto --> CalculoPd[Pd = Dto * 100 / Tarifa] CalculoPd --> Escribir[Escribir "Porcentaje de descuento:", Pd] Escribir --> Fin([Fin]) </pre>	<p>PROGRAMA: Descuento MÓDULO: Principal</p> <p>INICIO</p> <p>DATOS:</p> <table> <tr><td>VARIABLES</td><td></td></tr> <tr><td>Tarifa</td><td>Numérico Entero</td></tr> <tr><td>Precio</td><td>Numérico Entero</td></tr> <tr><td>Dto</td><td>Numérico Entero</td></tr> <tr><td>Pd</td><td>Numérico Real</td></tr> </table> <p>ALGORITMO:</p> <ul style="list-style-type: none"> Leer Tarifa Leer Precio $Dto = Tarifa - Precio$ $Pd = Dto * 100 / Tarifa$ Escribir "Porcentaje de descuento:", Pd <p>FIN</p>	VARIABLES		Tarifa	Numérico Entero	Precio	Numérico Entero	Dto	Numérico Entero	Pd	Numérico Real
VARIABLES											
Tarifa	Numérico Entero										
Precio	Numérico Entero										
Dto	Numérico Entero										
Pd	Numérico Real										

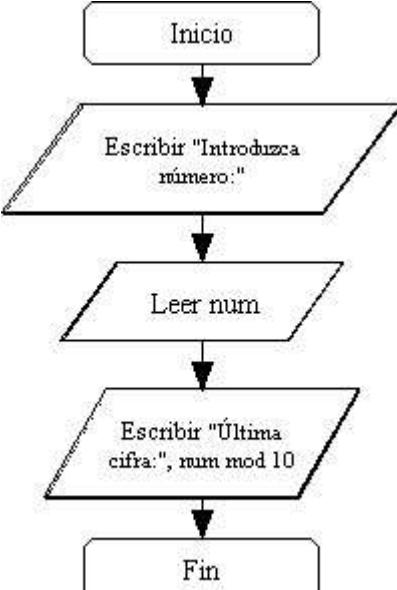
Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

3. Diseñar el algoritmo (ordinograma y pseudocódigo) correspondiente a un programa que pida por teclado dos números enteros y muestre su suma, resta, multiplicación, división y el resto (módulo) de la división. Si la operación no es commutativa, también se mostrará el resultado invirtiendo los operadores.

Ordinograma	Pseudocódigo				
 <pre> graph TD Inicio([Inicio]) --> Leer[/Leer num1, num2/] Leer --> Suma[/Escribir "n1 + n2 = ", num1+num2/] Suma --> Resta[/Escribir "n1 - n2 = ", num1-num2/] Resta --> RestaInv[/Escribir "n2 - n1 = ", num2-num1/] RestaInv --> Multiplicacion[/Escribir "n1 * n2 = ", num1*num2/] Multiplicacion --> Division[/Escribir "n1 / n2 = ", num1/num2/] Division --> DivisionInv[/Escribir "n2 / n1 = ", num2/num1/] Fin([Fin]) </pre>	<p>PROGRAMA: Operaciones_aritméticas MÓDULO: Principal</p> <p>INICIO</p> <p>DATOS:</p> <p>VARIABLES</p> <table> <tr> <td>num1</td> <td>Numérico Entero</td> </tr> <tr> <td>num2</td> <td>Numérico Entero</td> </tr> </table> <p>ALGORITMO:</p> <ul style="list-style-type: none"> Leer num1, num2 Escribir "n1 + n2 = ", num1+num2 Escribir "n1 - n2 = ", num1-num2 Escribir "n2 - n1 = ", num2-num1 Escribir "n1 * n2 = ", num1*num2 Escribir "n1 / n2 = ", num1/num2 Escribir "n2 / n1 = ", num2/num1 Escribir "n1 mod n2 = ", num1 mod num2 Escribir "n2 mod n1 = ", num2 mod num1 <p>FIN</p>	num1	Numérico Entero	num2	Numérico Entero
num1	Numérico Entero				
num2	Numérico Entero				

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

4. Diseñar el algoritmo (ordinograma y pseudocódigo) correspondiente a un programa que obtiene la última cifra de un número introducido.

Ordinograma	Pseudocódigo
 <pre> graph TD Inicio([Inicio]) --> P1[/Escribir "Introduzca número:"/] P1 --> P2[/Leer num/] P2 --> P3[/Escribir "Última cifra:", num mod 10/] P3 --> Fin([Fin]) </pre>	<p>PROGRAMA: Última_cifra MÓDULO: Principal</p> <p>INICIO</p> <p>DATOS: VARIABLES num Numérico Entero</p> <p>ALGORITMO:</p> <p>Escribir "Introduzca número:" Leer num Escribir "Última cifra: ", num mod 10</p> <p>FIN</p>

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

5. Diseñar el algoritmo (ordinograma y pseudocódigo) correspondiente a un programa que calcule el área y el perímetro de un triángulo rectángulo dada la base y la altura.

Ordinograma	Pseudocódigo										
<pre> graph TD Inicio([Inicio]) --> Leer[/Leer base, altura/] Leer --> Area[area = base * altura / 2] Area --- Line1[] Line1 --- Hipotenusa[hipotenusa = raiz (base^2 + altura^2)] Line1 --- Line2[] Line2 --- Perimetro[perimetro = base + altura + hipotenusa] Perimetro --- EscribirA[/Escribir "Área = ", area/] EscribirA --- EscribirP[/Escribir "Perímetro = ", perimetro/] EscribirP --- Fin([Fin]) </pre>	<p>PROGRAMA: Triángulo_rectángulo MÓDULO: Principal</p> <p>INICIO</p> <p>DATOS:</p> <table> <tr> <td>base</td> <td>Numérico Entero</td> </tr> <tr> <td>altura</td> <td>Numérico Entero</td> </tr> <tr> <td>area</td> <td>Numérico Real</td> </tr> <tr> <td>hipotenusa</td> <td>Numérico Real</td> </tr> <tr> <td>perimetro</td> <td>Numérico Real</td> </tr> </table> <p>ALGORITMO:</p> <ul style="list-style-type: none"> Leer base, altura area = base * altura / 2 hipotenusa = Raiz (base^2 + altura^2) perimetro = base + altura + hipotenusa Escribir "Área = ", area Escribir "Perímetro = ", perimetro <p>FIN</p>	base	Numérico Entero	altura	Numérico Entero	area	Numérico Real	hipotenusa	Numérico Real	perimetro	Numérico Real
base	Numérico Entero										
altura	Numérico Entero										
area	Numérico Real										
hipotenusa	Numérico Real										
perimetro	Numérico Real										

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

6. Diseñar el algoritmo (ordinograma y pseudocódigo) correspondiente a un programa que tras introducir una medida expresada en centímetros la convierta en pulgadas (1 pulgada = 2,54 centímetros)

<i>Ordinograma</i>	<i>Pseudocódigo</i>										
<pre> graph TD Inicio([Inicio]) --> Output1[/Escribir "Valor en centímetros:"/] Output1 --> Leer[Leer cm] Leer --> Process[pulgadas = cm / 2.54] Process --> Output2[/Escribir "Pulgadas: ", pulgadas/] Output2 --> Fin([Fin]) </pre>	PROGRAMA: Cent_Pulgadas MÓDULO: Principal INICIO DATOS: <table style="margin-left: 20px;"> <tr> <td>CONSTANTES</td> <td></td> </tr> <tr> <td>CM_PULG</td> <td>Numérico Real = 2.54</td> </tr> <tr> <td>VARIABLES</td> <td></td> </tr> <tr> <td>cm</td> <td>Numérico Real</td> </tr> <tr> <td>pulgadas</td> <td>Numérico Real</td> </tr> </table> ALGORITMO: <ul style="list-style-type: none"> Escribir “Valor en centímetros:” Leer cm pulgadas = cm / CM_PULG Escribir “Pulgadas:”, pulgadas FIN	CONSTANTES		CM_PULG	Numérico Real = 2.54	VARIABLES		cm	Numérico Real	pulgadas	Numérico Real
CONSTANTES											
CM_PULG	Numérico Real = 2.54										
VARIABLES											
cm	Numérico Real										
pulgadas	Numérico Real										

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

7. Diseñar el algoritmo (ordinograma y pseudocódigo) correspondiente a un programa que exprese en horas, minutos y segundos un tiempo expresado en segundos.

Ordinograma	Pseudocódigo								
<pre> graph TD Init([Init]) --> Leer{Leer segundos} Leer --> Div1[horas = segundos / 3600] Div1 --> Mod1[segundos = segundos mod 3600] Mod1 --> Div2[minutos = segundos / 60] Div2 --> Mod2[segundos = segundos mod 60] Mod2 --> Escribir{Escribir horas, "h ", minutos, "m ", segundos, "s"} Escribir --> Fin([Fin]) </pre>	<p>PROGRAMA: Tiempo_segundos MÓDULO: Principal</p> <p>INICIO</p> <p>DATOS:</p> <table> <tr> <td>VARIABLES</td> <td></td> </tr> <tr> <td>segundos</td> <td>Numérico Entero</td> </tr> <tr> <td>minutos</td> <td>Numérico Entero</td> </tr> <tr> <td>horas</td> <td>Numérico Entero</td> </tr> </table> <p>ALGORITMO:</p> <ul style="list-style-type: none"> Leer segundos horas = segundos / 3600 segundos = segundos mod 3600 minutos = segundos / 60 segundos = segundos mod 60 Escribir horas, "h ", minutos, "m ", segundos, "s" <p>FIN</p>	VARIABLES		segundos	Numérico Entero	minutos	Numérico Entero	horas	Numérico Entero
VARIABLES									
segundos	Numérico Entero								
minutos	Numérico Entero								
horas	Numérico Entero								

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

8. Diseñar el algoritmo (ordinograma y pseudocódigo) correspondiente a un programa que pida el total de kilómetros recorridos, el precio de la gasolina (por litro), el dinero de gasolina gastado en el viaje y el tiempo que se ha tardado (en horas y minutos) y que calcule:

- Consumo de gasolina (en litros y euros) por cada 100 km.
- Consumo de gasolina (en litros y euros) por cada km.
- Velocidad media (en km/h y m/s).

Ordinograma	Pseudocódigo																																																				
<pre> graph TD Inicio([Inicio]) --> Leer[Leer km, precio, dinero, horas, minutos] Leer --> litros[dinero/precio] litros --> litrosKm[litros/km] litrosKm --> dineroKm[dinero/km] dineroKm --> horastotal[horas + minutos/60] horastotal --> Kmh[km/h = km/horastotal] Kmh --> ms["ms = (km*1000)/(horastotal*3600)"] ms --> Tlitros100["Tlitros100 = litroskm*100"] Tlitros100 --> Teuros100["Teuros100 = dineroskm*100"] Teuros100 --> Consumo100Km["Consumo de gasolina por cada 100 Km"] Consumo100Km --> Fin([Fin]) </pre>	<p>PROGRAMA: Consumo_viaje MÓDULO: Principal</p> <p>INICIO</p> <p>DATOS:</p> <table> <tbody> <tr><td>km</td><td>Numérico Entero</td></tr> <tr><td>precio</td><td>Numérico Real</td></tr> <tr><td>dinero</td><td>Numérico Real</td></tr> <tr><td>horas</td><td>Numérico Entero</td></tr> <tr><td>minutos</td><td>Numérico Entero</td></tr> <tr><td>litros</td><td>Numérico Real</td></tr> <tr><td>litroskm</td><td>Numérico Real</td></tr> <tr><td>dinerskm</td><td>Numérico Real</td></tr> <tr><td>horastotal</td><td>Numérico Real</td></tr> <tr><td>kmh</td><td>Numérico Real</td></tr> <tr><td>ms</td><td>Numérico Real</td></tr> <tr><td>Tlitros100</td><td>Numérico Real</td></tr> <tr><td>Teuros100</td><td>Numérico Real</td></tr> </tbody> </table> <p>VARIABLES</p> <table> <tbody> <tr><td>km</td><td>Numérico Entero</td></tr> <tr><td>precio</td><td>Numérico Real</td></tr> <tr><td>dinero</td><td>Numérico Real</td></tr> <tr><td>horas</td><td>Numérico Entero</td></tr> <tr><td>minutos</td><td>Numérico Entero</td></tr> <tr><td>litros</td><td>Numérico Real</td></tr> <tr><td>litroskm</td><td>Numérico Real</td></tr> <tr><td>dinerskm</td><td>Numérico Real</td></tr> <tr><td>horastotal</td><td>Numérico Real</td></tr> <tr><td>kmh</td><td>Numérico Real</td></tr> <tr><td>ms</td><td>Numérico Real</td></tr> <tr><td>Tlitros100</td><td>Numérico Real</td></tr> <tr><td>Teuros100</td><td>Numérico Real</td></tr> </tbody> </table> <p>ALGORITMO:</p> <p>Leer km, precio, dinero, horas, minutos</p> <p>** Litros consumidos totales $\text{litros} = \text{dinero} / \text{precio}$</p> <p>** Litros por km $\text{litroskm} = \text{litros} / \text{km}$</p> <p>** Dinero por km $\text{dinerskm} = \text{dinero} / \text{km}$</p> <p>** Tiempo empleado, convertido a horas $\text{horastotal} = \text{horas} + \text{minutos} / 60$</p> <p>** Velocidad media (km/h, m/s) $\text{kmh} = \text{km} / \text{horastotal}$ $\text{ms} = (\text{km} * 1000) / (\text{horastotal} * 3600)$</p> <p>** Consumos por cada 100 km $\text{Tlitros100} = \text{litroskm} * 100$ $\text{Teuros100} = \text{dinerskm} * 100$</p> <p>Escribir "Consumo de gasolina cada 100 Km" Escribir "En litros:", Tlitros100 Escribir "En euros:", Teuros100 Escribir "Consumo de gasolina por cada Km" Escribir "En litros:", litroskm Escribir "En euros:", dinerskm Escribir "Velocidad media en Km/h:", kmh Escribir "Velocidad media en m/s:", ms</p>	km	Numérico Entero	precio	Numérico Real	dinero	Numérico Real	horas	Numérico Entero	minutos	Numérico Entero	litros	Numérico Real	litroskm	Numérico Real	dinerskm	Numérico Real	horastotal	Numérico Real	kmh	Numérico Real	ms	Numérico Real	Tlitros100	Numérico Real	Teuros100	Numérico Real	km	Numérico Entero	precio	Numérico Real	dinero	Numérico Real	horas	Numérico Entero	minutos	Numérico Entero	litros	Numérico Real	litroskm	Numérico Real	dinerskm	Numérico Real	horastotal	Numérico Real	kmh	Numérico Real	ms	Numérico Real	Tlitros100	Numérico Real	Teuros100	Numérico Real
km	Numérico Entero																																																				
precio	Numérico Real																																																				
dinero	Numérico Real																																																				
horas	Numérico Entero																																																				
minutos	Numérico Entero																																																				
litros	Numérico Real																																																				
litroskm	Numérico Real																																																				
dinerskm	Numérico Real																																																				
horastotal	Numérico Real																																																				
kmh	Numérico Real																																																				
ms	Numérico Real																																																				
Tlitros100	Numérico Real																																																				
Teuros100	Numérico Real																																																				
km	Numérico Entero																																																				
precio	Numérico Real																																																				
dinero	Numérico Real																																																				
horas	Numérico Entero																																																				
minutos	Numérico Entero																																																				
litros	Numérico Real																																																				
litroskm	Numérico Real																																																				
dinerskm	Numérico Real																																																				
horastotal	Numérico Real																																																				
kmh	Numérico Real																																																				
ms	Numérico Real																																																				
Tlitros100	Numérico Real																																																				
Teuros100	Numérico Real																																																				

Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a

9. Diseñar el algoritmo (ordinograma y pseudocódigo) correspondiente a un programa que al introducir una cantidad de dinero expresado en euros nos indique cuántos billetes y monedas se puede tener como mínimo.

<i>Ordinograma</i>	<i>Pseudocódigo</i>																																		
<pre> graph TD Inicio([Inicio]) --> Escritor[Escibir "Importe en euros."] Escritor --> Leer[Leer cant_euros] Leer --> B500["b_500 = cant_euros div 500"] B500 --> CantEuros["cant_euros = cant_euros - b_500*500"] CantEuros --> B200["b_200 = cant_euros div 200"] B200 --> CantEuros["cant_euros = cant_euros - b_200*200"] CantEuros --> B100["b_100 = cant_euros div 100"] B100 --> CantEuros["cant_euros = cant_euros - b_100*100"] CantEuros --> B50["b_50 = cant_euros div 50"] B50 --> CantEuros["cant_euros = cant_euros - b_50*50"] CantEuros --> B20["b_20 = cant_euros div 20"] B20 --> CantEuros["cant_euros = cant_euros - b_20*20"] CantEuros --> B10["b_10 = cant_euros div 10"] B10 --> CantEuros["cant_euros = cant_euros - b_10*10"] CantEuros --> B5["b_5 = cant_euros div 5"] B5 --> CantEuros["cant_euros = cant_euros - b_5*5"] CantEuros --> M2["m_2 = cant_euros div 2"] M2 --> CantEuros["cant_euros = cant_euros - m_2*2"] CantEuros --> M1["m_1 = cant_euros div 1"] M1 --> CantEuros["cant_euros = cant_euros - m_1*1"] CantEuros --> Fin1((1)) </pre>	<p>PROGRAMA: Billetes_Monedas MÓDULO: Principal</p> <p>INICIO</p> <p>DATOS:</p> <table> <tr><td>cant_euros</td><td>Numérico Real</td></tr> <tr><td>resto</td><td>Numérico Real</td></tr> <tr><td>b_500</td><td>Numérico Entero</td></tr> <tr><td>b_200</td><td>Numérico Entero</td></tr> <tr><td>b_100</td><td>Numérico Entero</td></tr> <tr><td>b_50</td><td>Numérico Entero</td></tr> <tr><td>b_20</td><td>Numérico Entero</td></tr> <tr><td>b_10</td><td>Numérico Entero</td></tr> <tr><td>b_5</td><td>Numérico Entero</td></tr> <tr><td>m_2</td><td>Numérico Entero</td></tr> <tr><td>m_1</td><td>Numérico Entero</td></tr> <tr><td>m_050</td><td>Numérico Entero</td></tr> <tr><td>m_020</td><td>Numérico Entero</td></tr> <tr><td>m_010</td><td>Numérico Entero</td></tr> <tr><td>m_005</td><td>Numérico Entero</td></tr> <tr><td>m_002</td><td>Numérico Entero</td></tr> <tr><td>m_001</td><td>Numérico Entero</td></tr> </table> <p>ALGORITMO:</p> <pre> Escribir "Introduzca importe en euros: " Leer cant_euros b_500 = cant_euros div 500 ** Se obtiene el resto con una operación matemática ** ya que no se puede usar el operador mod con reales cant_euros = cant_euros - b_500 * 500 b_200 = cant_euros div 200 cant_euros = cant_euros - b_200 * 200 b_100 = cant_euros div 100 cant_euros = cant_euros - b_100 * 100 b_50 = cant_euros div 50 cant_euros = cant_euros - b_50 * 50 b_20 = cant_euros div 20 cant_euros = cant_euros - b_20 * 20 b_10 = cant_euros div 10 cant_euros = cant_euros - b_10 * 10 b_5 = cant_euros div 5 cant_euros = cant_euros - b_5 * 5 m_2 = cant_euros div 2 cant_euros = cant_euros - m_2 * 2 m_1 = cant_euros div 1 cant_euros = cant_euros - m_1 * 1 m_050 = cant_euros div 0.50 cant_euros = cant_euros - m_050 * 0.50 m_020 = cant_euros div 0.20 cant_euros = cant_euros - m_020 * 0.20 m_010 = cant_euros div 0.10 cant_euros = cant_euros - m_010 * 0.10 m_005 = cant_euros div 0.05 cant_euros = cant_euros - m_005 * 0.05 m_002 = cant_euros div 0.02 cant_euros = cant_euros - m_002 * 0.02 m_001 = cant_euros div 0.01 cant_euros = cant_euros - m_001 * 0.01 Escribir b_500, b_200, b_100, b_50, b_20, b_10, b_5, m_2, m_1, m_050, m_020, m_010, m_005, m_002, m_001 FIN </pre>	cant_euros	Numérico Real	resto	Numérico Real	b_500	Numérico Entero	b_200	Numérico Entero	b_100	Numérico Entero	b_50	Numérico Entero	b_20	Numérico Entero	b_10	Numérico Entero	b_5	Numérico Entero	m_2	Numérico Entero	m_1	Numérico Entero	m_050	Numérico Entero	m_020	Numérico Entero	m_010	Numérico Entero	m_005	Numérico Entero	m_002	Numérico Entero	m_001	Numérico Entero
cant_euros	Numérico Real																																		
resto	Numérico Real																																		
b_500	Numérico Entero																																		
b_200	Numérico Entero																																		
b_100	Numérico Entero																																		
b_50	Numérico Entero																																		
b_20	Numérico Entero																																		
b_10	Numérico Entero																																		
b_5	Numérico Entero																																		
m_2	Numérico Entero																																		
m_1	Numérico Entero																																		
m_050	Numérico Entero																																		
m_020	Numérico Entero																																		
m_010	Numérico Entero																																		
m_005	Numérico Entero																																		
m_002	Numérico Entero																																		
m_001	Numérico Entero																																		

Esta guía de estudio, fue vendida por www.gulaceneval.mx Todos los Derechos Reservados a

10. Suponiendo que una paella se puede cocinar exclusivamente con arroz y gambas, y que para cada cuatro personas se utiliza medio kilo de arroz y un cuarto de kilo de gambas, escribir un programa que pida por pantalla el número de comensales para la paella, el precio por kilo de los ingredientes y muestre las cantidades de los ingredientes necesarios y el coste de la misma.

<i>Ordinograma</i>	<i>Pseudocódigo</i>																		
<pre> graph TD Inicio([Inicio]) --> Leer[/Leer comensales, precio_arroz, precio_gambas/] Leer --> CantArroz[cant_arroz = comensales * 0.5 / 4] CantArroz --> CantGambas[cant_gambas = comensales * 0.25 / 4] CantGambas --> CosteArroz[coste_arroz = cant_arroz * precio_arroz] CosteArroz --> CosteGambas[coste_gambas = cant_gambas * precio_gambas] CosteGambas --> CosteTotal[coste_total = coste_arroz + coste_gambas] CosteTotal --> Escribir[/Escribir cant_arroz, cant_gambas, coste_arroz, coste_gambas, coste_total/] Escribir --> Fin([Fin]) </pre>	<p>PROGRAMA: Paella MÓDULO: Principal</p> <p>INICIO</p> <p>DATOS:</p> <table> <tbody> <tr><td>VARIABLES</td><td></td></tr> <tr><td>comensales</td><td>Numérico Entero</td></tr> <tr><td>precio_arroz</td><td>Numérico Real</td></tr> <tr><td>precio_gambas</td><td>Numérico Real</td></tr> <tr><td>cant_arroz</td><td>Numérico Real</td></tr> <tr><td>cant_gambas</td><td>Numérico Real</td></tr> <tr><td>coste_gambas</td><td>Numérico Real</td></tr> <tr><td>coste_arroz</td><td>Numérico Real</td></tr> <tr><td>coste_total</td><td>Numérico Real</td></tr> </tbody> </table> <p>ALGORITMO:</p> <pre> Leer comensales, precio_arroz, precio_gambas cant_arroz = comensales * 0.5 / 4 cant_gambas = comensales * 0.25 / 4 coste_arroz = cant_arroz * precio_arroz coste_gambas = cant_gambas * precio_gambas coste_total = coste_arroz + coste_gambas Escribir cant_arroz, cant_gambas, coste_arroz, coste_gambas, coste_total </pre> <p>FIN</p>	VARIABLES		comensales	Numérico Entero	precio_arroz	Numérico Real	precio_gambas	Numérico Real	cant_arroz	Numérico Real	cant_gambas	Numérico Real	coste_gambas	Numérico Real	coste_arroz	Numérico Real	coste_total	Numérico Real
VARIABLES																			
comensales	Numérico Entero																		
precio_arroz	Numérico Real																		
precio_gambas	Numérico Real																		
cant_arroz	Numérico Real																		
cant_gambas	Numérico Real																		
coste_gambas	Numérico Real																		
coste_arroz	Numérico Real																		
coste_total	Numérico Real																		

Esta guía de estudio, fue vendida por www.guiaceneval.mx Todos los Derechos Reservados a EDICT._GUIA_MEX - Editorial Guias Mexico

Lectura 17. Problemario de algoritmos resueltos con diagramas de flujo y pseudocódigo



PROBLEMARIO DE **ALGORITMOS** RESUELTOS CON DIAGRAMAS DE FLUJO Y PSEUDOCÓDIGO

CIENCIAS BÁSICAS

textosuniversitarios

Francisco Javier Pinales Delgado
César Eduardo Velázquez Amador



Esta guía fue vendida por www.gulacaneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulacaneval.com.mx Todos los Derechos Reservados a
EDICT_GUA_MEX / Editorial Guías México



**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

PROBLEMARIO DE ALGORITMOS RESUELTOS CON DIAGRAMAS DE FLUJO Y PSEUDOCÓDIGO

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

PROBLEMARIO DE ALGORITMOS RESUELTOS CON DIAGRAMAS DE FLUJO Y PSEUDOCÓDIGO

Francisco Javier Pinales Delgado
César Eduardo Velázquez Amador



Esta guía fue vendida por www.gulacaneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulacaneval.com.mx Todos los Derechos Reservados.
EDICT_GUA_MEX / Editorial Guías México

PROBLEMARIO DE ALGORITMOS RESUELTOS

CON DIAGRAMAS DE FLUJO

Y PSEUDOCÓDIGO

D.R. © Universidad Autónoma de Aguascalientes
Av. Universidad No. 940
Ciudad Universitaria
C.P. 20131, Aguascalientes, Ags.
<http://www.uaa.mx/direcciones/dgdv/editorial/>

© Francisco Javier Pinales Delgado
César Eduardo Velázquez Amador

ISBN: 978-607-8285-96-9

Impreso y hecho en México / Printed and made in Mexico

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

Índice 9 Prólogo

	UNIDAD I.
	HERRAMIENTAS DE PROGRAMACIÓN
13	PARA LA SOLUCIÓN DE PROBLEMA CON COMPUTADORAS
15	Herramientas
16	Identificadores
17	Pseudocódigo
17	Diagramas de flujo
20	Diagramas Nassi-Schneiderman N/S
	UNIDAD II.
	SOLUCIÓN DE PROBLEMAS
25	CON ESTRUCTURAS SECUENCIALES
27	Introducción
28	Estructuras de control
28	Estructuras secuenciales
43	Problemas resueltos
45	Problemas propuestos
	UNIDAD III.
	SOLUCIÓN DE PROBLEMAS
47	CON ESTRUCTURAS SELECTIVAS
49	Introducción
88	Estructuras selectivas
90	Problemas resueltos
90	Problemas propuestos
	UNIDAD IV.
93	SOLUCIÓN DE PROBLEMAS CON ESTRUCTURAS REPETITIVAS
95	Introducción
95	Estructuras repetitivas o de ciclo
138	Problemas resueltos
139	Problemas propuestos

	UNIDAD V.
	INTRODUCCIÓN A LOS ARREGLOS UNIDIMENSIONALES
141	Y MULTIDIMENSIONALES (VECTORES Y MATRICES)
143	Introducción
144	Arreglos unidimensionales (vectores)
157	Arreglos bidimensionales (tablas)
168	Problemas resueltos
169	Problemas propuestos

APÉNDICE. Solución de problemas propuestos

- Soluciones de la unidad dos
- Soluciones de la unidad tres
- Soluciones de la unidad cuatro
- Soluciones de la unidad cinco

PRÓLOGO

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

El propósito de este libro es proporcionar a los alumnos que recién inician sus estudios en el área de computación una serie de problemas representativos, los cuales están resueltos algorítmicamente con detalle. En el área de programación existen diferentes herramientas que auxilian en la solución de problemas, pero seleccionar una de ellas para comenzar a introducir al estudiante en el área se vuelve un poco complicado, dado que cada una posee ventajas y desventajas; éstas son percibidas por los estudiantes, y si adoptan alguna herramienta con mayor facilidad, presentan cierto rechazo hacia las otras, por considerarlas más complicadas. Por tal motivo, en este libro se presentan tres herramientas para tratar de ayudar a los estudiantes a desarrollar una lógica apropiada para el planteamiento y solución de un problema (pseudocódigo, diagramas de flujo y diagramas Nassi-Schneiderman).

Los problemas que se plantean están enfocados en utilizar las tres estructuras básicas de la programación (secuencias, decisiones y ciclos), de tal forma que el alumno se vaya enrolando paso a paso en la solución de problemas cada vez más complejos, de aquí que el formato de este libro dedique una unidad a cada tipo de estructura, concluyendo finalmente con un capítulo del tratamiento de arreglos, tan útiles en la solución de problemas.

Definitivamente el objetivo de este libro no es establecer un patrón para resolver los problemas, tan sólo es el de proporcionar ayuda a los alumnos para desarrollar una lógica apropiada mediante la utilización de una de las herramientas para la solución de problemas, los cuales, posteriormente, podrán ser implementados en la computadora mediante un lenguaje de programación.

Queremos agradecer a todas aquellas personas que contribuyeron para la realización de este proyecto, especialmente a las autoridades de la Universidad Autónoma de Aguascalientes, por darnos las facilidades para poder realizar este trabajo. A las profesoras Ma. Guadalupe Mendoza y Lorena Pinales Delgado, por apoyar en la revisión de este libro; a Luz Patricia Pinales Delgado, por su colaboración en la realización de esta obra.

Francisco Javier Pinales Delgado
Cesar Eduardo Velázquez Amador

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

UNIDAD I

HERRAMIENTAS DE PROGRAMACIÓN

PARA LA SOLUCIÓN DE PROBLEMA

CON COMPUTADORAS

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

Herramientas

Para implementar la solución de un problema mediante el uso de una computadora es necesario establecer una serie de pasos que permitan resolver el problema, a este conjunto de pasos se le denomina algoritmo, el cual debe tener como característica final la posibilidad de transcribirlo fácilmente a un lenguaje de programación, para esto se utilizan herramientas de programación, las cuales son métodos que permiten la elaboración de algoritmos escritos en un lenguaje entendible.

Un algoritmo, aparte de tener como característica la facilidad para transcribirlo, debe ser:

1. Preciso. Debe indicar el orden en el cual debe realizarse cada uno de los pasos que conducen a la solución del problema.
2. Definido. Esto implica que el resultado nunca debe cambiar bajo las mismas condiciones del problema, éste siempre debe ser el mismo.
3. Finito. No se debe caer en repeticiones de procesos de manera innecesaria; deberá terminar en algún momento.

Por consiguiente, el algoritmo es una serie de operaciones detalladas y no ambiguas para ejecutar paso a paso que conducen a la resolución de un problema, y se representan mediante una herramienta o técnica.¹ O bien, es una forma de describir la solución de un problema planteado en forma adecuada y de manera genérica.

Además de esto, se debe considerar que el algoritmo, que posteriormente se transformará en un programa de computadora, debe considerar las siguientes partes:

- Una descripción de los datos que serán manipulados.
- Una descripción de acciones que deben ser ejecutadas para manipular los datos.
- Los resultados que se obtendrán por la manipulación de los datos.

¹ Luis Joyanes Aguilar, *Metodología de la programación, diagramas de flujo, algoritmos y programación estructurada*, España, Mc Graw Hill, 1993.

Las herramientas o técnicas de programación que más se utilizan y que se emplearán para la representación de algoritmos a lo largo del libro son dos:

1. Pseudocódigo.
2. Diagramas de flujo.

Y alternativamente se presentarán soluciones de problemas donde se utilicen:

3. Diagramas Nassi-Schneiderman (N/S).

Identificadores

Antes de analizar cada una las herramientas que se utilizan en representación de algoritmos para la solución de problemas, se establecerá qué son los identificadores que se utilizan dentro de un algoritmo.

Los identificadores son los nombres que se les asignan a los objetos, los cuales se pueden considerar como variables o constantes, éstos intervienen en los procesos que se realizan para la solución de un problema, por consiguiente, es necesario establecer qué características tienen.

Para establecer los nombres de los identificadores se deben respetar ciertas reglas que establecen cada uno de los lenguajes de programación, para el caso que nos ocupa se establecen de forma indistinta según el problema que se esté abordando, sin seguir regla alguna, generalmente se utilizará la letra, o las letras, con la que inicia el nombre de la variable que representa el objeto que se va a identificar.

Constante

Un identificador se clasifica como constante cuando el valor que se le asigna a este identificador no cambia durante la ejecución o proceso de solución del problema. Por ejemplo, en problemas donde se utiliza el valor de PI, si el lenguaje que se utiliza para codificar el programa y ejecutarlo en la computadora no lo tiene definido, entonces se puede establecer de forma constante estableciendo un identificador llamado PI y asignarle el valor correspondiente de la siguiente manera:

$$\text{PI} = 3.1416.$$

De igual forma, se puede asignar valores constantes para otro identificadores según las necesidades del algoritmo que se esté diseñando.

Variables

Los identificadores de tipo variable son todos aquellos objetos cuyo valor cambia durante la ejecución o proceso de solución del problema. Por ejemplo, el sueldo, el pago, el descuento, etcétera, que se deben calcular con un algoritmo determinado, o en su caso, contar con el largo (L) y ancho (A) de un rectángulo que servirán para calcular y obtener su área. Como se puede ver, tanto L como A son variables que se proporcionan para que el algoritmo pueda funcionar, y no necesariamente se calculen dentro del proceso de solución.

Tipos de variables

Los elementos que cambian durante la solución de un problema se denominan variables, se clasifican dependiendo de lo que deben representar en el algoritmo, por lo cual pueden ser: de tipo entero, real y *string* o de cadena, sin embargo, existen otros tipos de variables que son permitidos con base en el lenguaje de programación que se utilice para crear los programas, por consiguiente, al momento de estudiar algún lenguaje de programación en especial se deben dar a conocer esas clasificaciones.

Para el caso de este libro, se denominará variables de tipo entero a todas aquellas cuyo valor no tenga valores decimales; contrario a las de tipo real, la cual podrá tomar valores con decimales. Como ejemplo de variables enteras se puede considerar el número de personas, días trabajados, edad de una persona, etcétera. Y para el caso de reales, se puede considerar el sueldo de una persona, el porcentaje de equis cantidad, etcétera.

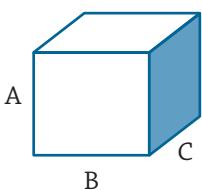
En caso de que las variables tomen valores de caracteres, se designarán *string* o de cadena; como ejemplo de éstas se pueden mencionar el sexo de una persona, falso o verdadero, el nombre de una persona, el tipo de sangre, etcétera.

Pseudocódigo

Sin duda, en el mundo de la programación el pseudocódigo es una de las herramientas más conocidas para el diseño de solución de problemas por computadora. Esta herramienta permite pasar casi de manera directa la solución del problema a un lenguaje de programación específico. El pseudocódigo es una serie de pasos bien detallados y claros que conducen a la resolución de un problema.

La facilidad de pasar casi de forma directa el pseudocódigo a la computadora ha dado como resultado que muchos programadores implementen de forma directa los programas en la computadora, cosa que no es muy recomendable, sobre todo cuando no se tiene la suficiente experiencia para tal aventura, pues se podrían tener errores propios de la poca experiencia acumulada con la solución de diferentes problemas.

Por ejemplo, el pseudocódigo para determinar el volumen de una caja de dimensiones A, B y C se puede establecer de la siguiente forma:



1. Inicio.
2. Leer las medidas A, B y C.
3. Realizar el producto de $A * B * C$ y guardarlo en V
 $(V = A * B * C)$.
4. Escribir el resultado V.
5. Fin.

Como se puede ver, se establece de forma precisa la secuencia de los pasos por realizar; además, si se le proporciona siempre los mismos valores a las variables A, B y C, el resultado del volumen será el mismo y, por consiguiente, se cuenta con un final.

Diagramas de flujo

Los diagramas de flujo son una herramienta que permite representar visualmente qué operaciones se requieren y en qué secuencia se deben efectuar para solucionar un problema dado. Por consiguiente, un diagrama de flujo es la representación gráfica mediante símbolos especiales, de los pasos o procedimientos de manera secuencial y lógica que se deben realizar para solucionar un problema dado.

Los diagramas de flujo desempeñan un papel vital en la programación de un problema, ya que facilitan la comprensión de problemas complicados y sobre todo aquellos en que sus procesos son muy largos;² generalmente, los diagramas de flujo se dibujan antes de comenzar a programar el código fuente, que se ingresará posteriormente a la computadora.

Los diagramas de flujo facilitan la comunicación entre los programadores y los usuarios, además de que permiten de una manera más rápida detectar los posibles errores de lógica que se presenten al implementar el algoritmo. En la tabla 1.1 se muestran algunos de los principales símbolos utilizados para construir un diagrama de flujo.

Dentro de los diagramas de flujo se pueden utilizar los símbolos que se presentan en la tabla 1.2, con los cuales se indican las operaciones que se efectuarán a los datos con el fin de producir un resultado.

Símbolo	Significado
	Terminal / Inicio.
	Entrada de datos.
	Proceso.
	Decisión.
	Decisión múltiple.
	Imprimir resultados.
	Flujo de datos.
	Conejeros.

Tabla 1.1 Principales símbolos utilizados para construir los diagramas de flujo.

² *Idem.*

Símbolo	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
^	Exponenciación
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
< >	Diferente que
=	Igual que

Tabla 1.2 Principales símbolos utilizados en los diagramas de flujo para indicar las operaciones que se realizan para producir un resultado.

Por ejemplo, se puede establecer la solución del diagrama de flujo para determinar el volumen de una caja de dimensiones A, B y C como se muestra en la figura 1.1.

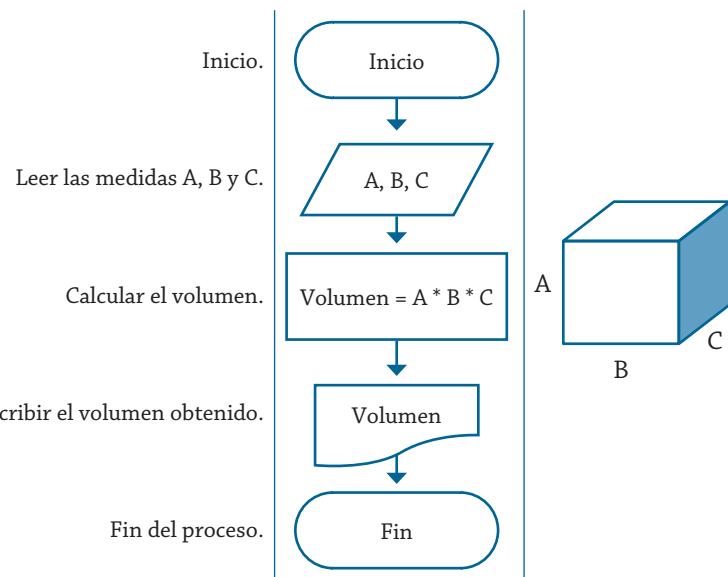


Figura 1.1 Diagrama de flujo para obtener el volumen de un cubo.

Y como se puede ver de manera gráfica, se establece de forma precisa la secuencia de los pasos por realizar para obtener el resultado del volumen. Como se puede verificar, son los mismos pasos que se establecieron en el algoritmo presentado previamente mediante el pseudocódigo.

Diagramas Nassi-Schneiderman N/S

El diagrama N-S es una técnica en la cual se combina la descripción textual que se utiliza en el pseudocódigo y la representación gráfica de los diagramas de flujo. Este tipo de técnica se presenta de una manera más compacta que las dos anteriores, contando con un conjunto de símbolos muy limitado para la representación de los pasos que se van a seguir por un algoritmo; por consiguiente, para remediar esta situación, se utilizan expresiones del lenguaje natural, sinónimos de las palabras propias de un lenguaje de programación (leer, hacer, escribir, repetir, etcétera).

Por ejemplo, se puede establecer la solución del diagrama N/S para determinar el volumen de una caja de dimensiones A, B y C como se muestra en la figura 1.2.

Como se puede ver de este ejemplo, los diagramas N/S son como los diagramas de flujo en el que se omiten las flechas de unión y las cajas son contiguas. Las acciones sucesivas se escriben dentro de las cajas sucesivas y, como en los diagramas de flujo, se pueden escribir diferentes acciones en una caja.³

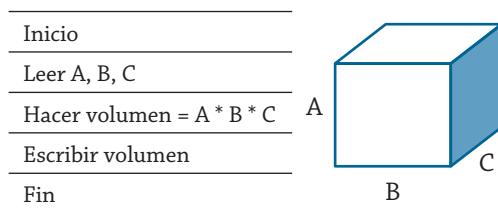


Figura 1.2 Diagrama N/S para obtener el volumen de un cubo.

Establecer cuál herramienta utilizar para representar los algoritmos diseñados para la solución de problemas estará en función del gusto y preferencia del programador, y quizás no tanto en función de la complejidad de los problemas, ya que si bien es cierto que los diagramas N/S tienen pocos símbolos, presentan la ventaja de que por lo compacto que resultan sus representaciones suelen ser más fáciles de leer y de transcribir al lenguaje de programación que se utilizará, pero luego resulta complicado acomodar las acciones al construir el diagrama.

Los símbolos más utilizados en diagrama N/S corresponden a un tipo de estructura para la solución del problema, esas estructuras pueden ser: secuenciales de decisión y de ciclo. Estas estructuras de los diagramas N/S se presentan en la tabla 1.3.

³ Luis Joyanes Aguilar, *Turbo Basic Manual de Programación*, España, Mc Graw Hill, 1990.

Símbolo	Tipo de estructura
	Secuencial
	Selectiva
	De ciclo

Tabla 1.3 Principales estructuras utilizadas para construir los diagramas N/S.

A continuación, se muestran ejemplos sobre cómo utilizar las estructuras de los diagramas N/S, tal es el caso de la figura 1.3 que muestra un diagrama N/S con el algoritmo para obtener el área de un rectángulo, en el cual la solución tiene una estructura secuencial.

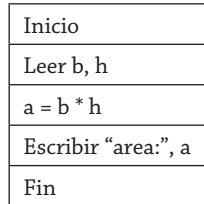


Figura 1.3 Diagrama N/S con una estructura secuencial.

Para una estructura de decisión se muestra la figura 1.4, en la cual se tiene la solución de un algoritmo para determinar cuál de dos cantidades es la mayor.

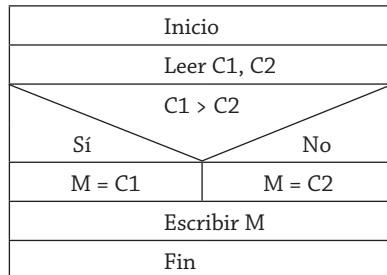


Figura 1.4 Estructura selectiva de un diagrama N/S.

Finalmente para una estructura de ciclo, el símbolo que se utiliza es como el que se muestra en la figura 1.5, en el cual están presentes una combinación de estructuras secuenciales con la de ciclo. En este diagrama se presenta la solución de la suma de diez cantidades cualesquiera.

Inicio
Hacer $1 = 10$
Mientras $1 \leq 10$
Leer C
Hacer $S = S + C$
Hacer $1 = 1 + 1$
Fin mientras
Escribir S
Fin

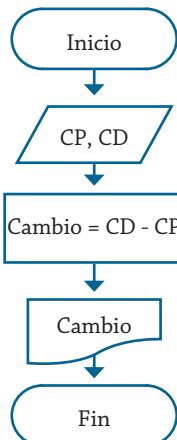
Figura 1.5 Estructura de ciclo de un diagrama N/S.

Como se puede ver, dentro de estos símbolos se utilizan palabras reservadas como: Inicio, Fin, Leer, Escribir, Mientras, Repita, Hasta, Para, Incrementar, Decremento, Hacer Función, etcétera.

En algunos casos se acostumbra indicar el tipo de las variables que se utilizarán en el proceso, que para el caso de los diagramas de flujo y el pseudocódigo representa en la tabla de variables que se ha venido utilizando (Entero, Real, Carácter o Cadena).

También es importante señalar que antes de presentar cualquier solución de un problema es necesario analizar el problema para entender qué es lo que se quiere obtener, con qué se cuenta y cómo se obtendrá lo deseado. En otras palabras, cómo está conformado el sistema: entrada, proceso y salida. No establecer con claridad lo que se tiene puede traer consigo una solución totalmente errónea; para que esto quede más claro, considere el siguiente ejemplo. Se requiere un algoritmo para determinar el cambio que recibirá una persona que adquiere un producto en la tienda.

Posiblemente alguien piense que la solución de este problema requiere una gran cantidad de pasos probablemente demasiado complicados, o por el contrario, que es demasiado sencillo, que no tiene ninguna complejidad. La cuestión es: ¿quién puede tener la razón? La respuesta puede ser que los dos, todo dependerá de cómo se entienda su planteamiento, si se plantea un razonamiento sencillo la solución puede ser la mostrada en la figura 1.6.



En esta solución lo que se propone es determinar el cambio que recibirá una persona, para esto es necesario conocer cuánto cuesta el producto (CP) y qué cantidad de dinero disponible se tiene, y resolver el problema mediante una simple diferencia entre lo que se pagó y el costo del producto.

Figura 1.6 Diagrama de flujo para determinar el cambio que recibirá una persona al adquirir un producto.

Ahora, si el mismo problema que se planteó se piensa en otros aspectos, de tal forma que para la solución se planteara algún cuestionamiento como: “¿Se debe considerar que el dinero alcanzó para comprar el artículo?”, la solución que se propondría ya no sería igual que la anterior, y podría plantearse de la forma como se muestra en la figura 1.7.

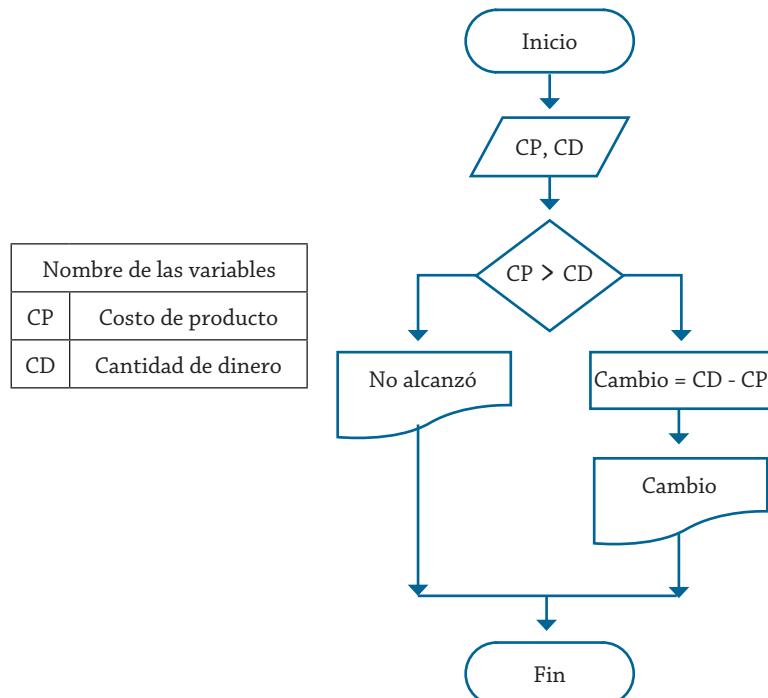


Figura 1.7 Diagrama de flujo para determinar el cambio que recibirá una persona al adquirir un producto.

Como se puede ver, en ocasiones exponer la solución de un problema dado dependerá de cómo se considere su planteamiento, y también tendrá mucho que ver la forma en la que el diseñador lo conceptualice; debido a esto, es muy importante, cuando se realicen algoritmos para la solución de problemas prácticos, que se plantee de manera correcta lo que se quiere y se aclaren los puntos necesarios que permitan diseñar la solución más óptima, pues hay que recordar que un algoritmo es siempre perfectible.

Para los siguientes capítulos se propondrá la solución de problemas donde se utilicen para su representación principalmente pseudocódigo y diagramas de flujo, y en otros casos diagramas N/S.

Pero antes de pasar al planteamiento y solución de problemas, es necesario dejar bien claro que las soluciones planteadas en este texto no son únicas, y pueden ser mejoradas por los lectores.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

UNIDAD II

SOLUCIÓN DE PROBLEMAS SECUENCIALES

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

Introducción

Para la solución de cualquier problema que se vaya a representar mediante alguna de las herramientas que se han mencionado, siempre tendremos que representar mediante letras, abreviaciones o palabras completas los elementos que intervienen en el proceso de solución, a estos elementos se les denomina variables o constantes. Por ejemplo: sueldo con **S**; horas trabajadas con **HT**; edad con **E**, o bien con la palabra completa según el gusto de cada diseñador.

Con base en esto, para facilitar la lectura de un algoritmo se recomienda crear una tabla donde se declaran las variables que se utilizarán y sus características o tipo, tal y como se muestra en la tabla 2.1, que muestra las variables que se utilizarían para obtener el área de un rectángulo.

Nombre de la variable	Descripción	Tipo
A	Altura del rectángulo	Real
B	Base del rectángulo	Real
Área	Área del rectángulo	Real

Tabla 2.1. Declaración de las variables que se utilizarán para obtener el área de un rectángulo.

Como se puede ver en la tabla 2.1, se utilizarán las variables A y B para representar la altura y la base de un rectángulo, respectivamente, a las cuales se les podrán asignar diferentes valores, y al utilizar esos valores y aplicar la fórmula correspondiente se podrá obtener el área del rectángulo, la cual es asignada a la variable denominada Área. Además, se describe que esas variables son de tipo real, lo cual implica que podrán tomar valores fraccionarios, pero también pudieron haber sido enteras.

Como ya se mencionó anteriormente, los tipos de variables que existen son: enteras, reales y *string* o de cadena; sin embargo, existen otros tipos que son permitidos con base en el lenguaje de programación que se utilice para crear los programas.

Estructuras de control

Sin importar qué herramienta o técnica se utilice para la solución de un problema dado, ésta tendrá una estructura, que se refiere a la secuencia en que se realizan las operaciones o acciones para resolver el problema; esas estructuras pueden ser: secuenciales, de decisión y de ciclo o repetición, las cuales se analizarán en su momento.

Debe tenerse presente que la solución de un problema dado mediante el uso de una computadora es un sistema, el cual debe tener una entrada de datos, los cuales serán procesados para obtener una salida, que es la solución o información que se busca. En la figura 2.1 se muestra el esquema de un sistema que transforma los datos en información mediante un proceso.



Figura 2.1. Un sistema de transformación.

Estructuras secuenciales

En este tipo de estructura las instrucciones se realizan o se ejecutan una después de la otra y, por lo general, se espera que se proporcione uno o varios datos, los cuales son asignados a variables para que con ellos se produzcan los resultados que representen la solución del problema que se planteó. Los algoritmos tienen como fin actuar sobre los datos proporcionados por el usuario, a los que se les aplican procesos con el fin de generar la información o un resultado. El algoritmo es realmente la representación funcional de un sistema, como el que se muestra en la figura 2.1.

Para resolver un problema mediante la utilización de cualquier herramienta es necesario entender y establecer con qué datos se cuenta, los procesos que se deben realizar y la secuencia apropiada para obtener la solución que se desea.

Ejemplo 2.1

Se desea implementar un algoritmo para obtener la suma de dos números cualesquiera. Se debe partir de que para poder obtener la suma es necesario contar con dos números, pues el proceso que debemos realizar es precisamente la suma de éstos, la cual se asigna a una variable que se reporta como resultado del proceso.

Los pasos por seguir son los mostrados en el pseudocódigo 2.1, que corresponde al algoritmo que permite determinar la suma de dos números cualesquiera.

1. Inicio
2. Leer A, B
3. Hacer $S = A + B$
4. Escribir S
5. Fin

Pseudocódigo 2.1 Algoritmo para determinar la suma de dos números cualesquiera.

Como se puede ver, **A** y **B** representan los valores para sumar, y **S** el resultado de la suma. Al representar la solución del problema utilizando pseudocódigo, se está utilizando un lenguaje que comúnmente utilizamos, sólo que de una forma ordenada y precisa.

Es recomendable indicar mediante una tabla las variables que se utilizan, señalando lo que representan y sus características, esta acción facilitará la lectura de la solución de un problema dado, sin importar qué herramienta de programación se esté utilizando para la representación de la solución del problema. Para el problema de la suma de dos números, la tabla 2.2 muestra las variables utilizadas en la solución.

Nombre de la variable	Descripción	Tipo
A	Primer número para sumar	Entero
B	Segundo número para sumar	Entero
S	Resultado de la suma	Entero

Tabla 2.2 Variables utilizadas para determinar la suma de dos números cualesquiera.

La construcción de las tablas de variables se puede realizar en forma paralela o, bien, al término del pseudocódigo o del diagrama según sea el caso.

La representación del algoritmo mediante la utilización de un diagrama de flujo sería como el que se muestra en el diagrama de flujo 2.1.

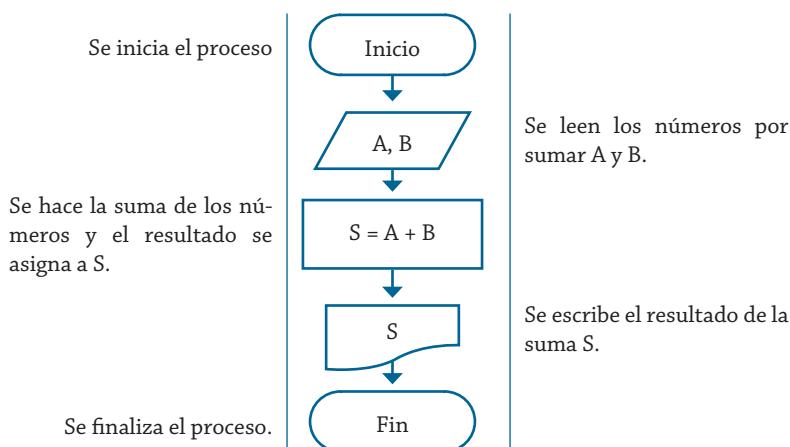


Diagrama de flujo 2.1 Algoritmo para determinar la suma de dos números.

De igual forma, como en el pseudocódigo, **A** y **B** representan los valores que se van a sumar, y **S** el resultado de la suma. Ahora el resultado se presenta de manera gráfica.

Ahora bien, si se plantea la solución del mismo problema, pero ahora utilizando los diagramas de Nassi-Schneiderman, la solución sería como la mostrada en el diagrama N/S 2.1.

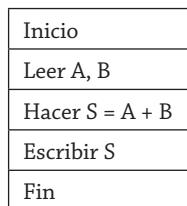


Diagrama N/S 2.1 Algoritmo para determinar la suma de dos números.

Como se puede ver, el proceso de solución es idéntico en las tres herramientas, lo que cambia es la forma en que se presenta; para una herramienta se utilizan sólo palabras; para los otros dos métodos se utilizan elementos gráficos, y como se puede ver, los diagramas N/S son casi diagramas de flujo normales donde sólo se omiten las flechas de unión.

A continuación, se planteará una serie de problemas; en algunos casos se presentará el pseudocódigo como solución y en otros el diagrama de flujo, o en su caso, ambos.

Ejemplo 2.2

Un estudiante realiza cuatro exámenes durante el semestre, los cuales tienen la misma ponderación. Realice el pseudocódigo y el diagrama de flujo que representen el algoritmo correspondiente para obtener el promedio de las calificaciones obtenidas.

Las variables que se van a utilizar en la solución de este problema se muestran en la tabla 2.3.

Nombre de la variable	Descripción	Tipo
C1, C2, C3, C4	Calificaciones obtenidas	Real
S	Suma de calificaciones	Real
P	Promedio calculado	Real

Tabla 2.3 Variables utilizadas para determinar el promedio de cuatro calificaciones.

Por consiguiente, el pseudocódigo 2.2 muestra la solución correspondiente.

1. Inicio
2. Leer C1, C2, C3, C4
3. Hacer S = C1 + C2 + C3 + C4
4. Hacer P = S/4
5. Escribir P
6. Fin

Pseudocódigo 2.2 Algoritmo para determinar el promedio de cuatro calificaciones.

Para explicar este proceso, primeramente se parte de que para poder obtener un promedio de calificaciones es necesario conocer estas calificaciones, las cuales las tenemos que leer de alguna parte (C1, C2, C3, C4); posteriormente, se tienen que sumar para saber el total de calificaciones obtenidas (S), y con base en el número de calificaciones proporcionadas (4), poder calcular el promedio obtenido (P) y presentar el resultado obtenido, éste de manera escrita.

Ahora bien, el diagrama de flujo 2.2 muestra la representación correspondiente mediante la cual se debe utilizar el mismo razonamiento que se utilizó al crear el pseudocódigo.

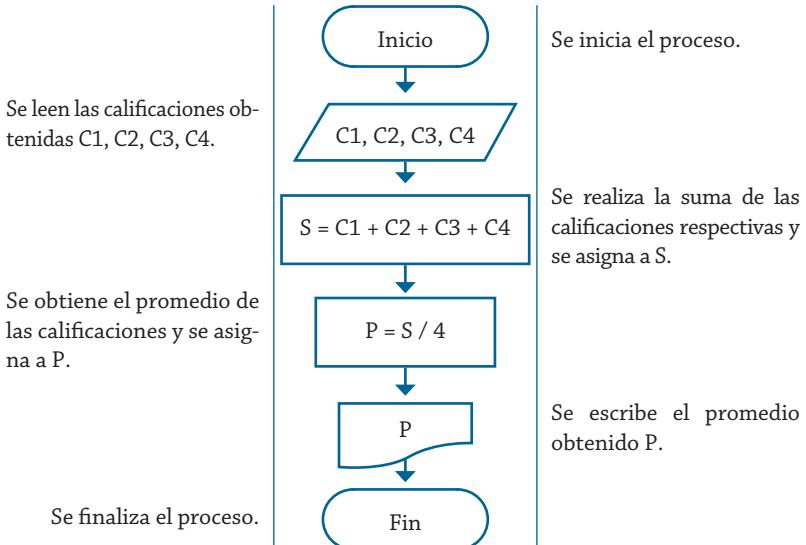


Diagrama de flujo 2.2 Algoritmo para determinar el promedio de cuatro calificaciones.

Como se puede ver, prácticamente lo que se tiene es el pseudocódigo, pero ahora presentado en forma gráfica, que es una de las características de los diagramas de flujo.

Como una herramienta alternativa de solución del problema, se presenta el diagrama N/S 2.2.

Inicio
Leer C1, C2, C3, C4
Hacer S = C1 + C2 + C3 + C4
Hacer P = S / 4
Escribir P
Fin

Diagrama N/S 2.2 Algoritmo para determinar el promedio de cuatro calificaciones.

Se puede observar que realmente es una combinación de pseudocódigo y de un diagrama de flujo, sólo que para este tipo de diagrama se omiten las flechas de flujo.

Ejemplo 2.3

Se requiere conocer el área de un rectángulo. Realice un algoritmo para tal fin y represéntelo mediante un diagrama de flujo y el pseudocódigo para realizar este proceso.

Como se sabe, para poder obtener el área del rectángulo, primeramente se tiene que conocer la base y la altura, y una vez obtenidas se presenta el resultado.

La tabla 2.4 muestra las variables que se van a utilizar para elaborar el algoritmo correspondiente.

Nombre de la variable	Descripción	Tipo
A	Altura del rectángulo	Real
B	Base del rectángulo	Real
Área	Área del rectángulo	Real

Fórmula: Área = (base * altura)

Tabla 2.4 Variables utilizadas para determinar el área de un rectángulo.

El diagrama de flujo 2.3 muestra la solución correspondiente al algoritmo apropiado, de acuerdo a lo planteado anteriormente.

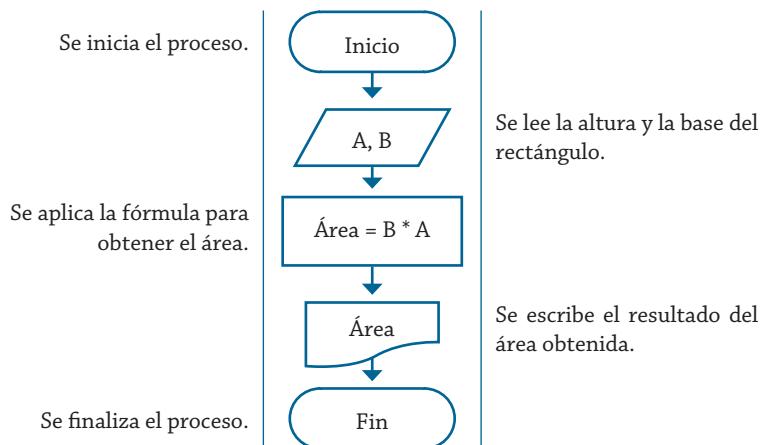


Diagrama de flujo 2.3 Algoritmo para determinar el área de un rectángulo.

La estructura del pseudocódigo 2.3 muestra el algoritmo que permite obtener el área del rectángulo.

1. Inicio
2. Leer A, B
3. Hacer Área = B * A
4. Escribir Área
5. Fin

Pseudocódigo 2.3 Algoritmo para determinar el área de un rectángulo.

Y de igual forma, el diagrama N/S 2.3 muestra la solución correspondiente.

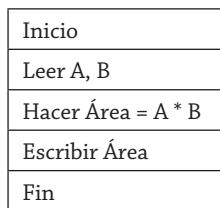


Diagrama N/S 2.3 Algoritmo para determinar el área de un rectángulo.

Ejemplo 2.4

Se requiere obtener el área de una circunferencia. Realizar el algoritmo correspondiente y representarlo mediante un diagrama de flujo y el pseudocódigo correspondiente.

De igual forma que en los problemas anteriores, es importante establecer la tabla de variables que se utilizarán para la solución del problema, pero ahora previamente se analizará qué se requiere para obtener el área de la circunferencia.

Si se analiza la fórmula que se utiliza para tal fin, se puede establecer que se requiere un valor de radio solamente y que se debe dar un valor constante, que es el valor de PI, que se establece como 3.1416. Con esto ahora se puede establecer la tabla 2.5 con las variables correspondientes.

Nombre de la variable	Descripción	Tipo
R	Radio de la circunferencia	Real
PI	El valor de 3.1416	Real
Área	Área de la circunferencia	Real

$$\text{Fórmula: Área} = \text{PI} * \text{R}^2$$

Tabla 2.5 Variables utilizadas para determinar el área de una circunferencia.

A partir de esto se obtendría el diagrama de flujo 2.4, que muestra el algoritmo correspondiente para la solución del problema.

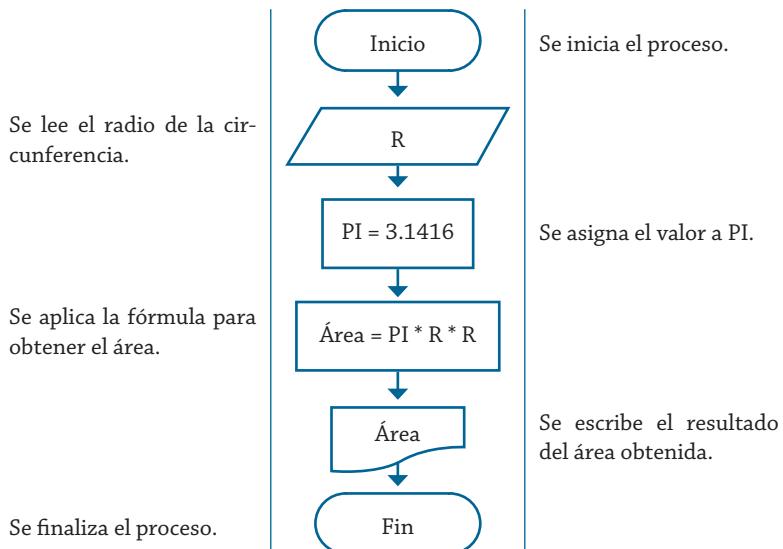


Diagrama de flujo 2.4 Algoritmo para determinar el área de una circunferencia.

Ahora, de igual forma se puede establecer la representación mediante el pseudocódigo 2.4.

1. Inicio
2. Leer R
3. Hacer PI = 3.1416
4. Hacer Área = PI * R * R
5. Escribir Área
6. Fin

Pseudocódigo 2.4 Algoritmo para determinar el área de una circunferencia.

De la misma forma, el diagrama N/S 2.4 muestra la solución correspondiente a este problema, mediante esta herramienta.

Inicio
Leer R
PI = 3.1416
Hacer Área = PI * R * R
Escribir Área
Fin

Diagrama N/S 2.4 Algoritmo para determinar el área de una circunferencia.

Como se puede ver, los diagramas N/S que resultaron en la solución de los problemas anteriores son realmente sencillos en la solución de problemas de tipo secuenciales, por tal motivo, por el momento sólo se presentarán soluciones con dos de las herramientas que se tiene contemplado analizar en el presente libro.

Ejemplo 2.5

Una empresa constructora vende terrenos con la forma A de la figura 2.2. Realice un algoritmo y represéntelo mediante un diagrama de flujo y el pseudocódigo para obtener el área respectiva de un terreno de medidas de cualquier valor.

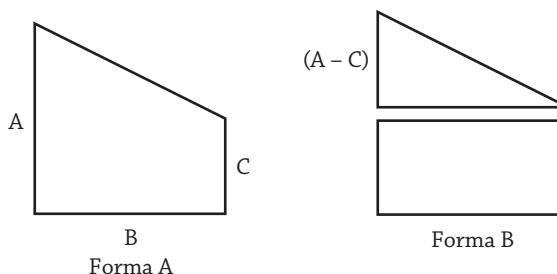


Figura 2.2 Forma del terreno y cómo se puede dividir.

Para resolver este problema se debe identificar que la forma A está compuesta por dos figuras: un **triángulo** de base B y de altura $(A - C)$; y por otro lado, un **rectángulo** que tiene base B y altura C. Con estas consideraciones se puede establecer la tabla 2.6 con las variables que se requieren para implementar el algoritmo de solución.

Nombre de la variable	Descripción	Tipo	
B	Base del triángulo y del rectángulo	Real	
A	Altura del triángulo y rectángulo unidos	Real	
C	Altura del rectángulo	Real	
	Fórmula		
AT	Área del triángulo	Área = (base * altura)/ 2	Real
AR	Área del rectángulo	Área = (base * altura)	Real
Área	Área de la figura	Área = AT + AR	Real

Tabla 2.6 Variables utilizadas para determinar el área de un terreno.

Por consiguiente, como se puede ver, se establecen variables para las respectivas áreas de las figuras que conforman el terreno, las cuales determinarán el área total del respectivo terreno.

Ahora, con estas consideraciones, se puede representar el algoritmo mediante el diagrama de flujo 2.5, el cual permite la solución del problema.

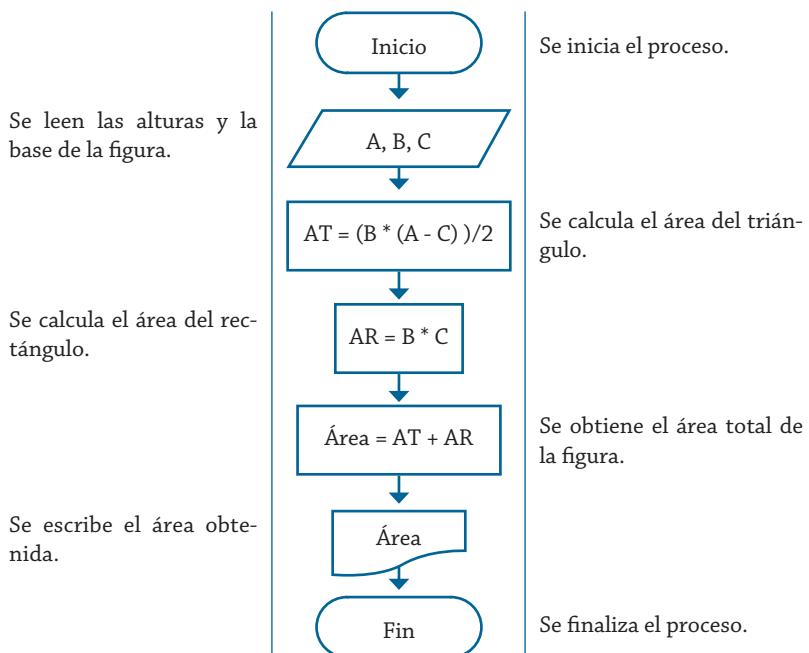


Diagrama de flujo 2.5 Algoritmo para determinar el área de un terreno.

De igual forma, el pseudocódigo 2.5 muestra la solución correspondiente mediante este método de representación.

1. Inicio
2. Leer A, B, C
3. Hacer $AT = (B * (A - C)) / 2$
4. Hacer $AR = B * C$
5. Hacer $\text{Área} = AT + AR$
6. Escribir Área
7. Fin

Pseudocódigo 2.5 Algoritmo para determinar el área de un terreno.

Ejemplo 2.6

Se requiere obtener el área de la figura 2.3 de la forma A. Para resolver este problema se puede partir de que está formada por tres figuras: dos triángulos rectángulos, con H como hipotenusa y R como uno de los catetos, que también es el radio de la otra figura, una semicircunferencia que forma la parte circular (ver forma B). Realice un algoritmo para resolver el problema y represéntelo mediante el diagrama de flujo y el pseudocódigo.

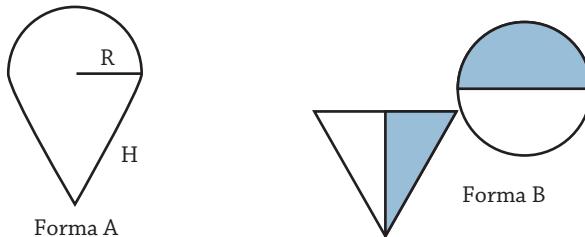


Figura 2.3 Forma del terreno y cómo se puede interpretar.

Por lo tanto, para poder resolver el problema, se tiene que calcular el cateto faltante, que es la altura del triángulo, con ésta se puede calcular el área del triángulo, y para obtener el área total triangular se multiplicará por dos. Por otro lado, para calcular el área de la parte circular, se calcula el área de la circunferencia y luego se divide entre dos, ya que representa sólo la mitad del círculo. De este análisis se puede obtener la tabla 2.7, que contiene las variables requeridas para plantear el algoritmo con la solución respectiva.

Nombre de la variable	Descripción	Tipo
R	Base del triángulo rectángulo y radio	Real
H	Hipotenusa del triángulo rectángulo	Real
C	Cateto faltante	Real
AT	Área triangular	Real
AC	Área circular	Real
PI	El valor de 3.1416	Real
Área	Área de la figura	Real
SQRT	Indica obtener raíz cuadrada	---

Tabla 2.7 Variables utilizadas para obtener el área de una figura.

Con esas consideraciones, la solución se puede representar mediante el diagrama de flujo 2.6.

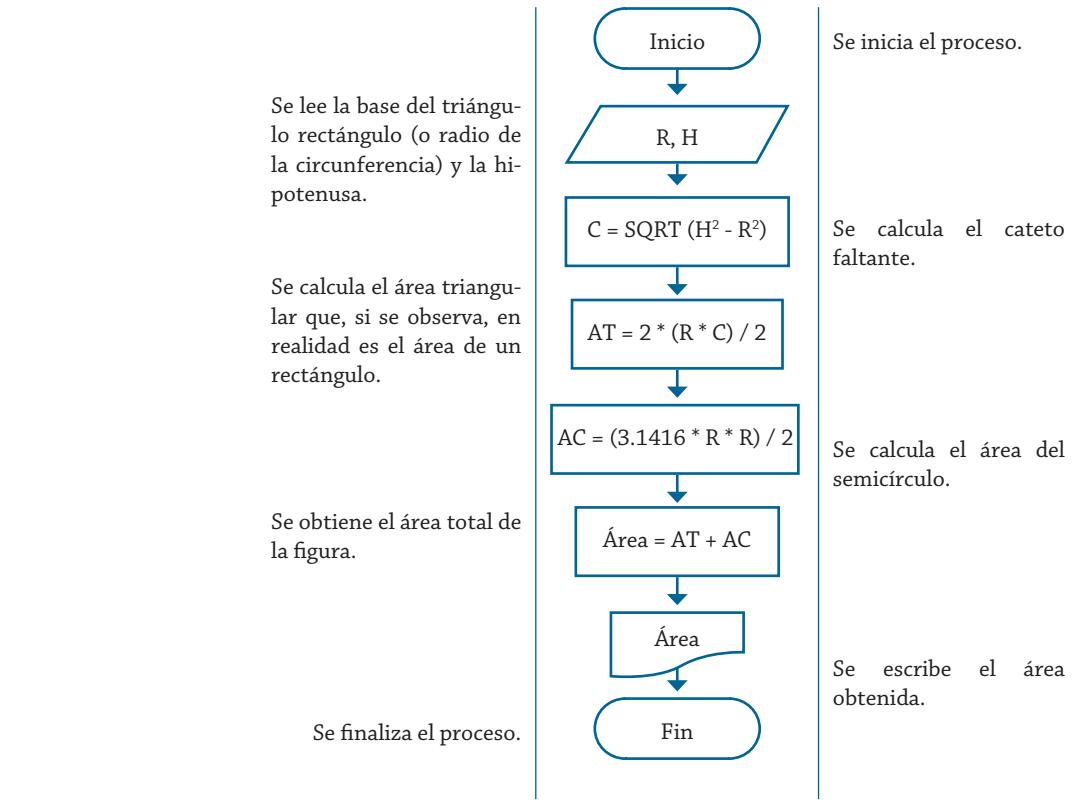


Diagrama de flujo 2.6 Algoritmo para obtener el área de una figura.

El pseudocódigo 2.6 representa el algoritmo de solución para este problema.

1. Inicio
2. Leer R, H
3. Hacer $C = \sqrt{H^2 - R^2}$
4. Hacer $AT = 2 * (R * C) / 2$
5. Hacer $AC = (\pi * R * R) / 2$
6. Hacer $\text{Área} = AT + AC$
7. Escribir Área
8. Fin

Pseudocódigo 2.6 Algoritmo para obtener el área de una figura.

Ejemplo 2.7

Un productor de leche lleva el registro de lo que produce en litros, pero cuando entrega le pagan en galones. Realice un algoritmo, y represéntelo mediante un diagrama de flujo y el pseudocódigo, que ayude al productor a saber cuánto recibirá por la entrega de su producción de un día (1 galón = 3.785 litros).

Si se analiza el problema se puede establecer que los datos que se necesitan para resolver el problema son los que se muestran en la tabla 2.8.

Nombre de la variable	Descripción	Tipo
L	Cantidad de litros que produce	Entero
PG	Precio del galón	Real
TG	Cantidad de galones que produce	Real
GA	Ganancia por la entrega de leche	Real

Tabla 2.8 Variables utilizadas para determinar la ganancia por la producción de leche.

El pseudocódigo 2.7 representa el algoritmo de la solución para determinar la ganancia por la venta de la leche producida.

1. Inicio
2. Leer L, PG
3. Hacer TG = (L / 3.785)
4. Hacer GA = PG * TG
5. Escribir GA
6. Fin

Pseudocódigo 2.7 Algoritmo para determinar la ganancia por la venta de leche.

De igual manera, el diagrama de flujo 2.7 muestra el algoritmo para la solución del problema.

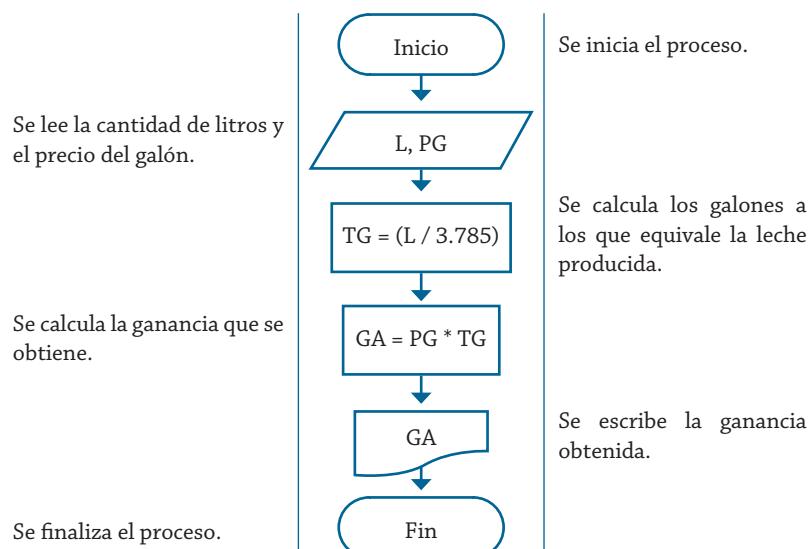


Diagrama de flujo 2.7 Algoritmo para determinar la ganancia por la venta de leche.

Ejemplo 2.8

Se requiere obtener la distancia entre dos puntos en el plano cartesiano, tal y como se muestra en la figura 2.4. Realice un diagrama de flujo y pseudocódigo que representen el algoritmo para obtener la distancia entre esos puntos.

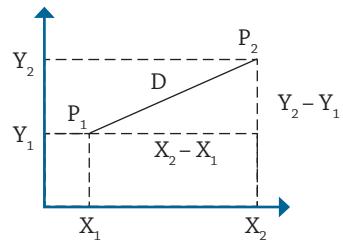


Figura 2.4 Representación gráfica de los puntos en el plano cartesiano.

Para resolver este problema es necesario conocer las coordenadas de cada punto (X, Y), y con esto poder obtener el cateto de abscisas y el de ordenadas, y mediante estos valores obtener la distancia entre P_1 y P_2 , utilizando el teorema de Pitágoras (ver figura 2.4). Por consiguiente, se puede establecer que las variables que se requieren para la solución de este problema son las mostradas en la tabla 2.9.

Nombre de la variable	Descripción	Tipo
X_1, X_2	Abscisas	Real
Y_1, Y_2	Ordenadas	Real
X	Cateto de las abscisas	Real
Y	Cateto de las ordenadas	Real
D	Distancia entre puntos	Real

Tabla 2.9 Variables utilizadas para obtener la distancia entre dos puntos.

Con base en lo anterior se puede constituir el diagrama de flujo 2.8, el cual corresponde al algoritmo para resolver este problema.

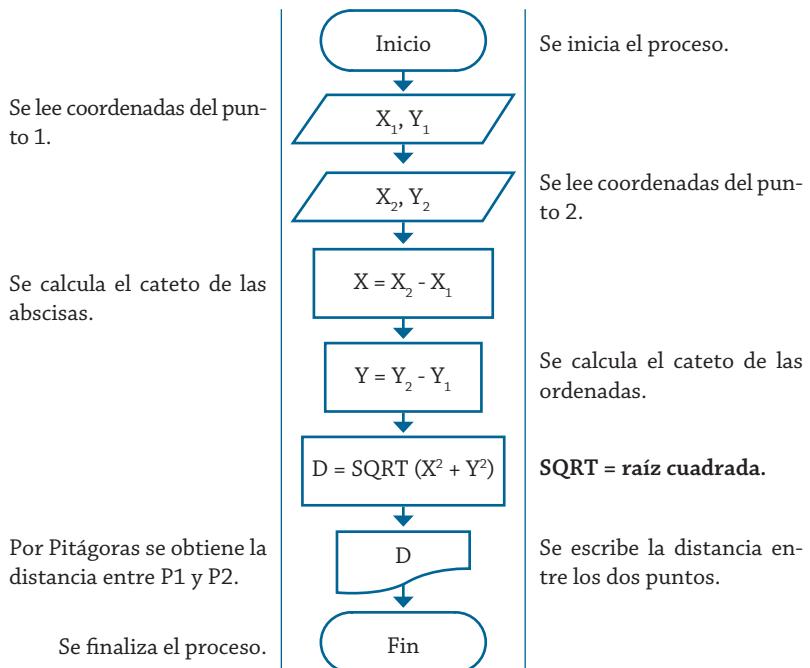


Diagrama de flujo 2.8 Algoritmo para obtener la distancia entre dos puntos.

El pseudocódigo 2.8 muestra el algoritmo correspondiente a la solución de este problema.

1. Inicio
2. Leer X_1, Y_1
3. Leer X_2, Y_2
4. Hacer $X = X_2 - X_1$
5. Hacer $Y = Y_2 - Y_1$
6. Hacer $D = \text{SQRT}(X * X + Y * Y)$
7. Escribir D
8. Fin

Pseudocódigo 2.8 Algoritmo para obtener la distancia entre dos puntos.

Ejemplo 2.9

Se requiere determinar el sueldo semanal de un trabajador con base en las horas que trabaja y el pago por hora que recibe. Realice el diagrama de flujo y el pseudocódigo que representen el algoritmo de solución correspondiente.

Para obtener la solución de este problema es necesario conocer las horas que labora cada trabajador y cuánto se le debe pagar por cada hora que labora, con base en esto se puede determinar que las variables que se requieren utilizar son las que se muestran en la Tabla 2.10.

Nombre de variable	Descripción	Tipo
HT	Horas trabajadas	Real
PH	Pago por hora	Real
SS	Sueldo semanal	Real

Tabla 2.10 Variables utilizadas para obtener el sueldo semanal de un trabajador.

El pseudocódigo 2.9 muestra el algoritmo con la solución correspondiente a este problema.

1. Inicio
2. Leer HT, PH
3. Hacer SS = HT*PH
4. Escribir SS
5. Fin

Pseudocódigo 2.9 Algoritmo para obtener el sueldo semanal de un trabajador.

Con base en lo anterior, se puede establecer que el diagrama de flujo 2.9 representa el algoritmo para resolver el problema.

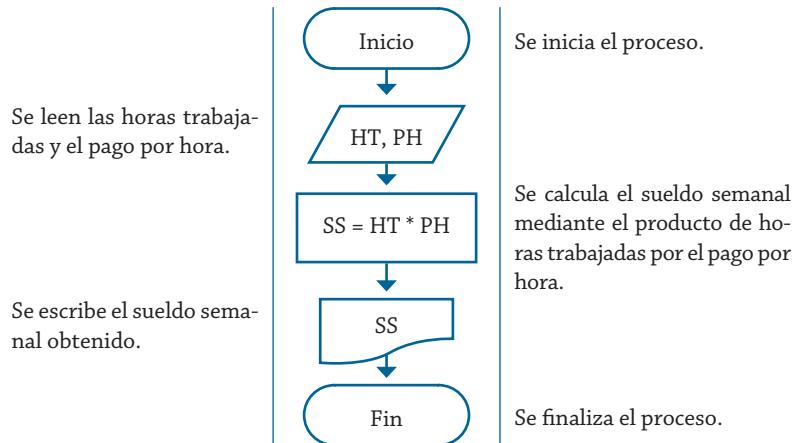


Diagrama de flujo 2.9 Algoritmo para obtener el sueldo semanal de un trabajador.

Ejemplo 2.10

Una modista, para realizar sus prendas de vestir, encarga las telas al extranjero. Para cada pedido, tiene que proporcionar las medidas de la tela en pulgadas, pero ella generalmente las tiene en metros. Realice un algoritmo para ayudar a resolver el problema, determinando cuántas pulgadas debe pedir con base en los metros que requiere. Represéntelo mediante el diagrama de flujo y el pseudocódigo (1 pulgada = 0.0254 m).

Prácticamente la solución de este problema radica en convertir los metros requeridos en pulgadas, por lo que para resolver el problema es adecuado utilizar las variables mostradas en la tabla 2.11.

Nombre de la variable	Descripción	Tipo
CM	Cantidad de metros que requiere	Real
PG	Pulgadas que debe pedir	Real

Tabla 2.11 Variables utilizadas para convertir los centímetros a pulgadas.

El pseudocódigo 2.10 muestra el algoritmo con la solución correspondiente a este problema.

1. Inicio
2. Leer CM
3. Hacer PG = CM / 0.0254 m
4. Escribir PG
5. Fin

Pseudocódigo 2.10 Algoritmo para convertir los metros a pulgadas.

Por consiguiente, se puede establecer que el diagrama de flujo 2.10 representa el algoritmo para resolver el problema.

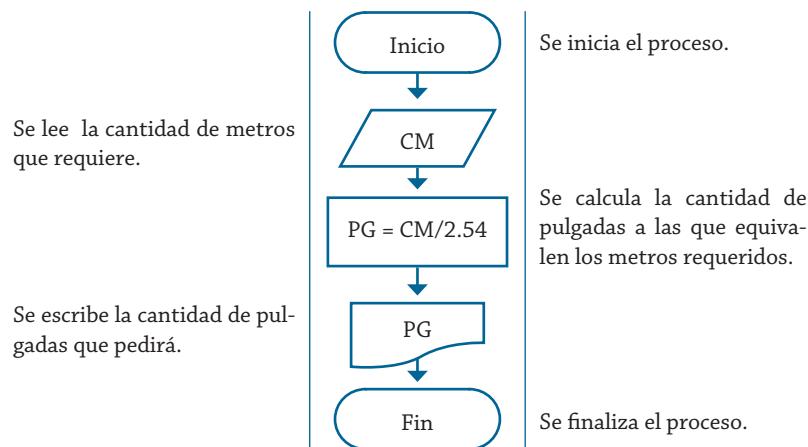


Diagrama de flujo 2.10 Algoritmo para convertir los metros a pulgadas.

Ejemplo 2.11

La conagua requiere determinar el pago que debe realizar una persona por el total de metros cúbicos que consume de agua al llenar una alberca (ver figura 2.5). Realice un algoritmo y represéntelo mediante un diagrama de flujo y el pseudocódigo que permita determinar ese pago.

Las variables requeridas para la solución de este problema se muestran en la tabla 2.12.

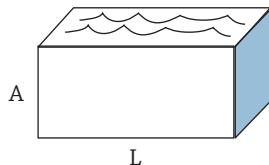


Figura 2.5 Forma de la alberca.

Nombre de la variable	Descripción	Tipo
A	Altura de la alberca	Real
L	Largo de la alberca	Real
N	Ancho de la alberca	Real
CM	Costo del metro cúbico	Real
V	Volumen de la alberca	Real
PAG	Pago a realizar por el consumo	Real

Fórmula: $V = (\text{largo} * \text{ancho} * \text{altura})$

Tabla 2.12 Variables utilizadas para determinar el pago por el agua requerida.

El diagrama de flujo 2.11 muestra el algoritmo correspondiente para determinar el pago.

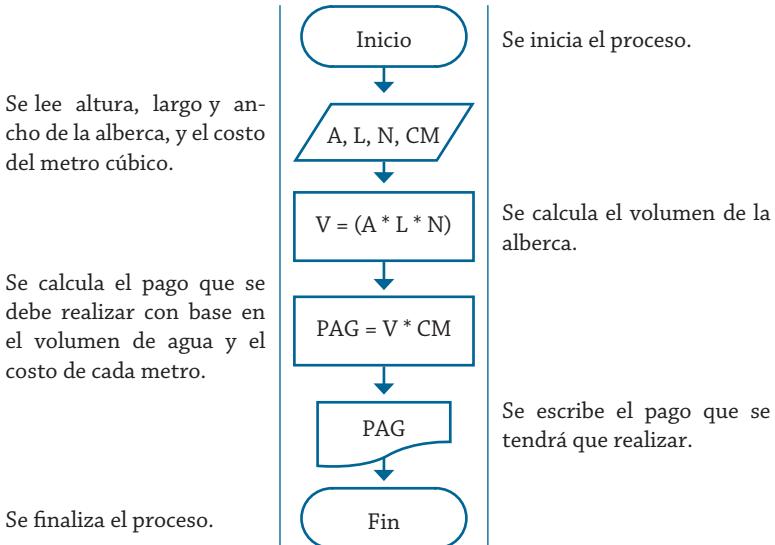


Diagrama de flujo 2.11 Algoritmo para determinar el pago por el agua requerida.

El pseudocódigo 2.11 muestra el algoritmo correspondiente para establecer el pago por los metros cúbicos consumidos.

1. Inicio
2. Leer A, L, N, CM
3. Hacer $V = (A * L * N)$
4. Hacer $PAG = V * CM$
5. Escribir PAG
6. Fin

Pseudocódigo 2.11 Algoritmo para determinar el pago por el agua requerida.

Problemas propuestos

- 2.1 Realice un diagrama de flujo y pseudocódigo que representen el algoritmo para obtener el área de un triángulo.
- 2.2 Una empresa importadora desea determinar cuántos dólares puede adquirir con equis cantidad de dinero mexicano. Realice un diagrama de flujo y pseudocódigo que representen el algoritmo para tal fin.
- 2.3 Una empresa que contrata personal requiere determinar la edad de las personas que solicitan trabajo, pero cuando se les realiza la entrevista sólo se les pregunta el año en que nacieron. Realice el diagrama de flujo y pseudocódigo que representen el algoritmo para solucionar este problema.
- 2.4 Un estacionamiento requiere determinar el cobro que debe aplicar a las personas que lo utilizan. Considere que el cobro es con base en las horas que lo disponen y que las fracciones de hora se toman como completas y realice un diagrama de flujo y pseudocódigo que representen el algoritmo que permita determinar el cobro.
- 2.5 Pinturas “La brocha gorda” requiere determinar cuánto cobrar por trabajos de pintura. Considere que se cobra por m² y realice un diagrama de flujo y pseudocódigo que representen el algoritmo que le permita ir generando presupuestos para cada cliente.

- 2.6 Se requiere determinar la hipotenusa de un triángulo rectángulo. ¿Cómo sería el diagrama de flujo y el pseudocódigo que representen el algoritmo para obtenerla? Recuerde que por Pitágoras se tiene que: $C^2 = A^2 + B^2$.
- 2.7 La compañía de autobuses “La curva loca” requiere determinar el costo que tendrá el boleto de un viaje sencillo, esto basado en los kilómetros por recorrer y en el costo por kilómetro. Realice un diagrama de flujo y pseudocódigo que representen el algoritmo para tal fin.
- 2.8 Se requiere determinar el tiempo que tarda una persona en llegar de una ciudad a otra en bicicleta, considerando que lleva una velocidad constante. Realice un diagrama de flujo y pseudocódigo que representen el algoritmo para tal fin.
- 2.9 Se requiere determinar el costo que tendrá realizar una llamada telefónica con base en el tiempo que dura la llamada y en el costo por minuto. Realice un diagrama de flujo y pseudocódigo que representen el algoritmo para tal fin.
- 2.10 La CONAGUA requiere determinar el pago que debe realizar una persona por el total de metros cúbicos que consume de agua. Realice un diagrama de flujo y pseudocódigo que representen el algoritmo que permita determinar ese pago.
- 2.11 La compañía de luz y sombras (CLS) requiere determinar el pago que debe realizar una persona por el consumo de energía eléctrica, la cual se mide en kilowatts (KW). Realice un diagrama de flujo y pseudocódigo que representen el algoritmo que permita determinar ese pago.
- 2.12 Realice un diagrama de flujo y pseudocódigo que representen el algoritmo para determinar cuánto pagará finalmente una persona por un artículo equis, considerando que tiene un descuento de 20%, y debe pagar 15% de IVA (debe mostrar el precio con descuento y el precio final).
- 2.13 Realice un diagrama de flujo y pseudocódigo que representen el algoritmo para determinar cuánto dinero ahorra una persona en un año si considera que cada semana ahorra 15% de su sueldo (considere cuatro semanas por mes y que no cambia el sueldo).
- 2.14 Una empresa desea determinar el monto de un cheque que debe proporcionar a uno de sus empleados que tendrá que ir por equis número de días a la ciudad de Monterrey; los gastos que cubre la empresa son: hotel, comida y 100.00 pesos diarios para otros gastos. El monto debe estar desglosado para cada concepto. Realice un diagrama de flujo y pseudocódigo que representen el algoritmo que determine el monto del cheque.
- 2.15 Se desea calcular la potencia eléctrica de circuito de la figura 2.6. Realice un diagrama de flujo y el pseudocódigo que representen el algoritmo para resolver el problema. Considere que: $P = V \cdot I$ y $V = R \cdot I$.

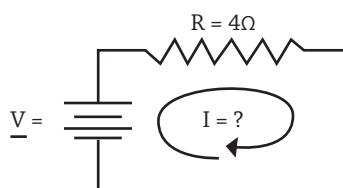


Figura 2.6 Circuito eléctrico.

- 2.16 Realice pseudocódigo y diagrama de flujo que representen el algoritmo para preparar una torta.

- 2.17 Realice pseudocódigo y diagrama de flujo que representen el algoritmo para confeccionar una prenda de vestir.
- 2.18 Realice pseudocódigo y diagrama de flujo que representen el algoritmo para preparar un pastel.
- 2.19 Realice el diagrama de flujo y pseudocódigo que representen el algoritmo para encontrar el área de un cuadrado.
- 2.20 Realice el diagrama de flujo y pseudocódigo que representen el algoritmo para determinar el promedio que obtendrá un alumno considerando que realiza tres exámenes, de los cuales el primero y el segundo tienen una ponderación de 25%, mientras que el tercero de 50%.
- 2.21 Realice un diagrama de flujo y pseudocódigo que representen el algoritmo para determinar aproximadamente cuántos meses, semanas, días y horas ha vivido una persona.
- 2.22 Se requiere un algoritmo para determinar el costo que tendrá realizar una llamada telefónica con base en el tiempo que dura la llamada y en el costo por minuto. Represente la solución mediante el diagrama de flujo y pseudocódigo.
- 2.23 El hotel “Cama Arena” requiere determinar lo que le debe cobrar a un huésped por su estancia en una de sus habitaciones. Realice un diagrama de flujo y pseudocódigo que representen el algoritmo para determinar ese cobro.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

UNIDAD III

SOLUCIÓN DE PROBLEMAS

CON ESTRUCTURAS SELECTIVAS

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

Introducción

Como se puede observar, los problemas que se han presentado hasta el momento no implican cuestionamientos como: “qué pasa si no le gusta con azúcar”, o bien, “qué pasa si le gusta más caliente”, esto en el algoritmo de preparar una taza de café, donde se puede seguir haciendo muchos cuestionamientos que conducen a tomar una decisión. Por consiguiente, los algoritmos, en determinados momentos, requieren ser selectivos en lo que respecta a las acciones que deben seguir, basándose en una respuesta de un determinado cuestionamiento que se formuló para la solución del problema planteado.

De aquí que las estructuras selectivas para los algoritmos sean tan importantes, de modo que en la mayoría de los problemas se tiene presente una estructura selectiva, que implica seguir o no un determinado flujo de secuencia del problema en cuestión.

Estructuras selectivas

En los algoritmos para la solución de problemas donde se utilizan estructuras selectivas se emplean frases que están estructuradas de forma adecuada dentro del pseudocódigo. En el caso del diagrama de flujo, también se estructura de una forma semejante. Ambos casos se muestran en la figura 3.1. En el caso del diagrama N/S con estructuras selectivas, se representa como se muestra en la figura 3.2.

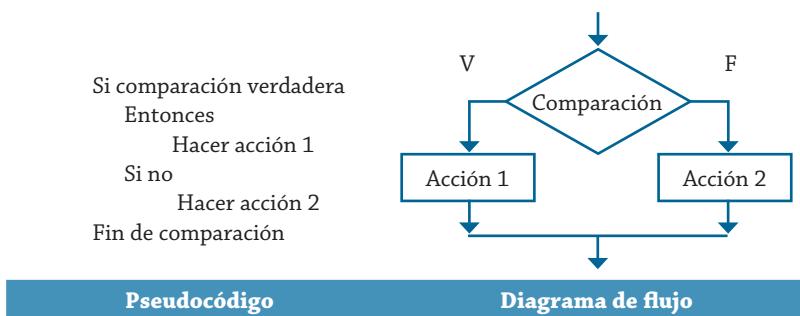


Figura 3.1 Forma de representar el algoritmo de una estructura selectiva.

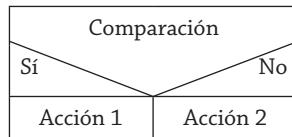


Figura 3.2 Forma de representar una estructura selectiva en el diagrama N/S.

Ejemplo 3.1

Se desea implementar un algoritmo para determinar cuál de dos valores proporcionados es el mayor. Representarlo con pseudocódigo, diagrama de flujo y diagrama N/S.

El pseudocódigo 3.1 presenta el algoritmo que permite determinar cuál de dos cantidades proporcionadas es la mayor.

1. Inicio
2. Leer A, B
3. Si $A > B$
 - Entonces
 - Hacer $M = A$
 - Si no
 - Hacer $M = B$
 - Fin de comparación
4. Escribir “el mayor es”, M
5. Fin

Pseudocódigo 3.1 Algoritmo para determinar cuál de dos cantidades es la mayor.

Como se puede ver, lo que se hace es comparar los dos valores que están asignados en las variables A y B respectivamente, que previamente se deben obtener mediante su lectura; posteriormente se comparan para determinar qué proceso hacer, en el caso de que A sea mayor que B, lo que procede es asignar A en la variable M; en caso contrario, el valor que se asigna a M es el que se guarda en B.

Una vez que se ha determinado cuál es el mayor y que se guardó en la variable M, lo que procede es escribir el resultado, con lo cual se concluye el proceso de solución.

Se puede establecer que la lectura del pseudocódigo o del diagrama de flujo debe ser de la siguiente forma:

Leer A y B, comparar si A es mayor que B, de ser verdad asignar el valor de A en la variable M, escribir M y fin. Éste sería el seguimiento que se daría en caso de ser verdad la comparación de variables, pero en caso de ser falso el proceso cambia, dado que el valor que tomaría la variable M es el de B para escribir este valor y finalizar el proceso. Como se puede ver, primeramente se debe seguir el camino de afirmación hasta llegar al fin, y después se recorre el de negación, esto es sólo para verificar la funcionalidad del algoritmo.

Partiendo del planteamiento del problema se puede establecer que las variables que se deben utilizar son las mostradas en la tabla 3.1.

Nombre de la variable	Descripción	Tipo
A	Primer valor para comparar	Entero
B	Segundo valor para comparar	Entero
M	Resultado de la comparación	Entero

Tabla 3.1 Variables utilizadas para determinar cuál de dos cantidades es la mayor.

De la misma forma, el diagrama de flujo 3.1 muestra el algoritmo que permite establecer cuál de las dos cantidades es la mayor.

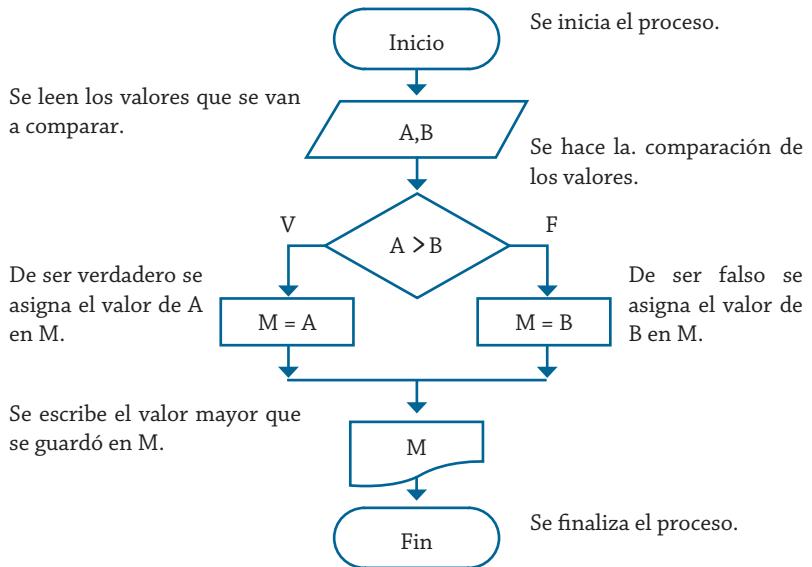


Diagrama de flujo 3.1 Algoritmo para determinar cuál de dos cantidades es la mayor.

Ahora, de una manera gráfica, se puede ver cuál es el proceso que se sigue para lograr la solución del problema planteado. Por otro lado, el diagrama N/S 3.1 presenta el algoritmo utilizando esta herramienta.

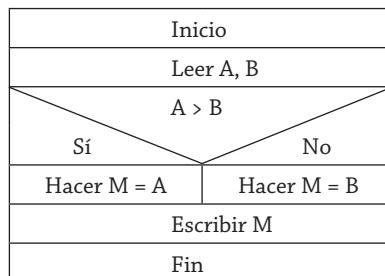


Diagrama N/S 3.1 Algoritmo para determinar cuál de dos cantidades es la mayor.

Como se puede ver, de nueva cuenta sí se sabe lo que se tiene que hacer; utilizar una u otra herramienta para presentar los algoritmos de solución a problemas es prácticamente indistinto.

Ejemplo 3.2

Realice un algoritmo para determinar si un número es positivo o negativo. Represéntelo en pseudocódigo, diagrama de flujo y diagrama N/S.

Como ya se mencionó anteriormente, para resolver cualquier problema se debe partir de la primicia de conocer qué variables son necesarias para resolverlo, sobre todo en aquéllos que no requieren de muchos identificadores en el proceso de solución, cuando esto sucede se puede proceder a generar primeramente la tabla de variables, aunque es posible establecerla al final o paralelamente al momento de la solución del problema, ya que a medida que se avanza con la solución surge la necesidad de utilizar nuevas variables.

Para este caso, la tabla 3.2 muestra las variables que se requieren en la solución del problema.

Nombre de la variable	Descripción	Tipo
NÚM	Valor para determinar su signo	Entero
R	Resultado del signo del valor	<i>String</i>

Tabla 3.2 Variables utilizadas para determinar si un número es positivo o negativo.

Mediante el pseudocódigo 3.2 represente el algoritmo que permite determinar si el número que se proporciona es positivo o negativo.

1. Inicio.
2. Leer NÚM
3. Si NÚM > = 0
 Entonces
 Hacer R = “POSITIVO”
 Si no
 Hacer R = “NEGATIVO”
 Fin de comparación
4. Escribir “el número es”, R
5. Fin

Pseudocódigo 3.2 Algoritmo para determinar si un número es positivo o negativo.

Como se puede ver, para determinar si un número es positivo o negativo, sólo es necesario establecer si éste es mayor o igual a cero; si el resultado de la comparación es afirmativa, a la variable R se le asignará el valor de “POSITIVO”, si resulta una negación, por consiguiente, el valor que tome R será de “NEGATIVO”.

Ahora, el diagrama de flujo 3.2 muestra el algoritmo que permite obtener la solución del problema tal y como se presenta mediante la utilización de pseudocódigo.

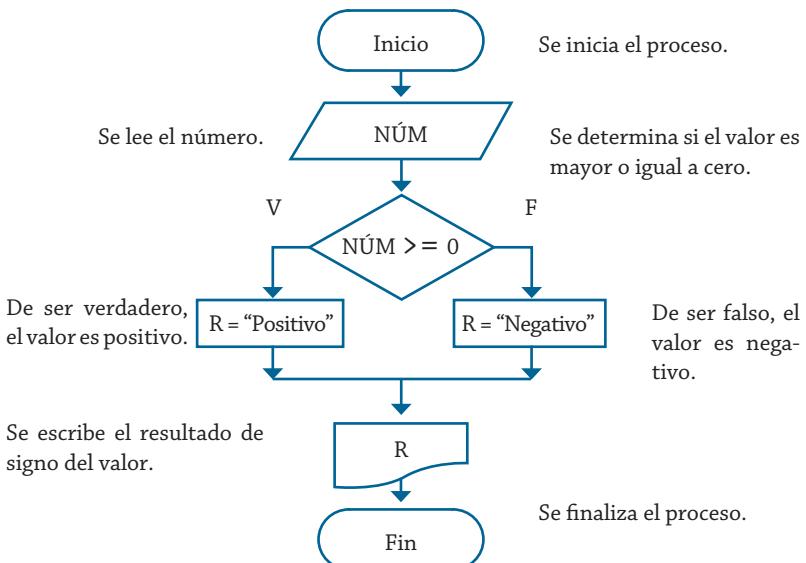


Diagrama de flujo 3.2 Algoritmo para determinar si un número es positivo o negativo.

De nueva cuenta y de manera gráfica, se puede ver cuál es el proceso que se sigue para lograr la solución del problema planteado. El diagrama N/S 3.2 muestra el algoritmo mediante esta herramienta.

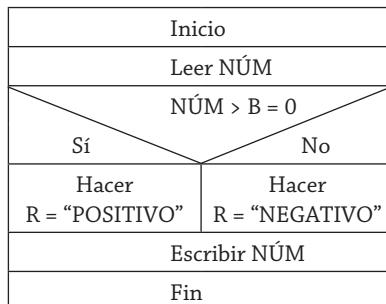


Diagrama N/S 3.2 Algoritmo para determinar si un número es positivo o negativo.

Si se compara el diagrama de flujo con el diagrama N/S, se puede observar que prácticamente son iguales, sólo que al diagrama N/S le faltan las líneas de flujo de datos que se utilizan en los diagramas de flujo. Así, decidir cuál herramienta es la más apropiada para la representación de los algoritmos dependerá básicamente del gusto del diseñador del algoritmo.

Ejemplo 3.3

Realice un algoritmo para determinar cuánto se debe pagar por equis cantidad de lápices considerando que si son 1000 o más el costo es de 85¢; de lo contrario, el precio es de 90¢. Represéntelo con el pseudocódigo, el diagrama de flujo y el diagrama N/S.

Partiendo de que ahora ya se tiene un poco más de experiencia en la formulación de algoritmos para la solución de problemas, se puede partir de nueva cuenta con establecer la tabla de variables que se pueden utilizar

en el planteamiento de la solución de un problema. Por consiguiente, la tabla 3.3 muestra las variables que se utilizan en la solución del problema.

Nombre de la variable	Descripción	Tipo
X	Cantidad de lápices	Entero
PAG	Pago que se realizará por los lápices	Real

Tabla 3.3 Variables utilizadas para determinar cuánto se paga por equis cantidad de lápices.

Cabe mencionar de nueva cuenta que el nombre de los identificadores que se utilizan son asignados de forma arbitraria por parte del diseñador del algoritmo.

Una vez que se determinaron las variables que se van a utilizar, el pseudocódigo 3.3 muestra el algoritmo correspondiente para determinar cuánto se debe pagar.

1. Inicio
2. Leer X
3. Si $X \geq 1000$
 Entonces
 Hacer $PAG = X * 0.85$
 Si no
 Hacer $PAG = X * 0.90$
 Fin de comparación
4. Escribir “el pago es”, PAG
5. Fin

Pseudocódigo 3.3 Algoritmo para determinar cuánto se paga por equis cantidad de lápices.

Ahora el algoritmo correspondiente se puede representar de la forma mostrada en el diagrama de flujo 3.3, el cual permite obtener el pago que se va a realizar por la compra de la cantidad equis de lápices, donde se puede resumir que si son más de mil, el número de lápiz se multiplica por 0.85, de lo contrario, el producto se efectúa por 0.90, con lo cual se obtiene el resultado que se busca.

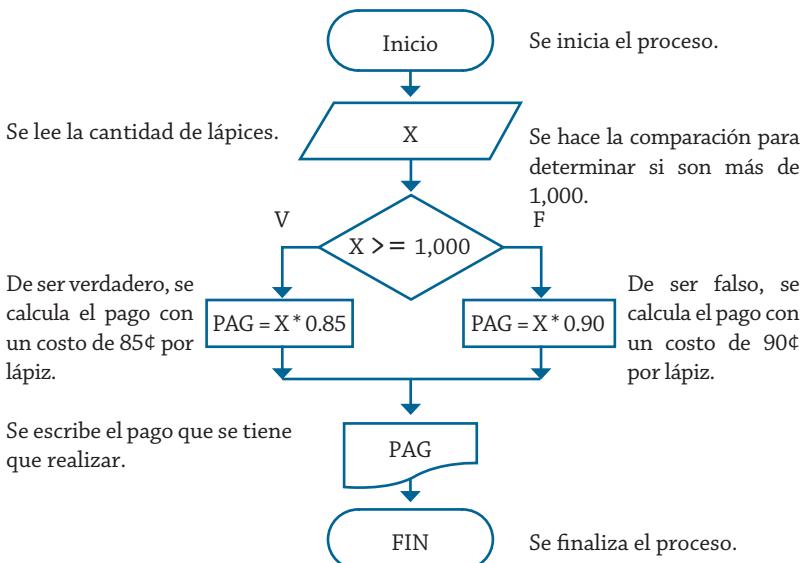


Diagrama de flujo 3.3 Algoritmo para determinar cuánto se paga por equis cantidad de lápices.cantidad de lápices.

El diagrama N/S 3.3 muestra el algoritmo de la solución correspondiente al problema mediante la utilización de la herramienta de Nassi-Schneiderman.

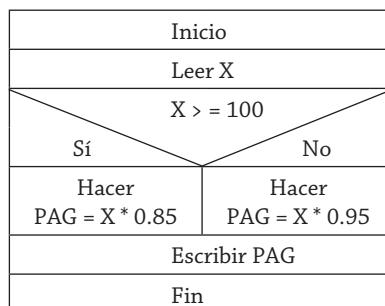


Diagrama N/S 3.3 Algoritmo para determinar cuánto se paga por equis cantidad de lápices.

Y como se puede ver, el diagrama N/S es semejante al diagrama de flujo que se estableció previamente.

Ejemplo 3.4

Almacenes “El harapiento distinguido” tiene una promoción: a todos los trajes que tienen un precio superior a \$2500.00 se les aplicará un descuento de 15 %, a todos los demás se les aplicará sólo 8 %. Realice un algoritmo para determinar el precio final que debe pagar una persona por comprar un traje y de cuánto es el descuento que obtendrá. Represéntelo mediante el pseudocódigo, el diagrama de flujo y el diagrama N/S.

El pseudocódigo 3.4 representa el algoritmo para determinar el descuento y el precio final que tendrá un determinado traje.

1. Inicio
2. Leer CT
3. Si CT > 2500
 - Entonces
 - Hacer DE = CT * 0.15
 - Si no
 - Hacer DE = CT * 0.08
- Fin de comparación
4. Hacer PF = CT - DE
5. Escribir "El precio final es", PF
6. Escribir "El descuento es" DE
7. Fin

Pseudocódigo 3.4 Algoritmo para determinar cuánto se paga por adquirir un traje.

Como se puede ver, una vez que se obtuvo el descuento que se aplicará, se hace un solo cálculo para determinar el precio final de la prenda; sin embargo, también se puede hacer de la siguiente forma:

```

Entonces
  Hacer DE = CT * 0.15
  Hacer PF = CT - DE
Si no
  Hacer DE = CT * 0.08
  Hacer PF = CT - DE

```

Realizar este cálculo del precio final inmediatamente después de haber obtenido el descuento implica procesos de más, ya que la manera como se realizó en el pseudocódigo 3.2.4 es más eficiente y correcta, pues se tiene el ahorro de un proceso, pero habrá algoritmos en los que el ahorro sea más significativo.

Con base en el pseudocódigo que se estableció se puede obtener la tabla 3.4, que contiene las variables que intervienen en el proceso de solución del problema.

Nombre de la variable	Descripción	Tipo
CT	Costo del traje	Real
DE	Descuento que se obtendrá	Real
PF	Precio final del traje	Real

Tabla 3.4 Variables utilizadas para determinar cuánto se paga por adquirir un traje.

Mientras que el diagrama de flujo 3.4 muestra la representación de este mismo algoritmo con esta herramienta, el diagrama N/S 3.4 muestra el resultado utilizando los diagramas de Nassi Schneiderman. Si se realiza una comparación entre estas dos herramientas, se puede observar que su estructuración no cambia, si no que es la misma, sólo se están omitiendo las flechas de flujo de datos.

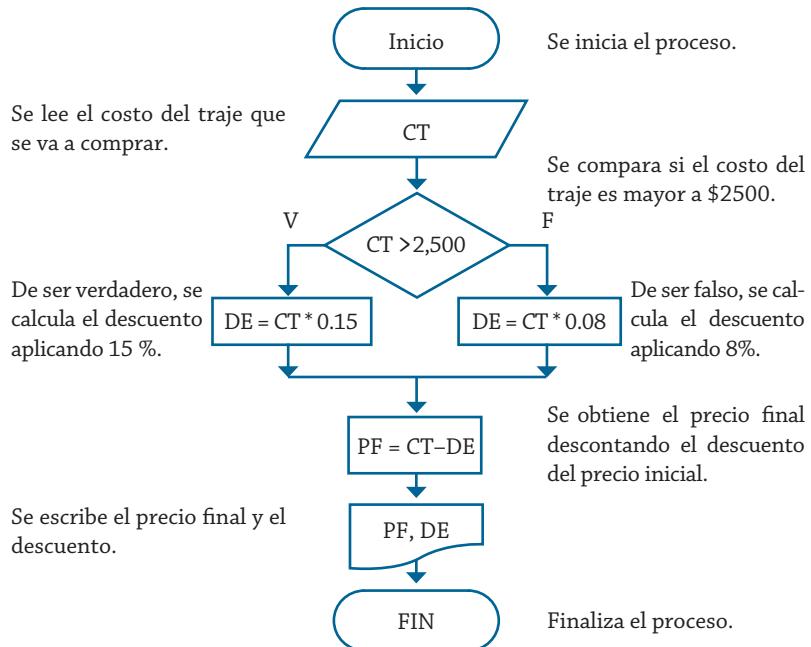


Diagrama de flujo 3.4 Algoritmo para determinar cuánto se paga por adquirir un traje.

Inicio	
Leer CT	
CT > 2,500	
Sí	No
Hacer $DE = CT * 0.15$	Hacer $DE = CT * 0.08$
Hacer $PF = CT - DE$	
Escribir PF, DE	
Fin	

Diagrama N/S 3.4 Algoritmo para determinar cuánto se paga por adquirir un traje.

Hasta ahora, los problemas vistos sólo presentan una decisión para realizar un determinado proceso; sin embargo, en algunas ocasiones es necesario elaborar estructuras selectivas en cascada, esto significa que después de haber realizado una comparación selectiva es necesario realizar otra comparación selectiva como resultado de la primera condición. En la figura 3.3 se presentan las formas correcta e incorrecta de estructurar el pseudocódigo para este caso:

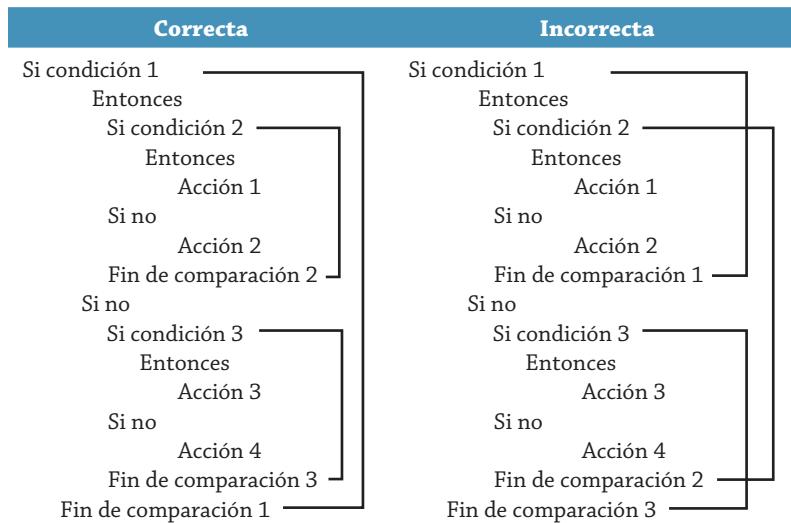


Figura 3.3 Forma correcta e incorrecta de representar una estructura selectiva anidada.

Como se puede ver, en la estructuración la primera condición que se abre es la última que se cierra, en la figura 3.4 se muestra el respectivo diagrama de flujo, en el cual se tiene el mismo principio mostrado en la figura 3.3.

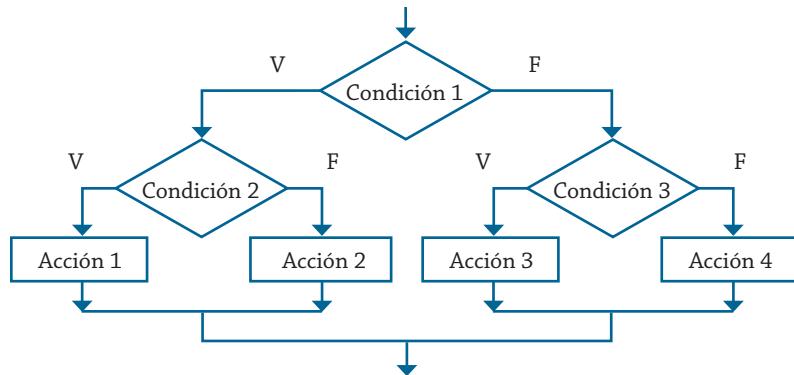


Figura 3.4 Forma de estructurar un diagrama de flujo con condiciones anidadas.

En el ejemplo 3.5 se considera este tipo de situaciones, y en los problemas subsiguientes se presentan estructuras un poco diferentes pero bajo el mismo principio de operación.

Ejemplo 3.5

Se requiere determinar cuál de tres cantidades proporcionadas es la mayor. Realizar su respectivo algoritmo y representarlo mediante un diagrama de flujo, pseudocódigo y diagrama N/S.

Las variables que intervienen en la solución de este problema se muestran en la tabla 3.5.

Nombre de la variable	Descripción	Tipo
A	Primer valor	Entero o real
B	Segundo valor	Entero o real
C	Tercer valor	Entero o real
M	Valor mayor	Entero o real

Tabla 3.5 Variables utilizadas para determinar cuál de tres cantidades es la mayor.

El diagrama de flujo 3.5 muestra la estructura del algoritmo correspondiente para la solución de este problema.

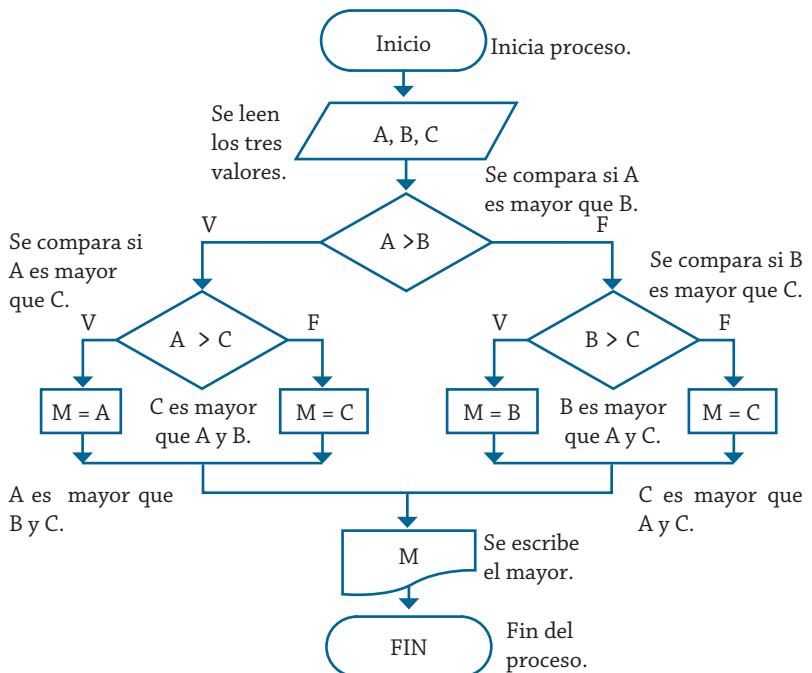


Diagrama de flujo 3.5. Algoritmo para determinar cuál de tres cantidades es la mayor.

Como se puede ver, primeramente se compara si A es mayor que B; de ser verdad, entonces ahora se compara A contra C, y finalmente esta comparación determinará cuál de los tres valores es el mayor. Si de nuevo cuenta A fue la mayor, o en su caso C, ya no es necesario compararlo contra B, ya que inicialmente A fue mayor que B. En el caso de que A no fuera mayor que B, entonces se procede a realizar la comparación de B contra C y con esto se determina cuál es la mayor de las tres cantidades.

En la representación de esta solución se muestran estructuras selectivas en cascada, ya que así lo amerita la solución del problema. El pseudo-código 3.5 presenta el algoritmo correspondiente mediante la utilización de esta herramienta.

1. Inicio
2. Leer A, B, C
3. Si $A > B$

Entonces

Si $A > C$

Entonces

$M = A$

Si no

$M = C$

Fin compara

Si no

Si $B > C$

Entonces

$M = B$

Si no

$M = C$

Fin compara

Fin compara
4. Escribir "El mayor es", M
5. Fin

Pseudocódigo 3.5 Para determinar cuál de tres cantidades es la mayor.

Por otro lado, el diagrama N/S 3.5 presenta el algoritmo de solución mediante esta herramienta.

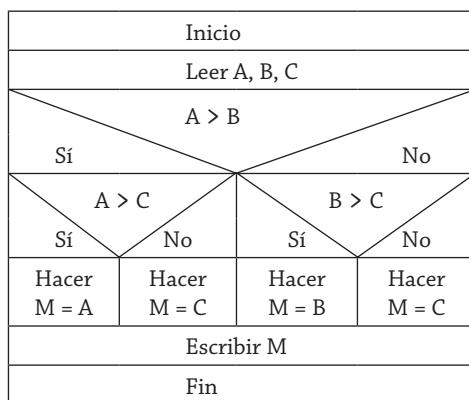


Diagrama N/S 3.5 Algoritmo para determinar cuál de tres cantidades es la mayor.

Un algoritmo es perfectible, o en su caso puede ser sustituido por otro con otras características que conducen a la misma solución, la diferencia que se presenta se puede basar en la eficiencia que presente uno con respecto a otro de los algoritmos; en la mayoría de los casos esta eficiencia se mide con respecto al número de pasos y variables que intervienen en el proceso de solución del problema, que se puede reflejar en el tiempo de respuesta. Para este problema 3.2.5 se puede establecer un algoritmo de solución como el que se presenta mediante el diagrama de flujo 3.6.

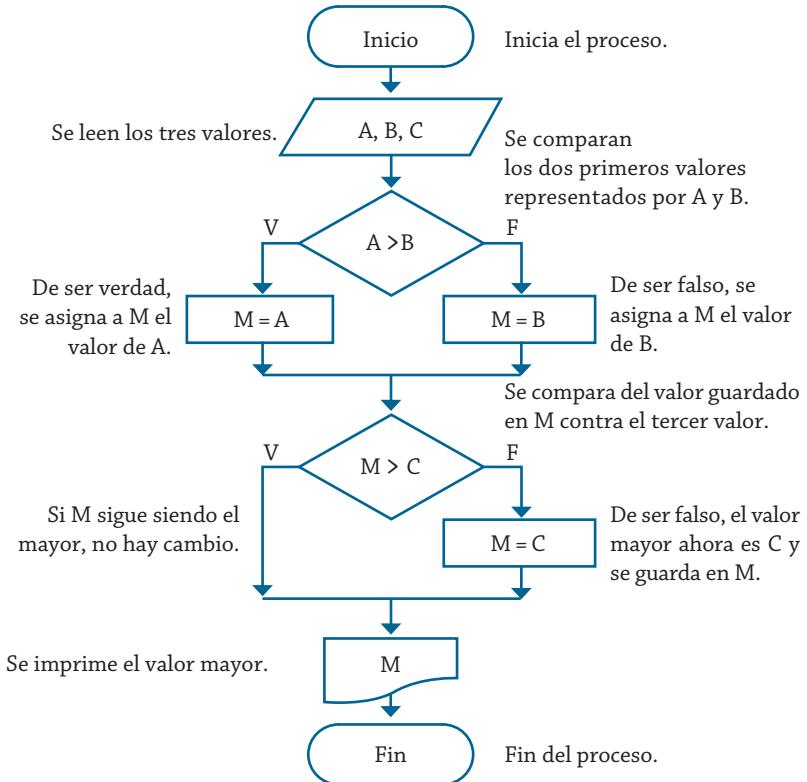


Diagrama N/S 3.5 Algoritmo para determinar cuál de tres cantidades es la mayor.

Como se puede ver, en esta solución que se presenta ahora sólo se compara los dos primeros valores (A y B), de los cuales se almacena el mayor en una variable auxiliar (M), la cual se compara con el tercer valor (C), y a partir de esta comparación se establece cuál valor es el mayor. Nótese que en ambas alternativas no se considera determinar el nombre de la variable, sino sólo se pide el valor que se almacena en ella.

Ejemplo 3.6

“La langosta ahumada” es una empresa dedicada a ofrecer banquetes; sus tarifas son las siguientes: el costo de platillo por persona es de \$95.00, pero si el número de personas es mayor a 200 pero menor o igual a 300, el costo es de \$85.00. Para más de 300 personas el costo por platillo es de \$75.00. Se requiere un algoritmo que ayude a determinar el presupuesto que se debe presentar a los clientes que deseen realizar un evento. Mediante pseudocódigo, diagrama de flujo y un diagrama N/S represente su solución.

Para la solución del problema se requiere saber el número de personas que se presupuestarán para el banquete, y con base en éstas determinar el costo del platillo que en cierta forma es constante, con éste se determinará cuánto debe pagar el cliente en total, de aquí que la tabla 3.6 muestre las variables que se utilizarán para la solución del problema. El pseudocódigo 3.6 presenta el algoritmo de solución de este problema.

Nombre de la variable	Descripción	Tipo
NP	Número de personas	Entero
TOT	Total que se va a pagar por el banquete	Real

Tabla 3.6 Variables utilizadas para determinar el presupuesto de un banquete.

1. Inicio
2. Leer NP
3. Si $NP > 300$
 - Entonces

$$Hacer TOT = NP * 75$$
 - Si no
 - Si $NP > 200$
 - Entonces

$$Hacer TOT = NP * 85$$
 - Si no

$$Hacer TOT = NP * 95$$
4. Escribir "El total es", TOT
5. Fin

Pseudocódigo 3.6 Algoritmo para determinar el presupuesto de un banquete.

Con el algoritmo representado mediante el pseudocódigo correspondiente ya establecido ahora se facilita presentarlo con el diagrama N/S 3.6 o bien con el diagrama de flujo 3.7.

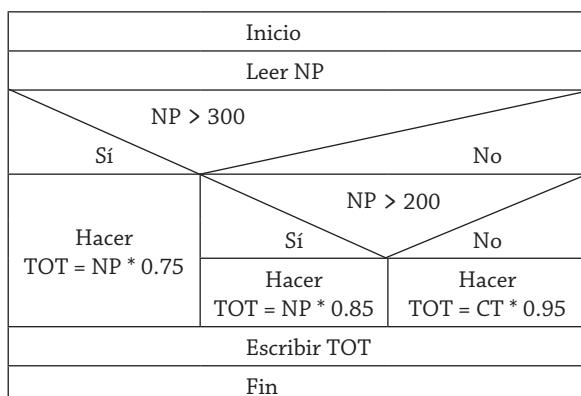


Diagrama N/S 3.6 Algoritmo para determinar el presupuesto de un banquete.

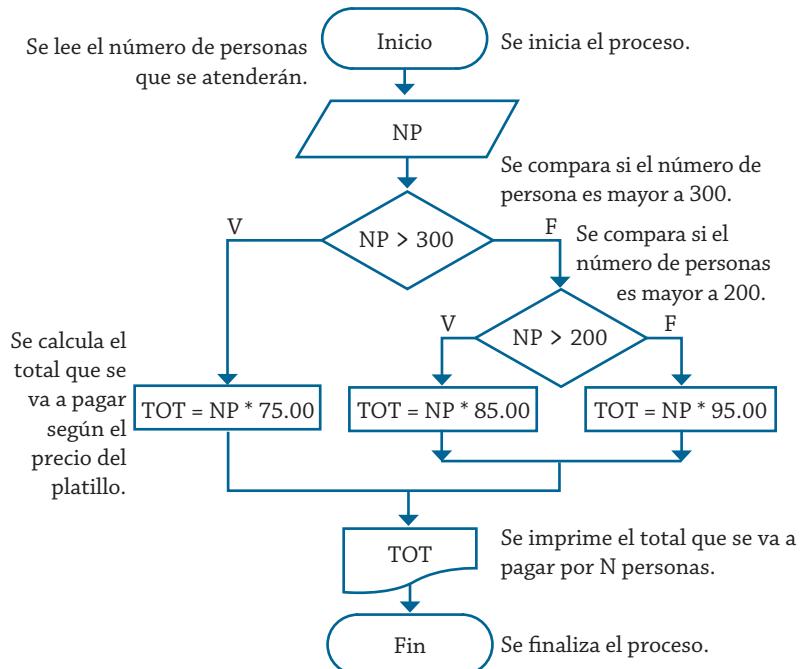


Diagrama de flujo 3.7 Algoritmo para determinar el presupuesto de un banquete.

En muchas ocasiones un mismo proceso se puede dividir en más procesos sin que esto altere el resultado como se puede ver en el diagrama de flujo 3.8, que muestran una alternativa para la solución del mismo problema. Para esta alternativa primeramente se asigna el precio al platillo y seguido a esto se calcula el pago total.

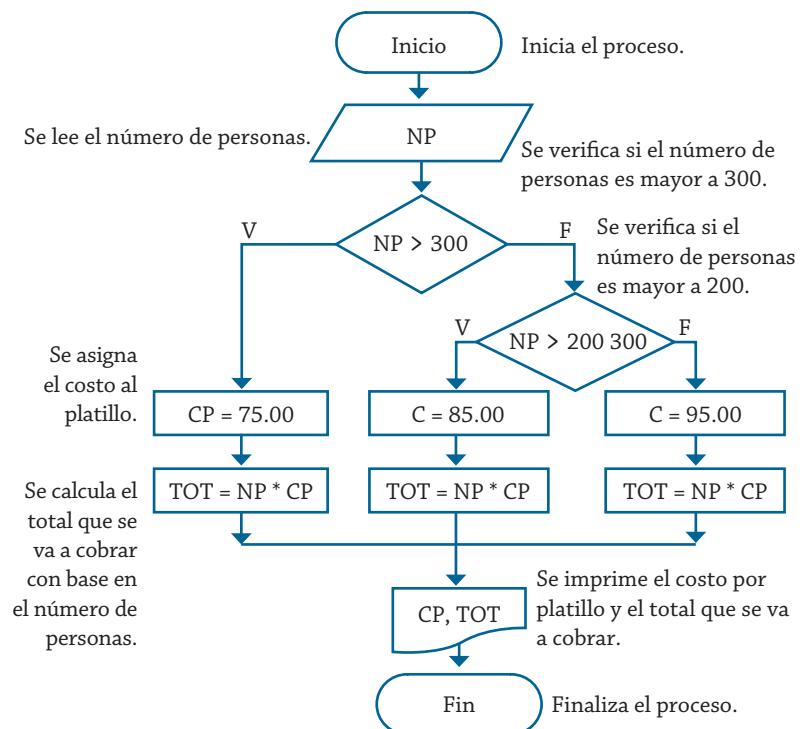


Diagrama de flujo 3.8 Algoritmo para determinar el presupuesto de un banquete.

Sin embargo, se pudiera presentar otra alternativa de solución, que tendría la forma del diagrama de flujo 3.9, en la cual se podrá observar que el pago total se realiza mediante un proceso común para las tres asignaciones de precio de platillo.

Si se analizan las tres alternativas en cuestión de número de procesos empleados, la primera alternativa se puede considerar como la más eficiente, dado que emplea menos procesos o instrucciones para su solución; sin embargo, las tres alternativas son válidas y correctas, ya que cumplen con las características y condiciones que debe tener todo algoritmo: resolver de forma eficiente un problema dado. En la solución de muchos problemas es recomendable utilizar esta última opción, donde se emplea un proceso común para varias alternativas selectivas que se presentan en la solución del problema. Pero como se ha mencionado anteriormente, la forma dependerá del diseñador del mismo y de las necesidades que se tengan que cubrir con la solución que se establezca.

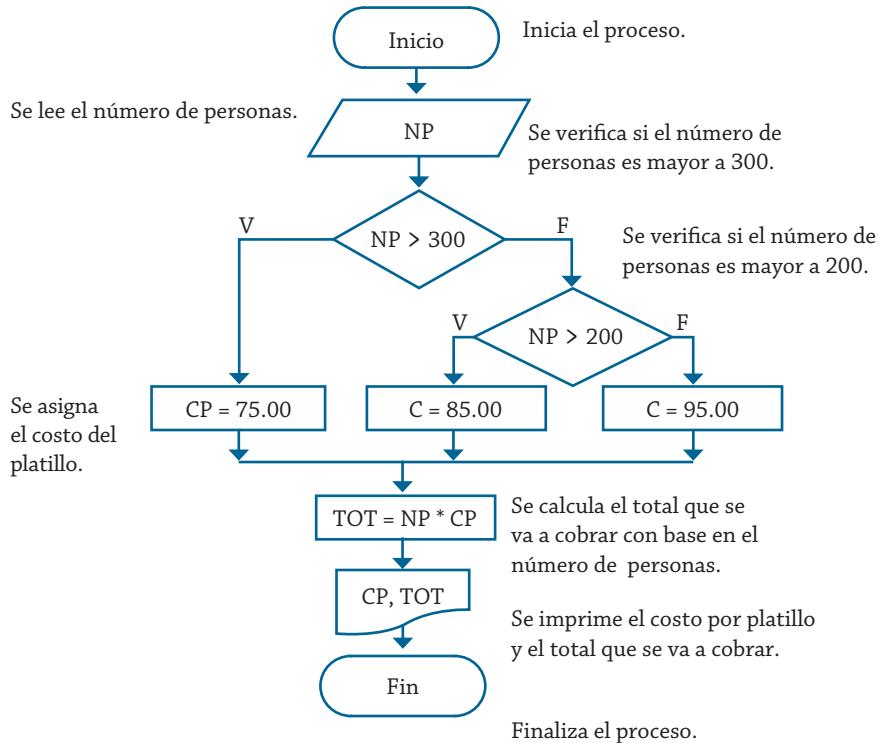


Diagrama de flujo 3.9 Algoritmo para determinar el presupuesto de un banquete.

Ejemplo 3.7

La asociación de vinicultores tiene como política fijar un precio inicial al kilo de uva, la cual se clasifica en tipos **A** y **B**, y además en tamaños **1** y **2**. Cuando se realiza la venta del producto, ésta es de un solo tipo y tamaño, se requiere determinar cuánto recibirá un productor por la uva que entrega en un embarque, considerando lo siguiente: si es de tipo **A**, se le cargan 20¢ al precio inicial cuando es de tamaño **1**; y 30¢ si es de tamaño **2**. Si es de tipo **B**, se rebajan 30¢ cuando es de tamaño **1**, y 50¢ cuando es de tamaño **2**. Realice un algoritmo para determinar la ganancia obtenida y representelo mediante diagrama de flujo, pseudocódigo y diagrama N/S.

Realizando un análisis de los datos que se requieren y de los resultados que se deben obtener, se puede determinar que son los que se muestran en la tabla 3.7, y con base en esto se puede representar el algoritmo con el diagrama de flujo 3.10.

Nombre de la variable	Descripción	Tipo
TI	Tipo de la uva	String
TA	Tamaño de la uva	Entero
P	Precio de la uva	Real
K	Kilos de producción	Entero
GA	Ganancia obtenida	Real

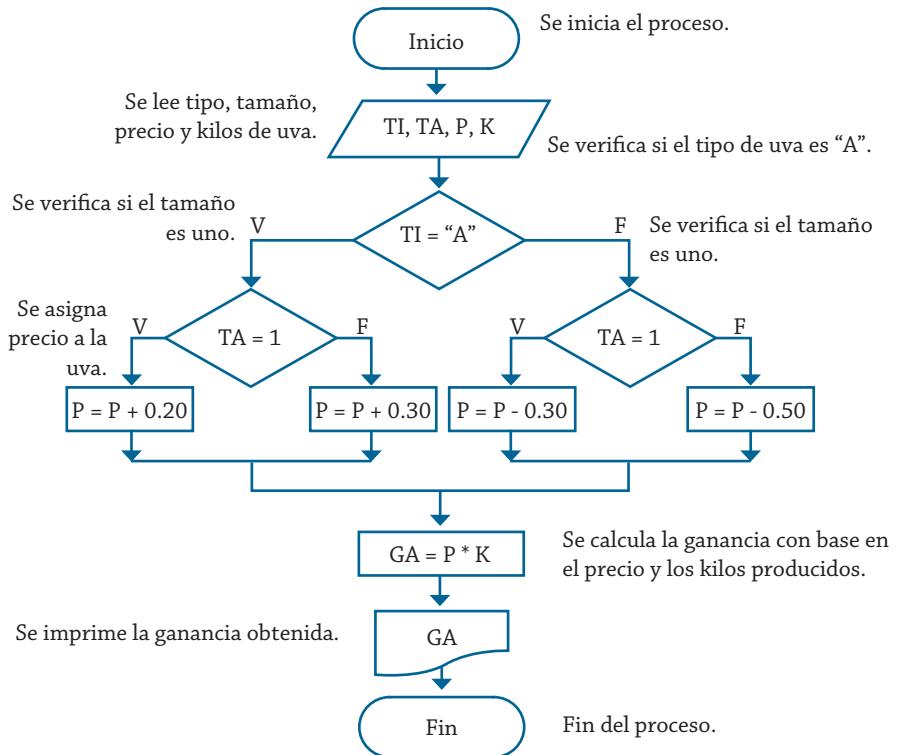


Diagrama de flujo 3.10 Algoritmo para determinar las ganancias por la venta de la uva.

El pseudocódigo 3.7 muestra el algoritmo correspondiente.

1. Inicio
2. Leer TI, TA, P, K
3. Si TI = "A"
 - Entonces
 - Si TA = 1
 - Entonces
 - P = P + 0.20
 - Si no
 - P = P + 0.30
 - Si no
 - Si TA = 1
 - Entonces
 - P = P - 0.30
 - Si no
 - P = P - 0.50
 4. Hacer GA = P * K
 5. Escribir "La ganancia es", GA
 6. Fin

Pseudocódigo 3.7 Algoritmo para determinar las ganancias por la venta de la uva.

Por otro lado, el diagrama N/S 3.7 presenta el algoritmo con la utilización de esta herramienta.

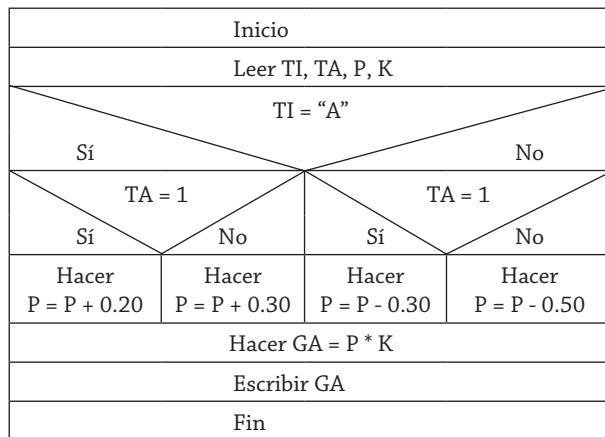


Diagrama N/S 3.7 Algoritmo para determinar las ganancias por la venta de la uva.

Como se puede ver en la solución de este problema, el cálculo de la ganancia por cada productor se realiza mediante un proceso común para todas las alternativas del precio de la uva, éste es un caso de los que se mencionaron en el problema 3.2.6, si la ganancia se hubiera obtenido después de cada asignación de precio de la uva, esto traería como consecuencia el incremento de tres procesos más de los que se emplean con la opción de solución planteada para este problema.

Ejemplo 3.8

El director de una escuela está organizando un viaje de estudios, y requiere determinar cuánto debe cobrar a cada alumno y cuánto debe pagar a la compañía de viajes por el servicio. La forma de cobrar es la siguiente: si son 100 alumnos o más, el costo por cada alumno es de \$65.00; de 50 a 99 alumnos, el costo es de \$70.00, de 30 a 49, de \$95.00, y si son menos de 30, el costo de la renta del autobús es de \$4000.00, sin importar el número de alumnos.

Realice un algoritmo que permita determinar el pago a la compañía de autobuses y lo que debe pagar cada alumno por el viaje (represente en pseudocódigo, diagrama de flujo y diagrama N/S la solución).

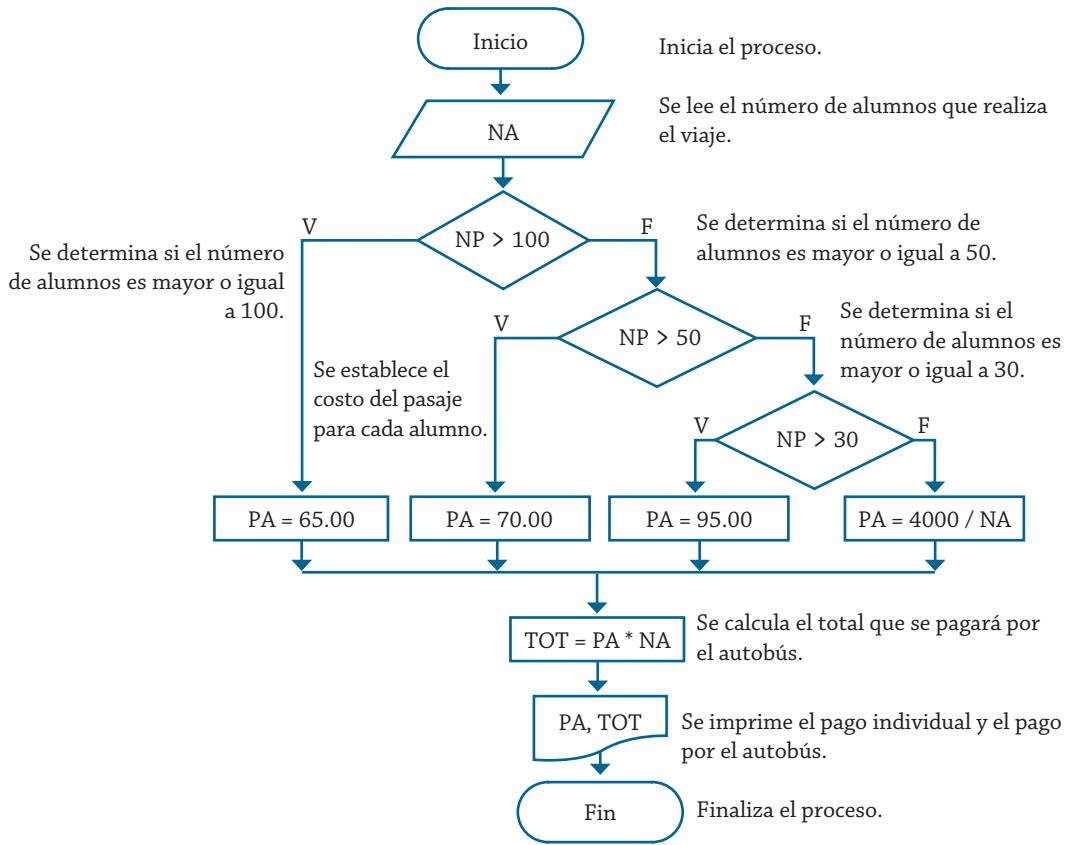
Al realizar un análisis del problema, se puede deducir que las variables que se requieren como datos son el número de alumnos (NA), con lo que se puede calcular el pago por alumno (PA) y el costo total del viaje (TOT). Las características de estas variables se muestran en la tabla 3.8.

Nombre de la variable	Descripción	Tipo
NA	Número de alumnos que realizan el viaje	Entero
PA	Pago por alumno	Real
TOT	Total que va a pagar a la empresa por el viaje	Real

A partir de lo anterior, se puede establecer el pseudocódigo 3.8, el cual presenta la solución del problema. Y de igual forma, lo presenta el diagrama de flujo 3.11.

1. Inicio
2. Leer NA
3. Si $NA \geq 100$
Entonces
 Hacer $PA = 65.0$
- Si no
 Si $NA \geq 50$
 Entonces
 Hacer $PA = 70.0$
- Si no
 Si $NA \geq 30$
 Entonces
 Hacer $PA = 95.0$
- Si no
 Hacer $PA = 4000 / NA$
- Fin compara
- Fin compara
- Fin compara
4. Hacer $TOT = PA * NA$
5. Escribir “El pago individual es”, PA
6. Escribir “El pago total es”, TOT
7. Fin

Pseudocódigo 3.8 Algoritmo para determinar el total que se va a pagar por el viaje.



Como se puede ver, con base en el número de alumnos, se asigna el pago de los mismos de manera directa cuando se presenta que éstos son mayores a 30, para el caso de que no sea así se debe proceder a determinar mediante la división de los 4000 pesos que cuesta todo el camión entre el número de alumnos que viajarán.

El diagrama N/S 3.8 muestra el algoritmo mediante esta herramienta.

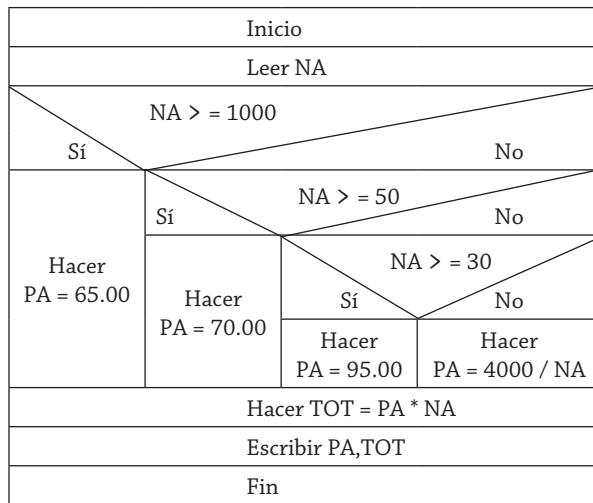


Diagrama N/S 3.8 Algoritmo para determinar el total que se va a pagar por el viaje.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

Ejemplo 3.9

La política de la compañía telefónica “chimefón” es: “Chisme + x –”. Cuando se realiza una llamada, el cobro es por el tiempo que ésta dura, de tal forma que los primeros cinco minutos cuestan \$ 1.00 c/u, los siguientes tres, 80¢ c/u, los siguientes dos minutos, 70¢ c/u, y a partir del décimo minuto, 50¢ c/u.

Además, se carga un impuesto de 3 % cuando es domingo, y si es día hábil, en turno matutino, 15 %, y en turno vespertino, 10 %. Realice un algoritmo para determinar cuánto debe pagar por cada concepto una persona que realiza una llamada. Represéntelo en diagrama de flujo, en pseudocódigo y en diagrama N/S.

Al analizar el problema se puede identificar que será necesario conocer como datos la duración de la llamada, así como el día y turno en que se realiza. Con base en esto se podrá determinar cuál será el pago que se efectuará por el tiempo que dura la llamada y el impuesto que deberá pagar en función del día y del turno en que se realiza. La tabla 3.9 muestra las variables que se van a utilizar.

Nombre de la variable	Descripción	Tipo
TI	Tiempo	Entero
DI	Tipo de día	String
TU	Turno	String
PAG	Pago por el tiempo	Real
IMP	Impuesto	Real
TOT	Total que se va a pagar	Real

Tabla 3.9 Variables utilizadas para determinar el costo de una llamada telefónica.

El algoritmo correspondiente se muestra en el diagrama de flujo 3.12, que resuelve este problema.

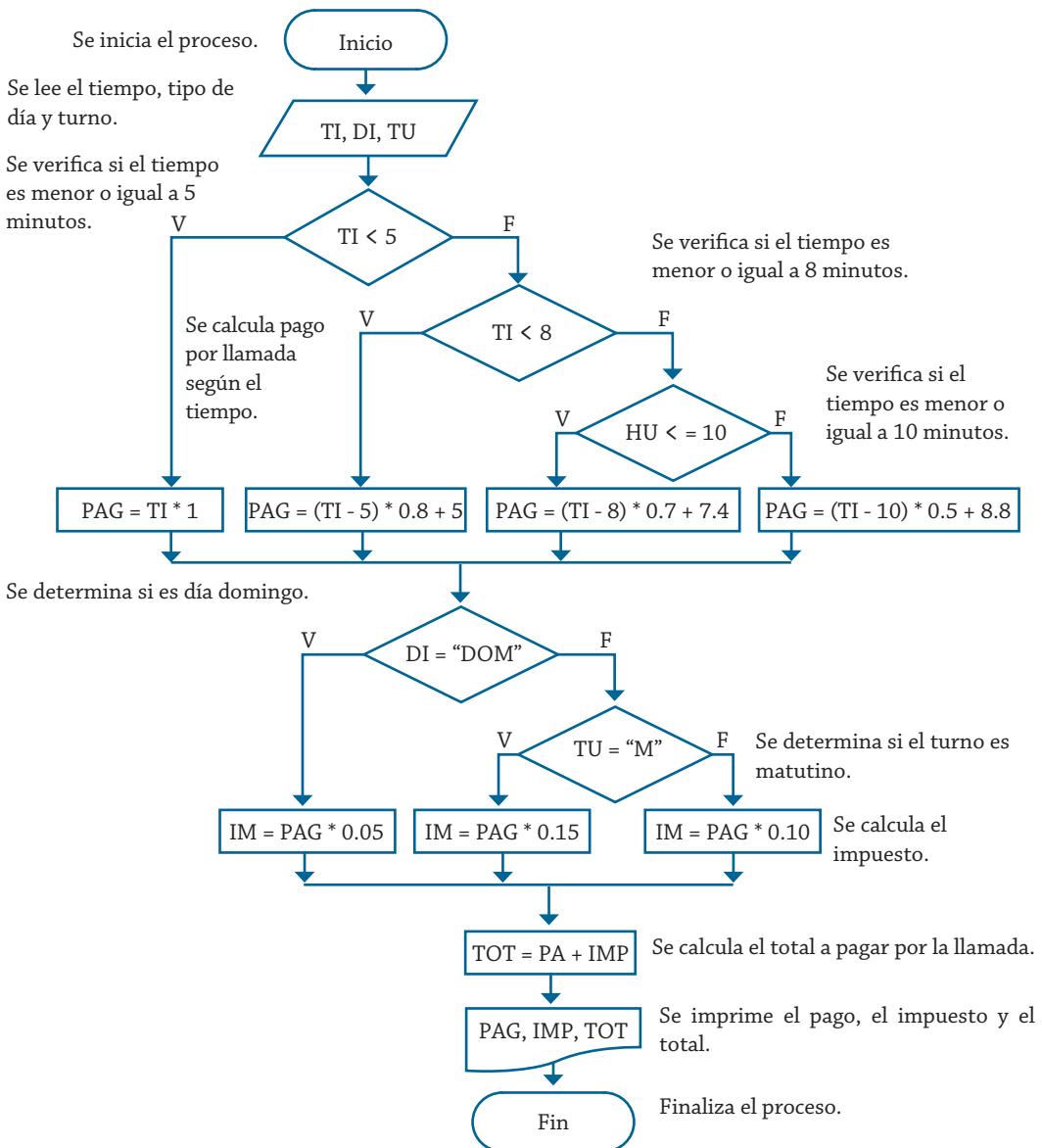


Diagrama de flujo 3.12 Algoritmo para determinar el costo de una llamada telefónica.

Como se puede ver, cuando el tiempo es menor o igual a cinco, el pago se obtiene directamente del producto de tiempo por el costo de un peso, sin embargo, cuando el tiempo es mayor a cinco pero menor o igual a ocho, el cálculo del pago involucra operaciones como la diferencia del tiempo menos cinco, dado que son los primeros cinco minutos los que tienen un costo de cinco pesos, los cuales posteriormente se suman. De igual forma se procede para los otros intervalos de tiempo, donde se le resta el tiempo y se suma lo que se pagó por los minutos previos al rango en cuestión.

La solución propuesta se muestra en el diagrama N/S 3.9 y en el pseudocódigo 3.9 con las respectivas herramientas.

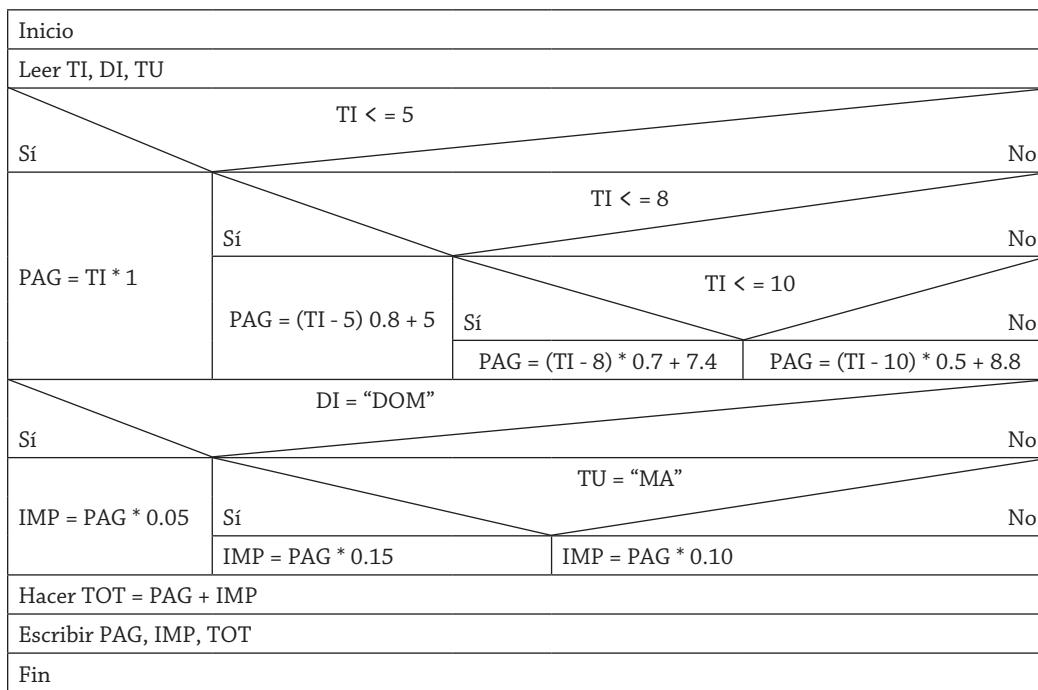


Diagrama N/S 3.9 Algoritmo para determinar el costo de una llamada telefónica.

```

1. Inicio.
2. Leer TI, DI, TU
3. Si TI < = 5
   Entonces
      Hacer PAG = TI * 1
   Si no
      Si TI < = 8
      Entonces
         Hacer PAG = (TI - 5) * 0.8 + 5.0
      Si no
         Si NA < = 10
         Entonces
            Hacer PAG = (TI - 8) * 0.7 + 7.4
         Si no
            Hacer PAG = (TI - 10) * 0.5 + 8.8
         Fin comparar
      Fin de comparar
      Fin comparar
   4. Si DI = "DOM"
      Entonces
         Hacer IM = PAG * 0.05
      Si no
         Si TU = "M"
         Entonces
            Hacer IMP = PAG * 0.15
         Si no
            Hacer IMP = PAG * 0.10
         Fin comparar
         Fin comparar
   5. Hacer TOT = PAG + IMP
   6. Escribir "El pago es", PA
   7. Escribir "El impuesto es", IMP
   8. Escribir "El pago total es", TOT
   9. Fin

```

Pseudocódigo 3.9 Algoritmo para determinar el costo de una llamada telefónica.

Ejemplo 3.10

Una compañía de viajes cuenta con tres tipos de autobuses (A, B y C), cada uno tiene un precio por kilómetro recorrido por persona, los costos respectivos son \$2.0, \$2.5 y \$3.0. Se requiere determinar el costo total y por persona del viaje considerando que cuando éste se presupuesta debe haber un mínimo de 20 personas, de lo contrario el cobro se realiza con base en este número límite.

Con la información correspondiente se puede establecer las variables que se van a utilizar, las cuales se muestran en la tabla 3.10.

Nombre de la variable	Descripción	Tipo
TI	Tipo autobús	String
KM	Kilómetros por recorrer	Entero
NPR	Número de personas real	Entero
CK	Costo por kilómetro	Real
NP	Número de personas para presupuestar	Entero
CP	Costo por persona	Real
TO	Costo total del viaje	Real

Tabla 3.10 Variables utilizadas para determinar el costo del viaje individual y colectivo.

Mientras que el diagrama de flujo 3.13 presenta el algoritmo que permite resolver el problema planteado, el pseudocódigo 3.10 y el diagrama N/S 3.10 presentan la solución correspondiente que permite obtener el costo por persona y el costo que tendrá el viaje en total, mediante cada una de las herramientas.

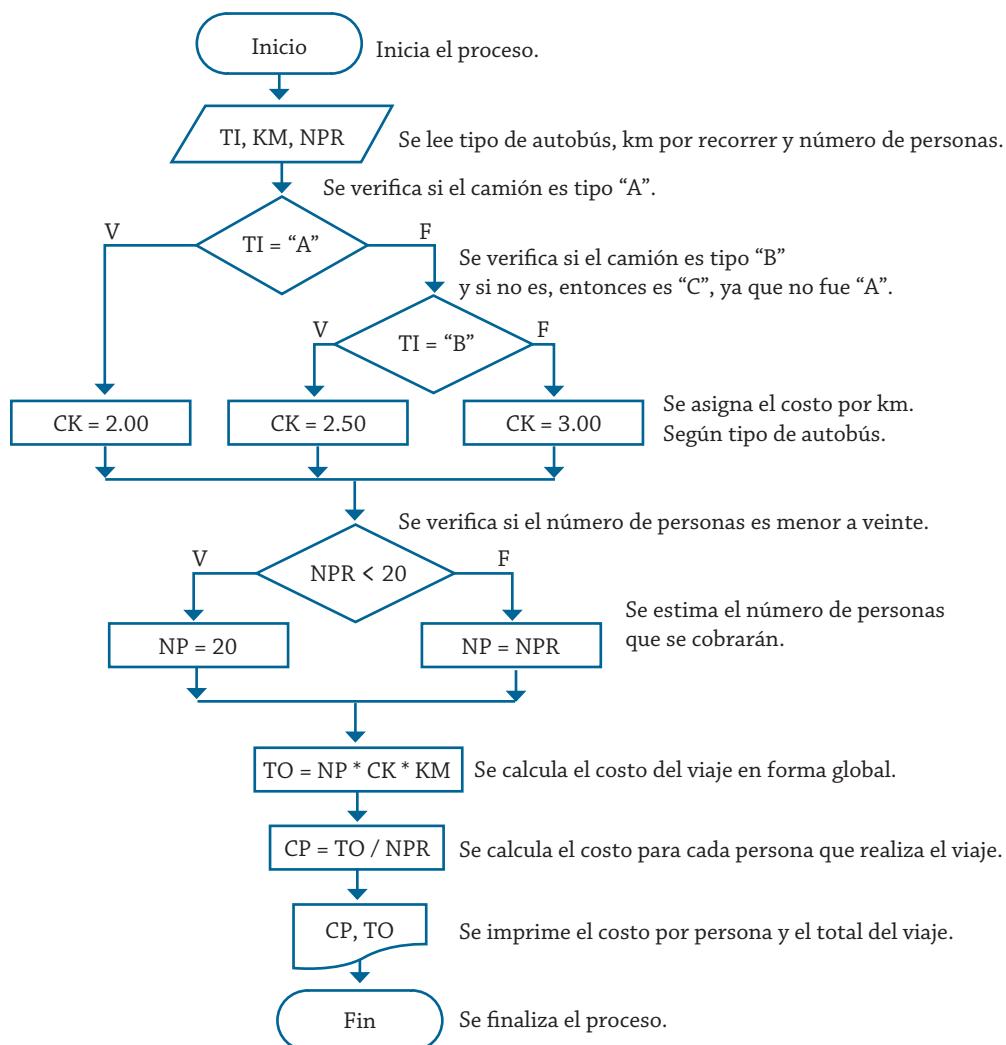


Diagrama de flujo 3.13 Algoritmo para determinar el costo del viaje individual y colectivo.

1. Inicio.
2. Leer TI, KM, NPR
3. Si TI = "A"
 - Entonces
 - Hacer CK = 2.00
 - Si no
 - Si TI = "B"
 - Entonces
 - Hacer CK = 2.50
 - Si no
 - Hacer CK = 3.00
- Fin comparar
- Fin de comparar
4. Si NPR < 20
 - Entonces
 - Hacer NP = 20
 - Si no
 - Hacer NP = NPR
- Fin comparar
5. Hacer TO = NP * CK * KM
6. Hacer PC = TO / NPR
7. Escribir "La persona pagará", CP
8. Escribir "El costo del viaje", TO
9. Fin

Pseudocódigo 3.10. Algoritmo para determinar el costo del viaje individual y colectivo.

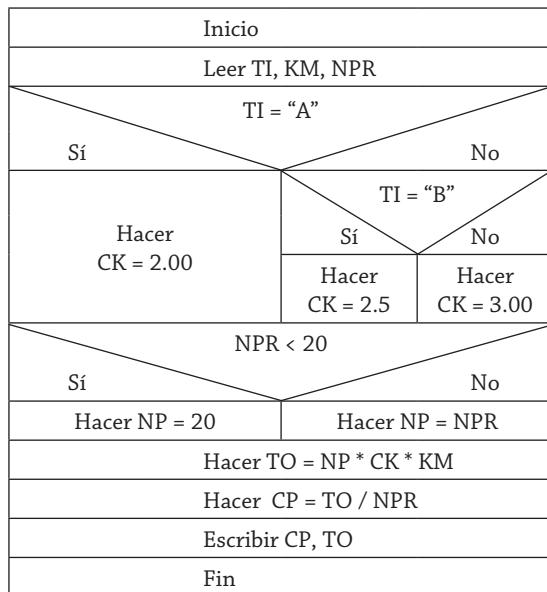


Diagrama N/S 3.10. Algoritmo para determinar el costo del viaje individual y colectivo.

Ejemplo 3.11

“El náufrago satisfecho” ofrece hamburguesas sencillas, dobles y triples, las cuales tienen un costo de \$20.00, \$25.00 y \$28.00 respectivamente. La empresa acepta tarjetas de crédito con un cargo de 5 % sobre la compra. Suponiendo que los clientes adquieren sólo un tipo de hamburguesa, realice un algoritmo para determinar cuánto debe pagar una persona por N hamburguesas. Represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S.

En la tabla 3.11 se muestran las variables que se requieren utilizar en el algoritmo para la solución del problema. El diagrama de flujo 3.14 presenta de forma gráfica ese algoritmo.

Nombre de la variable	Descripción	Tipo
TI	Tipo de hamburguesa	<i>String</i>
N	Número de hamburguesas	Entero
TP	Tipo de pago	<i>String</i>
PA	Precio de la hamburguesa	Real
CA	Cargo por el uso de tarjeta	Real
TO	Total sin cargo	Real
TOT	Total con cargo	Real

Tabla 3.11 Variables utilizadas para determinar el pago por N hamburguesas.

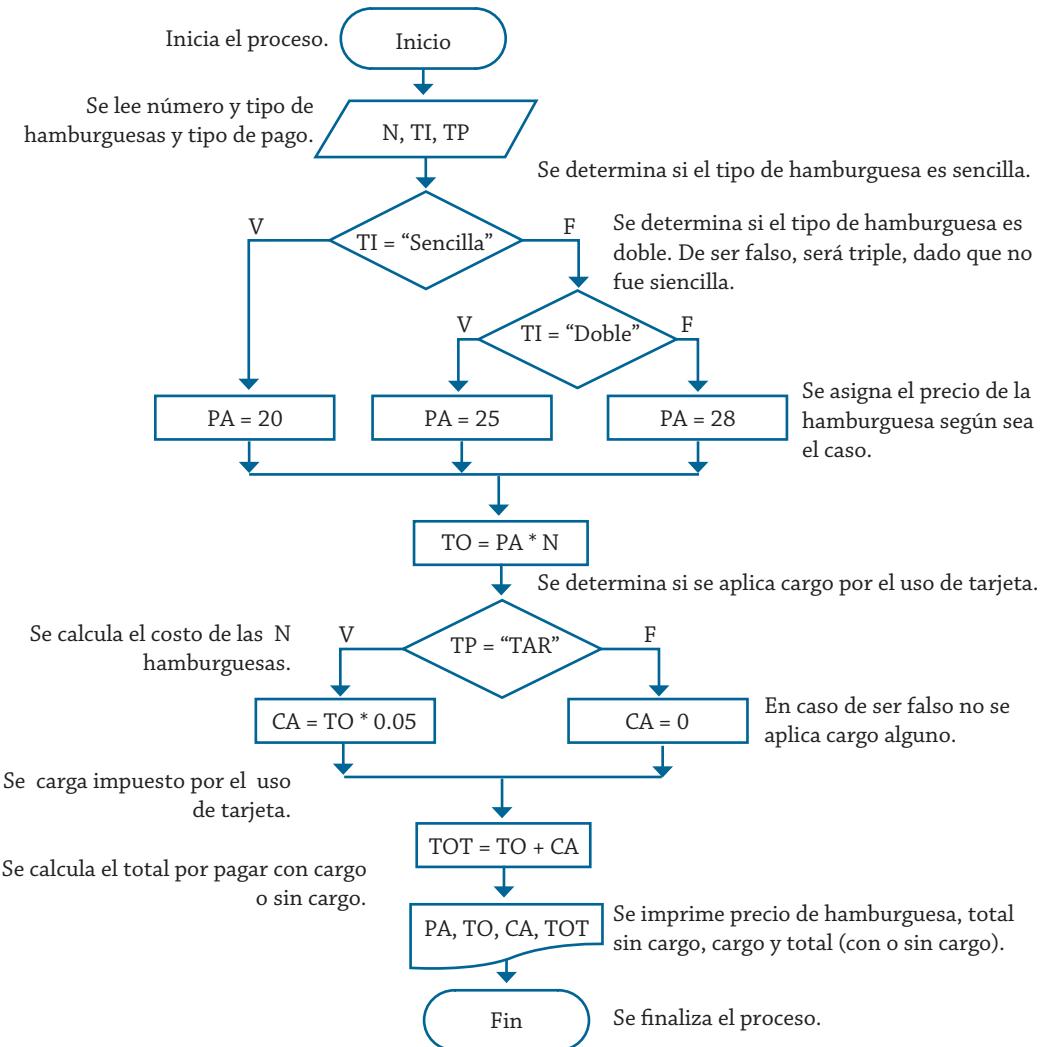


Diagrama de flujo 3.14 Algoritmo para determinar el pago por N hamburguesas.

El pseudocódigo 3.11 y el diagrama N/S 3.11 muestran la representación correspondiente al algoritmo de solución.

```

1. Inicio
2. Leer N, TI, TP
3. Si TI = "Sencilla"
   Entonces
      Hacer PA = 20.00
   Si no
      Si TI = "Doble"
         Entonces
            Hacer PA = 25.00
         Si no
            Hacer PA = 28.00
      Fin comparar
   Fin de comparar
4. Hacer TO = PA * N
5. Si TP = "Tarjeta"
   Entonces
      Hacer CA = TO * 0.05
   Si no
      Hacer CA = 0
   Fin comparar
6. Hacer TOT = TO + CA
7. Escribir "La hamburguesa costo", PA
8. Escribir "El total sin cargo", TO
9. Escribir "El cargo es", CA
10. Escribir "El total por pagar es", TOT
11. Fin

```

Pseudocódigo 3.11 Algoritmo para determinar el pago por N hamburguesas.

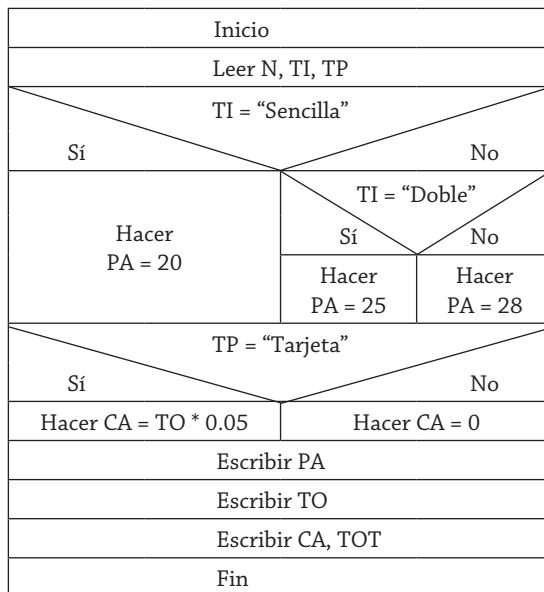


Diagrama N/S 3.11 Algoritmo para determinar el pago por N hamburguesas.

Ejemplo 3.12

El consultorio del Dr. Lorenzo T. Mata Lozano tiene como política cobrar la consulta con base en el número de cita, de la siguiente forma:

- Las tres primeras citas a \$200.00 c/u.
- Las siguientes dos citas a \$150.00 c/u.
- Las tres siguientes citas a \$100.00 c/u.
- Las restantes a \$50.00 c/u, mientras dure el tratamiento.

Se requiere un algoritmo para determinar:

- a) Cuánto pagará el paciente por la cita.
- b) El monto de lo que ha pagado el paciente por el tratamiento.

Para la solución de este problema se requiere saber qué número de cita se efectuará, con el cual se podrá determinar el costo que tendrá la consulta y cuánto se ha gastado en el tratamiento. Con este análisis se puede determinar que las variables que se van a utilizar son las que se muestran en la tabla 3.12.

Nombre de la variable	Descripción	Tipo
NC	Número de consulta	Entero
CC	Costo de la cita	Real
TOT	Costo del tratamiento	Real

Tabla 3.12 Variables utilizadas para determinar el costo de la consulta y del tratamiento.

Con la tabla de variables establecidas previamente, el diagrama de flujo 3.15 que representa el algoritmo de solución para este problema es el siguiente.

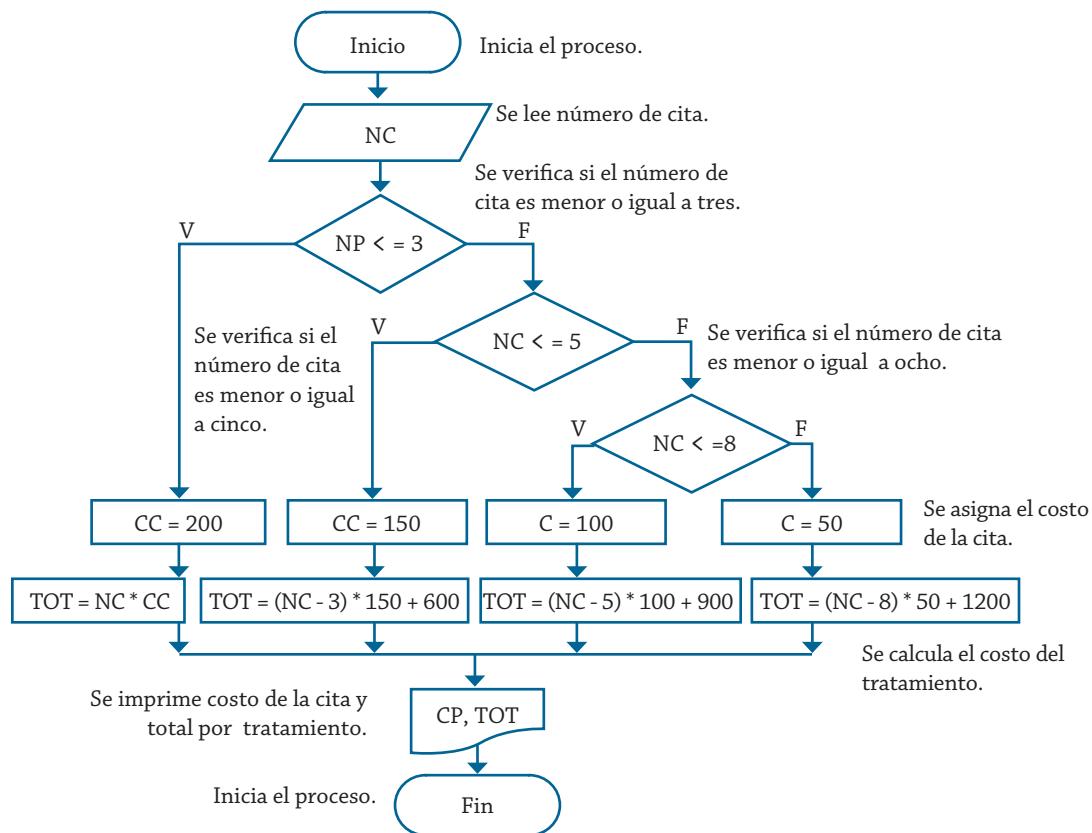


Diagrama de flujo 3.15 Algoritmo para determinar el costo de la consulta y del tratamiento.

Como se puede ver, con base en el número de cita se establece el precio, y según el rango del número de cita, se establece el costo del tratamiento. En cada proceso se le carga un valor constante (600, 900 y 1200), que corresponde a las citas previas, y este número de citas consideradas se restan del número de citas para determinar el monto de las citas en este rango de costo.

Con estas mismas consideraciones, el pseudocódigo 3.12 y el diagrama N/S 3.12 muestran la representación correspondiente al algoritmo de solución.

1. Inicio
2. Leer NC
3. Si $NC <= 3$
 - Entonces
 - Hacer $CC = 200$
 - Hacer $TOT = NC * CC$
 - Si no
 - Si $NC <= 5$
 - Entonces
 - Hacer $CC = 150$
 - Hacer $TOT = (NC - 3) * 150 + 600$
 - Si no
 - Si $NC <= 8$
 - Entonces
 - Hacer $CC = 100$
 - Hacer $TOT = (NC - 5) * 100 + 900$
 - Si no
 - Hacer $CC = 50$
 - Hacer $TOT = (NC - 8) * 50 + 1200$
 - Fin compara
 - Fin compara
 - Fin condición
 - 4. Escribir "El costo de la consulta es", CC
 - 5. Escribir "El costo del tratamiento es", TOT
 - 6. Fin

Pseudocódigo 3.12 Algoritmo para determinar el costo de la consulta y del tratamiento.

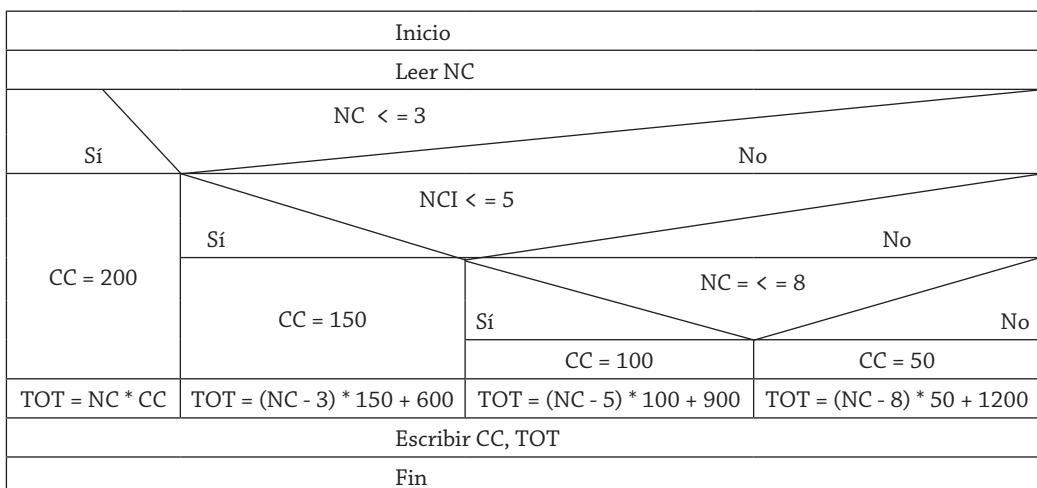


Diagrama N/S 3.12 Algoritmo para determinar el costo de la consulta y del tratamiento

Ejemplo 3.13

Fábricas “El cometa” produce artículos con claves (1, 2, 3, 4, 5 y 6). Se requiere un algoritmo para calcular los precios de venta, para esto hay que considerar lo siguiente:

Costo de producción = materia prima + mano de obra + gastos de fabricación.

Precio de venta = costo de producción + 45 % de costo de producción.

El costo de la mano de obra se obtiene de la siguiente forma: para los productos con clave 3 o 4 se carga 75 % del costo de la materia prima; para los que tienen clave 1 y 5 se carga 80 %, y para los que tienen clave 2 o 6, 85 %.

Para calcular el gasto de fabricación se considera que si el artículo que se va a producir tiene claves 2 o 5, este gasto representa 30 % sobre el costo de la materia prima; si las claves son 3 o 6, representa 35 %; si las claves son 1 o 4, representa 28 %. La materia prima tiene el mismo costo para cualquier clave.

Represente mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S la solución de este problema.

Con las consideraciones anteriores se puede establecer la tabla 3.13 de variables requeridas para el planteamiento del algoritmo correspondiente.

Nombre de la variable	Descripción	Tipo
C	Clave del artículo	Entero
MP	Costo de materia prima	Real
MO	Costo de mano de obra	Real
GF	Gastos de fabricación	Real
CP	Costo de producción	Real
PV	Precio de venta	Real

Tabla 3.13 Variables utilizadas para determinar el precio de venta de un artículo.

Para el planteamiento de la solución de este problema se utilizarán los operadores lógicos (O) o (Y). Cuando se utiliza (O), para que la condición sea verdadera, al menos un valor de los comparados debe ser verdadero; cuando se utiliza (Y), para que la condición sea verdadera, todos los valores comparados deben ser verdaderos. Si en ambos casos no se cumple con esto, la condición será falsa.

Con base en lo anterior, el diagrama de flujo 3.16 muestra el corres-

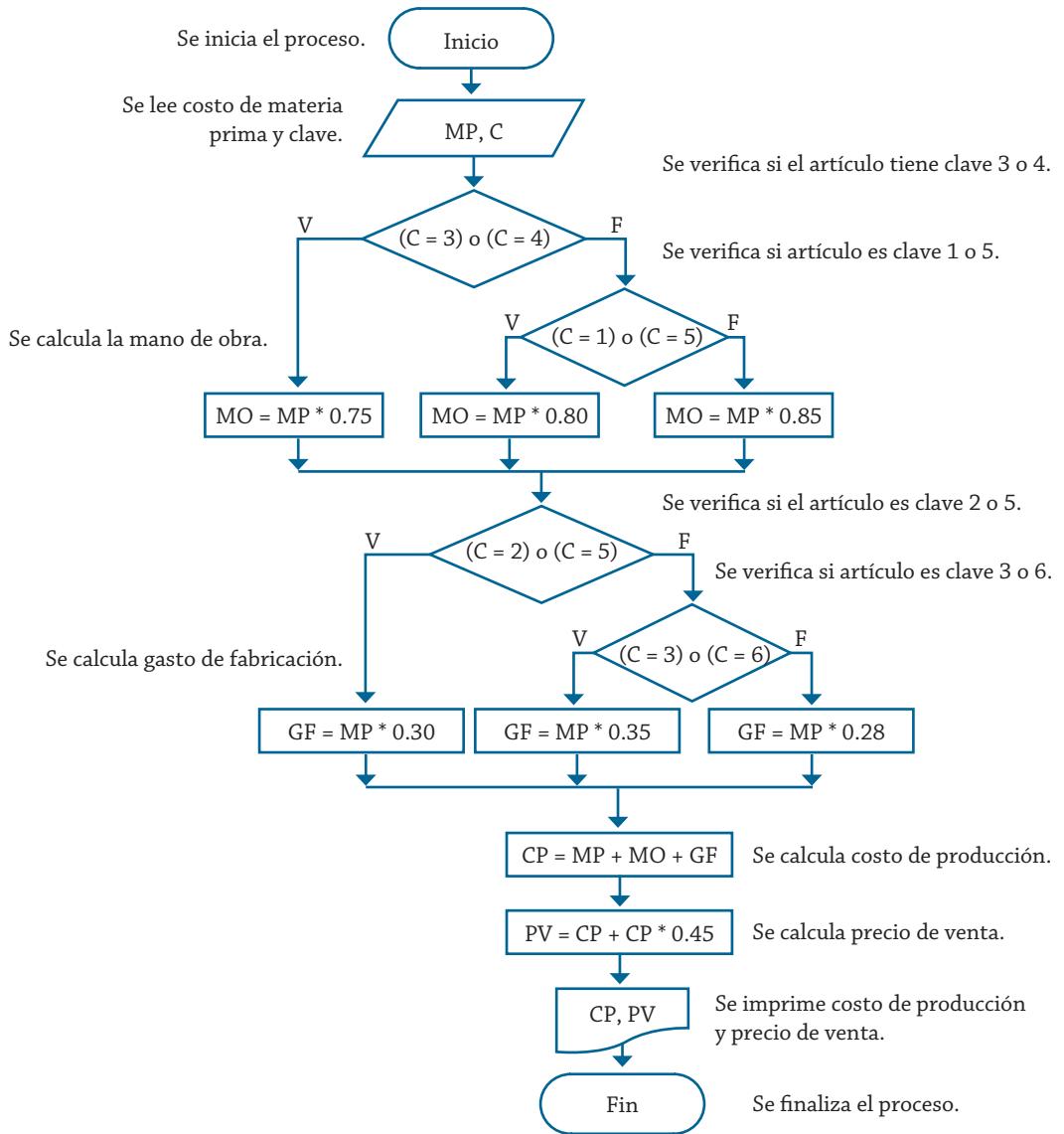


Diagrama de flujo 3.16 Algoritmo para determinar el precio de venta de un artículo.

Cuando se tienen este tipo de situaciones, si no se plantea la solución del problema con la utilización de los operadores lógicos (O) o (Y), se tienen que utilizar más comparaciones, de tal forma que permitan discernir las diferentes alternativas que se puedan presentar para la solución del problema.

El algoritmo representado mediante pseudocódigo no sufre modificación alguna que no se contemple en el diagrama de flujo, de tal forma que éste se muestra en el pseudocódigo 3.13.

```

1. Inicio
2. Leer MP, C
3. Si (C = 3) o (C = 4)
   Entonces
      Hacer MO = MP * 0.75
   Si no
      Si (C = 1) o (C = 5)
      Entonces
         Hacer MO = MP * 0.80
      Si no
         Hacer MO = MP * 0.85
   Fin compara
Fin de compara
4. Si (C = 2) o (C = 5)
   Entonces
      Hacer GF = MP * 0.30
   Si no
      Si (C = 3) o (C = 6)
      Entonces
         Hacer GF = MP * 0.35
      Si no
         Hacer GF = MP * 0.28
   Fin compara
Fin compara
5. Hacer CP = MP + MO + GF
6. Hacer PV = CP + CP * 0.45
7. Escribir "El costo de producción es", CP
8. Escribir "El precio de venta es", PV
9. Fin

```

Pseudocódigo 3.13 Algoritmo para determinar el precio de venta de un artículo.

El diagrama N/S 3.13 muestra el algoritmo correspondiente a esta herramienta. Tiene el mismo grado de complejidad que cualquiera de las otras dos herramientas empleadas previamente; sin embargo, en ocasiones al utilizar los diagramas N/S lo que se complica es organizar la estructura que va resultando al momento de estar implementando el algoritmo, pero con la adquisición de experiencia en la utilización de cada una de estas herramientas, finalmente resulta indiferente utilizarlas, pero sin duda alguna, cada diseñador podrá tener sus preferencias por alguna en especial.

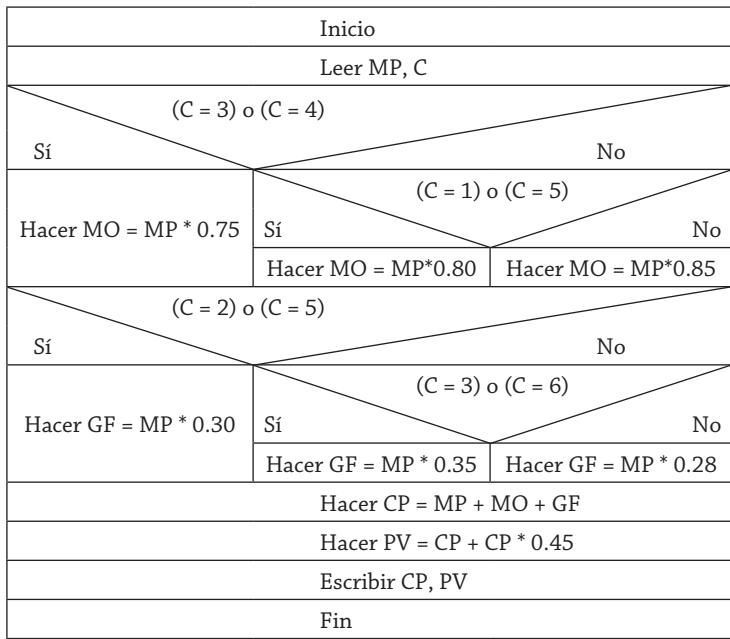


Diagrama N/S 3.13 Algoritmo para determinar el precio de venta de un artículo.

Ejemplo 3.14

Una compañía de paquetería internacional tiene servicio en algunos países de América del Norte, América Central, América del Sur, Europa y Asia. El costo por el servicio de paquetería se basa en el peso del paquete y la zona a la que va dirigido. Lo anterior se muestra en la tabla 3.14:

Zona	Ubicación	Costo/gramo
1	América del Norte	\$11.00
2	América Central	\$10.00
3	América del Sur	\$12.00
4	Europa	\$24.00
5	Asia	\$27.00

Tabla 3.14 Costos por el servicio de paquetería con base en el peso y la zona.

Parte de su política implica que los paquetes con un peso superior a 5 kg no son transportados, esto por cuestiones de logística y de seguridad. Realice un algoritmo para determinar el cobro por la entrega de un paquete o, en su caso, el rechazo de la entrega; represéntelo mediante diagrama de flujo, diagrama N/S y pseudocódigo.

Para la solución de este problema se utilizará el símbolo de decisión múltiple, en los lenguajes de programación la sentencia CASE. Cuando se utiliza esta alternativa se debe considerar que el elemento selector debe ser de tipo ordinal (que sigue un orden estricto, como ejemplo a, b, d, e, c, etcétera; sin embargo, a, c, b no tiene el orden exigido). Para este caso se utiliza el número de zona que es ordinal (1, 2, 3, 4 y 5).

Con esta consideración el diagrama de flujo 3.17 representa el algoritmo correspondiente para obtener el costo que tendrá enviar un paquete

a una zona determinada considerando que si no es del 1 al 4, es 5. Pero de igual forma se puede considerar zona no válida.

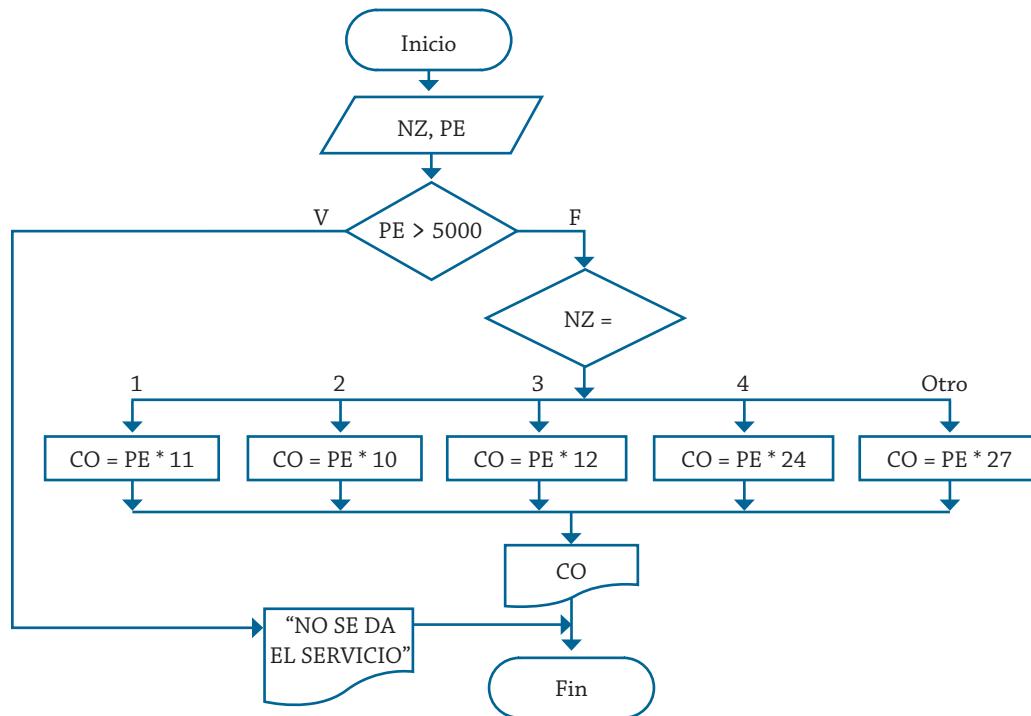


Diagrama de flujo 3.17 Algoritmo para determinar el costo por el servicio de paquetería.

El pseudocódigo 3.14 presenta la forma de escribir el algoritmo que corresponde a la solución del problema. La forma de estructurarlo al momento de pasarlo al lenguaje de programación dependerá básicamente del lenguaje que se utilice, haciendo referencia a dónde ubicar la impresión del costo del servicio, pero básicamente su estructura estaría dada por:

1. Inicio
2. Leer NZ, PE
3. SI PE > 5000
 - Entonces
 - Escribir “No se puede dar el servicio”
 - Si no
 - SI NZ igual a
 - 1: Hacer CO = PE * 11
 - 2: Hacer CO = PE * 10
 - 3: Hacer CO = PE * 12
 - 4: Hacer CO = PE * 24
 - Si no
 - Hacer CO = PE * 27
 - Fin compara
 - Escribir “el costo del servicio es”, CO
 - Fin compara
 4. Fin

Pseudocódigo 3.14 Algoritmo para determinar el costo por el servicio de paquetería.

De igual forma, el diagrama N/S 3.14 se puede representar de la siguiente forma, en la que se considera una estructura selectiva múltiple.

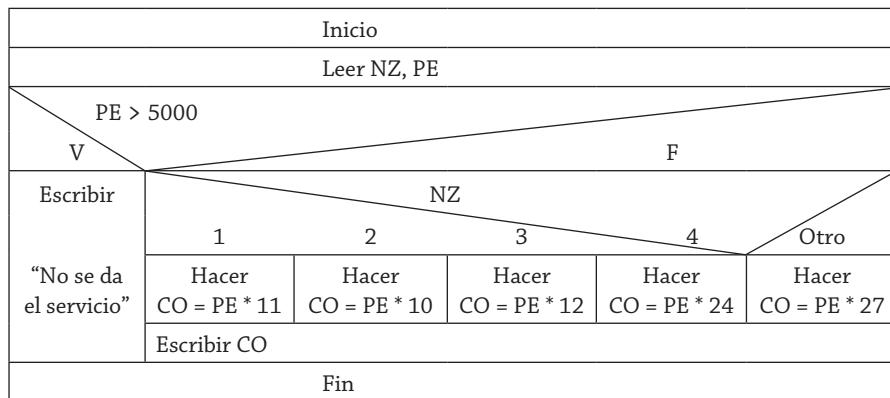


Diagrama N/S 3.14 Algoritmo para determinar el costo por el servicio de paquetería.

Por consiguiente, las variables que se utilizan para la solución de este problema se muestran en la tabla 3.15.

Nombre de la variable	Descripción	Tipo
NZ	Zona donde se dirige el paquete	Entero
PE	Peso del paquete en gramos	Entero
CO	Costo de la entrega	Real

Tabla 3.15 Variables utilizadas para determinar el servicio de paquetería.

Ejemplo 3.15

El banco “Pueblo desconocido” ha decidido aumentar el límite de crédito de las tarjetas de crédito de sus clientes, para esto considera que si su cliente tiene tarjeta tipo 1, el aumento será de 25 %; si tiene tipo 2, será de 35 %; si tiene tipo 3, de 40 %, y para cualquier otro tipo, de 50 %. Ahora bien, si la persona cuenta con más de una tarjeta, sólo se considera la de tipo mayor o la que el cliente indique. Realice un algoritmo y represente su diagrama de flujo y el pseudocódigo para determinar el nuevo límite de crédito que tendrá una persona en su tarjeta.

Nombre de la variable	Descripción	Tipo
TT	Tipo de tarjeta	Entero
LA	Límite actual de crédito	Real
AC	Aumento de crédito	Real
NC	Nuevo límite de crédito	Real

Tabla 3.16 Variables utilizadas para determinar el nuevo límite de crédito.

De igual forma que el problema anterior, la solución de éste se puede plantear con un proceso de solución múltiple, dado que el elemento selector, que es el tipo de tarjeta, es de tipo ordinal; en estas circunstancias el pseudocódigo 3.15 y el diagrama de flujo 3.18 muestran una solución a este problema.

1. Inicio
2. Leer TT, LA

Si NZ Igual a

 - 1: Hacer $AC = LA * 0.25$
 - 2: Hacer $AC = LA * 0.35$
 - 3: Hacer $AC = LA * 0.40$

Si no

Hacer $AC = LA * 0.50$

Fin de comparación
3. Hacer $NC = LA + AC$
4. Escribir “El aumento de crédito”, AC
5. Escribir “Nuevo límite de crédito”, NC
6. Fin

Pseudocódigo 3.15 Algoritmo para determinar el nuevo límite de crédito.

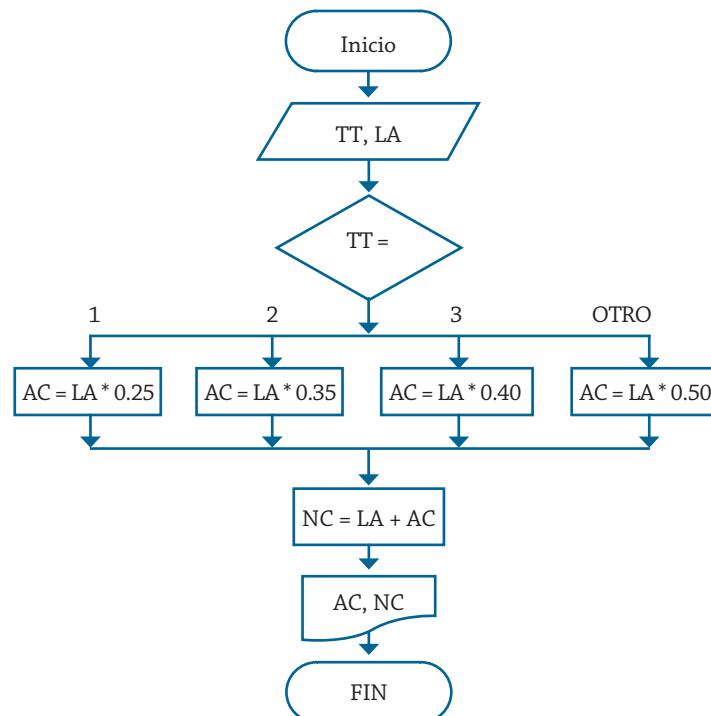


Diagrama de flujo 3.18 Algoritmo para determinar el nuevo límite de crédito.

Problemas propuestos

- 3.1 Realice un algoritmo para determinar si una persona puede votar con base en su edad en las próximas elecciones. Construya el diagrama de flujo, el pseudocódigo y el diagrama N/S.
- 3.2 Realice un algoritmo para determinar el sueldo semanal de un trabajador con base en las horas trabajadas y el pago por hora, consi-

derando que después de las 40 horas cada hora se considera como excedente y se paga el doble. Construya el diagrama de flujo, el pseudocódigo y el diagrama N/S.

- 3.3 El 14 de febrero una persona desea comprarle un regalo al ser querido que más aprecia en ese momento, su dilema radica en qué regalo puede hacerle, las alternativas que tiene son las siguientes:

Regalo	Costo
Tarjeta	\$10.00 o menos
Chocolates	\$11.00 a \$100.00
Flores	\$101.00 a \$250.00
Anillo	Más de \$251.00

Se requiere un diagrama de flujo con el algoritmo que ayude a determinar qué regalo se le puede comprar a ese ser tan especial por el día del amor y la amistad.

- 3.4 El dueño de un estacionamiento requiere un diagrama de flujo con el algoritmo que le permita determinar cuánto debe cobrar por el uso del estacionamiento a sus clientes. Las tarifas que se tienen son las siguientes:

Las dos primeras horas a \$5.00 c/u.

Las siguientes tres a \$4.00 c/u.

Las cinco siguientes a \$3.00 c/u.

Después de diez horas el costo por cada una es de dos pesos.

- 3.5 Se tiene el nombre y la edad de tres personas. Se desea saber el nombre y la edad de la persona de menor edad. Realice el algoritmo correspondiente y represéntelo con un diagrama de flujo, pseudocódigo y diagrama N/S.

- 3.6 Realice el diagrama de flujo, el pseudocódigo y el diagrama N/S que muestren el algoritmo para determinar el costo y el descuento que tendrá un artículo. Considere que si su precio es mayor o igual a \$200 se le aplica un descuento de 15%, y si su precio es mayor a \$100 pero menor a \$200, el descuento es de 12%, y si es menor a \$100, sólo 10%.

- 3.7 El presidente de la república ha decidido estimular a todos los estudiantes de una universidad mediante la asignación de becas mensuales, para esto se tomarán en consideración los siguientes criterios:

Para alumnos mayores de 18 años con promedio mayor o igual a 9, la beca será de \$2000.00; con promedio mayor o igual a 7.5, de \$1000.00; para los promedios menores de 7.5 pero mayores o iguales a 6.0, de \$500.00; a los demás se les enviará una carta de invitación incitándolos a que estudien más en el próximo ciclo escolar.

A los alumnos de 18 años o menores de esta edad, con promedios mayores o iguales a 9, se les dará \$3000; con promedios menores a 9 pero mayores o iguales a 8, \$2000; para los alumnos con promedios menores a 8 pero mayores o iguales a 6, se les dará \$100, y a los alumnos que tengan promedios menores a 6 se les enviará carta de invitación. Realice el algoritmo correspondiente y represéntelo con un diagrama de flujo.

- 3.8 Cierta empresa proporciona un bono mensual a sus trabajadores, el cual puede ser por su antigüedad o bien por el monto de su sueldo (el que sea mayor), de la siguiente forma:
Cuando la antigüedad es mayor a 2 años pero menor a 5, se otorga 20 % de su sueldo; cuando es de 5 años o más, 30 %. Ahora bien, el bono por concepto de sueldo, si éste es menor a \$1000, se da 25 % de éste, cuando éste es mayor a \$1000, pero menor o igual a \$3500, se otorga 15% de su sueldo, para más de \$3500. 10%. Realice el algoritmo correspondiente para calcular los dos tipos de bono, asignando el mayor, y represéntelo con un diagrama de flujo y pseudocódigo.
- 3.9 Una compañía de seguros para autos ofrece dos tipos de póliza: cobertura amplia (A) y daños a terceros (B). Para el plan A, la cuota base es de \$1,200, y para el B, de \$950. A ambos planes se les carga 10% del costo si la persona que conduce tiene por hábito beber alcohol, 5% si utiliza lentes, 5% si padece alguna enfermedad –como deficiencia cardiaca o diabetes–, y si tiene más de 40 años, se le carga 20%, de lo contrario sólo 10%. Todos estos cargos se realizan sobre el costo base. Realice diagrama de flujo y diagrama N/S que represente el algoritmo para determinar cuánto le cuesta a una persona contratar una póliza.
- 3.10 Represente un algoritmo mediante un diagrama de flujo y el pseudocódigo para determinar a qué lugar podrá ir de vacaciones una persona, considerando que la línea de autobuses “La tortuga” cobra por kilómetro recorrido. Se debe considerar el costo del pasaje tanto de ida, como de vuelta; los datos que se conocen y que son fijos son: México, 750 km; P.V., 800 km; Acapulco, 1200 km, y Cancún, 1800 km. También se debe considerar la posibilidad de tener que quedarse en casa.
- 3.11 Se les dará un bono por antigüedad a los empleados de una tienda. Si tienen un año, se les dará \$100; si tienen 2 años, \$200, y así sucesivamente hasta los 5 años. Para los que tengan más de 5, el bono será de \$1000. Realice un algoritmo y represéntelo mediante el diagrama de flujo, el pseudocódigo y diagrama N/S que permita determinar el bono que recibirá un trabajador.
- 3.12 Realice un algoritmo que permita determinar el sueldo semanal de un trabajador con base en las horas trabajadas y el pago por hora, considerando que a partir de la hora número 41 y hasta la 45, cada hora se le paga el doble, de la hora 46 a la 50, el triple, y que trabajar más de 50 horas no está permitido. Represente el algoritmo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S.
- 3.13 Los alumnos de una escuela desean realizar un viaje de estudios, pero requieren determinar cuánto les costará el pasaje, considerando que las tarifas del autobús son las siguientes: si son más de 100 alumnos, el costo es de \$20; si son entre 50 y 100, \$35; entre 20 y 49, \$40, y si son menos de 20 alumnos, \$70 por cada uno. Realice el algoritmo para determinar el costo del pasaje de cada alumno. Represente el algoritmo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S.
- 3.14 Realice un algoritmo que, con base en una calificación proporcionada (0-10), indique con letra la calificación que le corresponde: 10 es “A”, 9 es “B”, 8 es “C”, 7 y 6 son “D”, y de 5 a 0 son “F”. Represente el diagrama de flujo, el pseudocódigo y el diagrama N/S correspondiente.
- 3.15 Realice un algoritmo que, con base en un número proporcionado (1-7), indique el día de la semana que le corresponde (L-D). Re-

presente el diagrama de flujo, el pseudocódigo y el diagrama N/S correspondiente.

- 3.16 El secretario de educación ha decidido otorgar un bono por desempeño a todos los profesores con base en la puntuación siguiente:

Puntos	Premio
0 - 100	1 salario
101 - 150	2 salarios mínimos
151 - en adelante	3 salarios mínimos

Realice un algoritmo que permita determinar el monto de bono que percibirá un profesor (debe capturar el valor del salario mínimo y los puntos del profesor). Represente el algoritmo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S.

- 3.17 Realice un algoritmo y represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S que permitan determinar qué paquete se puede comprar una persona con el dinero que recibirá en diciembre, considerando lo siguiente:

- Paquete A. Si recibe \$50,000 o más se comprará una televisión, un modular, tres pares de zapatos, cinco camisas y cinco pantalones.
- Paquete B. Si recibe menos de \$50,000 pero más (o igual) de \$20,000, se comprará una grabadora, tres pares de zapatos, cinco camisas y cinco pantalones.
- Paquete C. Si recibe menos de \$20,000 pero más (o igual) de \$10,000, se comprará dos pares de zapatos, tres camisas y tres pantalones.
- Paquete D. Si recibe menos de \$10,000, se tendrá que conformar con un par de zapatos, dos camisas y dos pantalones.

- 3.18 Realice un algoritmo y represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S que permitan determinar la cantidad del bono navideño que recibirá un empleado de una tienda, considerando que si su antigüedad es mayor a cuatro años o su sueldo es menor de dos mil pesos, le corresponderá 25 % de su sueldo, y en caso contrario sólo le corresponderá 20 % de éste.

- 3.19 La secretaría de salud requiere un diagrama de flujo que le represente el algoritmo que permite determinar qué tipo de vacuna (A, B o C) debe aplicar a una persona, considerando que si es mayor de 70 años, sin importar el sexo, se le aplica la tipo C; si tiene entre 16 y 69 años, y es mujer, se le aplica la B, y si es hombre, la A; si es menor de 16 años, se le aplica la tipo A, sin importar el sexo.

- 3.20 Realice un algoritmo para resolver el siguiente problema: una fábrica de pantalones desea calcular cuál es el precio final de venta y cuánto ganará por los N pantalones que produzca con el corte de alguno de sus modelos, para esto se cuenta con la siguiente información:

- a) Tiene dos modelos A y B, tallas 30, 32 y 36 para ambos modelos.
- b) Para el modelo A se utiliza 1.50 m de tela, y para el B 1.80 m.
- c) Al modelo A se le carga 80 % del costo de la tela, por mano de obra. Al modelo B se le carga 95 % del costo de la tela, por el mismo concepto.

- d) A las tallas 32 y 36 se les carga 4 % del costo generado por mano de obra y tela, sin importar el modelo.
 - e) Cuando se realiza el corte para fabricar una prenda sólo se hace de un solo modelo y una sola talla.
 - f) Finalmente, a la suma de estos costos se les carga 30%, que representa la ganancia extra de la tienda.
- 3.21 El banco “Bandido de peluche” desea calcular para uno de sus clientes el saldo actual, el pago mínimo y el pago para no generar intereses. Los datos que se conocen son: saldo anterior del cliente, monto de las compras que realizó y el pago que depositó en el corte anterior. Para calcular el pago mínimo se debe considerar 15% del saldo actual, y para no generar intereses corresponde 85% del saldo actual, considerando que este saldo debe incluir 12% de los intereses causados por no realizar el pago mínimo y \$200 por multa por el mismo motivo. Realice el algoritmo correspondiente y represéntelo mediante el diagrama de flujo y pseudocódigo.

UNIDAD IV

SOLUCIÓN DE PROBLEMAS

CON ESTRUCTURAS REPETITIVAS

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

Introducción

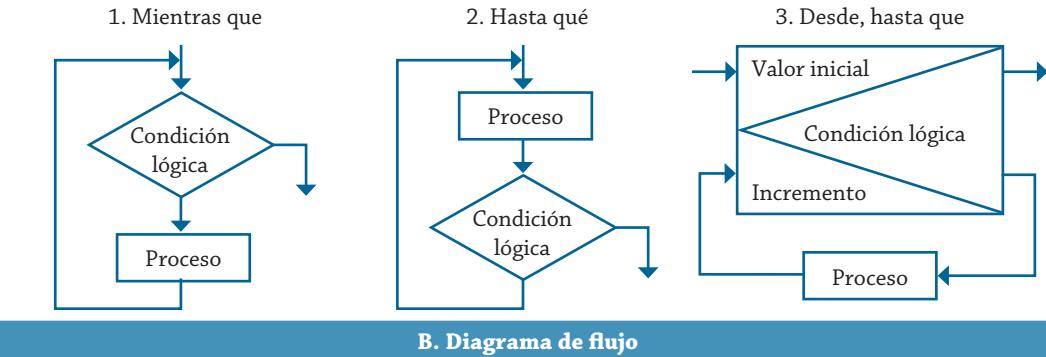
Como se ha podido observar hasta el momento, las soluciones planteadas a los problemas propuestos han sido para una persona, un objeto o cosa, pero siempre de manera unitaria, tanto en las soluciones que se plantearon con estructuras secuenciales como con las decisivas; sin embargo, debemos considerar que cuando se plantean problemas como calcular un sueldo cabe la posibilidad de que el cálculo se tenga que hacer para dos o más empleados, un proceso de cálculo que por lógica debe ser el mismo para cada uno, pero donde existe la posibilidad de que los parámetros que determinan ese sueldo sean los que cambien.

También se puede considerar el caso del cobro de una llamada realizada por una persona, pero también puede ser que se considere el cobro de N llamadas efectuadas por la misma persona, donde lo que puede cambiar es el tiempo, o la tarifa, que puede depender de alguna condición. De igual forma se pueden presentar muchos casos donde el proceso se debe repetir varias veces. Por tal motivo se emplean estructuras denominadas repetitivas, de ciclo o de bucle, e independientemente del nombre que se les aplique, lo que importa es que permiten que un proceso pueda realizarse N veces, donde sólo cambien los parámetros que se utilizan en el proceso.

Estructuras repetitivas o de ciclo

Cuando se requiere que un proceso se efectúe de manera cíclica, se emplean estructuras que permiten el control de ciclos, esas estructuras se emplean con base en las condiciones propias de cada problema, los nombres con los que se conocen éstas son: "Mientras que", "Repite hasta que" y "Desde, hasta que". En la figura 4.1 se presentan las formas de estas estructuras mediante un diagrama de flujo y el pseudocódigo correspondiente.

Para el caso de la estructura "Mientras que", el ciclo se repite hasta que la condición lógica resulta ser falsa; en tanto que en la estructura "Hasta que", el ciclo se repite siempre y cuando el resultado de la condición lógica sea falso; además, como se puede ver en la figura 4.1, en la estructura "Mientras que" primero se evalúa y luego se realiza el proceso; y para el caso de "Hasta que", primero se realiza el proceso y luego se evalúa, por consiguiente este tipo de estructura siempre realizará por lo menos un proceso.



B. Diagrama de flujo

Mientras Condición lógica

Proceso

Sin Mientras

Repite

Proceso

Hasta Condición lógica

Desde valor inicial **Hasta** valor Final

Proceso

Fin Desde

A. Pseudocódigo

Figura 4.1 Estructuras de control de ciclos.

Las estructuras de tipo “Desde” se aplican cuando se tiene definido el número de veces que se realizará el proceso dentro del ciclo, lo que la hace diferente de las otras es que aquellas se pueden utilizar hasta que las condiciones cambien dentro del mismo ciclo, estas condiciones pueden deberse a un dato proporcionado desde el exterior, o bien, al resultado de un proceso ejecutado dentro del mismo, el cual marca el final. Además, en el ciclo “Desde”, su incremento es automático, por lo cual no se tiene que efectuar mediante un proceso adicional, como en los otros dos tipos.

En los siguientes ejemplos se mostrará la aplicación de los tres tipos de ciclos antes mencionados.

Ejemplo 4.1

Se requiere un algoritmo para obtener la suma de diez cantidades mediante la utilización de un ciclo “Mientras”. Realice el diagrama de flujo, el pseudocódigo y el diagrama N/S para representarlo.

Con base en lo que se requiere determinar se puede establecer que las variables requeridas para la solución del problema son las mostradas en la tabla 4.1.

Nombre de la variable	Descripción	Tipo
C	Contador	Entero
VA	Valor por sumar	Real
SU	Suma de los valores	Real

Tabla 4.1 Variables utilizadas para obtener la suma de diez cantidades.

La solución de este problema mediante el ciclo Mientras, que también es conocido como ciclo While en los diferentes lenguajes de programación, se puede establecer mediante el diagrama de flujo 4.1

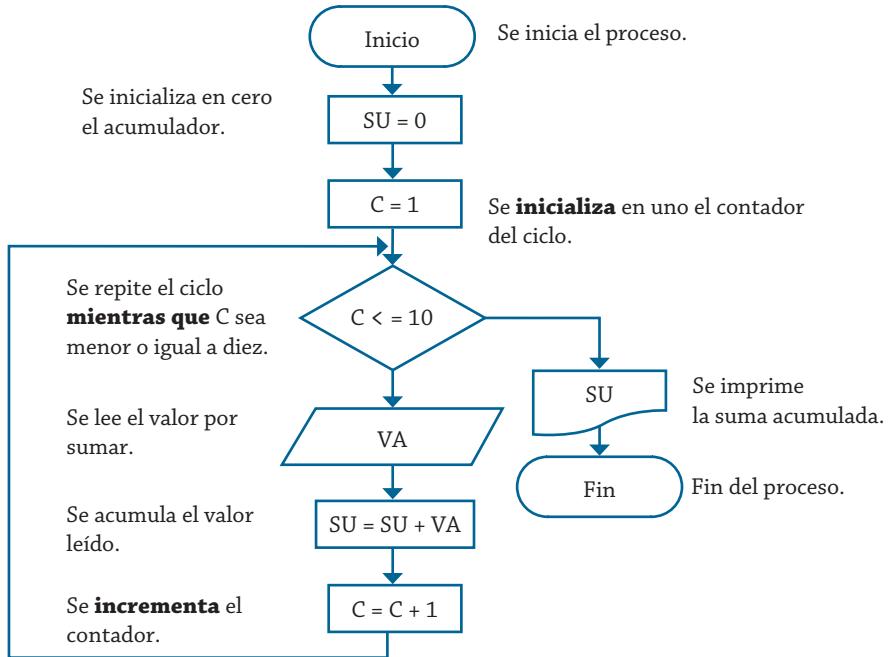


Diagrama de flujo 4.1 Algoritmo para obtener la suma de diez cantidades con ciclo Mientras.

De esta solución planteada se puede ver, primeramente, que el contador del ciclo “C” se inicializa en uno, posteriormente se verifica que éste sea menor o igual a diez, que es lo que debe durar el ciclo (diez veces), ya dentro del ciclo el contador se incrementa por cada vuelta que dé y se realice el proceso de leer un valor y acumularlo en la suma.

En general, todo ciclo debe tener un valor inicial, un incremento y un verificador que establezca el límite de ejecución (inicializa, incrementa, “mientras que”).

El pseudocódigo 4.1 y el diagrama N/S 4.1 presentan el algoritmo correspondiente de la solución de este problema mediante la utilización de estas herramientas.

1. Inicio
2. Hacer $SU = 0$
3. Hacer $C = 1$
4. Mientras $C <= 10$
 - Leer VA
 - Hacer $SU = SU + VA$
 - Hacer $C = C + 1$
5. Escribir SU
6. Fin

Pseudocódigo 4.1 Algoritmo para obtener la suma de diez cantidades con ciclo Mientras.

Inicio
Hacer $SU = 0$
Hacer $C = 1$
Mientras $C \leq 10$
Leer VA
Hacer $SU = SU + VA$
Hacer $C = C + 1$
Fin mientras
Escribir SU
Fin

Diagrama N/S 4.1 Algoritmo para obtener la suma de diez cantidades con ciclo Mientras.

Ejemplo 4.2

Se requiere un algoritmo para obtener la suma de diez cantidades mediante la utilización de un ciclo Repite. Realice el diagrama de flujo, el pseudocódigo y diagrama N/S para representarlo.

La solución de este problema mediante el ciclo Repite, que también es conocido como ciclo Repeat en los diferentes lenguajes de programación, se puede establecer mediante el diagrama de flujo 4.2.

Las variables que se requieren son las que se muestran en la tabla 4.1 (el ejemplo anterior y el presente son el mismo, lo que cambia es el planteamiento de solución del problema en lo que respecta al tipo de ciclo por utilizar).

El diagrama de flujo 4.2 muestra la solución correspondiente mediante la utilización de este tipo de ciclo.

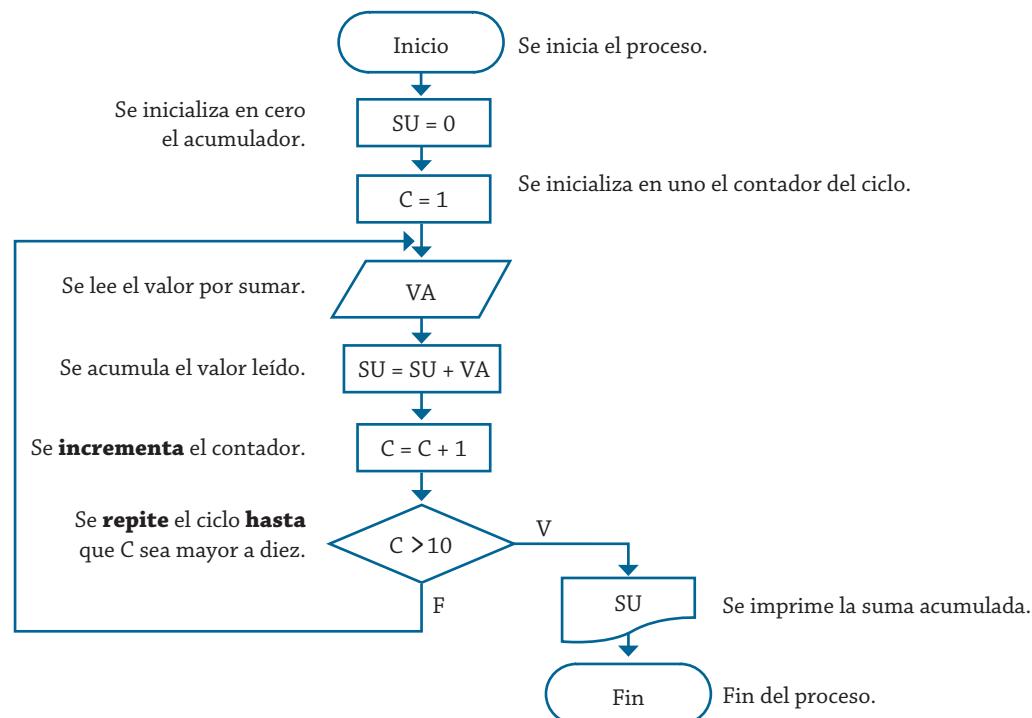


Diagrama de flujo 4.2 Algoritmo para obtener la suma de diez cantidades con ciclo Repite.

El pseudocódigo 4.2 y el diagrama N/S 4.2 presentan el algoritmo correspondiente de la solución de este problema mediante la utilización de estas herramientas.

1. Inicio
2. Hacer $SU = 0$
3. Hacer $C = 1$
4. Repite
 Leer VA
 Hacer $SU = SU + VA$
 Hacer $C = C + 1$
 Hasta que $C > 10$
5. Escribir SU
6. Fin

Pseudocódigo 4.2 Algoritmo para obtener la suma de diez cantidades con ciclo Repite.

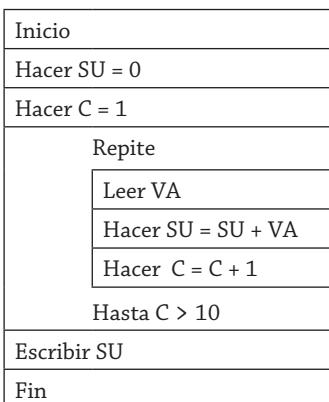


Diagrama N/S 4.2 Algoritmo para obtener la suma de diez cantidades con ciclo Repite.Repite.

Como se puede ver, el ciclo tiene un valor inicial, un incremento y un verificador, el cual establece el límite de ejecución, tal y como se tiene para el ciclo Mientras, mostrado en el ejemplo anterior; si se compara la solución planteada en el ejemplo anterior con ésta, se podrá observar que cuando se emplea el ciclo Mientras, primero se evalúa la terminación del ciclo y posteriormente se realiza el proceso, y en este caso, primero se ejecuta el proceso y posteriormente se evalúa la terminación del ciclo.

Ejemplo 4.3

Se requiere un algoritmo para obtener la suma de diez cantidades mediante la utilización de un ciclo Desde. Realice el diagrama de flujo, el pseudocódigo y el diagrama N/S para representarlo.

El ciclo Desde también es conocido como ciclo For en los diferentes lenguajes de programación. Se utilizarán las mismas variables mostradas en la tabla 4.1. El diagrama de flujo 4.3 muestra la solución correspondiente utilizando el ciclo Desde.

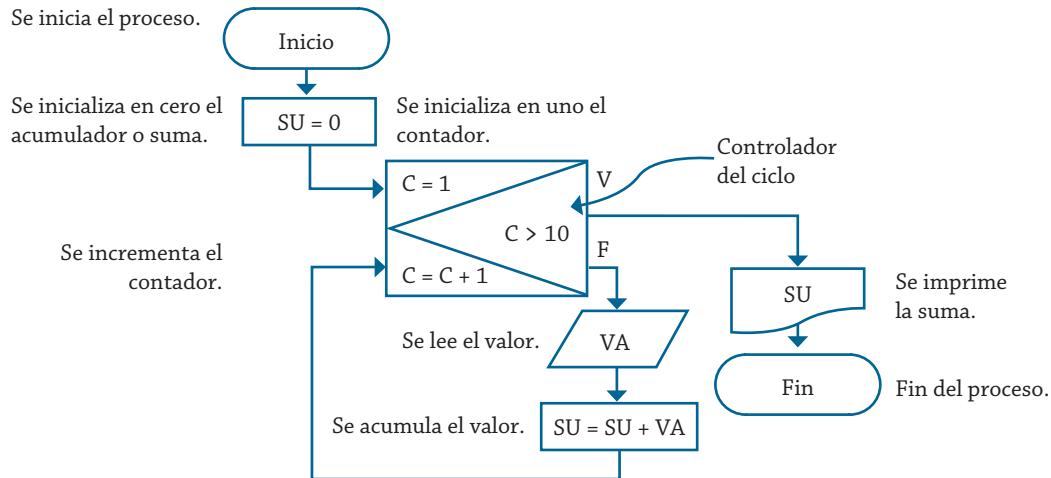


Diagrama de flujo 4.3 Algoritmo para obtener la suma de diez cantidades con ciclo Desde.

El pseudocódigo 4.3 y el diagrama N/S 4.3 presentan el algoritmo correspondiente de la solución de este problema mediante la utilización de estas herramientas.

1. Inicio
2. Hacer $SU = 0$
3. Desde $C = 1$ hasta $C = 10$
 - Leer VA
 - Hacer $SU = SU + VA$
 - Fin desde
4. Escribir SU
5. Fin

Pseudocódigo 4.3 Algoritmo para obtener la suma de diez cantidades con ciclo Desde.

Inicio
Hacer $SU = 0$
Desde $C = 1$ hasta 10
Leer VA
Hacer $SU = SU + VA$
Fin desde
Escribir SU
Fin

Diagrama N/S 4.3 Algoritmo para obtener la suma de diez cantidades con ciclo Desde.

Como se mencionó, este tipo de estructura para el control de ciclos se utiliza exclusivamente cuando el número de veces que se realizará el ciclo está bien definido; sin embargo, como se pudo ver en los dos ejemplos anteriores, esta condición está presente y no fue un impedimento para utilizarlos en la solución del problema.

Además, se debe observar que el incremento de la variable que controla el ciclo no se indica en este tipo de estructura, ya que el incremento o decremento de la variable se realiza de manera automática; cuando el caso es decremento la forma de indicarlo dependerá del lenguaje de programación que se esté utilizando.

Ejemplo 4.4

Se requiere un algoritmo para obtener la edad promedio de un grupo de N alumnos. Realice el diagrama de flujo, el pseudocódigo y el diagrama N/S para representarlo, utilizando los tres tipos de estructuras de ciclo.

La tabla 4.2 muestra las variables que se van a utilizar para la solución del problema, sin importar qué estructura de ciclo se utilice; por consiguiente, es la misma para los tres tipos de ciclo para los que se dará la solución.

Nombre de la variable	Descripción	Tipo
C	Contador	Entero
ED	Edad de cada alumno	Entero
SU	Suma de las edades	Entero
NU	Número de alumnos	Entero
PR	Edad promedio	Real

Tabla 4.2 Variables utilizadas para obtener la edad promedio de N alumnos.

El diagrama de flujo 4.4, el pseudocódigo 4.4 y el diagrama N/S 4.4 muestran el algoritmo de solución mediante la utilización de un ciclo Mientras.

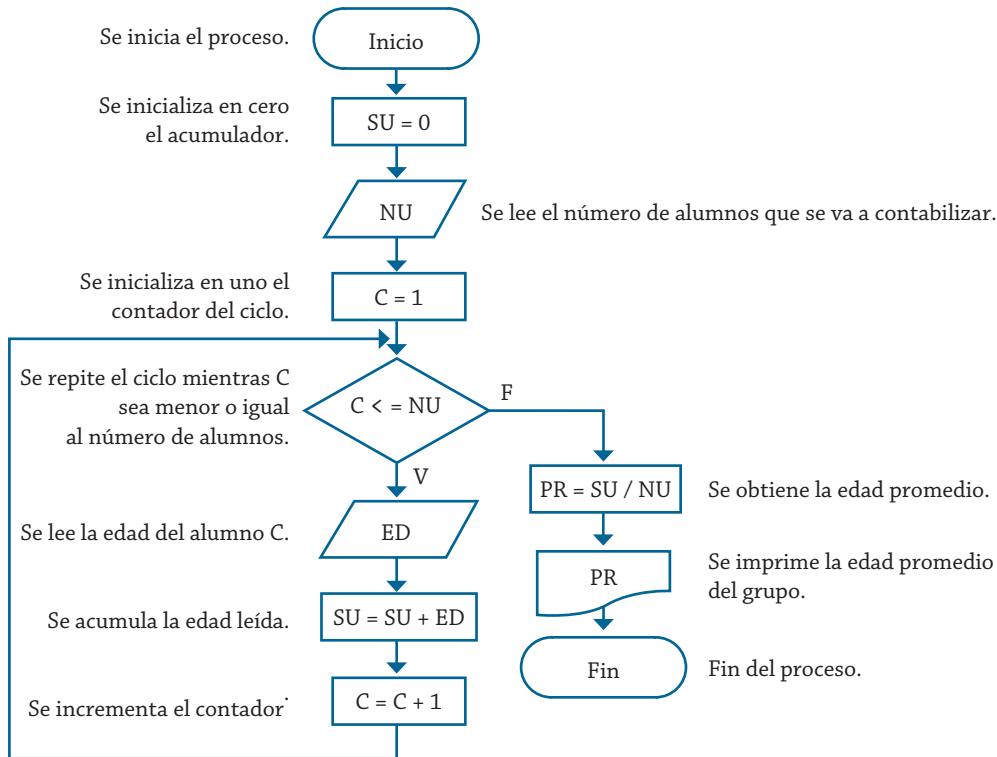


Diagrama de flujo 4.4 Algoritmo para obtener la edad promedio de N alumnos con ciclo Mientras.

1. Inicio
2. Hacer $SU = 0$
3. Leer NU
4. Hacer $C = 1$
5. Mientras $C \leq NU$
 - Leer ED
 - Hacer $SU = SU + ED$
 - Hacer $C = C + 1$
6. Hacer $PR = SU / NU$
7. Escribir PR
8. Fin

Pseudocódigo 4.4 Algoritmo para obtener la edad promedio de N alumnos con ciclo Mientras.

Inicio
Hacer SU = 0
Leer NU
Hacer C = 1
Mientras C < = 10
Leer ED
Hacer SU = SU + ED
Hacer C = C + 1
Fin mientras
Hacer PR = SU / NU
Escribir PR
Fin

Diagrama N/S 4.4 Algoritmo para obtener la edad promedio de N alumnos con ciclo Mientras.

El pseudocódigo 4.5, el diagrama de flujo 4.5 y el diagrama N/S 4.5 muestran el algoritmo de solución mediante la utilización de un ciclo Repite.

1. Inicio
2. Hacer SU = 0
3. Leer NU
4. Hacer C = 1
5. Repite
 - Leer ED
 - Hacer SU = SU + ED
 - Hacer C = C + 1
 - Hasta C > NU
6. Hacer PR = SU / NU
7. Escribir PR
8. Fin

Pseudocódigo 4.5 Algoritmo para obtener la edad promedio de N alumnos con ciclo Repite.

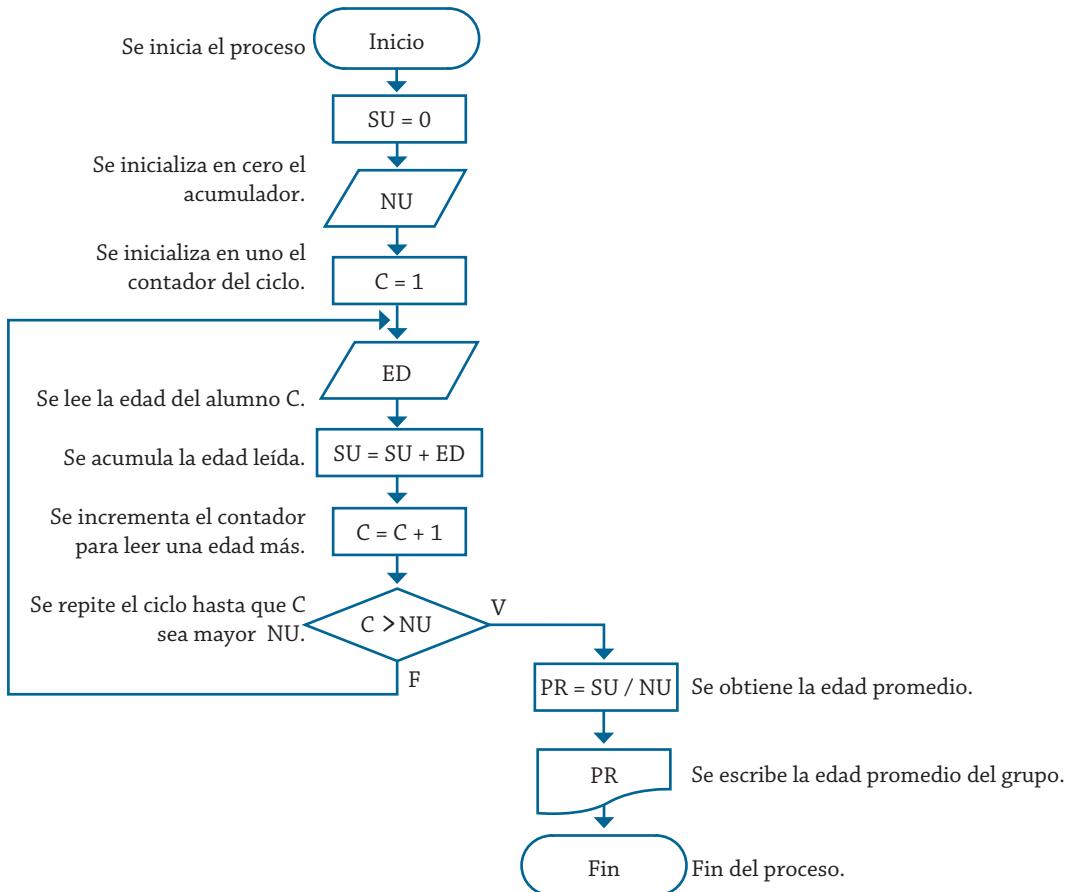


Diagrama de flujo 4.5 Algoritmo para obtener la edad promedio de N alumnos con ciclo Repite.

Inicio
Hacer $SU = 0$
Ler NU
Hacer $C = 1$
Repite
Ler ED
Hacer $SU = SU + ED$
Hacer $C = C + 1$
Hasta $C > NU$
Hacer $PR = SU / NU$
Escribir PR
Fin

Diagrama N/S 4.5 Algoritmo para obtener la edad promedio de N alumnos con ciclo Repite.

Para plantear la solución mediante un ciclo Repite se partió del entendido de que se contaría con la edad de una persona por lo menos.

El diagrama de flujo 4.6, el pseudocódigo 4.6 y el diagrama N/S 4.6 muestran el algoritmo de solución mediante la utilización de un ciclo Desde.

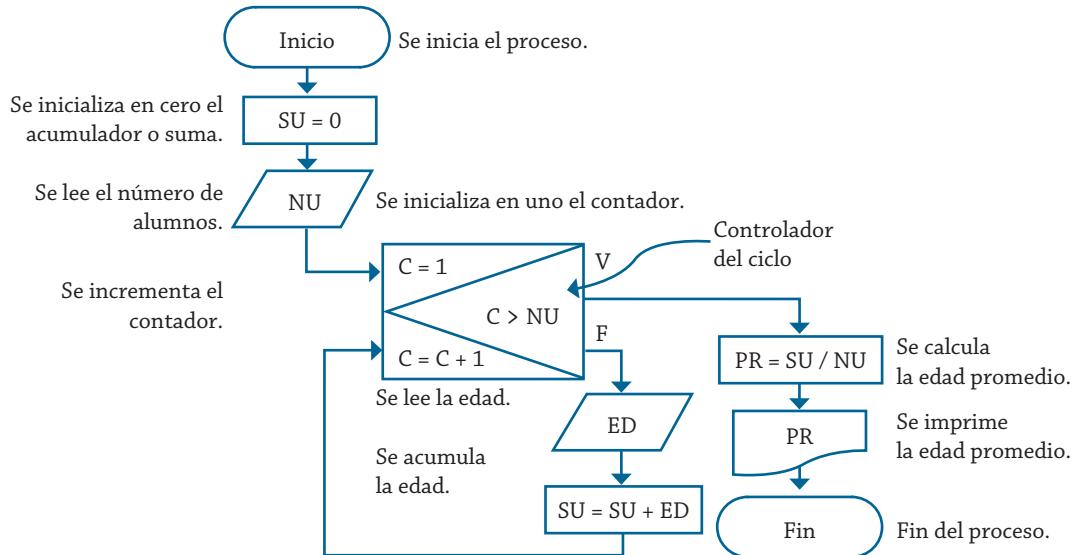


Diagrama de flujo 4.6 Algoritmo para obtener la edad promedio de N alumnos con ciclo Desde.

1. Inicio
2. Hacer $SU = 0$
3. Leer NU
4. Desde $C = 1$ hasta $C = NU$
 - Leer ED
 - Hacer $SU = SU + ED$
 - Fin desde
5. Hacer $PR = SU / NU$
6. Escribir PR
7. Fin

Pseudocódigo 4.6 Algoritmo para obtener la edad promedio de N alumnos con ciclo Desde.

Iniciar
Hacer $SU = 0$
Leer NU
Desde $C = 1$ hasta NU
Leer ED
Hacer $SU = SU + ED$
Fin desde
Hacer $PR = SU / NU$
Escribir PR
Fin

Diagrama N/S 4.6 Algoritmo para obtener la edad promedio de N alumnos con ciclo Desde.

No se debe perder de vista que para plantear la solución de este problema con ciclo Desde, se partió de que se conocía el número de personas que se les tomaría su edad.

Ejemplo 4.5

Se requiere un algoritmo para obtener la estatura promedio de un grupo de personas, cuyo número de miembros se desconoce, el ciclo debe efectuarse siempre y cuando se tenga una estatura registrada. Realice el diagrama de flujo, el pseudocódigo y el diagrama N/S para representarlo, utilizando el ciclo apropiado.

Como se puede ver, para resolver este problema no se puede utilizar el ciclo Desde, ya que no se tiene el número de personas exacto, que es lo que en un momento determinaría el número de veces que el proceso que se encuentra dentro del ciclo se ejecute, para este caso es necesario contar al menos con la estatura de una persona (para que tenga caso realizar el proceso del ciclo). Por otro lado, si se utiliza el ciclo Repite, se ejecutará por lo menos una vez y hasta que se le proporcione una estatura menor o igual a cero, por tal motivo no es muy conveniente utilizarlo, ya que se debe tener al menos una estatura para realizar lo que se pretende con el algoritmo.

El ciclo que es apropiado para utilizar en la solución de este problema es Mientras, ya que este ciclo se realiza siempre y cuando se cuente con una estatura mayor a cero, de una manera natural sin forzar el proceso en ningún momento, y en caso de que no se tenga estatura registrada el promedio es cero, y se debe indicar que no existe ninguna estatura registrada.

La tabla 4.3 muestra las variables que se van a utilizar para la solución de este problema. La representación del algoritmo para este problema se presenta mediante el diagrama de flujo 4.7, el pseudocódigo 4.7 y el diagrama N/S 4.7, en los cuales se utiliza el ciclo Mientras.

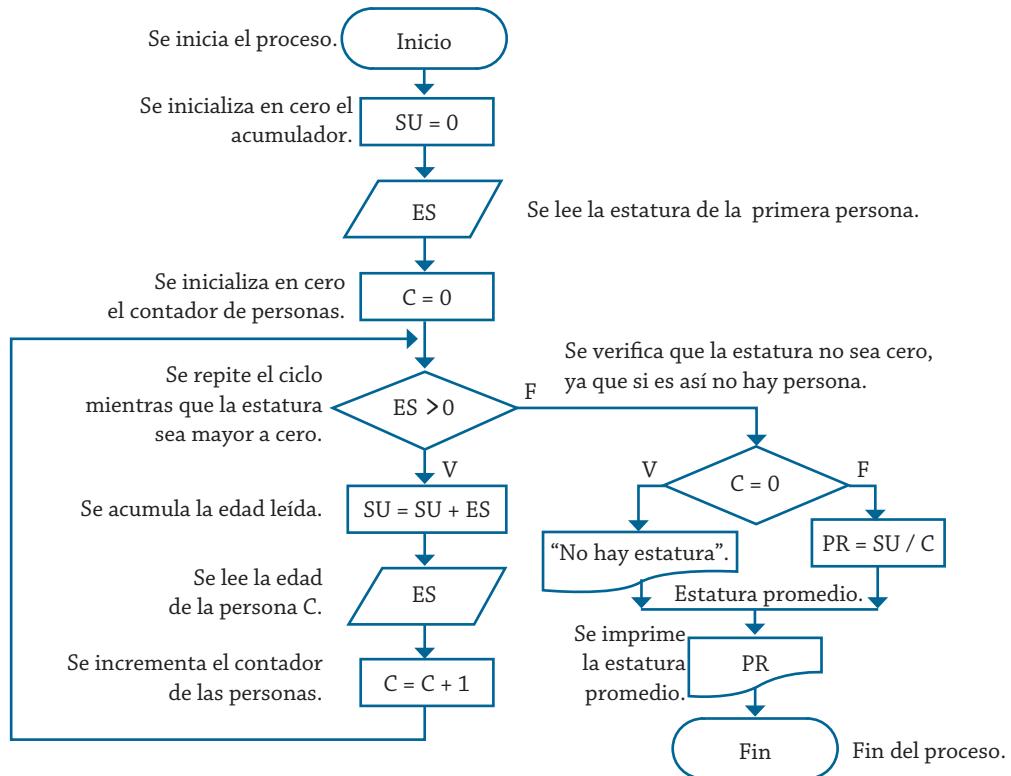


Diagrama de flujo 4.7 Algoritmo para obtener la estatura promedio de un número desconocido de personas.

Nombre de la variable	Descripción	Tipo
C	Contador de personas	Entero
ES	Estatura de cada persona	Real
SU	Suma de las estaturas	Real
PR	Estatura promedio	Real

Tabla 4.3 Variables utilizadas para obtener la estatura promedio de un número desconocido de personas.

```

1. Inicio
2. Hacer SU = 0
3. Leer ES
4. Hacer C = 0
5. Mientras ES > 0
    Hacer SU = SU + ES
    Leer ES
    Hacer C = C + 1
Fin mientras
6. Si C = 0
    Entonces
        Escribir "No hay estaturas"
    Si no
        Hacer PR = SU / C
    Fin compara
7. Escribir PR
8. Fin

```

Pseudocódigo 4.7 Algoritmo para obtener la estatura promedio de un número desconocido de personas.

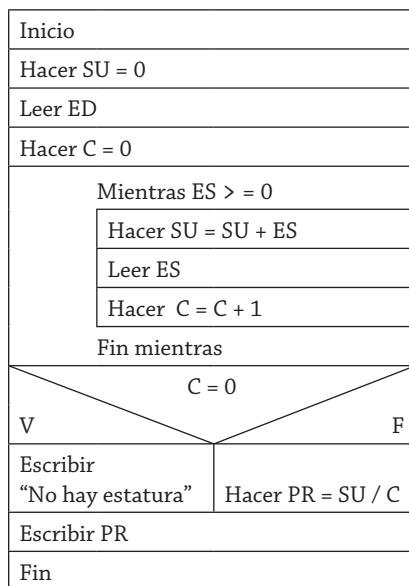


Diagrama N/S 4.7 Algoritmo para obtener la estatura promedio de un número desconocido de personas.

Ejemplo 4.6

Se requiere un algoritmo para determinar cuánto ahorrará una persona en un año, si al final de cada mes deposita variables cantidades de dinero; además, se requiere saber cuánto lleva ahorrado cada mes. Realice el diagrama de flujo, el pseudocódigo y el diagrama N/S para representarlo, utilizando un ciclo apropiado.

La tabla 4.4 muestra las variables que se requieren para plantear la solución del problema.

Nombre de la variable	Descripción	Tipo
AH	Ahorro mensual	Real
M	Contador del mes	Entero
CA	Cantidad que se va a ahorrar	Entero

Tabla 4.4 Variables utilizadas para determinar el ahorro de una persona en un año.

Este problema se puede resolver mediante la utilización de cualquiera de los ciclos, dado que se conoce el número de veces que se debe efectuar el ciclo, pero se debe considerar que en caso de utilizar el ciclo Repite, al menos para un mes se debe tener el ahorro.

La solución para este problema utilizando el ciclo Mientras, se puede plantear mediante el pseudocódigo 4.8, o con el diagrama de flujo 4.8, y en su caso con el diagrama N/S 4.8.

1. Inicio
2. Hacer $AH = 0$
3. Hacer $M = 1$
4. Mientras $M <= 12$
 - Leer CA
 - Hacer $AH = AH + CA$
 - Hacer $M = M + 1$
 - Escribir "El ahorro del mes:", M, "es", AH
5. Escribir "El ahorro final es:", AH
6. Fin

Pseudocódigo 4.8 Algoritmo para determinar el ahorro de una persona en un año.

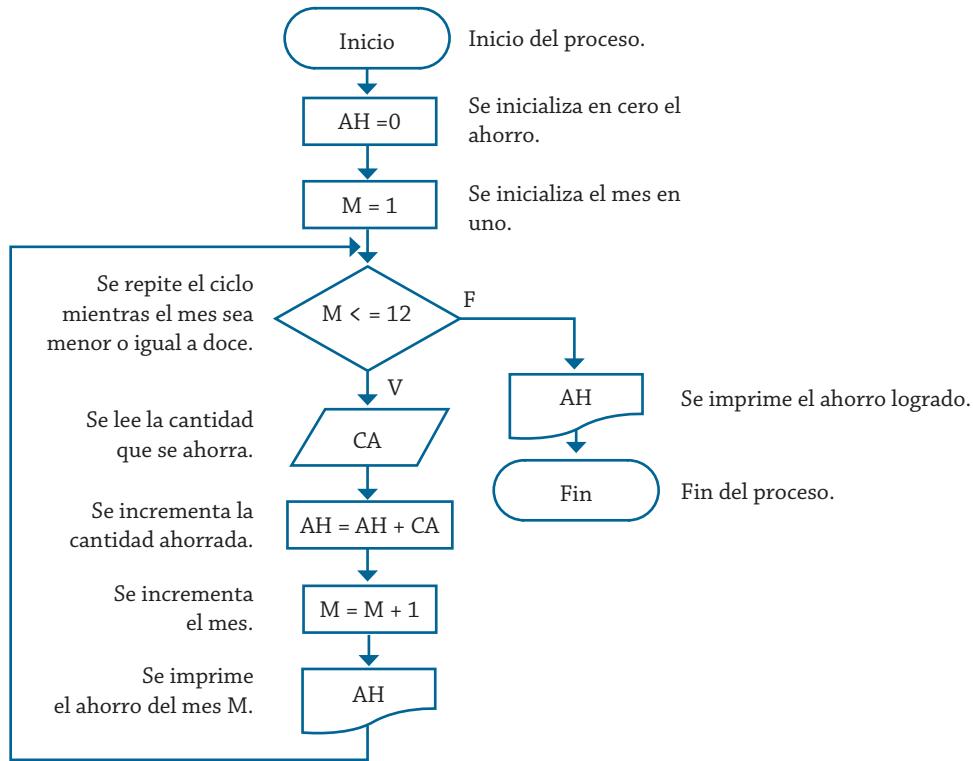


Diagrama de flujo 4.8 Algoritmo para determinar el ahorro de una persona en un año.

Inicio
Hacer AH = 0
Hacer M = 1
Mientras M <= 12
Leer CA
Hacer AH = AH + CA
Hacer M = M + 1
Escribir AH
Fin mientras
Escribir AH
Fin

Diagrama N/S 4.8 Algoritmo para determinar el ahorro de una persona en un año.

Ejemplo 4.7

Se requiere un algoritmo para determinar, de N cantidades, cuántas son menores o iguales a cero y cuántas mayores a cero. Realice el diagrama de flujo, el pseudocódigo y el diagrama N/S para representarlo, utilizando el ciclo apropiado.

La tabla 4.5 muestra las variables que se requieren para plantear la solución del problema.

Nombre de la variable	Descripción	Tipo
C	Contador	Entero
CA	Cantidad por leer	Entero
CP	Cantidadas positivas	Entero
CN	Cantidadas negativas	Entero
NU	Número de cantidades	Real

Tabla 4.5 Variables utilizadas para determinar el número de cantidades positivas y negativas.

Este problema se puede resolver mediante la utilización de cualquiera de los ciclos, dado que se conoce el número de cantidades para capturar, que es el número de veces que se debe efectuar el ciclo, pero se debe considerar que en caso de utilizar el ciclo Repite, al menos debe haber una cantidad leída o por leer.

La solución para este problema utilizando el ciclo Repite se puede plantear mediante el pseudocódigo 4.9 o con el diagrama N/S 4.9, o en su caso con el diagrama de flujo 4.9.

1. Inicio
2. Hacer $CP = 0$
3. Hacer $CN = 0$
4. Leer NU
5. Hacer $C = 1$
6. Repite
 - Leer CA
 - Si $CA > 0$
 - Entonces
 - Hacer $CP = CP + 1$
 - Si no
 - Hacer $CN = CN + 1$
 - Fin compara
 - Hacer $C = C + 1$
 - Hasta $C > NU$
7. Escribir "Positivos:", CP
8. Escribir "Negativos:", CN
9. Fin

Pseudocódigo 4.9 Algoritmo para determinar el número de cantidades positivas y negativas.

```
graph TD; Inicio[Inicio] --> HacerCP[Hacer CP = 0]; HacerCP --> HacerCN[Hacer CN = 0]; HacerCN --> LeerNU[Leer NU]; LeerNU --> HacerC1[Hacer C = 1]; HacerC1 --> Repite[Repite]; Repite --> LeerCA[Leer CA]; LeerCA --> CAgt0{CA > 0}; CAgt0 -- Sí --> CPplus1[CP = CP + 1]; CAgt0 -- No --> CNplus1[CN = CN + 1]; CPplus1 --> HacerCplus1[Hacer C = C + 1]; HacerCplus1 --> HastaCgtNU[Hasta C > NU]; HastaCgtNU --> EscribirCP[Escribir CP, CN]; EscribirCP --> Fin[Fin]
```

Diagrama N/S 4.9 Algoritmo para determinar el número de cantidades positivas y negativas.

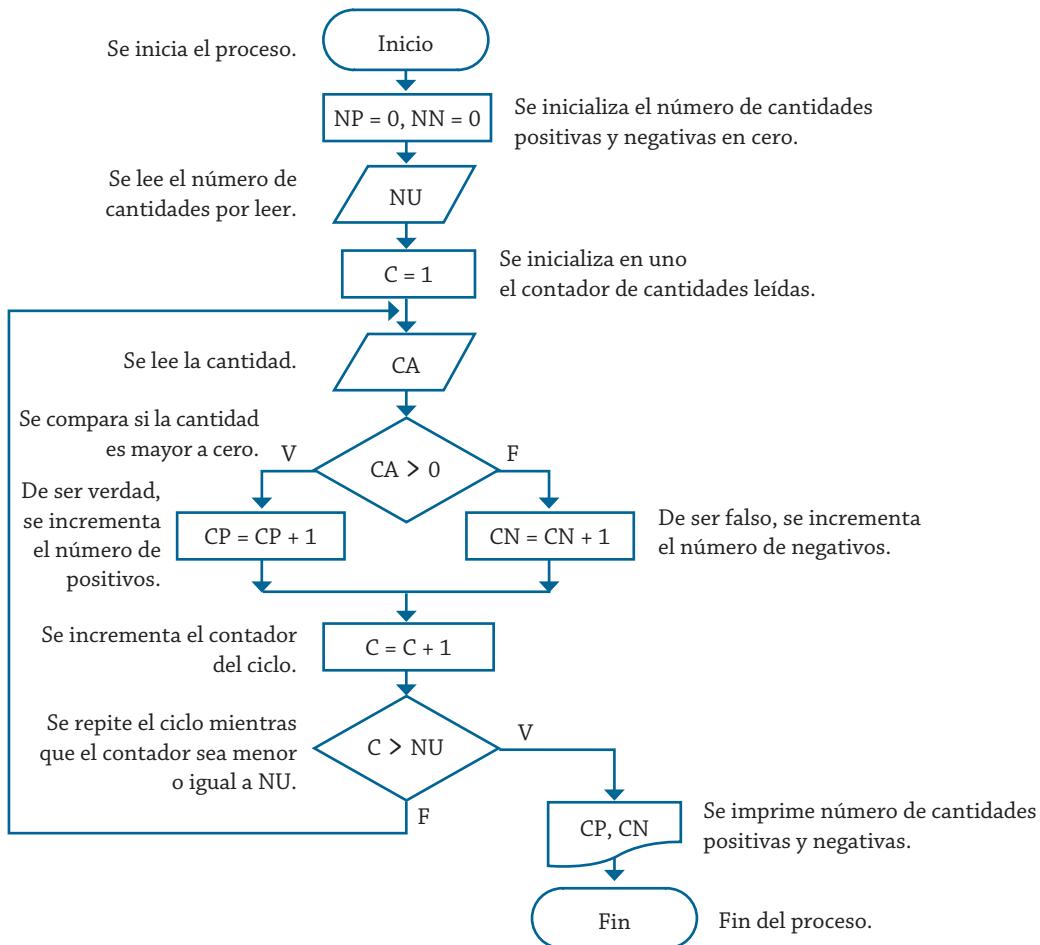


Diagrama de flujo 4.9 Algoritmo para determinar el número de cantidades positivas y negativas.

Ejemplo 4.8

Realice un algoritmo para generar e imprimir los números pares que se encuentran entre 0 y 100. Realice el diagrama de flujo, el pseudocódigo y el diagrama N/S para representarlo, utilizando el ciclo apropiado.

Para este caso se requiere únicamente una variable, tal y como se muestra en la tabla 4.6.

Nombre de la variable	Descripción	Tipo
N	Número par por generar	Entero

Tabla 4.6 Variables utilizadas para generar el número par.

La solución para este problema utilizando el ciclo Mientras se puede plantear mediante el diagrama de flujo 4.10, con el pseudocódigo 4.10 o en su caso con el diagrama N/S 4.10.

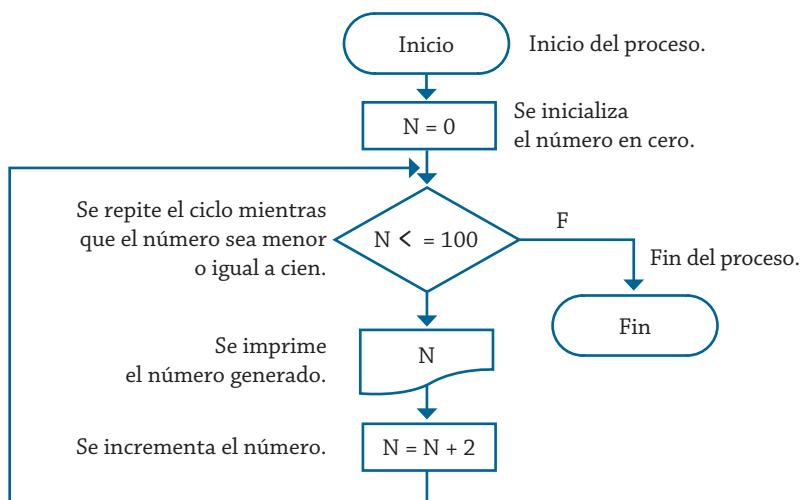


Diagrama de flujo 4.10 Algoritmo para generar los números pares entre 0 y 100.

Como se puede ver en el algoritmo, iniciando en cero, con sólo sumar dos a cada número generado resulta el siguiente número par, proceso que se repite hasta llegar al número cien, que es lo que se requería.

1. Inicio
2. Hacer $N = 0$
3. Mientras $N \leq 100$
 - Escribir N
 - Hacer $N = N + 2$
 - Fin mientras
4. Fin

Pseudocódigo 4.10 Algoritmo para generar los números pares entre 0 y 100.

Inicio
Hacer N = 0
Mientras N < = 100
Escribir N
Hacer N = N + 2
Fin desde
Fin

Diagrama N/S 4.10 Algoritmo para generar los números pares entre 0 y 100.

La solución para este problema utilizando el ciclo Repite se puede plantear mediante el pseudocódigo 4.11, con el diagrama de flujo 4.11 o en su caso con el diagrama N/S 4.11.

1. Inicio
2. Hacer N = 0
3. Repite
 - Escribir N
 - Hacer N = N + 2
 - Hasta N > 100
4. Fin

Pseudocódigo 4.11 Algoritmo para generar los números pares entre 0 y 100.

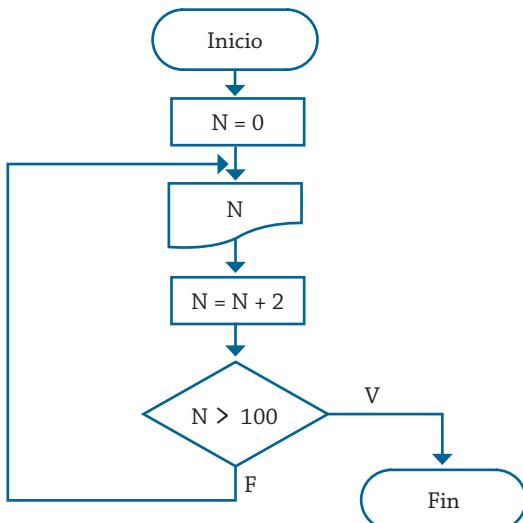


Diagrama de flujo 4.11 Algoritmo para generar los números pares entre 0 y 100.

En lo que respecta al diagrama N/S, prácticamente es el mismo que para el ciclo Mientras, sólo cambiando: "Mientras N < = 100" por "Repite" y "Fin mientras" por "Hasta N > 100". Y como se puede ver, al utilizar este ciclo por lo menos escribirá el cero, que corresponde al valor inicial de la variable, y hasta después de incrementar en dos el valor de la misma compara, de ahí que la condición que se establece es que la variable debe ser mayor a 100.

Ejemplo 4.9

Realice un algoritmo para generar N elementos de la sucesión de Fibonacci (0, 1, 1, 2, 3, 5, 8, 13,...). Realice el diagrama de flujo, el pseudocódigo y el diagrama N/S para representarlo, utilizando el ciclo apropiado.

El planteamiento del algoritmo correspondiente se hace a partir del análisis de la sucesión, en la que se puede observar que un tercer valor de la serie está dado por la suma de los dos valores previos, de aquí que se asignan los dos valores para sumar (0, 1), que dan la base para obtener el siguiente elemento que se busca, además, implica que el ciclo se efectúe dos veces menos.

Las variables que se requieren para la solución de este problema se muestran en la tabla 4.7. En lo que respecta a qué tipo de ciclo se debe utilizar, es indistinto, por lo cual se muestran las tres alternativas a continuación.

Nombre de la variable	Descripción	Tipo
A, B	Valores iniciales o previos	Entero
C	Valor generado	Entero
M	Contador del ciclo	Entero
N	Número de elementos de la serie	Entero

Tabla 4.7 Variables utilizadas para generar el número par.

La solución para este problema utilizando el ciclo Mientras se puede plantear mediante el pseudocódigo 4.12, con el diagrama de flujo 4.12, o en su caso con el diagrama N/S 4.12.

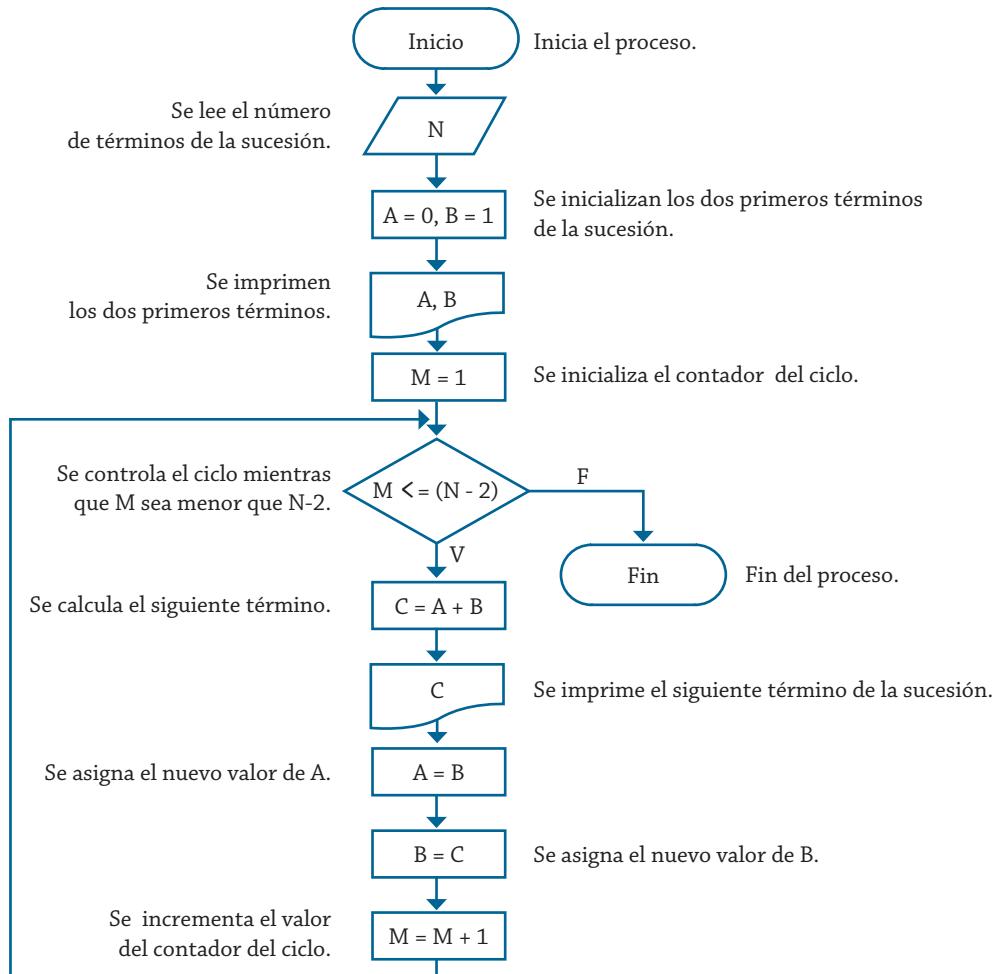


Diagrama de flujo 4.12 Algoritmo para generar N elementos de la sucesión de Fibonacci.

1. Inicio
2. Leer N
3. Hacer $A = 0$
4. Hacer $B = 1$
5. Escribir A, B
6. Hacer $M = 1$
7. Mientras $M \leq (N - 2)$
 - Hacer $C = A + B$
 - Escribir C
 - Hacer $A = B$
 - Hacer $B = C$
 - Hacer $M = M + 1$
8. Fin

Pseudocódigo 4.12 Algoritmo para generar N elementos de la sucesión de Fibonacci.

Inicio
Leer N
Hacer A = 0
Hacer B = 0, M =1
Escribir A, B
Mientras M < = (N - 2)
Hacer C = A + B
Escribir C
Hacer A = B
Hacer B = C
Hacer M = M + 1
Fin mientras
Fin

Diagrama N/S 4.12 Algoritmo para generar N elementos de la sucesión de Fibonacci.

La solución para este problema utilizando el ciclo Repite se puede plantear mediante el pseudocódigo 4.13, con el diagrama de flujo 4.13 o en su caso con el diagrama N/S 4.13.

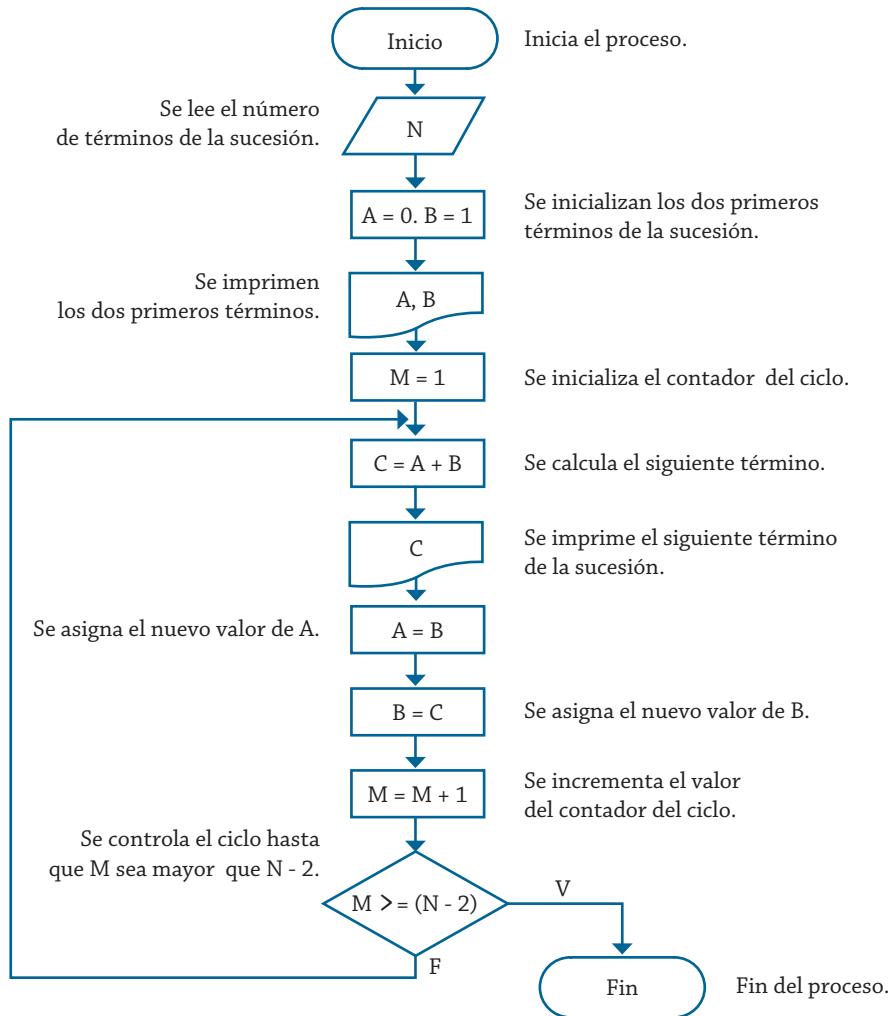


Diagrama de flujo 4.13 Algoritmo para generar N elementos de la sucesión de Fibonacci.

1. Inicio
2. Leer N
3. Hacer $A = 0$
4. Hacer $B = 1$
5. Escribir A, B
6. Repite
 - Hacer $C = A + B$
 - Escribir C
 - Hacer $A = B$
 - Hacer $B = C$
 - Hacer $M = M + 1$
 - Hasta $M > (N - 2)$
7. Fin

Pseudocódigo 4.13 Algoritmo para generar N elementos de la sucesión de Fibonacci.

Inicio
Leer N
Hacer A = 0. M = 1
Hacer B = 0
Escribir A. B
Repite
Hacer C = A + B
Escribir C
Hacer A = B
Hacer B = C
Hacer M = M + 1
Hasta M > (N - 2)
Fin

Diagrama N/S 4.13 Algoritmo para generar N elementos de la sucesión de Fibonacci.

El diagrama de flujo 4.14 muestra la solución mediante la utilización de un ciclo Desde. Se omite el pseudocódigo y el diagrama N/S para evitar ser repetitivos, dado que son semejantes las soluciones.

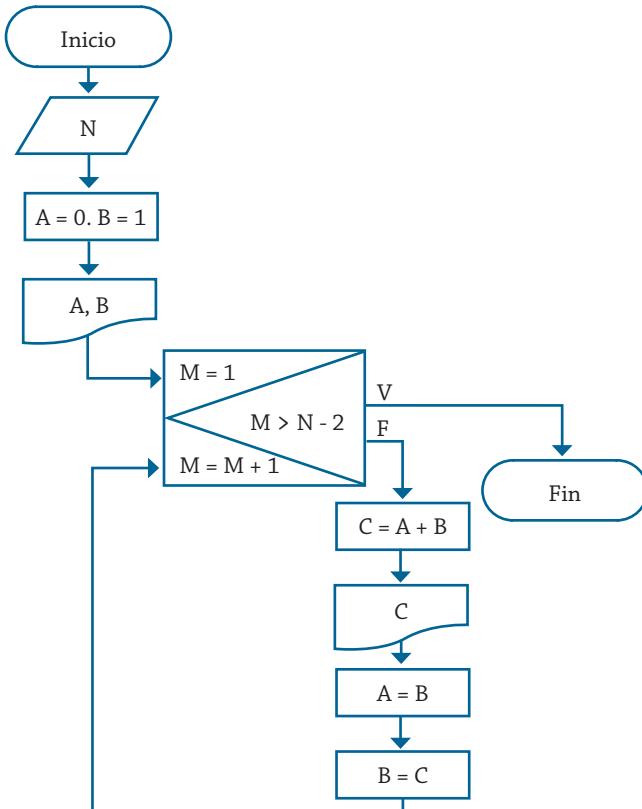


Diagrama de flujo 4.14 Algoritmo para generar N elementos de la sucesión de Fibonacci.

Ejemplo 4.10

Una empresa tiene el registro de las horas que trabaja diariamente un empleado durante la semana (seis días) y requiere determinar el total de éstas, así como el sueldo que recibirá por las horas trabajadas. Realice un algoritmo para determinar esto y represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S, utilizando el ciclo apropiado. La tabla 4.8 muestra las variables requeridas en la solución de este problema.

Nombre de la variable	Descripción	Tipo
D	Contador del ciclo de días	Entero
PH	Pago por hora	Real
SH	Horas trabajadas en la semana	Entero
HT	Horas trabajadas por día	Entero
SU	Sueldo semanal	Real

Tabla 4.8 Variables utilizadas para determinar las horas trabajadas y el sueldo de un empleado.

Dado que se conoce como dato que el total de días por contabilizar es seis, es posible realizar la solución con cualquier tipo de ciclo, por tal motivo se presenta la solución del problema mostrando los tres diagramas de flujo correspondientes, y para el ciclo Repite, el pseudocódigo y el diagrama N/S.

La solución para este problema utilizando el ciclo Mientras se puede plantear mediante el pseudocódigo 4.15, con el diagrama de flujo 4.15 o en su caso con el diagrama N/S 4.15.

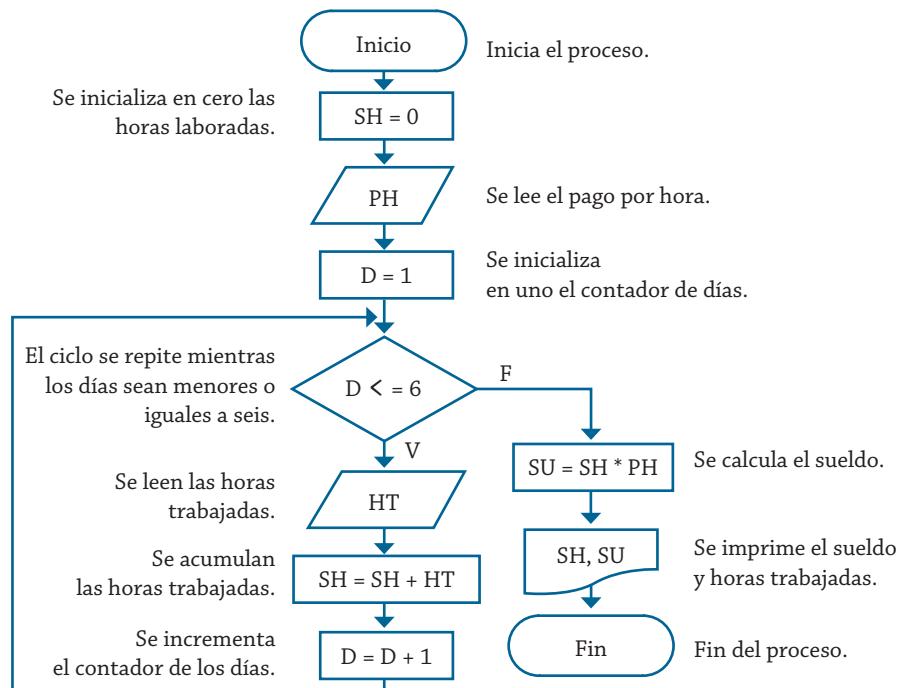


Diagrama de flujo 4.15 Algoritmo para determinar las horas trabajadas y el sueldo semanal de un empleado.

1. Inicio
2. Hacer SH = 0
3. Leer PH
4. Hacer D = 1
5. Mientras D < = 6
 - Leer HT
 - Hacer SH = SH + HT
 - Hacer D = D + 1
- Fin mientras
6. SU = SH * PH
7. Escribir "Las horas laboradas son =", SH
8. Escribir "El sueldo es =", SU
9. Fin

Pseudocódigo 4.15 Algoritmo para determinar las horas laboradas y el sueldo semanal de un empleado.

Inicio
Hacer SH = 0
Leer PH
Hacer D = 1
Mientras D < = 6
Leer HT
Hacer SH = SH + HT
Hacer D = D + 1
Fin mientras
Hacer SU = SH * PH
Escribir SH, SU
Fin

Diagrama N/S 4.15 Algoritmo para determinar las horas trabajadas y el sueldo semanal de un empleado.

De igual forma, para el ciclo Repite, la solución se puede representar mediante el diagrama de flujo 4.16, con el pseudocódigo 4.16 y el diagrama N/S 4.16, de donde se puede ver que básicamente la solución es la misma, lo que cambia es la estructura del control del ciclo; de igual forma, el diagrama de flujo 4.17 muestra la solución del problema mediante la utilización del ciclo Desde.

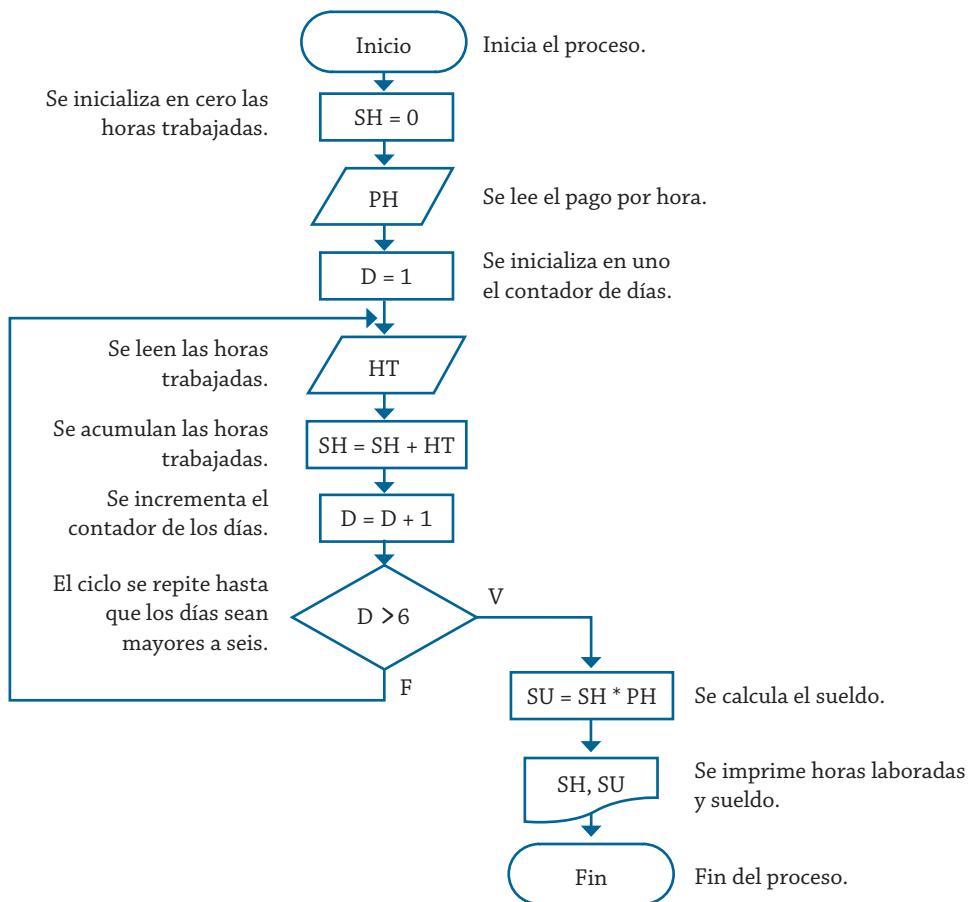


Diagrama de flujo 4.16 Algoritmo para determinar las horas trabajadas y el sueldo semanal de un empleado.

1. Inicio
2. Hacer $SH = 0$
3. Leer PH
4. Hacer $D = 1$
5. Repite
 - Leer HT
 - Hacer $SH = SH + HT$
 - Hacer $D = D + 1$
6. Hasta $D > 6$
7. $SU = SH * PH$
8. Escribir "Las horas laboradas son =", SH
9. Escribir "El sueldo es =", SU
10. Fin

Diagrama N/S 4.16 Algoritmo para determinar las horas trabajadas y el sueldo semanal de un empleado.

Inicio
Hacer SH = 0
Leer PH
Hacer D = 1
Repite
Leer HT
Hacer SH = SH + HT
Hacer D = D + 1
Mientras D > 6
Hacer SU = SH * PH
Escribir SU
Fin

Diagrama N/S 4.16 Algoritmo para determinar las horas trabajadas y el sueldo semanal de un empleado.

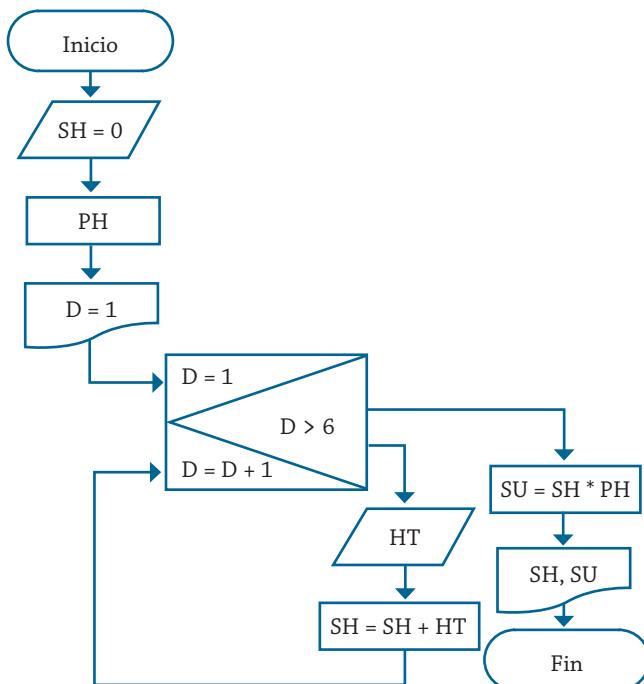


Diagrama de flujo 4.17 Algoritmo para determinar las horas trabajadas y el sueldo semanal de un empleado.

Ejemplo 4.11

Una persona se encuentra en el kilómetro 70 de la carretera Aguascalientes-Zacatecas, otra se encuentra en el km 150 de la misma carretera, la primera viaja en dirección a Zacatecas, mientras que la segunda se dirige a Aguascalientes, a la misma velocidad. Realice un algoritmo para determinar en qué kilómetro de esa carretera se encontrarán y represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S, utilizando el ciclo apropiado.

Las variables que se emplean se muestran en la tabla 4.9. Para plantear el algoritmo que dé la solución de este problema, no se puede utilizar un ciclo *Desde*, dado que se desconoce el número de veces que se debe efectuar el ciclo, por consiguiente, para la solución de este problema se pueden utilizar los ciclos *Mientras* o *Repite*, ya que se determinará la duración del ciclo cuando la distancia entre los dos puntos sea cero o menor que cero, como se ve en la solución planteada; con base en esto, el diagrama de flujo 4.18 muestra el algoritmo de solución mediante la utilización del ciclo *Mientras*.

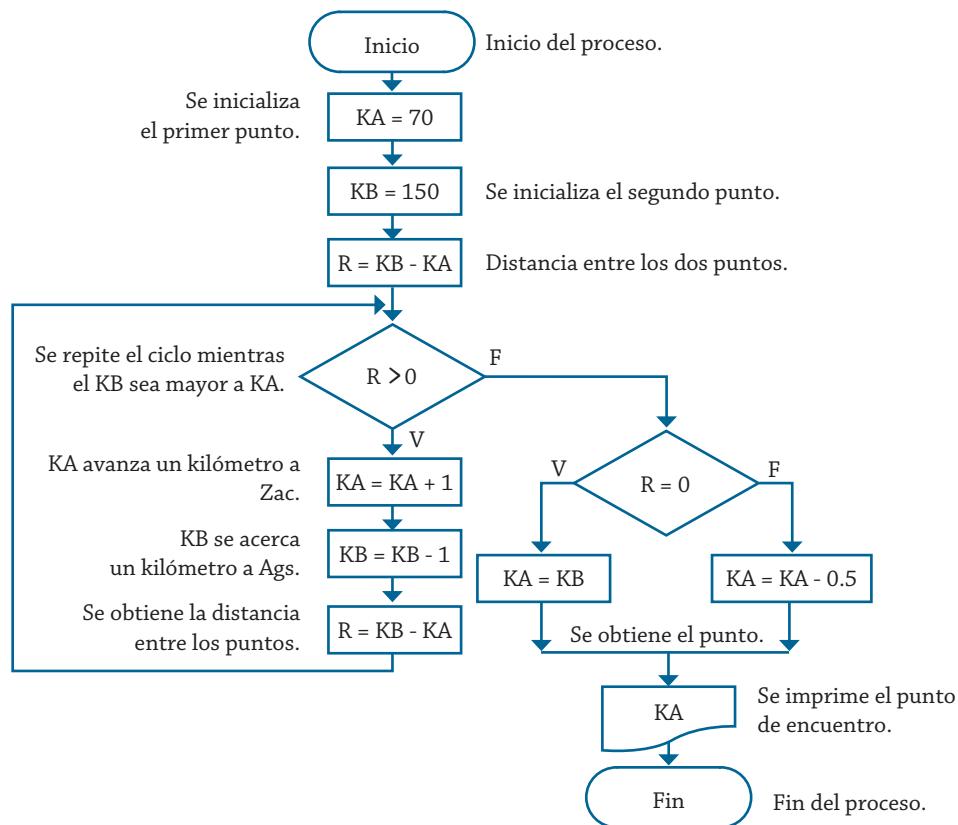


Diagrama de flujo 4.18 Algoritmo para determinar el punto de encuentro entre las dos personas.

Nombre de la variable	Descripción	Tipo
KA	Primer punto en la carretera	Real
KB	Segundo punto en la carretera	Real
R	Distancia entre los dos puntos	Entero

Tabla 4.9 Variables utilizadas para determinar el punto de encuentro entre las dos personas.

A partir del diagrama de flujo se puede observar que cuando se establece finalmente el punto de encuentro, si R fue igual a cero los dos valores de kilómetros son iguales, y en caso de que no se cumpla esta condición

se tiene que disminuir medio kilómetro al punto KA, dado que ahora la distancia entre KA y KB sería de un kilómetro, por lo tanto, el punto común para ambos es medio kilómetro antes del que ahora tiene KA, o bien medio kilómetro más del que ahora tiene KB, esto en lo que respecta a valores absolutos.

Si se planteara la solución utilizando el ciclo Repite, esta parte de la consideración de las distancias debería ser igual, lo que cambiaría sería básicamente el formato del ciclo, en lo que difiere del ciclo Mientras.

El pseudocódigo 4.17 y el diagrama N/S 4.17 representan la solución correspondiente a este planteamiento:

1. Inicio
2. Hacer KA = 70
3. Hacer KB = 150
4. Hacer R = KB - KA
5. Mientras R > 0
 - Hacer KA = KA + 1
 - Hacer KB = KB - 1
 - Hacer R = KB - KA
- Fin mientras
6. Si R = 0
 - Entonces
 - Hacer KA = KB
- Si no
 - Hacer KA = KA - 0.5
- Fin compara
7. Escribir "Punto de encuentro =", KA
8. Fin

Pseudocódigo 4.17 Algoritmo para determinar el punto de encuentro entre las dos personas.

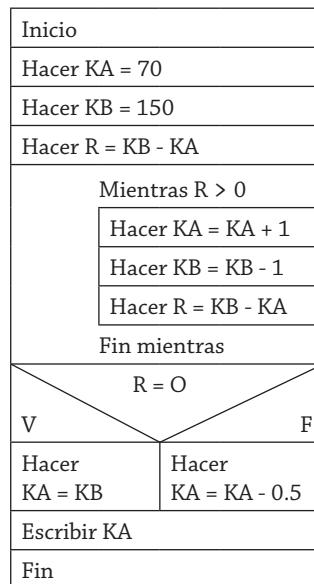


Diagrama N/S 4.17 Algoritmo para determinar el punto de encuentro entre las dos personas.

Ejemplo 4.12

Un empleado de la tienda “Tiki Taka” realiza N ventas durante el día, se requiere saber cuántas de ellas fueron mayores a \$1000, cuántas fueron mayores a \$500 pero menores o iguales a \$1000, y cuántas fueron menores o iguales a \$500. Además, se requiere saber el monto de lo vendido en cada categoría y de forma global. Realice un algoritmo que permita determinar lo anterior y representelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S, utilizando el ciclo apropiado.

La tabla 4.11 muestra las variables requeridas para plantear el algoritmo que permita obtener la solución del problema 4.12.

Nombre de la variable	Descripción	Tipo
N	Número de ventas	Real
CN	Contador de las ventas	Real
A	Ventas mayores a mil	Entero
B	Ventas mayores a quinientos pero menores o iguales a mil	Entero
C	Ventas menores o iguales a quinientos	Entero
V	Monto de la venta	Real
T1	Total de las ventas tipo A	Real
T2	Total de las ventas tipo B	Real
T3	Total de las ventas tipo C	Real
TT	Total de las ventas	Real

Tabla 4.11 Variables utilizadas para determinar el número de ventas de cada tipo y sus montos.

Es posible plantear este problema con cualquier tipo de ciclo, ya que previamente se puede saber cuántas ventas se realizan y cuál es el valor que determina el número de veces que se realiza el ciclo, por tal motivo, el pseudocódigo 4.18 muestra la solución mediante la utilización de un ciclo Mientras. De igual forma, el diagrama de flujo 4.19 y el diagrama N/S 4.18 representan la solución correspondiente a este problema.

1. Inicio
2. Leer N
3. Hacer A = 0, B = 0, C = 0
4. Hacer T1 = 0, T2 = 0, T3 = 0
5. Hacer TT = 0
6. Hacer CN = 1
7. Mientras CN < = N
 - Leer V
 - Si V > 1000
 - Entonces
 - Hacer A = A + 1
 - Hacer T1 = T1 + 1
 - Si no
 - Si V > 500
 - Entonces
 - Hacer B = B + 1
 - Hacer T2 = T2 + 1
 - Si no
 - Hacer C = C + 1
 - Hacer T3 = T3 * 1
 - Fin comparar
- Fin comparar
- Hacer TT = TT + V
- Hacer CN = CN + 1
- Fin mientras
- Escribir "Las ventas y el total de ventas 1 es:", A, T1
- Escribir "Las ventas y el total de ventas 2 es:", B, T2
- Escribir "Las ventas y el total de ventas 3 es:", C, T3
- Escribir "El total de ventas es:", TT
12. Fin

Pseudocódigo 4.18 Algoritmo para determinar el número de ventas de cada tipo y sus montos respectivos.

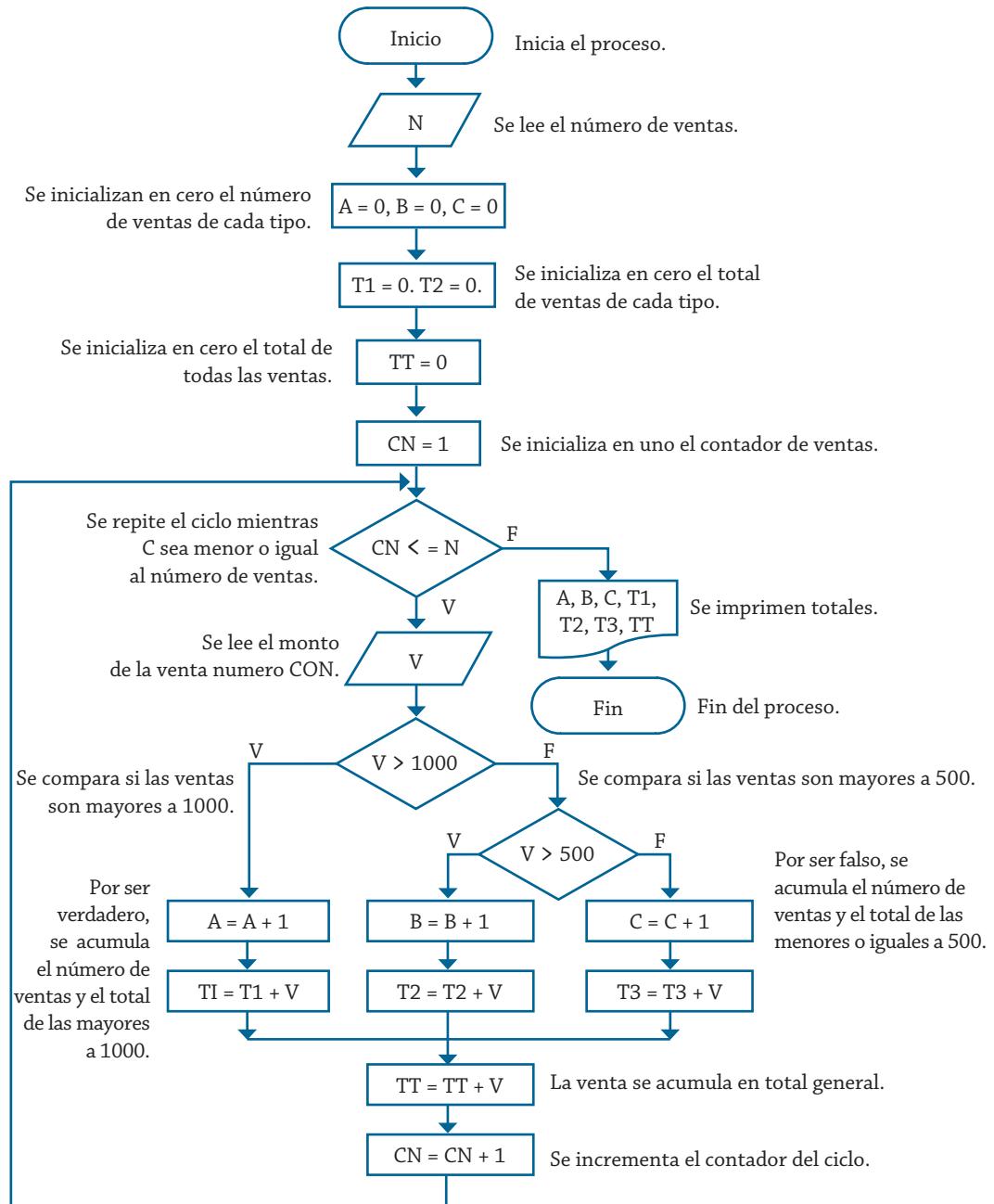


Diagrama de flujo 4.19 Algoritmo para determinar el número de ventas de cada tipo y sus montos respectivos.

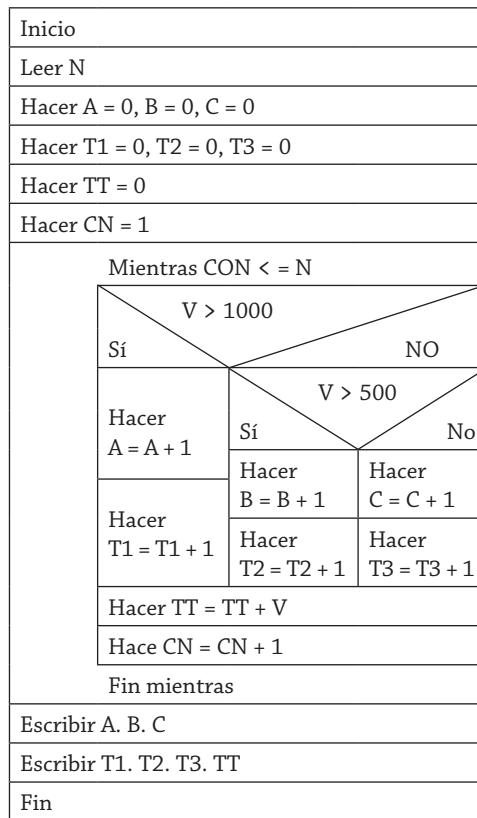


Diagrama N/S 4.18 Algoritmo para determinar el número de ventas de cada tipo y sus montos respectivos.

Sólo con el objetivo de realizar una comparación entre un ciclo Mientras y un ciclo Desde, se presenta la solución mediante el diagrama de flujo 4.20 con este tipo de ciclo.

Como se puede ver, si se comparan los dos diagramas de flujo que presentan la solución, básicamente son idénticos, lo que cambia es la forma de controlar el bucle, ya que en un ciclo Mientras la variable se inicializa, se compara y se incrementa en diferentes momentos del recorrido del ciclo, en cambio, en un ciclo Desde, la variable se inicializa, se compara y se incrementa en el mismo símbolo.

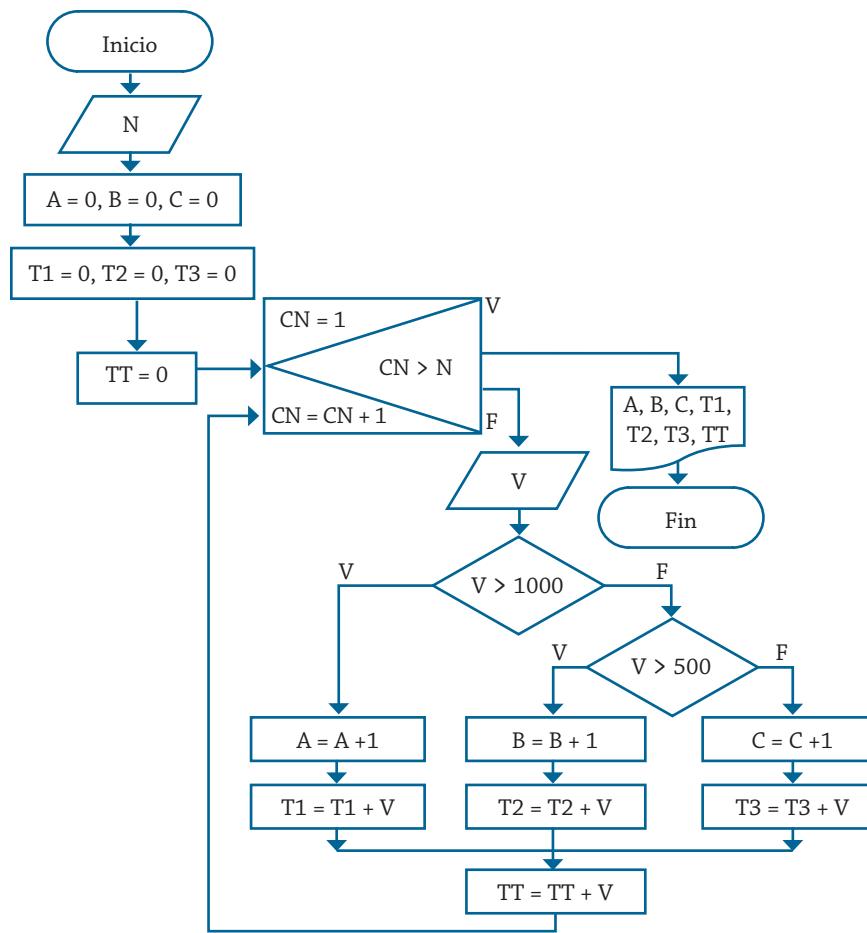


Diagrama de flujo 4.20 Algoritmo para determinar el número de ventas de cada tipo y sus montos respectivos.

Ejemplo 4.13

Una persona adquirió un producto para pagar en 20 meses. El primer mes pagó \$10, el segundo \$20, el tercero \$40 y así sucesivamente. Realice un algoritmo para determinar cuánto debe pagar mensualmente y el total de lo que pagó después de los 20 meses y represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S utilizando el ciclo apropiado.

La tabla 4.12 muestra las variables requeridas para plantear la solución del problema.

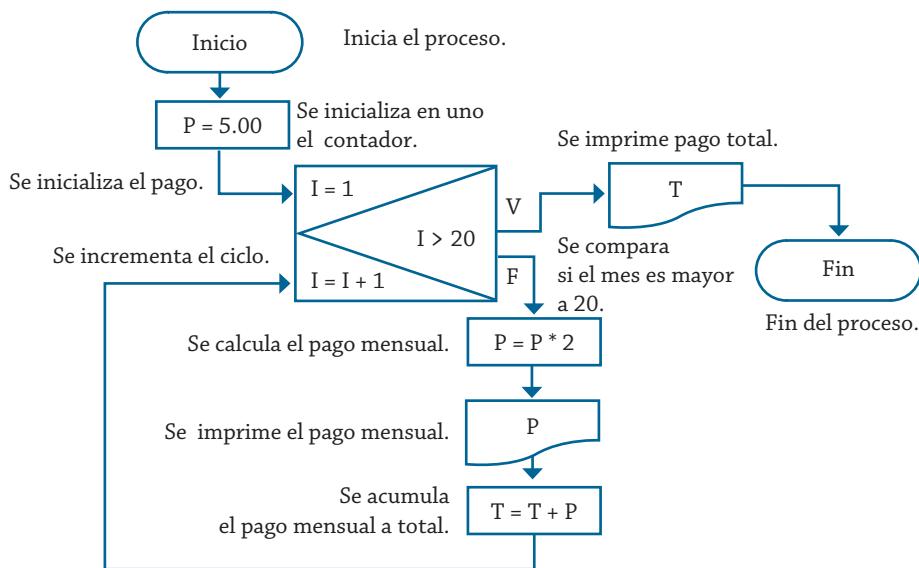
Nombre de la variable	Descripción	Tipo
I	Contador del ciclo de meses	Entero
P	Cantidad para pagar mensualmente	Real
T	Pago total acumulado	Real

Tabla 4.12 Variables utilizadas para determinar el pago mensual y el costo total del artículo.

Para resolver este problema se parte de que el pago inicial es de \$5, para que al momento de entrar al ciclo se consideren los 20 meses, con lo cual el primer mes será un pago de \$10, tal y como es la condición; además, esto simplifica el proceso, ya que si se inicializa en 10, se debería agregar una impresión más para el primer mes, y la duración del ciclo sería de 1 a 19 meses.

Como previamente se establece que el número de veces que se debe realizar el ciclo es 20, que corresponde al número de meses o de pagos, el ciclo que se aplicará es el de Desde.

El diagrama de flujo 4.21 muestra el algoritmo correspondiente para la solución de este problema.



Y el pseudocódigo 4.19 y diagrama N/S 4.19 muestran el algoritmo correspondiente a la solución del problema.

1. Inicio
2. Hacer $P = 5.0$
3. Desde $I = 1$ hasta $I = 20$
 - Hacer $P = P * 2$
 - Escribir “El pago mensual”, P
 - Hacer $T = T + P$
 - Fin desde
4. Escribir “Pago total”, T
5. Fin

Pseudocódigo 4.19 Algoritmo para determinar el pago mensual y costo total del artículo.

Inicio
Hacer P = 5.0
Desde I = 1 hasta 20
Hacer P = P * 2
Escribir P
Hacer T = T + P
Fin desde
Escribir T
Fin

Diagrama N/S 4.19 Algoritmo para determinar el pago mensual y costo total del artículo.

Ejemplo 4.14

Una empresa les paga a sus empleados con base en las horas trabajadas en la semana. Realice un algoritmo para determinar el sueldo semanal de N trabajadores y, además, calcule cuánto pagó la empresa por los N empleados. Represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S, utilizando el ciclo apropiado.

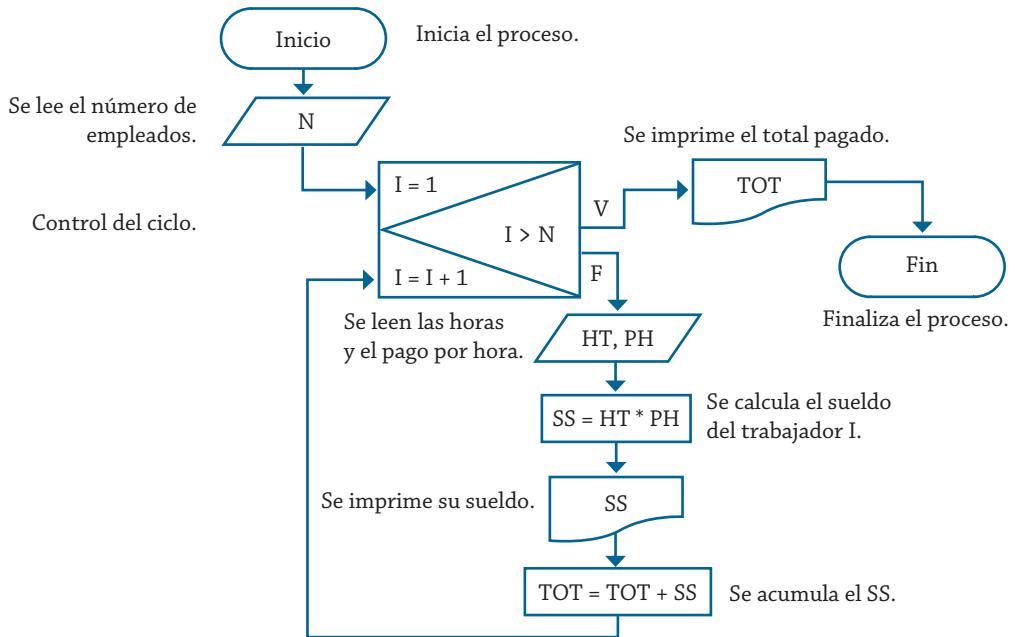
La tabla 4.13 muestra las variables requeridas para determinar el sueldo semanal de los N trabajadores con base en el total de horas trabajadas.

Nombre de la variable	Descripción	Tipo
N	Número de trabajadores	Entero
HT	Horas trabajadas	Real
PH	Pago por hora	Real
SS	Sueldo semanal	Real
I	Contador del ciclo de empleado	Entero

Tabla 4.13 Variables utilizadas para determinar el sueldo semanal de los N trabajadores.

Dado que previamente a realizar el proceso de calcular el sueldo de cada empleado se puede conocer el número de éstos, la solución se puede plantear sin ningún problema con cualquier tipo de ciclo. La solución aquí se planteará a partir de un ciclo *Desde*.

El algoritmo para la solución de este problema se presenta en el diagrama de flujo 4.22, el pseudocódigo 4.20 y el diagrama N/S 4.20.



1. Inicio
2. Leer N
3. Desde I = 1 hasta I = N
 - Leer HT, PH
 - Hacer $SS = HT * PH$
 - Escribir "el sueldo de trabajador", I, "es", SS
 - Hacer $TOT = TOT + SS$
 - Fin desde
4. Escribir "Pago total es =", TOT
5. Fin

Pseudocódigo 4.20 Algoritmo para determinar el sueldo semanal de los N trabajadores.

Inicio
Leer N
Desde I = 1 hasta N
Leer HT, PH
Hacer $SS = HT * PH$
Escribir SS
Hacer $TOT = TOT + SS$
Fin desde
Escribir TOT
Fin

Diagrama N/S 4.20 Algoritmo para determinar el sueldo semanal de los N trabajadores.

Ejemplo 4.15

Una empresa les paga a sus empleados con base en las horas trabajadas en la semana. Para esto, se registran los días que laboró y las horas de cada día. Realice un algoritmo para determinar el sueldo semanal de N trabajadores y además calcule cuánto pagó la empresa por los N empleados. Represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S, utilizando el ciclo apropiado.

El planteamiento de este problema es una alternativa del problema 4.14, ya que para el presente se debe acumular día con día las horas que labora cada trabajador, de tal forma que la tabla 4.14 muestra las variables requeridas.

Nombre de la variable	Descripción	Tipo
N	Número de trabajadores	Entero
HT	Horas trabajadas	Real
PH	Pago por hora	Real
SH	Suma de horas semanales	Entero
DT	Días laborados	Entero
SS	Sueldo semanal	Real
I	Contador del ciclo de empleado	Entero
D	Contador del ciclo de días	Entero

Tabla 4.14 Variables utilizadas para determinar el sueldo semanal de los N trabajadores, acumulando día con día la horas que laboran.

El diagrama de flujo 4.23 muestra el algoritmo para la solución de este problema.

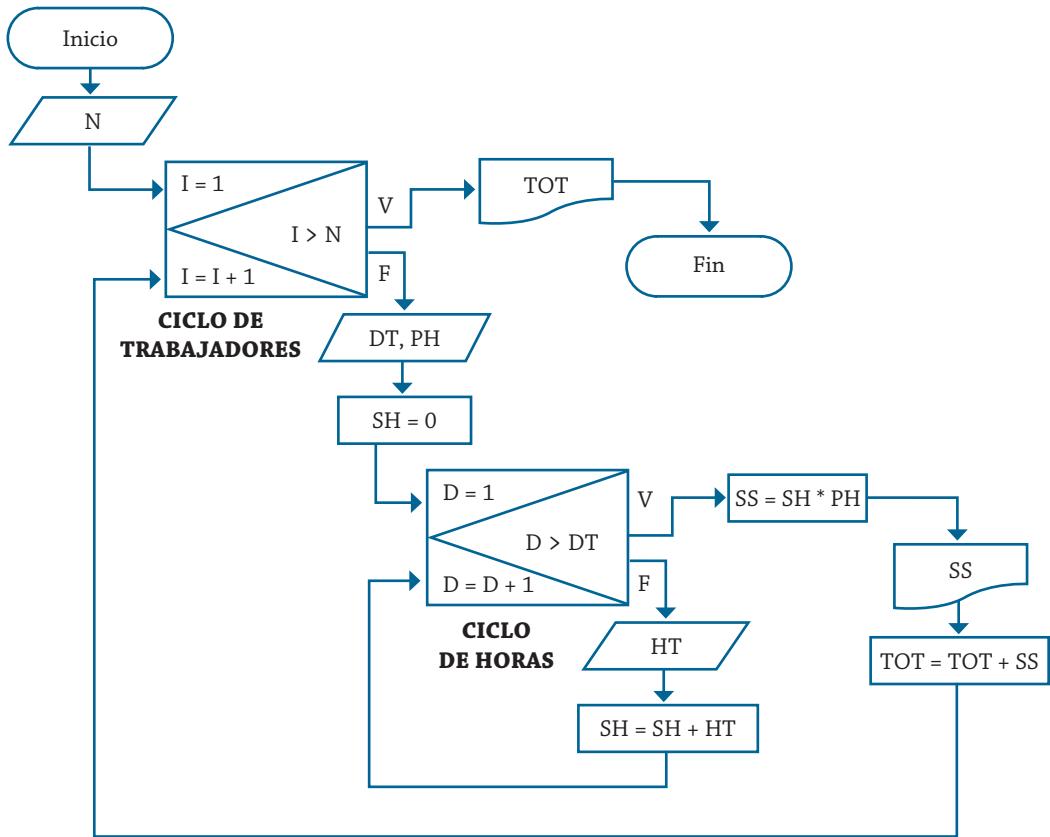


Diagrama de flujo 4.23 Algoritmo para determinar el sueldo semanal de los N trabajadores, acumulando día con día las horas que labora.

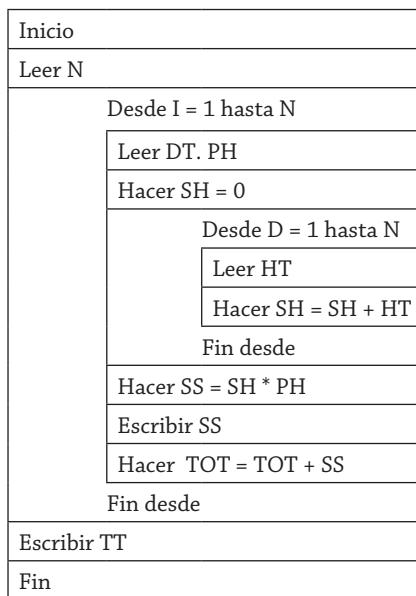
Como se puede ver a partir de la solución planteada, ahora se tienen dos ciclos, uno dentro del otro, cuando esto sucede es importante que el ciclo interno termine primero que el externo, ya que a cada valor del ciclo externo le corresponden los N valores del ciclo interno.

En este caso, a cada trabajador le corresponden DT días trabajados, cuyas horas laboradas cada uno de estos días deben ser proporcionadas, y esto se debe repetir para los N trabajadores, cuyo sueldo semanal requiere ser calculado.

De tal forma, el pseudocódigo 4.21 y el diagrama N/S 4.21 muestran el algoritmo correspondiente a la solución de este problema.

1. Inicio
2. Leer N
3. Desde I = 1 hasta N
 - Leer DT, PH
 - Hacer SH = 0
 - Desde D = 1 hasta DT
 - Leer HT
 - Hacer SH = SH + HT
 - Fin desde
 - Hacer SS = SH * PH
 - Escribir "El sueldo del trabajador", I, "es", SS
 - Hacer TOT = TOT + SS
- Fin desde
4. Escribir "El total que se pagó es", TOT
5. Fin

Pseudocódigo 4.21 Algoritmo para determinar el sueldo semanal de los N trabajadores, acumulando día con día las horas que labora.



Pseudocódigo 4.21 Algoritmo para determinar el sueldo semanal de los N trabajadores, acumulando día con día las horas que labora.

Ejemplo 4.16

La cadena de tiendas de autoservicio “El mandilón” cuenta con sucursales en C ciudades diferentes de la República, en cada ciudad cuenta con T tiendas y cada tienda cuenta con N empleados, asimismo, cada una registra lo que vende de manera individual cada empleado, cuánto fue lo que vendió cada tienda, cuánto se vendió en cada ciudad y cuánto recaudó la cadena en un solo día. Realice un algoritmo para determinar lo anterior y represéntelo mediante un diagrama de flujo, utilizando el ciclo apropiado.

El diagrama de flujo 4.24 muestra el algoritmo de solución mediante la utilización de ciclos *Desde*. La tabla 4.15 muestra las variables utilizadas.

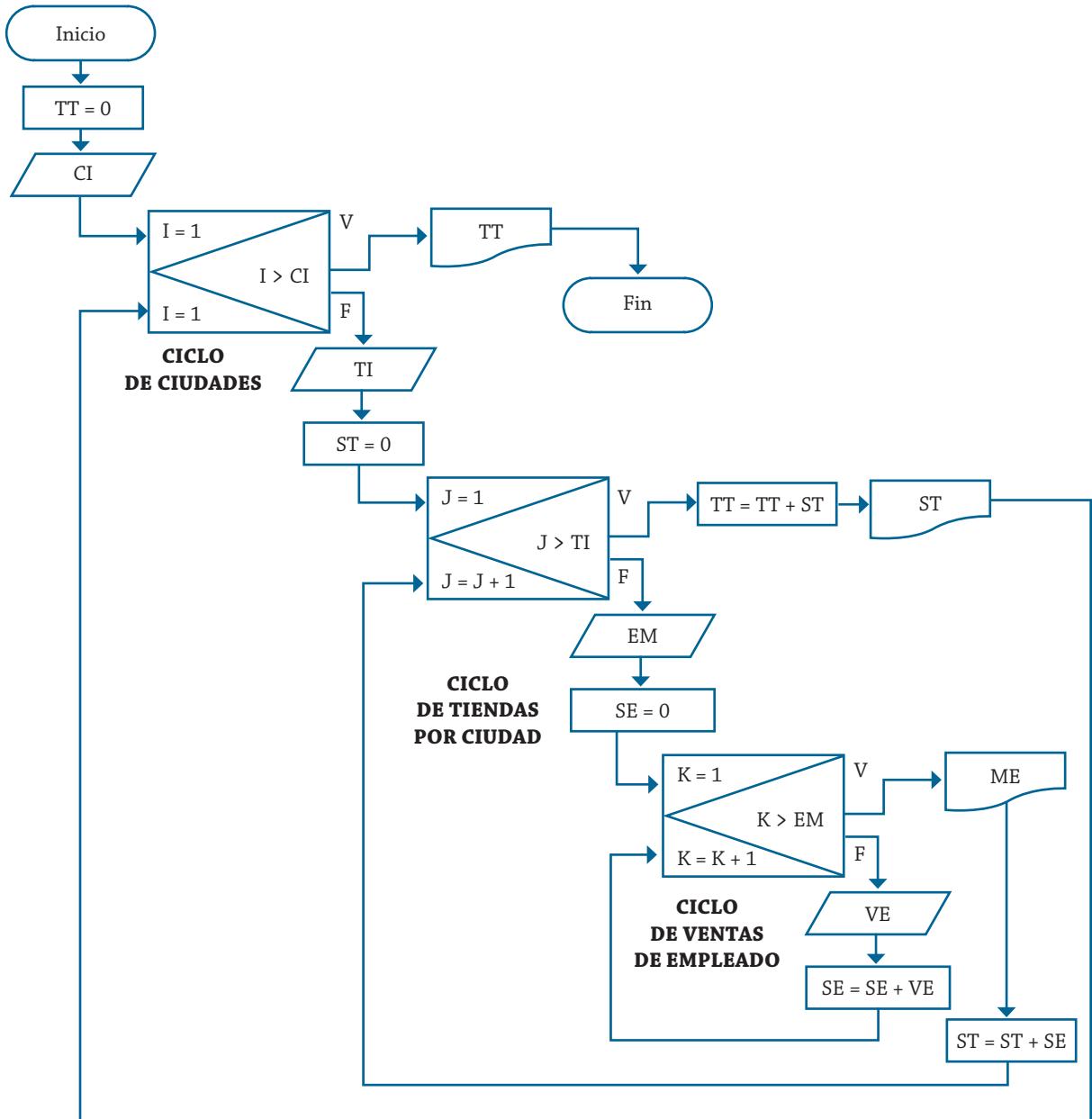


Diagrama de flujo 4.24 Algoritmo para determinar los montos de ventas por empleado, por tienda, por ciudad y el total.

Nombre de la variable	Descripción	Tipo
TT	Recaudado por la cadena	Real
CI	Ciudades donde tiene tiendas	Entero
TI	Número de tiendas por ciudad	Entero
ST	Venta en cada ciudad	Real
SE	Venta en cada tienda	Real
VE	Venta realizada por empleado	Real
EM	Número de empleados	Entero
I, J, K	Contadores de ciclo	Entero

Tabla 4.15 Variables utilizadas para determinar los montos de ventas por empleado, por tienda, por ciudad y el total.

Problemas propuestos

- 4.1 Un profesor tiene un salario inicial de \$1500, y recibe un incremento de 10 % anual durante 6 años. ¿Cuál es su salario al cabo de 6 años? ¿Qué salario ha recibido en cada uno de los 6 años? Realice el algoritmo y represente la solución mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S, utilizando el ciclo apropiado.
- 4.2 “El náufrago satisfecho” ofrece hamburguesas sencillas (S), dobles (D) y triples (T), las cuales tienen un costo de \$20, \$25 y \$28 respectivamente. La empresa acepta tarjetas de crédito con un cargo de 5 % sobre la compra. Suponiendo que los clientes adquieren N hamburguesas, las cuales pueden ser de diferente tipo, realice un algoritmo para determinar cuánto deben pagar. Represéntelo en diagrama de flujo, pseudocódigo y diagrama N/S.
- 4.3 Se requiere un algoritmo para determinar, de N cantidades, cuántas son cero, cuántas son menores a cero, y cuántas son mayores a cero. Realice el diagrama de flujo, el pseudocódigo y el diagrama N/S para representarlo, utilizando el ciclo apropiado.
- 4.4 Una compañía fabrica focos de colores (verdes, blancos y rojos). Se desea contabilizar, de un lote de N focos, el número de focos de cada color que hay en existencia. Desarrolle un algoritmo para determinar esto y represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S, utilizando el ciclo apropiado.
- 4.5 Se requiere un algoritmo para determinar cuánto ahorrará en pesos una persona diariamente, y en un año, si ahorra 3¢ el primero de enero, 9¢ el dos de enero, 27¢ el 3 de enero y así sucesivamente todo el año. Represeñe la solución mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S, utilizando el ciclo apropiado.
- 4.6 Resuelva el problema 4.1, mediante: a) un ciclo Repite y b) un ciclo Desde.
- 4.7 Resuelva el problema 4.2, mediante: a) un ciclo Mientras y b) un ciclo Desde.
- 4.8 Realice el algoritmo para determinar cuánto pagará una persona que adquiere N artículos, los cuales están de promoción. Considere que si su precio es mayor o igual a \$200 se le aplica un descuento de 15%, y si su precio es mayor a \$100 pero menor a \$200, el descuento es de 12%; de lo contrario, sólo se le aplica 10%. Se debe saber cuál es el costo y el descuento que tendrá cada uno de los artículos y finalmente cuánto se pagará por todos los artículos obtenidos. Represeñe la solución mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S.

- 4.9 Un cliente de un banco deposita equis cantidad de pesos cada mes en una cuenta de ahorros. La cuenta percibe un interés fijo durante un año de 10 % anual. Realice un algoritmo para determinar el total de la inversión final de cada año en los próximos N años. Represente la solución mediante el diagrama de flujo, el pseudocódigo y diagrama N/S.
- 4.10 Los directivos de equis escuela requieren determinar cuál es la edad promedio de cada uno de los M salones y cuál es la edad promedio de toda la escuela. Realice un algoritmo para determinar estos promedios y represente la solución mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S.
- 4.11 Realice un algoritmo y represéntelo mediante un diagrama de flujo para obtener una función exponencial, la cual está dada por:
- $$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots,$$
- 4.12 Se desea saber el total de una caja registradora de un almacén, se conoce el número de billetes y monedas, así como su valor. Realice un algoritmo para determinar el total. Represente la solución mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S.
- 4.13 Un vendedor ha realizado N ventas y desea saber cuántas fueron por 10,000 o menos, cuántas fueron por más de 10,000 pero por menos de 20,000, y cuánto fue el monto de las ventas de cada una y el monto global. Realice un algoritmo para determinar los totales. Represente la solución mediante diagrama de flujo, pseudocódigo y diagrama N/S.
- 4.14 Realice un algoritmo para leer las calificaciones de N alumnos y determine el número de aprobados y reprobados. Represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S.
- 4.15 Realice un algoritmo que determine el sueldo semanal de N trabajadores considerando que se les descuenta 5% de su sueldo si ganan entre 0 y 150 pesos. Se les descuenta 7% si ganan más de 150 pero menos de 300, y 9% si ganan más de 300 pero menos de 450. Los datos son horas trabajadas, sueldo por hora y nombre de cada trabajador. Represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S.
- 4.16 Realice un algoritmo donde, dado un grupo de números naturales positivos, calcule e imprima el cubo de estos números. Represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S.
- 4.17 Realice un algoritmo para obtener la tabla de multiplicar de un entero K comenzando desde el 1. Represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S.
- 4.18 En 1961, una persona vendió las tierras de su abuelo al gobierno por la cantidad de \$1500. Suponga que esta persona ha colocado el dinero en una cuenta de ahorros que paga 15% anual. ¿Cuánto vale ahora su inversión? $P(1+i)^n$. Realice un algoritmo para obtener este valor y represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S.
- 4.19 El gerente de una compañía automotriz desea determinar el impuesto que va a pagar por cada uno de los automóviles que posee, además del total que va a pagar por cada categoría y por todos los vehículos, basándose en la siguiente clasificación:

Los vehículos con clave 1 pagan 10% de su valor.

Los vehículos con clave 2 pagan 7% de su valor.

Los vehículos con clave 3 pagan 5% de su valor.

Realice un algoritmo para obtener la información y represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S. Los datos son la clave y costo de cada uno.

- 4.20 Realice un algoritmo para obtener el seno de un ángulo y represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S.

$$\text{Sen } x = (x - x^3/3! + x^5/5! - x^7/7! + \dots)$$

- 4.21 Realice un algoritmo para determinar qué cantidad de dinero hay en un monedero, considerando que se tienen monedas de diez, cinco y un peso, y billetes de diez, veinte y cincuenta pesos. Represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S.

- 4.22 El banco “Bandido de peluche” desea calcular para cada uno de sus N clientes su saldo actual, su pago mínimo y su pago para no generar intereses. Además, quiere calcular el monto de lo que ganó por concepto interés con los clientes morosos. Los datos que se conocen de cada cliente son: saldo anterior, monto de las compras que realizó y pago que depositó en el corte anterior. Para calcular el pago mínimo se considera 15% del saldo actual, y el pago para no generar intereses corresponde a 85% del saldo actual, considerando que el saldo actual debe incluir 12% de los intereses causados por no realizar el pago mínimo y \$200 de multa por el mismo motivo. Realice el algoritmo correspondiente y represéntelo mediante diagrama de flujo y pseudocódigo.

UNIDAD V
INTRODUCCIÓN A LOS ARREGLOS
UNIDIMENSIONALES Y MULTIDIMENSIONALES
(VECTORES Y MATRICES)

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

Introducción El planteamiento de algoritmos para la solución de problemas partió de solucionar problemas secuencialmente lineales, para luego llegar a los de ciclo. Si se comparan, se puede establecer que los secuenciales presentan la solución para un solo caso, mientras que en los de ciclo se repite N veces el procedimiento, que necesariamente es el mismo. En ambos casos, al realizar la captura o calcular el valor de una variable para un nuevo caso, los valores del anterior se pierden, debido a que el nuevo lo sustituye, pues se guardan en la memoria en una posición determinada. Por consiguiente, contar con estructuras dimensionales para las variables resulta muy apropiado y de gran utilidad, a esas estructuras se les denomina vectores o matrices, basándose en la dimensión con la que se trabaja; este tipo de arreglos permite guardar una serie de valores bajo el mismo nombre de la variable y al mismo tiempo. Para lograr esto, al nombre de la variable se le agrega entre corchetes uno, dos o varios subíndices, los cuales hacen referencia a la posición que guarda el dato dentro del arreglo. El número de subíndices hace referencia a la dimensión que tendrá el arreglo, por lo general se utilizan uno o dos, y en ocasiones hasta tres; sin embargo, podrían utilizarse más de tres, pero a medida que aumenta el número de dimensiones, aumenta la complejidad de los mismos, y como consecuencia, cambia la forma de trabajar con ellos, y en ocasiones es más complicado entenderlos.

Por lo tanto, se debe entender como arreglo a una estructura en la que se almacena una colección de datos del mismo tipo (ejemplo: las calificaciones de los alumnos de un grupo, sus edades, sus estaturas, etcétera). Estos arreglos se caracterizan por:

- 1) Almacenar sus elementos en una posición de memoria continua.
- 2) Tener un único nombre de variable.
- 3) Tener acceso directo o aleatorio a los elementos individuales del arreglo.
- 4) Tener homogéneos sus elementos.

En los diferentes lenguajes de programación, al momento de declarar las variables tipo arreglo, se deben establecer el tamaño y tipo de estas variables, o lo que es lo mismo, se debe determinar cuántos elementos y de qué tipo podrán almacenarse con el mismo nombre del dato.

A los arreglos, cuando son unidimensionales, se les denomina vectores o listas; cuando son multidimensionales, se les da el nombre de matrices o tablas.

Arreglos unidimensionales (vectores)

Los vectores son arreglos que contienen un solo índice que indica la posición que guarda el dato dentro del arreglo, esa posición es la física; algunos lenguajes de programación hacen referencia a la primera posición como lógica, de tal forma que se establece como la posición cero, de esta manera ésa es la posición lógica y no la física. En este libro, en la solución de los problemas se utilizará la posición física. Para fundamentar esto se analizará el ejemplo 5.1, mediante el cual se establecen las bases para la solución de problemas de este tipo.

Ejemplo 5.1

Suponga que tiene las edades de cuatro alumnos; si no cuenta con un arreglo o estructura de datos tipo vector, al trabajar con estos valores al mismo tiempo, tendría que definir cuatro variables para almacenar cada una de las edades en la memoria de la máquina, sin embargo, con un vector es posible guardar estas edades en una misma variable, y tener acceso a ella en cualquier momento. En la figura 5.1 se representará mediante un esquema cómo estarían integrados estos elementos dentro del arreglo.

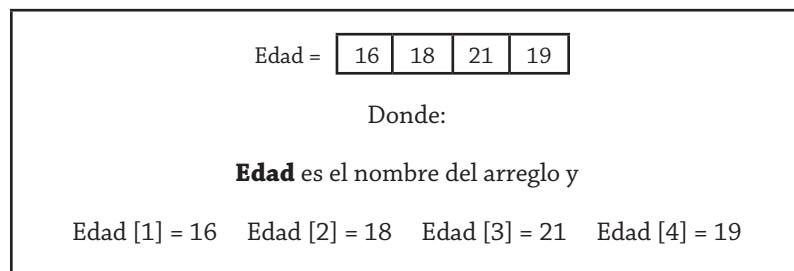


Figura 5.1 Forma en que se integran los elementos de un vector.

Partiendo del planteamiento del problema 5.1, si en lugar de tener sólo cuatro edades se tuvieran todas las edades de los alumnos de una escuela o, aún más complejo, se tratara de las edades de los habitantes de una ciudad, o alguna situación semejante, utilizar variables simples resultaría, si no imposible, sí lo bastante complejo para manipular las N variables por utilizar para guardar los datos correspondientes; no obstante, con un vector se pueden almacenar estos datos en una misma variable, en la que sólo se hace referencia a la posición que ocupa dentro del arreglo.

La forma de representar la captura e impresión en un diagrama de flujo y el respectivo pseudocódigo de un vector de N elementos se muestra en la figura 5.2.

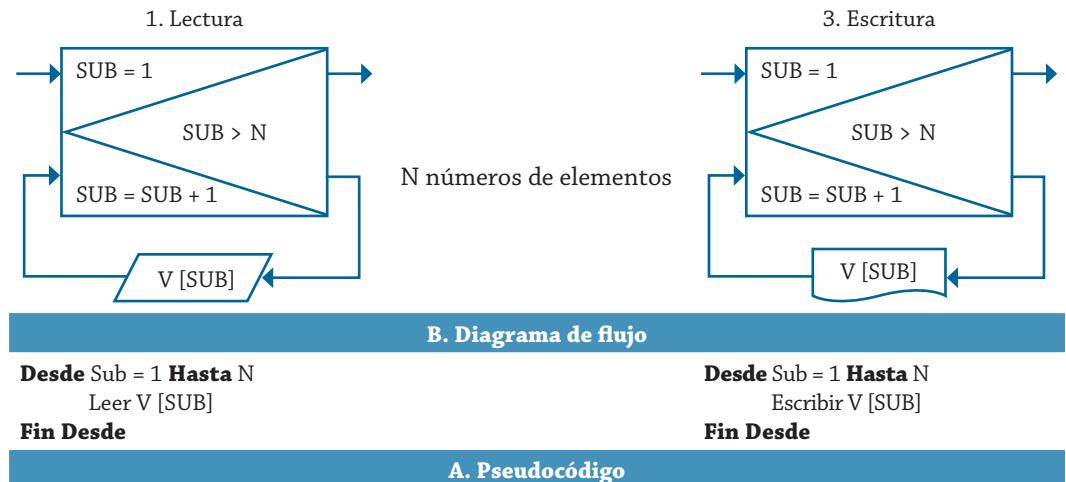


Figura 5.2 Cómo capturar e imprimir un vector de N elementos.

Con los elementos establecidos anteriormente, ahora se tiene la posibilidad de plantear algoritmos para la solución de problemas donde se requiera la utilización de variables tipo estructura, sin perder de vista que el control de un arreglo también se puede realizar mediante la utilización de un ciclo Mientras, o en su caso un Repite, esto basado en las necesidades o preferencias del programador.

Ejemplo 5.2

Se requiere obtener la suma de las cantidades contenidas en un arreglo de 10 elementos. Realice el algoritmo y represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S.

Con base en lo que se desea determinar, se puede establecer que las variables requeridas para la solución del problema son las mostradas en la tabla 5.1.

Nombre de la variable	Descripción	Tipo
I	Contador y subíndice	Entero
VA	Nombre del vector de valores	Real
SU	Suma de los valores	Real

Tabla 5.1 Variables utilizadas para obtener la suma de diez cantidades.

Para la solución de este problema se respetará el principio de que todo sistema tiene una entrada, un proceso y una salida. En consecuencia, el diagrama de flujo 5.1 muestra el algoritmo correspondiente para la solución de este problema, y de igual forma lo muestra el pseudocódigo 5.1 y el diagrama N/S 5.1.

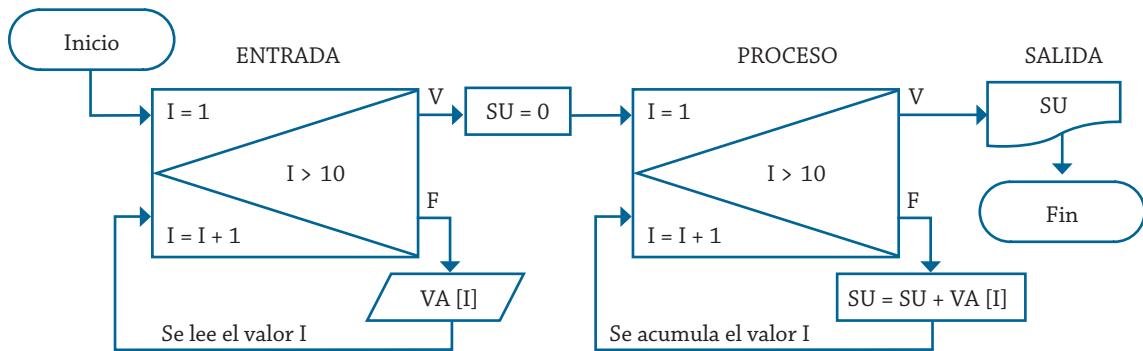


Diagrama de flujo 5.1 Algoritmo para obtener la suma de diez cantidades utilizando un vector.

1. Inicio
2. Desde $I = 1$ hasta $I = 10$
 Leer $VA[I]$ Fin desde
3. Hacer $SU = 0$
4. Desde $I = 1$ hasta $I = 10$
 Hacer $SU = SU + VA[I]$
 Fin desde
5. Escribir SU
6. Fin

Pseudocódigo 5.1 Algoritmo para obtener la suma de diez cantidades utilizando un vector.

Inicio
Desde $I = 1$ hasta 10
Leer $VA[I]$
Desde $I = 1$ hasta 10
Hacer $SU = SU + VA[I]$
Fin desde
Escribir SU
Fin

Diagrama N/S 5.1 Algoritmo para obtener la suma de diez cantidades utilizando un vector.

Como se puede ver en el diagrama 5.1, se indica la entrada, el proceso y la salida de manera especial, dado que cuando se diseña el algoritmo para un sistema complejo, seguir la regla de manejar lo más posible por separado estas partes de sistema permitirá que sea más fácil su mantenimiento y corrección, pero esto no quiere decir que no se puedan mezclar esos elementos, eso dependerá de las necesidades que se tenga para cada caso. En el diagrama de flujo 5.2, pseudocódigo 5.2 y diagrama N/S 5.2 se muestra una alternativa de solución haciendo la mezcla de estos elementos.

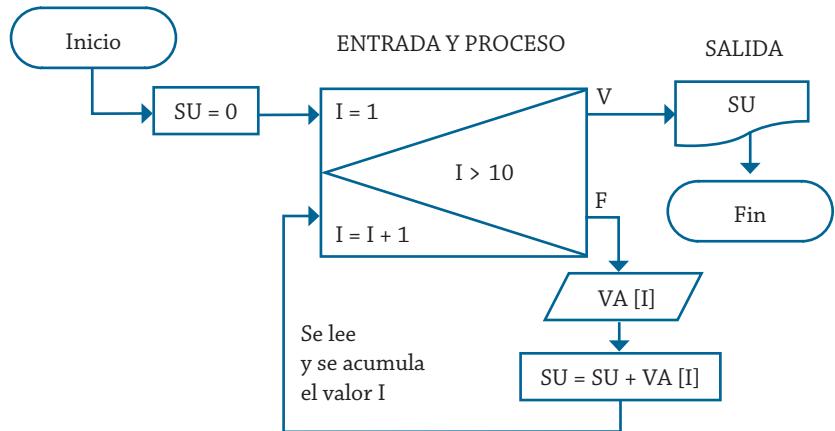


Diagrama de flujo 5.2 Algoritmo para obtener la suma de diez cantidades utilizando un vector.

1. Inicio
2. Hacer $SU = 0$
3. Desde $I = 1$ hasta $I = 10$
 - Leer $VA [I]$
 - Hacer $SU = SU + VA [I]$
 - Fin desde
4. Escribir SU
5. Fin

Pseudocódigo 5.2 Algoritmo para obtener la suma de diez cantidades utilizando un vector.

Inicio
Desde $I = 1$ hasta 10
Leer $VA [I]$
Hacer $SU = SU + VA [I]$
Fin desde
Escribir SU
Fin

Diagrama N/S 5.2 Algoritmo para obtener la suma de diez cantidades utilizando un vector.

Como se puede ver, las soluciones planteadas anteriormente son correctas, y en ambas, al momento de ir proporcionando los valores para el vector VA , éstos se van almacenando en la posición que tenga el subíndice I , ahora bien, si se trata de ahorrar código al momento de pasarlo a un lenguaje de programación en especial, la opción dos es la más adecuada; la primera alternativa es la que muestra más claridad en las partes que indican cómo está formado un sistema. Por consiguiente, el diseñador debe considerar qué es lo que desea, y optar por lo que más le convenga a sus propósitos.

En muchas ocasiones, el nombre del subíndice confunde a los que inician con el tratado de arreglos de este tipo, pero el nombre no debe

importar, sino el valor que toma en el momento que se utilice, esto se menciona porque en ocasiones se utiliza I en un ciclo y J en otro, y causa confusión acerca de si se hace referencia a la posición que se desea.

Ejemplo 5.3

Se requiere un algoritmo para obtener un vector (C) de N elementos que contenga la suma de los elementos correspondientes de otros dos vectores (A y B). Represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S.

La tabla 5.2 muestra las variables que se requieren para plantear el algoritmo que permita solucionar este problema.

Nombre de la variable	Descripción	Tipo
I	Contador y subíndice	Entero
A, B, C	Nombre de los vectores	Real
N	Número de elementos de cada arreglo	Entero

Tabla 5.2 Variables utilizadas para obtener un vector con la suma de los elementos correspondientes de otros dos vectores.

El diagrama de flujo 5.3 muestra el algoritmo correspondiente para la solución de este problema, también el pseudocódigo 5.3 y el diagrama N/S 5.3.

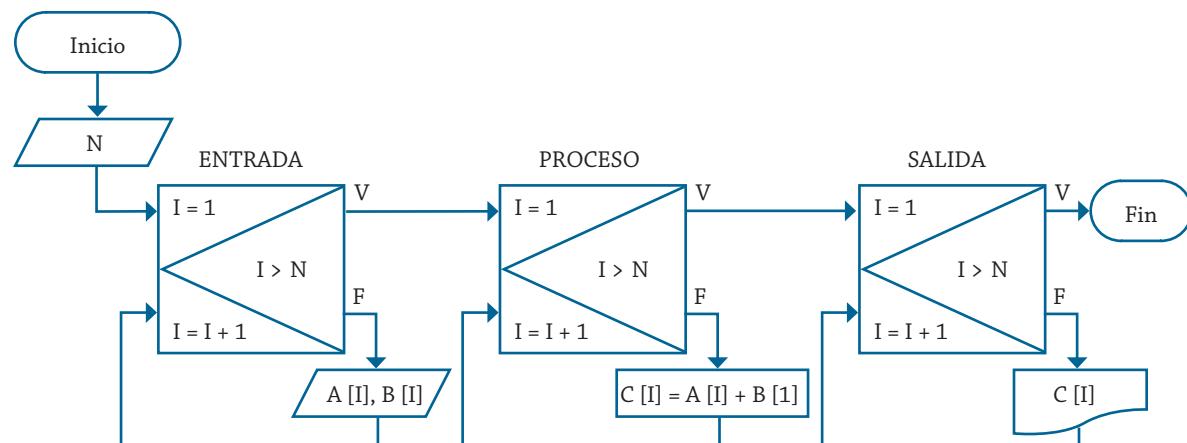


Diagrama de flujo 5.3 Algoritmo para obtener un vector con la suma de los elementos correspondientes de otros dos vectores.

Como se puede ver, el diagrama de flujo 5.3 cumple con lo establecido para un sistema en lo que respecta a sus partes (entrada, proceso y salida). En lo que respecta a la entrada, se puede observar que se capturan los valores de cada posición de los vectores A y B, la posición dentro del vector la marca el subíndice que está indicado por la variable I, la cual, a su vez, también sirve como controlador del ciclo Mientras. En lo que respecta a la parte de proceso, lo que se realiza es que se genera el vector C con la suma de los elementos de la posición correspondiente de los dos primeros vectores establecidos, mientras que para la salida el ciclo controla que se escriba cada uno de los elementos que se guardan en el vector C.

1. Inicio
2. Leer N
3. Desde I = N hasta I = N
 - Leer A [I], B [I]
 - Fin desde
4. Desde I = 1 hasta I = N
 - Hacer SU = A [I] + B [I]
 - Fin desde
5. Desde I = 1 hasta I = N
 - Escribir C [I]
 - Fin desde
6. Fin

Pseudocódigo 5.3 Algoritmo para obtener un vector con la suma de los elementos correspondientes de otros dos vectores.

De igual forma que en el problema anterior, se pueden realizar simultáneamente la entrada y el proceso sin ningún problema, incluso incluir la salida, todo en el mismo ciclo, pero esto no sería muy adecuado, dado que mezclaría la captura de datos con la escritura de resultados, lo cual no resulta muy estético ni recomendado, pues trae como consecuencia falta de claridad en los resultados.

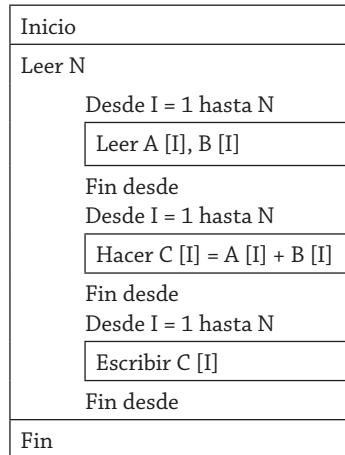


Diagrama N/S 5.3 Algoritmo para obtener un vector con la suma de los elementos correspondientes de otros dos vectores.

Ejemplo 5.4

Se tienen los nombres de los N alumnos de una escuela, además de su promedio general. Realice un algoritmo para capturar esta información, la cual se debe almacenar en arreglos, un vector para el nombre y otro para el promedio, después de capturar la información se debe ordenar con base en su promedio, de mayor a menor, los nombres deben corresponder con los promedios. Realice el algoritmo y represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S.

La tabla 5.3 muestra las variables requeridas para representar el algoritmo de solución de este problema.

Nombre de la variable	Descripción	Tipo
I, J	Contadores y subíndices	Entero
P	Vector de promedios	Real
N	Vector de nombres	String
A	Número de elementos de cada arreglo	Entero
PA	Variable auxiliar para promedio	Real
NA	Variable auxiliar de nombre de alumno	String

Tabla 5.3 Variables utilizadas para capturar los nombres y los promedios de N alumnos y ordenarlos de mayor a menor.

Por consiguiente, el diagrama de flujo 5.4 muestra el algoritmo para la solución de este problema. Como se puede ver, en la parte de entrada, se capturan los dos vectores, el de promedios y el de los nombres de cada alumno, mientras que en la parte del proceso se puede observar que se utilizan dos ciclos, uno para dar la posición del elemento que se comparará, y el otro para dar la posición del elemento con el cual se compara, dicho de otra forma, el primer elemento se compara contra el segundo, tercero y así sucesivamente. En caso de que se encuentre un elemento mayor en la comparación se hace un cambio de posición de los valores. En caso de que esto suceda, se almacena uno de los valores en una variable auxiliar para que no se pierda al momento de almacenar el nuevo valor que se almacenará en esa posición, al hacer esto se tienen valores repetidos en las dos posiciones que se están analizando, por lo que hay que cambiar el valor de la posición que resultó mayor por los valores almacenados en las variables auxiliares, de esta forma ya se habrá ordenado un valor mayor en una posición de subíndice menor. Pero si el valor de subíndice I es mayor que el valor de subíndice J, no se realiza ningún procedimiento, quedando los valores en su posición original. Por lo tanto, se puede ver que cuando I tiene un valor de uno, éste se compara contra todos los valores del arreglo a partir del valor dos, quedando finalmente el valor mayor en la posición uno, al incrementar el valor de I éste se compara contra todos los valores restantes del arreglo, esto se repite A veces hasta lograr que con este procedimiento se ordenen de mayor a menor los promedios y, por consecuencia, los nombres que les corresponden.

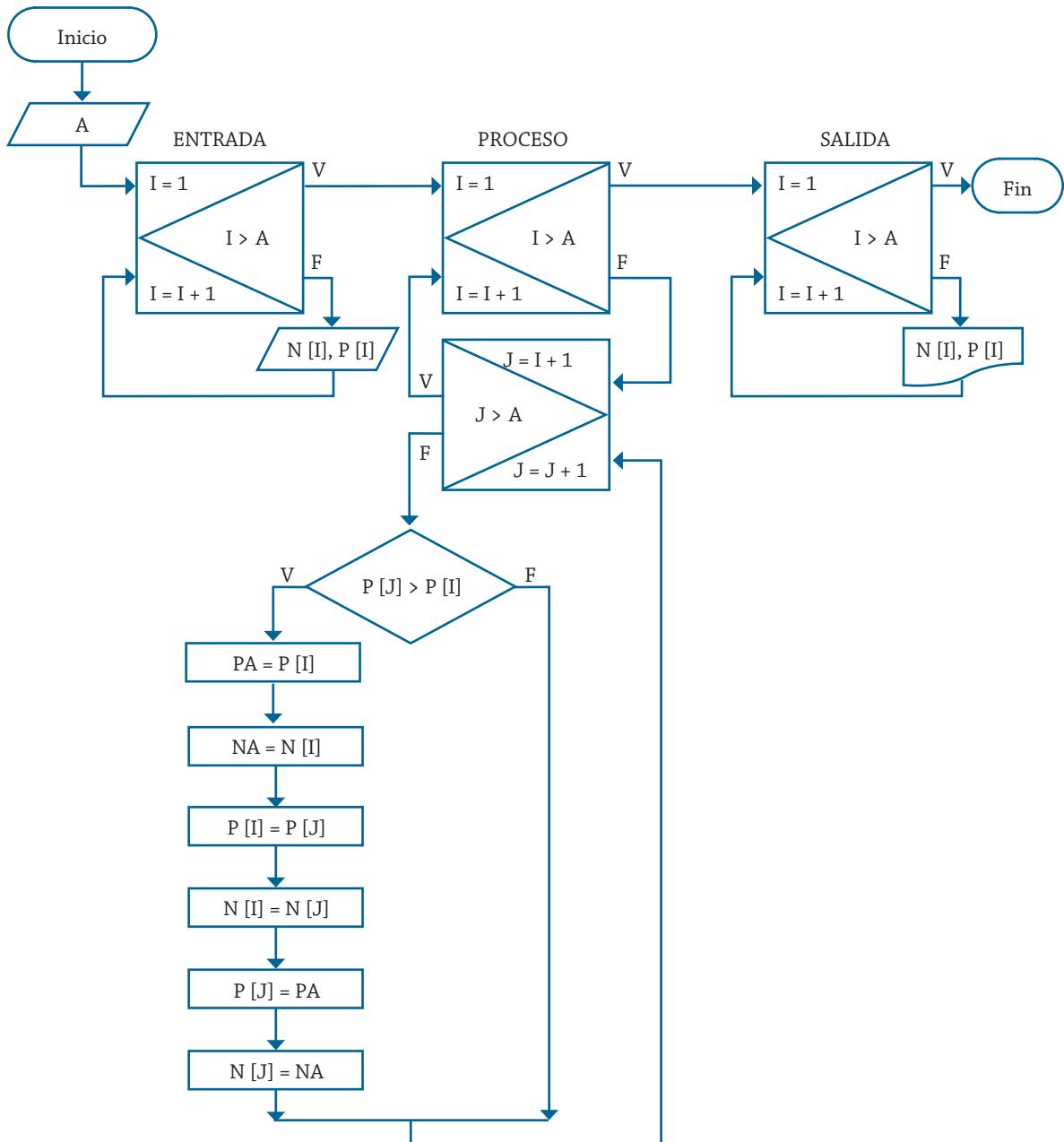


Diagrama de flujo 5.4 Algoritmo para ordenar de mayor a menor los promedios y nombres de N alumnos.

De igual forma, el pseudocódigo 5.4 muestra el algoritmo que corresponde a la solución de este problema y de igual forma el diagrama N/S 5.4 muestra la solución correspondiente mediante esta herramienta.

```

1. Inicio
2. Leer A
3. Desde I = 1 hasta I = A
   Leer N [I], P [I]
   Fin desde
4. Desde I = 1 hasta I = A
   Desde J = 1 hasta J = A
   Si P [J] > P [I]
   Entonces
      Hacer PA = P [I]
      Hacer NA = N [I]
      Hacer P [I] = P [J]
      Hacer N [I] = N [J]
      Hacer P [J] = PA
      Hacer N [J] = NA
   Fin compara
   Fin desde
   Fin desde
5. Desde I = 1 hasta I = A
   Escribir N [I], P [I]
   Fin desde
6. Fin

```

Pseudocódigo 5.4 Algoritmo para ordenar de mayor a menor los promedios y nombres de N alumnos.

Ejemplo 5.5

Cierta empresa requiere controlar la existencia de diez productos, los cuales se almacenan en un vector A, mientras que los pedidos de los clientes de estos productos se almacenan en un vector B. Se requiere generar un tercer vector C con base en los anteriores que represente lo que se requiere comprar para mantener el *stock* de inventario, para esto se considera lo siguiente: si los valores correspondientes de los vectores A y B son iguales se almacena este mismo valor, si el valor de B es mayor que el de A se almacena el doble de la diferencia entre B y A, si se da el caso de que A es mayor que B, se almacena B, que indica lo que se requiere comprar para mantener el *stock* de inventario. Realice el algoritmo y represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S.

La tabla 5.4 muestra las variables requeridas para representar el algoritmo de solución de este problema.

Nombre de la variable	Descripción	Tipo
I	Contador y subíndice	Entero
A	Vector de existencia	Entero
B	Vector de pedidos de clientes	Entero
C	Vector de requerimientos	Entero

Tabla 5.4 Variables utilizadas para obtener un vector con los pedidos requeridos para mantener un *stock* de existencias.

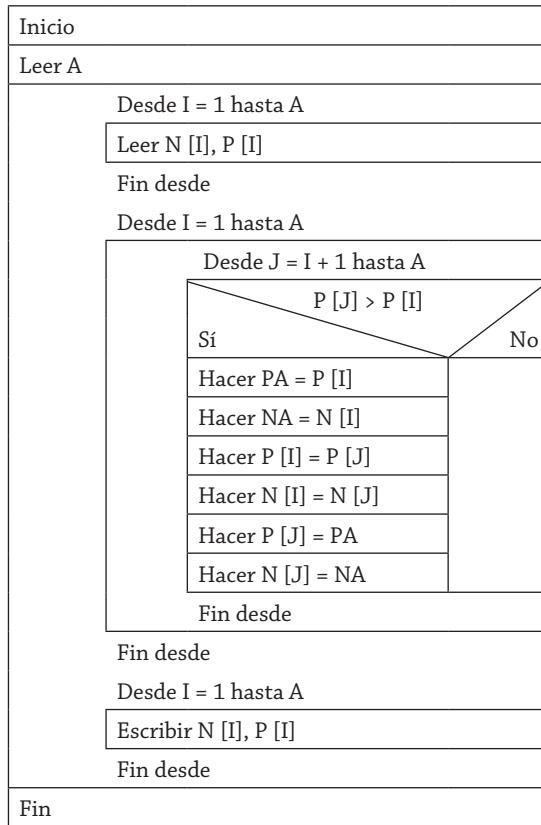


Diagrama N/S 5.4 Algoritmo para ordenar de mayor a menor los promedios y nombres de N alumnos.

Con base en lo anterior, se puede establecer el algoritmo de solución de la forma en que se muestra en el diagrama de flujo 5.5, donde se puede ver que se separa con claridad la entrada, el proceso y la salida, esto para respetar lo que se estableció desde un inicio en lo que se refiere a la definición de cómo está compuesto un sistema.

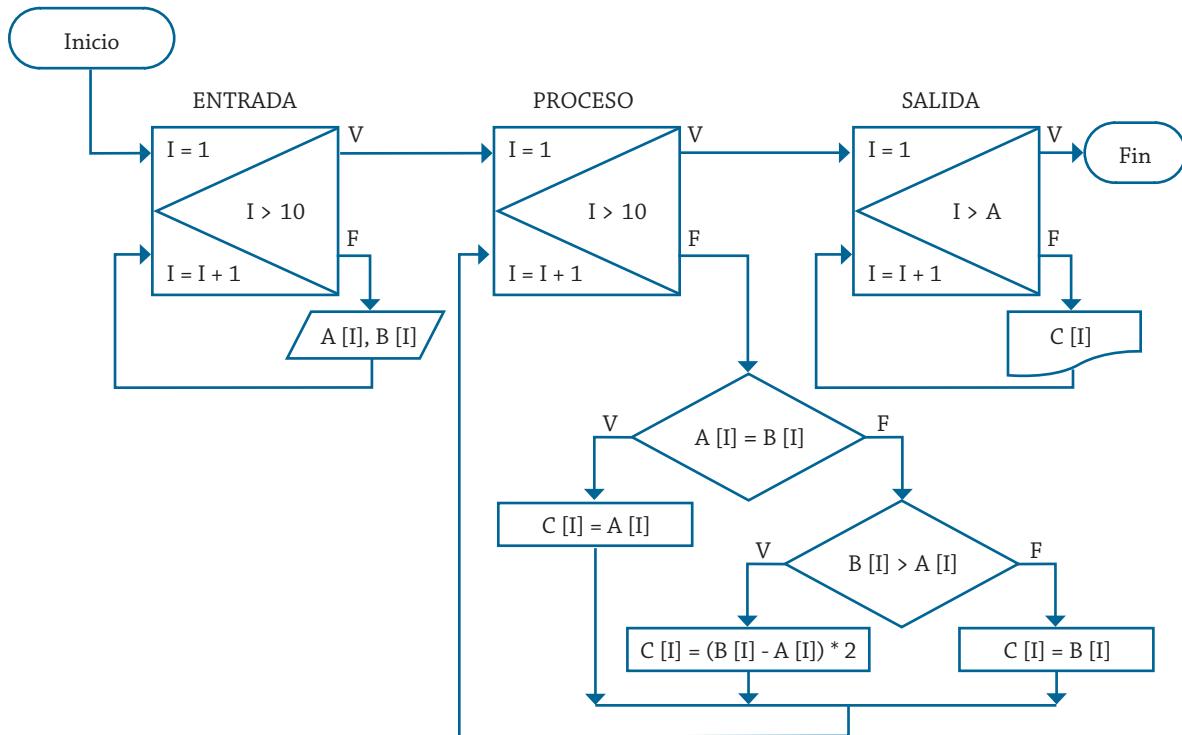


Diagrama de flujo 5.5 Algoritmo para obtener un vector con los pedidos requeridos para mantener un stock de existencias.

Ahora bien, el algoritmo se puede representar mediante el pseudocódigo 5.5, o bien con el diagrama N/S 5.5, que cumplen con las mismas características que el diagrama de flujo.

1. Inicio
2. Desde $I = 1$ hasta $I = 10$
 - Leer $A[I]$, $B[I]$
 - Fin desde
3. Desde $I = 1$ hasta $I = 10$
 - Si $A[I] = B[I]$
 - Entonces
 - Hacer $C[I] = A[I]$
 - Si no
 - Si $B[I] > A[I]$
 - Entonces
 - Hacer $C[I] = (B[I] - A[I]) * 2$
 - Si no
 - Hacer $C[I] = B[I]$
 - Fin compara
 - Fin compara
 - Fin desde
4. Desde $I = 1$ hasta $I = A$
 - Escribir $C[I]$
5. Fin

Pseudocódigo 5.5 Algoritmo para ordenar de mayor a menor los promedios y nombres de N alumnos.

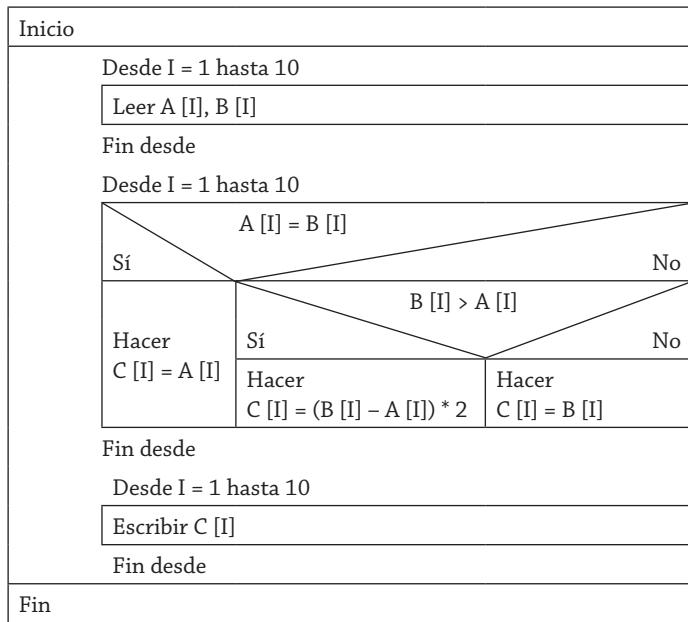


Diagrama N/S 5.5 Algoritmo para ordenar de mayor a menor los promedios y nombres de N alumnos.

Ejemplo 5.6

Realice un algoritmo que lea un vector de seis elementos e intercambie las posiciones de sus elementos, de tal forma que el primer elemento pase a ser el último y el último el primero, el segundo el penúltimo y así sucesivamente, e imprima ese vector. Represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S.

La tabla 5.5 muestra las variables requeridas para representar el algoritmo de solución de este problema.

Nombre de la variable	Descripción	Tipo
I	Contador y subíndice	Entero
J	Auxiliar para el subíndice	Entero
V	Vector de valores	Entero
AU	Auxiliar para guardar el valor de V	Entero

Tabla 5.5 Variables utilizadas para intercambiar de posición los elementos de un vector.

Ahora bien, el algoritmo solución para este problema se puede plantear de la forma en que se presenta en el diagrama de flujo 5.6.

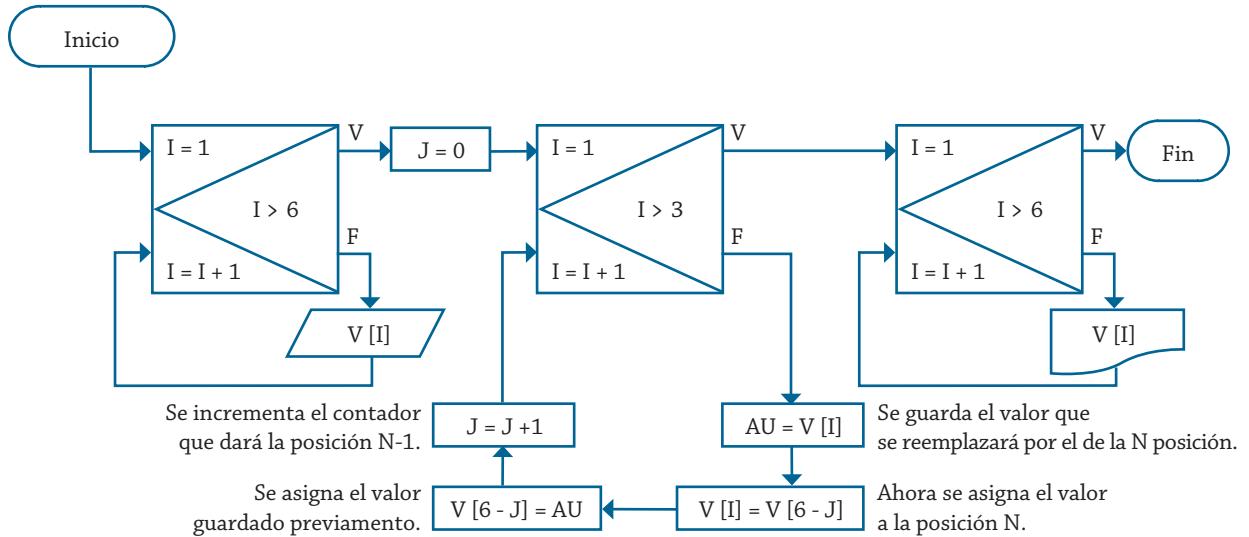


Diagrama de flujo 5.6 Algoritmo para intercambiar de posición los elementos de un vector.

Como se puede ver en esta solución planteada, de nueva cuenta se separa la Entrada del Proceso, en este caso en especial no se pueden combinar estas partes debido a la naturaleza del problema, y de igual forma, se pueden presentar muchos casos semejantes en los cuales es esencial realizar primero la captura de los datos para posteriormente procesarlos y presentar los resultados.

De igual modo, el algoritmo correspondiente se presenta en el pseudocódigo 5.6 y en el diagrama N/S 5.6.

1. Inicio
2. Desde $I = 1$ hasta $I = 6$
 - Leer $V[I]$
 - Fin desde
3. Hacer $J = 0$
4. Desde $I = 1$ hasta $I = 3$
 - Hacer $AU = V[I]$
 - Hacer $V[I] = V[6 - J]$
 - Hacer $V[6 - J] = AU$
 - Hacer $J = J + 1$
 - Fin desde
5. Desde $I = 1$ hasta $I = 6$
 - Escribir $V[I]$
 - Fin desde
6. Fin

Pseudocódigo 5.6 Algoritmo para intercambiar de posición los elementos de un vector.

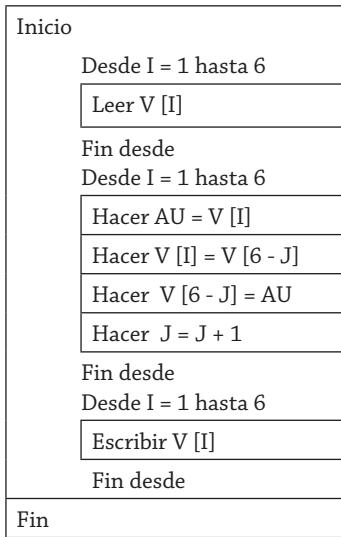


Diagrama N/S 5.6 Algoritmo para intercambiar de posición los elementos de un vector.

Para plantear la solución de este problema se partió de que los elementos del vector son seis, y con base en esto se puede establecer que el ciclo del proceso dura la mitad, o sea tres ciclos para lograr el resultado correcto, ahora bien, habrá casos en que las condiciones cambien, y por consiguiente, la solución también (ver ejemplo propuesto 5.1).

Arreglos bidimensionales (tablas)

Un arreglo bidimensional es un arreglo con dos índices, esto para localizar o almacenar un valor en el arreglo, por tal motivo se deben especificar dos posiciones (dos subíndices), uno para la fila y otro para la columna, a este tipo de arreglos indistintamente se les llama tablas o matrices. Para ejemplificar la forma en que están integradas y cómo se deben tratar veamos el ejemplo 5.1.

Ejemplo 5.1

Suponga que tiene tres calificaciones, de las cuatro que tres alumnos obtuvieron durante el período escolar. Esta información se puede almacenar de tal forma que los renglones representen las calificaciones de cada alumno, mientras que las columnas representen esas calificaciones, pero de cada materia en especial, esto lo podemos ver gráficamente en la figura 5.3.

		Materiales			
		1	2	3	4
Alumnos	Física	Química	Ética	Historia	
	1 Pepe	6	7	8	9
	2 Mary	6	9	7	9
	3 Chuy	8	9	7	6

CAL = CAL

CAL es el nombre del arreglo y

CAL [1,1] = 6 representa la calificación de Pepe en Física.
 CAL [2,2] = 9 representa la calificación de Mary en Química.
 CAL [3,4] = 6 representa la calificación de Chuy en Historia.

Figura 5.3 Forma en que se integran los elementos de un arreglo bidimensional (matriz).

Como se puede observar, el primer subíndice indica el renglón y el segundo la columna de la posición en que se encuentran los elementos correspondientes a la matriz de calificaciones llamada **CAL**. Como se puede ver, este arreglo muestra las calificaciones de cuatro materias comunes para tres alumnos diferentes, supongamos que no fuera posible guardar estos valores mediante un arreglo, nos obliga a pensar en el número de variables individuales que se requerirían para representar cada una de estas calificaciones, ahora bien, si pensáramos en N alumnos y M materias esto se complicaría $N \times M$ veces el número de variables requeridas para su representación, de aquí la importancia de poder contar con este tipo de arreglos para el tratamiento de datos. Por otro lado, la forma en que se realiza la captura e impresión de datos almacenados en un arreglo bidimensional se puede ver en la figura 5.4.

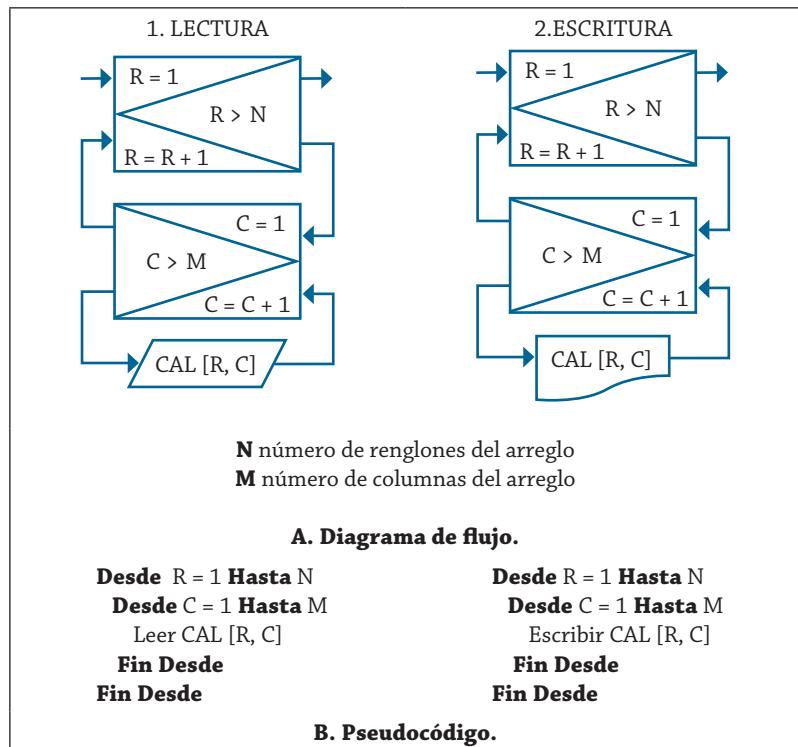


Figura 5.4 Forma de capturar e imprimir una matriz de $N \times M$ elementos.

Ejemplo 5.2

Se requiere determinar cuántos ceros se encuentran en un arreglo de cuatro renglones y cuatro columnas, las cuales almacenan valores comprendidos entre 0 y 9. Realice el algoritmo y represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S.

En la tabla 5.6 se muestran las variables que se requieren utilizar para generar el algoritmo solución de este problema.

Nombre de la variable	Descripción	Tipo
I	Contador y subíndice	Entero
J	Contador y subíndice	Entero
V	Nombre del arreglo de valores	Entero
NC	Contador de ceros en el arreglo	Entero

Tabla 5.6 Variables utilizadas para determinar el número de ceros en el arreglo.

El diagrama de flujo 5.7 muestra el algoritmo correspondiente para la solución a este planteamiento.

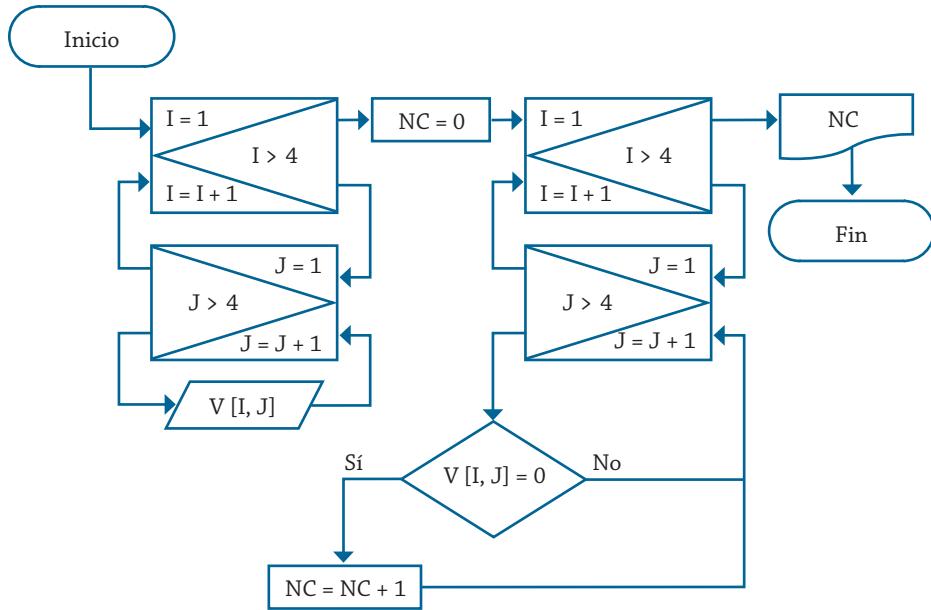


Diagrama de flujo 5.7 Algoritmo para determinar el número de ceros en el arreglo.

Como se puede ver en esta solución, se respetó de nueva cuenta la entrada, el proceso y la salida de manera independiente, con esto se puede dar más claridad a la solución planteada. De igual forma, el pseudocódigo 5.7 y el diagrama N/S 5.7 muestran el algoritmo solución para este problema.

1. Inicio
2. Desde $I = 1$ hasta $I = 4$
 - Desde $J = 1$ hasta $J = 4$
 - Leer $V [I, J]$
 - Fin desde
 - Fin desde
 - 3. Hacer $NC = 0$
 - 4. Desde $I = 1$ hasta $I = 4$
 - Desde $J = 1$ hasta $J = 4$
 - Si $V [I, J] = 0$
 - Entonces
 - Hacer $NC = NC + 1$
 - Fin compara
 - Fin desde
 - Fin desde
 - 5. Escribir NC
 - 6. Fin

Pseudocódigo 5.7 Algoritmo para determinar el número de ceros en el arreglo.

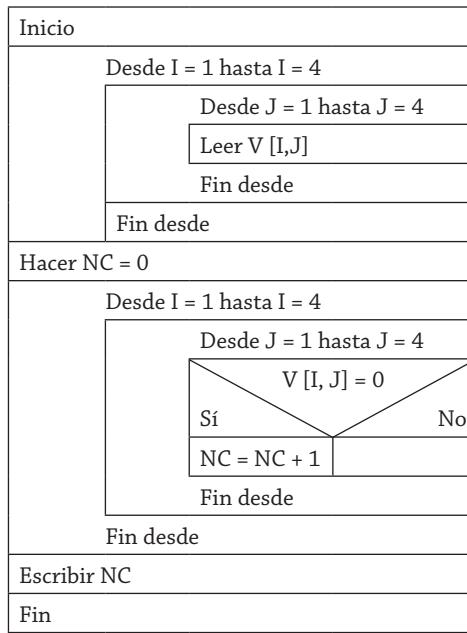


Diagrama N/S 5.7 Algoritmo para determinar el número de ceros en el arreglo.

Ejemplo 5.3

La empresa de transportes “The Big Old” cuenta con N choferes, de los cuales se conoce su nombre y los kilómetros que conducen durante cada día de la semana, esa información se guarda en un arreglo de $N \times 6$. Se requiere un algoritmo que capture esa información y genere un vector con el total de kilómetros que recorrió cada chofer durante la semana. Realice el algoritmo y represéntelo mediante el diagrama de flujo, el pseudocódigo y el diagrama N/S. Al final se debe presentar un reporte donde se muestre el nombre del chofer, los kilómetros recorridos cada día y el total de éstos, tal y como se muestra en la figura 5.5.

En la tabla 5.7 se muestran las variables que se requieren utilizar para generar el algoritmo solución de este problema. Y con las variables establecidas, el diagrama de flujo 5.8 muestra el algoritmo correspondiente para la solución a este planteamiento.

Nombre	Lun	Mar	Mié	Jue	Vie	Sáb	Tot K
NC ¹	K ^{1,1}	K ^{1,2}	K ^{1,3}	K ^{1,4}	K ^{1,5}	K ^{1,6}	TK ¹
NC ²	K ^{2,1}	K ^{2,2}	K ^{2,3}	K ^{2,4}	K ^{2,5}	K ^{2,6}	TK ²
:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:
NC ^{N-1}	K ^{N-1,1}	K ^{N-1,2}	K ^{N-1,3}	K ^{N-1,4}	K ^{N-1,5}	K ^{N-1,6}	TK ^{N-1}
NC ^N	K ^{N,1}	K ^{N,2}	K ^{N,3}	K ^{N,4}	K ^{N,5}	K ^{N,6}	TK ^N

Figura 5.5 Presentación de los datos y resultados requeridos.

Nombre de la variable	Descripción	Tipo
I	Contador y subíndice	Entero
J	Contador y subíndice	Entero
N	Número de choferes	Entero
K	Nombre del arreglo de kilómetros recorridos por día	Entero
NC	Nombre del arreglo con nombres de choferes	String
TK	Nombre del arreglo del total de kilómetros recorridos	Entero

Tabla 5.7 Variables utilizadas para determinar los kilómetros recorridos en la semana.

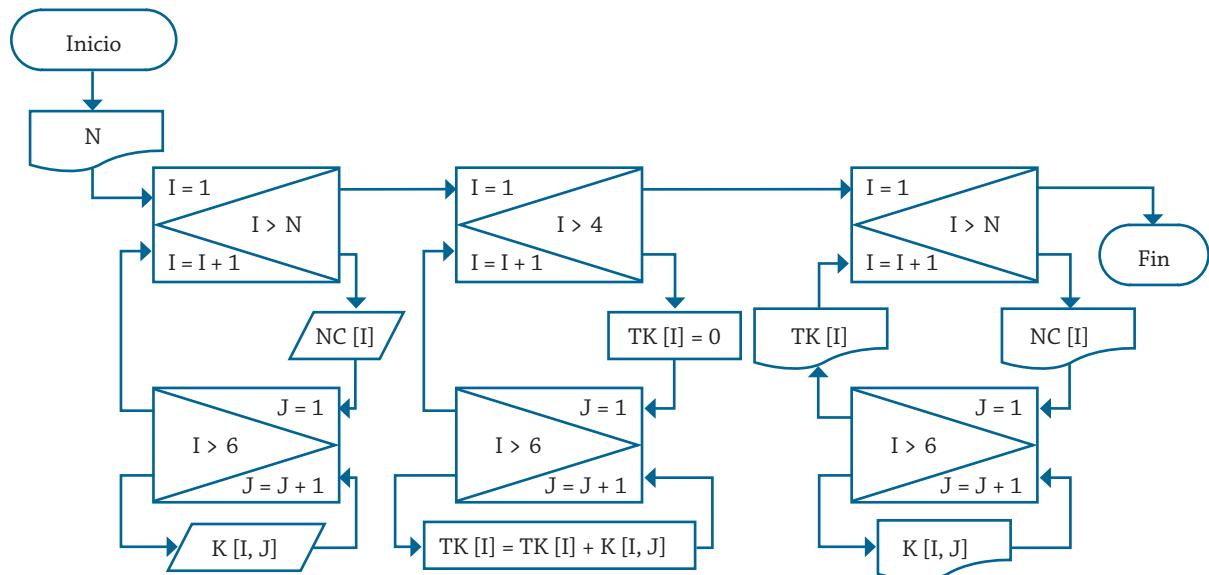


Tabla 5.7 Variables utilizadas para determinar los kilómetros recorridos en la semana.

Como se puede observar, en la parte de captura se inicia un ciclo, el cual se utiliza para dar un subíndice [I] a cada chofer, que permitirá almacenar en esta posición el respectivo nombre; posteriormente, se inicia otro ciclo, el cual se empleará para capturar los kilómetros recorridos por el chofer al día [J], por esta razón es que se solicita el valor K [I, J], que representa los kilómetros recorridos por el chofer I el día J.

De igual forma, en la parte del proceso, después de iniciar el primer ciclo se asigna el valor de cero al total de kilómetros recorridos por cada chofer, esto funciona para limpiar cualquier valor que pudiera estar almacenado en esa posición; posteriormente, se le integra uno a uno el valor correspondiente a cada día de la semana, que está denotado por la posición J, ya que como se mencionó anteriormente, la posición I corresponde al número o nombre del chofer respectivo.

En cuanto a la impresión o presentación de resultados, se puede observar que después de iniciar el primer ciclo se imprime el nombre del chofer correspondiente a la posición I, posteriormente se inicia el segundo ciclo, con el cual se podrán presentar los kilómetros recorridos cada día J por cada chofer I; al concluir el ciclo referente a los días de la semana se

imprime el total que se acumuló durante la semana para el chofer I. Al incrementar el valor de I se repetirá el proceso para el siguiente chofer, por consiguiente, este proceso permitirá generar una tabla de resultados como se solicitó previamente. Debe quedar claro que al momento de traducir este diagrama a cualquier lenguaje de programación es necesario considerar los formatos de impresión que cada lenguaje tiene.

Con base en lo anterior, el pseudocódigo 5.8 y el diagrama N/S 5.8 muestran la solución correspondiente. En ambos se podrá notar que la estructuración de la solución está bien definida en lo que respecta a las partes que integran cualquier sistema: entrada, proceso y salida.

```
1. Inicio.  
2. Leer N  
3. Desde I = 1 hasta I = N  
    Leer NC [I]  
        Desde J = 1 hasta J = 6  
        Leer K [I, J]  
        Fin desde  
    Fin desde  
4. Desde I = 1 hasta I = N  
    Hacer TK [I] = 0  
        Desde J = 1 hasta J = 6  
        Hacer TK [I] = TK [I] + K [I, J]  
        Fin desde  
    Fin desde  
5. Desde I = 1 hasta I = N  
    Escribir NC [I]  
        Desde J = 1 hasta J = 6  
        Escribir K [I, J]  
        Fin desde  
    Escribir TK [I]  
    Fin desde  
6. Fin
```

Pseudocódigo 5.8 Algoritmo para determinar el total de kilómetros recorridos en la semana.

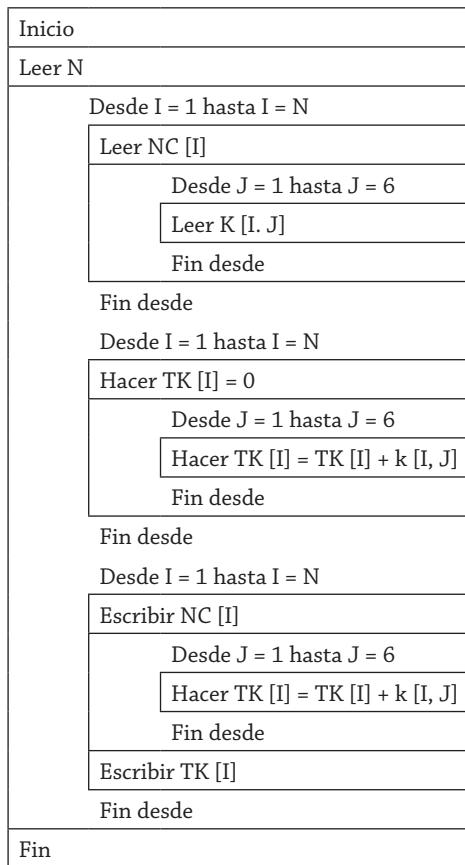


Diagrama N/S 5.8 Algoritmo para determinar el total de kilómetros recorridos en la semana.

Como se puede ver en las soluciones planteadas, cuando se respeta el principio de todo sistema, de separar en la medida de las posibilidades de diseño la entrada, el proceso y la salida, la consecuencia es la claridad y la facilidad en el manejo del algoritmo para posibles modificaciones en el momento de que cambien las condiciones del problema que se plantearon inicialmente.

Ejemplo 5.4

En un arreglo se tienen registradas las ventas de cinco empleados durante cinco días de la semana. Se requiere determinar cuál fue la venta mayor realizada. Realice un algoritmo para tal fin y represéntelo mediante diagrama de flujo, pseudocódigo y diagrama N/S.

Para resolver este problema se debe entender que en el arreglo al que se hace referencia, los renglones definen a los empleados y las columnas los días de la semana, en consecuencia, se trata de un arreglo de 5 x 5. Lo que se trata de encontrar es el valor mayor almacenado en el arreglo.

En la tabla 5.8 se muestran las variables que se requieren utilizar para generar el algoritmo solución de este problema.

Nombre de la variable	Descripción	Tipo
I	Contador y subíndice	Entero
J	Contador y subíndice	Entero
V	Nombre del arreglo de ventas	Entero
VA	Representa la venta mayor realizada	Entero

Tabla 5.8 Variables utilizadas para determinar la venta mayor de la semana.

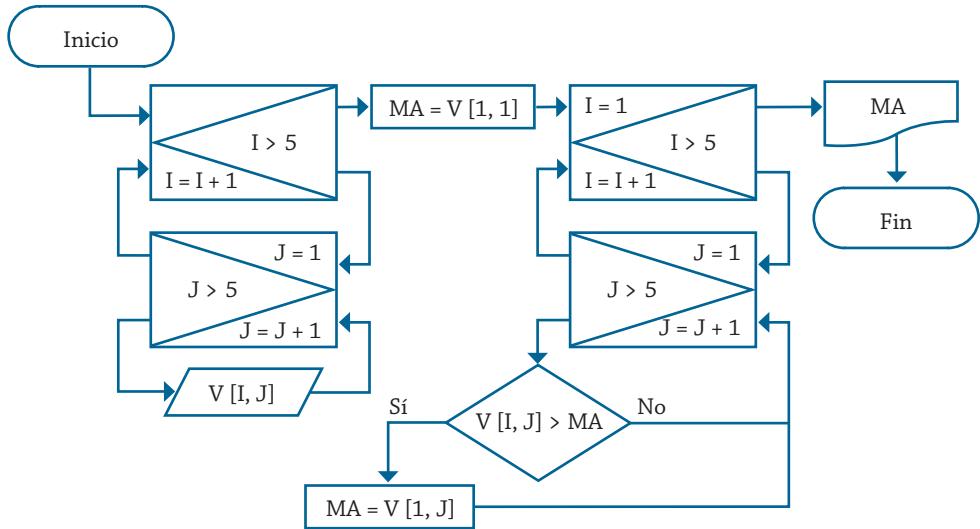


Diagrama de flujo 5.9 Algoritmo para determinar la venta mayor de la semana.

Como se puede ver, se asigna el valor del vendedor 1 del día 1 como venta máxima antes del proceso para determinar cuál de las ventas es la mayor, de igual forma, se asigna un valor de cero, ya que al comparar la venta de esta misma posición con el asignado, éste se verá reemplazado por este primer valor del arreglo.

El pseudocódigo 5.9 y el diagrama N/S 5.9 muestran el algoritmo de la solución correspondiente a este problema planteado.

1. Inicio
2. Desde $I = 1$ hasta $I = 5$
 - Desde $J = 1$ hasta $J = 5$
 - Leer $V[I, J]$
 - Fin desde
 - Fin desde
 - 3. Hacer $MA = V[1, 1]$
 - 4. Desde $I = 1$ hasta $I = 5$
 - Desde $J = 1$ hasta $J = 5$
 - Si $V[I, J] > MA$
 - Entonces
 - Hacer $MA = V[I, J]$
 - Fin comparar
 - Fin desde
 - Fin desde
 - 5. Escribir MA
 - 6. Fin

Pseudocódigo 5.9 Algoritmo para determinar la venta mayor de la semana.

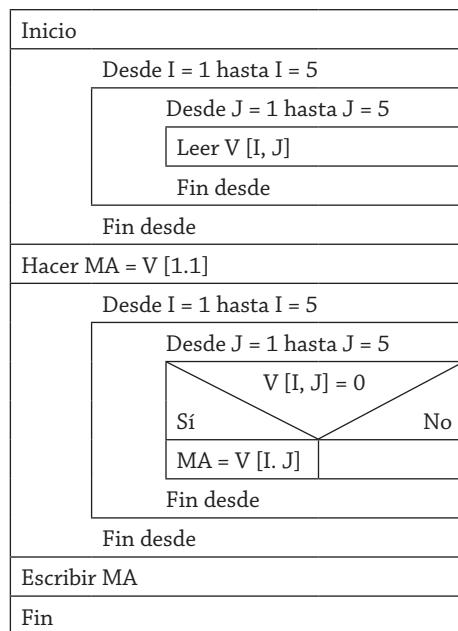


Diagrama N/S 5.9 Algoritmo para determinar la venta mayor de la semana.

Ejemplo 5.5

Realice un algoritmo para obtener una matriz como el resultado de la suma de dos matrices de orden M x N. Represéntelo mediante diagrama de flujo y pseudocódigo.

Nombre de la variable	Descripción	Tipo
I	Contador y subíndice	Entero
J	Contador y subíndice	Entero
A, B	Nombres de los arreglos por sumar	Entero
C	Nombre del arreglo resultante	Entero
M	Número de renglones del arreglo	Entero
N	Número de columnas del arreglo	Entero

Tabla 5.9 Variables utilizadas para obtener la suma de dos matrices.

Una vez que se establecieron las variables para elaborar el algoritmo, éste se puede representar mediante el pseudocódigo 5.10. De igual forma, el diagrama de flujo 5.10 y el diagrama N/S 5.10 muestran el algoritmo de solución para este problema.

1. Inicio.
2. Leer M, N
3. Desde I = 1 hasta I = M
Desde J = 1 hasta J = N
 Leer A [I, J], B [I, J]
 Fin desde
 Fin desde
4. Desde I = 1 hasta I = M
Desde J = 1 hasta J = N
 Hacer C [I, J] = A [I, J] + B [I, J]
 Fin desde
 Fin desde
5. Desde I = 1 hasta I = M
Desde J = 1 hasta J = N
 Escribir C [I, J]
 Fin desde
 Fin desde
6. Fin

Pseudocódigo 5.10 Algoritmo para obtener la suma de dos matrices.

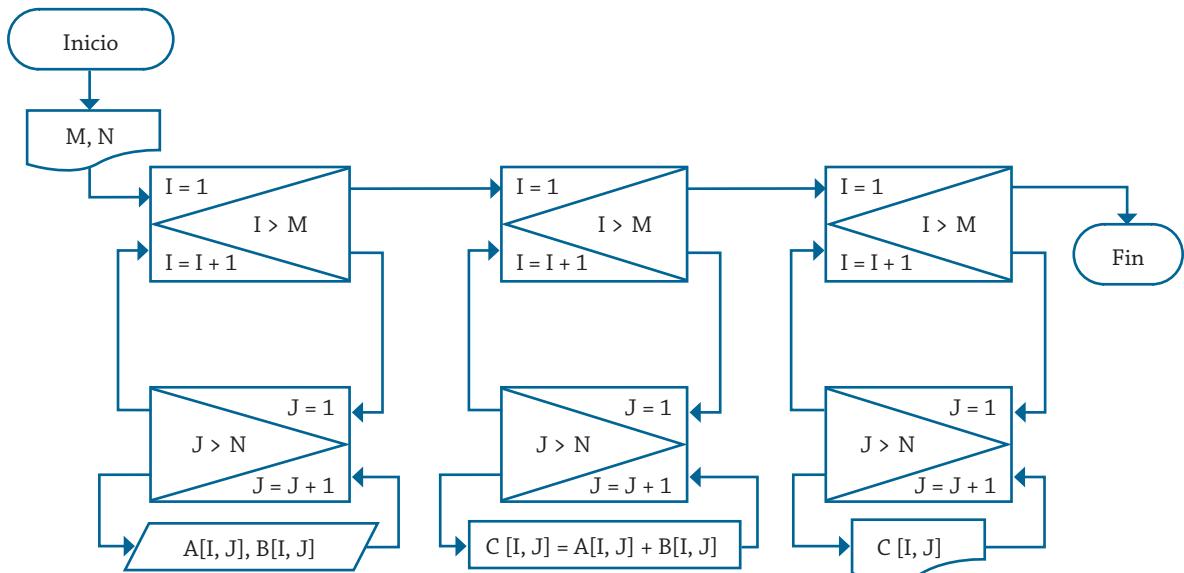


Diagrama de flujo 5.10 Algoritmo para obtener la suma de dos matrices.

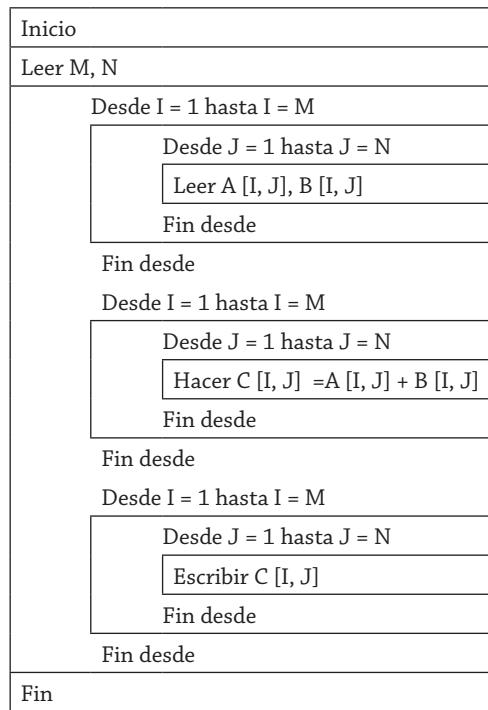


Diagrama N/S 5.10 Algoritmo para obtener la suma de dos matrices.

Como se puede observar, en salida sólo se presenta la matriz resultante. Al momento de implementar el resultado en algún lenguaje en especial se debe considerar su sintaxis, y en caso de que se requiera presentar las tres matrices, se debe poner atención sobre cómo ubicarlos en coordenadas de impresión que permiten manipular los mismos lenguajes.

Problemas propuestos

- 5.1 Realice y represente mediante un diagrama de flujo el algoritmo para obtener la matriz transpuesta de cualquier matriz de orden $M \times N$.
- 5.2 Realice y represente mediante un diagrama de flujo el algoritmo para obtener el producto de dos matrices de orden $M \times N$ y $P \times Q$.
- 5.3 Realice y represente mediante diagrama de flujo y pseudocódigo un algoritmo que lea un arreglo de M filas y N columnas y que calcule la suma de los elementos de la diagonal principal.
- 5.4 Realice un algoritmo para obtener una matriz como el resultado de la resta de dos matrices de orden $M \times N$. Represéntelo mediante diagrama de flujo y pseudocódigo.
- 5.5 Realice un diagrama de flujo que represente el algoritmo para determinar si una matriz es de tipo diagonal: es una matriz cuadrada en la cual todos sus elementos son cero, excepto los electos de la diagonal principal.
- 5.6 Realice y represente mediante diagrama de flujo y pseudocódigo un algoritmo que lea los nombres y las edades de diez alumnos, y que los datos se almacenen en dos vectores, y con base en esto se determine el nombre del alumno con la edad mayor del arreglo.
- 5.7 Modifique el problema del ejemplo 5.12, considerando que el vector tiene N elementos y que este número puede ser impar.

- 5.8 Realice un algoritmo que lea un vector y a partir de él forme un segundo vector, de tal forma que el primer elemento pase a ser el segundo, el segundo pase a ser el tercero, el último pase a ser el primero, y así sucesivamente. Represéntelo mediante un diagrama de flujo.
- 5.9 Se tiene un arreglo de 15 filas y 12 columnas. Realice un algoritmo que permita leer el arreglo y que calcule y presente los resultados siguientes:
- El menor elemento del arreglo; la suma de los elementos de las cinco primeras filas del arreglo; y el total de elementos negativos en las columnas de la quinta a la nueve.
- 5.10 Realice un algoritmo que lea dos vectores de cien elementos y que calcule la suma de éstos guardando su resultado en otro vector, el cual se debe presentar en forma impresa.
- 5.11 Se tienen dos matrices cuadradas (de 12 filas y 12 columnas cada una). Realice un algoritmo que lea los arreglos y que determine si la diagonal principal de la primera es igual a la diagonal principal de la segunda. (Diagonal principal es donde los subíndices I, J son iguales). Represeñe la solución mediante el diagrama de flujo y el pseudocódigo.
- 5.12 Se tiene una matriz de 12 filas por 19 columnas y se desea un algoritmo para encontrar todos sus elementos negativos y para que les cambie ese valor negativo por un cero. Realice un algoritmo para tal fin y represéntelo mediante diagrama N/S y pseudocódigo.
- 5.13 Se tiene en un arreglo cien elementos representando calificaciones de los estudiantes de una escuela. Realice un algoritmo que lea el arreglo y calcule la calificación promedio del grupo, además, que cuente los estudiantes que obtuvieron calificaciones arriba del promedio del grupo. Represéntelo mediante diagrama de flujo, diagrama N/S y pseudocódigo.
- 5.14 Realice un algoritmo que lea un vector de cien elementos y que calcule su magnitud y represéntelo mediante diagrama de flujo, diagrama N/S y pseudocódigo.
- 5.15 Realice un algoritmo que lea una matriz de cinco filas y seis columnas y que cuente los elementos negativos que contiene, así como también cuántos elementos de la diagonal principal son igual a cero. Represéntelo mediante diagrama de flujo, diagrama N/S y pseudocódigo.
- 5.16 Realice un algoritmo que calcule el producto de dos vectores. Uno de ellos es de una fila con diez elementos y el otro con una columna de diez elementos. Represéntelo mediante diagrama, diagrama N/S y pseudocódigo.
- 5.17 Una compañía de transporte cuenta con cinco choferes, de los cuales se conoce: nombre, horas trabajadas cada día de la semana (seis días) y sueldo por hora. Realice un algoritmo que:
- Calcule el total de horas trabajadas a la semana para cada trabajador.
 - Calcule el sueldo semanal para cada uno de ellos.
 - Calcule el total que pagará la empresa.
 - Indique el nombre del trabajador que labora más horas el día lunes.
 - Imprima un reporte con todos los datos anteriores.
- 5.18 Se tiene un arreglo de seis filas y ocho columnas y se sabe que se tiene un elemento negativo. Realice un algoritmo que indique la posición que ese elemento ocupa en el arreglo (en la fila y la columna en la que

se encuentra ese elemento). Represéntelo mediante diagrama, diagrama N/S y pseudocódigo.

- 5.19 Realice un algoritmo que lea una matriz de C columnas y R renglones. A partir de ella genere dos vectores que contengan la suma de sus renglones y la suma de sus columnas. Represéntelo mediante diagrama, diagrama N/S y pseudocódigo.
- 5.20 Realice un algoritmo que calcule el valor que se obtiene al multiplicar entre sí los elementos de la diagonal principal de una matriz de 5 por 5 elementos, represéntelo mediante diagrama, diagrama N/S y pseudocódigo.
- 5.21 Realice un algoritmo que a partir de la matriz del problema anterior encuentre cuántos elementos tienen valor par y cuántos valores impares. Represéntelo mediante diagrama, diagrama N/S y pseudocódigo.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

**PROBLEMARIO DE ALGORITMOS RESUELTOS
CON DIAGRAMAS DE FLUJO
Y PSEUDOCÓDIGO**

Primera edición 2014

El cuidado de la edición de este libro estuvo a cargo
del Departamento Editorial de la Dirección General de Difusión y Vinculación
de la Universidad Autónoma de Aguascalientes.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

EDICT._GUÍA_MEX - Editorial Guías México

Lectura 18. Panorámica de la teoría computacional: Desarrollo y problemas abiertos.

TRABAJO FIN DE MÁSTER

Facultad de Filosofía y Letras

Máster en Lógica y Filosofía de la Ciencia

Autor: Héctor Sanz Herranz

Tutor: Enrique Alonso González

*[Panorámica de la
teoría computacional:
Desarrollo y problemas
abiertos.]*



Universidad de Valladolid

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

Julio de 2017.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

PROLEGÓMENOS.

RESUMEN.

En este trabajo se exponen sendas introducciones a los conceptos propios de la teoría de la computación y de la teoría de la complejidad computacional, prestando especial atención a los movimientos filosóficos que propiciaron su desarrollo. Asimismo, se presenta una de las incógnitas sin resolver más importantes de la Matemática, la *conjetura de Cook*, así como sus consiguientes implicaciones al respecto de la naturaleza de los propios problemas.

ABSTRACT.

In this dissertation, two introductions are presented to the concepts of computational theory and the computational complexity theory paying special attention to the philosophical movements that led up to its development. Also presents one of the most important unresolved unknowns of Mathematics, the Cook's conjecture, as well as its consequent implications for the nature of the problems themselves.

TÉRMINOS CLAVE.

Clase de complejidad, ¿P = NP?, lenguaje formal, función computable, problema de decisión, programa de Hilbert, máquina de Turing.

KEY WORDS.

Complexity class, P = NP?, formal language, computable function, decision problem, Hilbert's program, Turing machine.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

AGRADECIMIENTOS.

Siempre he tratado de ser prudente a la hora de salvaguardar la intimidad de quienes se encuentran a mi alrededor e, incluso, la mía propia. Por otra parte, me he dado cuenta recientemente de que, aun cuando habitualmente dedico gran cantidad de tiempo a recopilar y estudiar las distintas cuestiones que acaparan mi interés, no termino de integrarlas a mi entendimiento hasta que no las comparto con alguien, lo cual puede resultar a menudo trágico para el resto dada la extraña naturaleza de las mismas. Por ello, mi agradecimiento más sincero será para quienes han decidido en alguna ocasión soportar en carnes propias mis pensamientos, ya sea prestándose para leer cuanto escribo, escuchar cuanto hablo o preguntar lo que creen que conozco.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

CONTENIDOS.

Introducción.	xi
1 Una introducción a la teoría de la computación.	1
1.1 Preludio matemático	3
1.2 La crisis de las matemáticas	5
1.3 El programa de Hilbert	7
1.4 Modelos computacionales	10
1.5 La máquina de Turing	15
1.6 Funciones computables y lenguajes	20
1.7 Referencias bibliográficas	23
2 Una introducción a la teoría de la complejidad.	27
2.1 ¿Qué es la complejidad?	29
2.2 Algunas nociones de la complejidad	30
2.3 Clases de complejidad	32
2.4 ¿P = NP?	35
2.5 Referencias Bibliográficas	37
Comentarios finales.	41
A Apéndice.	45
A.1 Sumar en distintos modelos computacionales	45
Bibliografía.	49

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

INTRODUCCIÓN.

La comunidad matemática no ha logrado alcanzar aún un consenso respecto a cuál ha de ser la labor primordial de los matemáticos. La tendencia predominante es la de contribuir a la ingente colección de resultados que se van acumulando en cada una de las ramas de la Matemática mientras que otros, por su parte, vuelcan sus esfuerzos en tratar de reconciliar algunas de ellas. Repárese en que se ha dado en llamar a Henri Poincaré como el *último matemático universal*, evidenciando la progresiva escisión del conocimiento matemático.

En cualquier caso, lo que es cierto es que los matemáticos trabajamos con *problemas*, esto es, los analizamos, los comprendemos, los revisamos, volvemos sobre ellos e, incluso, llegamos a desarrollar sentimientos encontrados hacia ellos. Sin embargo, nos estamos olvidando de lo más importante y es que, a menudo, también los *planteamos*. En efecto, no es nada excepcional que en el transcurso de una investigación, un matemático termine por proponer más interrogantes que aquellos a los que pretendía dar respuesta en un principio. De esta manera, es natural encontrarse con situaciones como la *paradoja de Russel*, el *teorema de Fermat* o la *conjetura de Cook*, donde los problemas toman el nombre del que para algunos es su *descubridor* y para otros, su *creador*.

De esta manera, existen dos posiciones antagónicas respecto al carácter ontológico de los problemas. Una de ellas defiende que los problemas gozan de una naturaleza objetiva aunque puede que no siempre exista una vía para *acceder* a ellos. En esta línea, se acepta que tanto su existencia como sus propiedades sean independientes de los sujetos racionales, quienes quedan relegados a un segundo plano pues resultan ser completamente contingentes. De esta manera, se entiende que se *descubre* un problema cuando se logra *acceder* a él. La otra posibilidad orbita en torno a los individuos y considera que su existencia es absolutamente necesaria para la de los *problemas*. En este sentido, los problemas se *crean* gracias al proceso cognoscitivo llevado a cabo por un sujeto racional con capacidad suficiente para ello. Esta parece ser la postura que habitualmente aceptamos en nuestra vida diaria. La Real Academia de la Lengua Española define *problema* como:

1. Cuestión que se trata de aclarar.
2. Proposición o dificultad de solución dudosa.
3. Conjunto de hechos o circunstancias que dificultan la consecución de algún fin.
4. Disgusto, preocupación.
5. Planteamiento de una situación cuya respuesta desconocida debe obtenerse a través de métodos científicos.

En efecto, se presupone la existencia de un sujeto que trata de aclarar o resolver una cuestión o dificultad (1, 2), o que pretende conseguir un determinado fin (3), o que está afecto de un disgusto o preocupación (4), o que plantea una cierta situación (5). Asimismo, podemos observar cómo se han introducido de soslayo ciertas nociones como las de *solución* o *dificultad* de un problema que resultan ser propiedades de los mismos. Aunque tales nociones nos resultan familiares es necesario realizar determinadas reflexiones al respecto antes de embarcarse en el estudio de las mismas del mismo modo que conviene elegir destino antes de pretender llegar a algún sitio pues, de otro modo, se corre el riesgo de no llegar a ninguna parte.

Como estábamos diciendo, a menudo asumimos que los problemas requieren de la existencia de individuos que los planteen o que los padeczan. Esta posición acaece en nuestra vida diaria cuando concedemos, por ejemplo, que para comprender totalmente el impacto de una determinada enfermedad es preciso haberla experimentado previamente. De hecho, conviene prestar atención a la connotación negativa de la palabra "problema". De esta manera, tendríamos que reconocer que las propiedades de los problemas dependen del sujeto racional al que conciernen y, puesto que la *dificultad* o *complejidad* de un problema resulta ser una de tales propiedades, también habrá de depender del mismo. Estudiar la complejidad de acuerdo con esta postura tiene complicaciones evidentes y quizás por ello los matemáticos habitualmente rehúsan cuestionar la naturaleza objetiva de los problemas. Como si de una escultura que trasciende a su autor se tratase, los problemas perduran mucho más que sus creadores. Por eso, aparentemente uno puede desprenderse de estos y analizar únicamente sus creaciones pero eso no debe implicar el dejar de ser conscientes de su carácter ontológico.

De ahora en adelante, nos ceñiremos únicamente a *problemas matemáticos* entendiendo estos bajo la acepción 5. de la definición general de problema, pero con la salvedad de no reconocer la necesidad de la existencia de un agente racional y aceptaremos, al menos inicialmente, que los problemas matemáticos disfrutan de una naturaleza objetiva y así también sus propiedades. De esta manera, corresponde a los matemáticos la labor de proponer una taxonomía para ellos y, para que esta sea suficientemente general, se ha de atender a las propiedades comunes a todos ellos: la *computabilidad* y la *complejidad*. Sin embargo, comprender fidedignamente tales conceptos resulta ser una ardua tarea que hemos de padecer (no olvidemos que, al fin y al cabo, estamos lidiando con problemas) y es precisamente el propósito de este trabajo el tratar de contribuir a ello.

En cuanto a la estructura de esta memoria conviene señalar que consta de dos partes diferenciadas. La primera de ellas versa sobre la *teoría de la computación*, siguiendo el trazado histórico de los acontecimientos que derivaron en su aparición, para acabar exponiendo el modelo computacional por antonomasia: la *máquina de Turing*. Respecto a la segunda, cabe apuntar que pretende constituir un acercamiento a la problemática de la *teoría de la complejidad* y, por ello, se ha tratado de evitar en medida de lo posible el incurrir en demasiados tecnicismos. Aun así, no hemos podido dejar de incluir sendas secciones que definen con el rigor debido las clases de complejidad más importantes. Por último, se acaba presentando la *conjetura de Cook*, uno de los problemas matemáticos

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

1

UNA INTRODUCCIÓN A LA *teoría de la computación.*

"La gente tiene estrellas que no son las mismas.
Para unos son guías, para otros luces pequeñas.
Pero para los que son sabios, son problemas."
— (El Principito) Antoine de Saint-Exupery.

Resumen: Se presentan los conceptos básicos de la *teoría de la computación* relatando la manera en que se ha llegado a ellos a partir de los problemas de la fundamentación de la matemática. Asimismo, se introducen distintos modelos computacionales y, en particular, se examina detalladamente el modelo de Turing.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

1.1. PRELUDIO MATEMATICO.

Antes de comenzar nuestra empresa será necesario disponer de las herramientas con las que vamos a trabajar. Por ello, hemos incluido esta sección que pretende introducir los conceptos básicos que utilizaremos a lo largo del texto.

Especial importancia van a tener los conjuntos no vacíos que son, además, finitos. A tales conjuntos los denominaremos *alfabetos* y a sus elementos, *letras*. Ahora estamos en condiciones de presentar lo que en este contexto vamos a entender por *palabra*.

Definición 1. (*Palabra, lenguaje*)

Dado un alfabeto, Σ , se dice que w es una palabra sobre Σ si, y solo si, w es una lista o cadena finita de letras de Σ . Formalmente, se representa como $w = w_1 \dots w_n$, donde $w_i \in \Sigma$, $i = 1, \dots, n$. Al número natural n se lo denomina longitud de la palabra, y se denota por $|w|$, mientras que al conjunto de todas las palabras de longitud exactamente n se lo denota por Σ^n . Por su parte, existe una única palabra de longitud nula, la palabra vacía, que se suele denotar por λ .

Al conjunto de todas las palabras sobre un alfabeto Σ se lo denota por Σ^* , y es posible identificarlos con la unión disjunta

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n.$$

A los conjuntos \mathbb{L} tales que $\mathbb{L} \subseteq \Sigma^*$ se los denomina lenguajes.

La noción que vamos a presentar a continuación va a resultar de vital importancia pues nos va a permitir comprender mejor el alcance de los resultados que se tratarán más adelante.

Definición 2. (*Codificar, código, codificación*)

Dados sendos alfabetos, Δ y Σ , codificar Δ en Σ^* consiste en dar una aplicación inyectiva $c : \Delta \rightarrow \Sigma$. En tal caso, se dice que Σ es el código y $c(\Delta)$ la codificación.

Cuando no se concrete la aplicación c utilizada, denotaremos por $\lfloor x \rfloor$ a la codificación de un cierto objeto x , es decir, identificaremos $\lfloor x \rfloor \equiv c(x)$. Lo más habitual será codificar en código binario, esto es, tomando $\Sigma = \{0, 1\}$. Demos paso a una nueva definición.

Definición 3. (*Función booleana, asignación booleana*)

Se dice que una función ψ es una función booleana de n argumentos si $\psi : \{0, 1\}^n \longrightarrow \{0, 1\}$. A cada elemento $z \in \{0, 1\}^n$ se lo llama asignación booleana.

Sin lugar a duda, la noción que va a jugar un papel central en este ensayo va a ser la de *algoritmo*. Si bien existen actualmente numerosos debates abiertos al respecto del concepto de *algoritmo*, comúnmente se acepta que este ha de ser un procedimiento definido mediante un conjunto finito de instrucciones que aplicado a una cantidad fija de elementos *input* produce eventualmente un elemento *output*. Durante el proceso de ejecución de un algoritmo se van generando distintos elementos conforme van aplicándose cada una de las instrucciones. Este proceso puede terminar debido a que no resten instrucciones por aplicar o, en cambio, puede no finalizar nunca. Por su parte, aunque actualmente se permite que existan algoritmos que solo aplican a lo sumo una instrucción en cada etapa (*deterministas*) y otros que gozan de más de una instrucción aplicable (*no deterministas*), conviene señalar que en lo que sigue, cuando utilicemos el término *algoritmo* nos referiremos únicamente a los deterministas salvo que se indique lo contrario.

Aunque existe cierto debate al respecto, habitualmente se acepta que los *inputs* y *outputs* de un algoritmo han de ser objetos finitos. De esta manera, no todos los números reales podrían servir de *input* a un algoritmo pues existen números cuyas infinitas cifras decimales no se siguen ningún comportamiento periódico (por ejemplo, el número π).

Asimismo, podría afirmarse que cada problema matemático no es más que una relación entre dos conjuntos. El primero, \mathfrak{A} , habría de ser el de todos los posibles *input* mientras que el segundo, \mathfrak{B} , consistiría en todas las posibles soluciones o *output*. Esta idea nos dirige inevitablemente a la noción de función. En efecto, podemos entender un problema como una función parcial ¹ $f : \mathfrak{A} \longrightarrow \mathfrak{B}$. Con esto en mente, se puede decir que un algoritmo materializa el problema y que guía al *procesador*, que puede ser un humano o una máquina, desde el *input* hasta el *output* mediante una secuencia finita de instrucciones. Asimismo, se dice que el procesador *computa* el problema cuando recorre tal secuencia de instrucciones.

Conviene señalar que la noción de *algoritmo*, al contrario de lo que ocurre con otros objetos matemáticos, no está libre de ambigüedades. Quizás sea consecuencia de que se puede acceder a ella siguiendo distintos planteamientos. En la sección venidera recorreremos uno que pretende plasmar la razón de ser de un algoritmo.

¹Se dice que una función $f : A \longrightarrow B$ es una *función parcial* si se permite que f pueda no estar definida para algunos elementos de A . Si f está definida para un cierto $a \in A$, se escribe $f(a) \downarrow$; en otro caso, $f(a) \uparrow$.

1.2. LA CRISIS DE LAS MATEMÁTICAS.

Idealmente, la manera en que la matemática adquiere conocimiento nuevo se basa en el denominado *método axiomático*. Partiendo de un conjunto de enunciados básicos que se denominan *axiomas*, el conocimiento se extiende vía generación sistemática de *proposiciones* que han de ser deducidas correctamente, esto es, siguiendo unas determinadas *reglas de deducción*. Así, para alcanzar una nueva proposición válida ha de recorrerse una secuencia finita de estados a la que usualmente se denomina *prueba*. Una proposición que goza de una prueba es lo que se denomina *teorema*. Al conjunto compuesto por los axiomas y los teoremas se lo conoce como *teoría*. Así, por definición, supondremos que todas las teorías son axiomatizables.

Desde los trabajos de Euclides hasta mediados del siglo XIX se requería que los axiomas fuesen enunciados cuya validez estuviese libre de cualquier duda, esto es, que resultasen *evidentes* a partir de la experiencia. Debido a este requerimiento, tácitamente se estaba asumiendo que los axiomas debían versar sobre *ideas*² cuya naturaleza objetiva estaba clara. De esta manera, los axiomas no necesitaban prueba alguna pues estaban claramente ratificados por la propia realidad. Sin embargo, a lo largo del siglo XIX surgieron serias dudas al respecto de qué era evidente y qué no, lo que llevó al abandono de los sistemas axiomáticos basados en la evidencia en beneficio de aquellos cuyos axiomas eran aceptados como meras *hipótesis*. Estos nuevos sistemas axiomáticos no cuestionaban en absoluto la naturaleza de los objetos que trataban. Así, por ejemplo, la axiomática de Peano propuesta en 1889 describía relaciones y propiedades relativas a los números naturales a partir de nociones más básicas sin entrar a valorar qué realidad material representa un número natural.

Una teoría erigida sobre un sistema axiomático hipotético resultó ser la *teoría de conjuntos* propuesta por el célebre matemático alemán Georg Cantor a finales del siglo XIX. El desarrollo de la teoría de conjuntos provocó que en la comunidad matemática cundiese el escepticismo. La reinterpretación del concepto de *infinito* trajo consigo numerosos resultados sumamente contraintuitivos y de lo más inesperados. Además, se había de permitir la extrapolación de razonamientos finitos al terreno de lo infinito. De hecho, el propio Cantor distinguía entre las nociones de *infinito actual* e *infinito potencial*. Por si fuera poco, en torno al año 1900, empezó a descubrirse que la teoría de conjuntos estaba afecta de numerosas *paradojas*³.

A partir de ese momento quedó claro que había que hacer algo para combatir las paradojas que habían ido apareciendo. El objetivo era realizar las modificaciones necesarias para lograr que se asentasen los fundamentos de las matemáticas y de otros métodos axiomáticos pero esta vez con garantías de no adolecer de paradojas. Las propuestas y críticas que se realizaron fueron substancialmente diferentes y, de todas ellas,

²De acuerdo con el punto de vista *platónico*, existe objetivamente un mundo no material en el que habitan las *ideas* abstractas y que es *accesible* únicamente a través de nuestro intelecto.

³Una paradoja es la situación en la que se alcanza una conclusión inaceptable a la que se llega siguiendo razonamientos aparentemente válidos. Las paradojas más conocidas que se descubrieron fueron las de Cantor, Russel y Burali-Forti.

las más importantes dieron lugar a tres corrientes de pensamiento: el *intuicionismo*, el *logicismo* y el *formalismo*.

El intuicionismo abogaba por una matemática fundamentada sobre la noción de *prueba*, apostando por una perspectiva no platonista en la que la existencia de los objetos estaba inherentemente ligada a la existencia de su constructibilidad mental. A partir de este principio, el intuicionismo logró recomponer algunas partes de la matemática clásica demostrando, además, que estaba desprovista de paradojas. Desafortunadamente, gran parte de las matemáticas no pudieron ser adaptadas a los principios intuicionistas.

Por su parte, el propósito del logicismo iba mucho más lejos pues este ansiaba fundamentar todas las matemáticas en la lógica. Los máximos exponentes de esta corriente fueron George Boole, Gottlob Frege, Giuseppe Peano, Bertrand Russell y Alfred Whitehead. Boole estaba preocupado por analizar rigurosamente los procesos de deducción lógica que venían empleándose desde Aristóteles. Sus investigaciones dimanaron en lo que se conoce actualmente como *cálculo proposicional*. Frege, por su parte, trató de demostrar que la aritmética era deducible a partir de la lógica. Concretamente, Frege planeaba definir las nociones propias de la aritmética valiéndose de las de la lógica y, al mismo tiempo, deducir los axiomas de la aritmética a partir de los de la lógica. Simultáneamente, Peano desarrolló otro lenguaje simbólico con capacidad suficiente para expresar las proposiciones propias de la matemática que incorporaba símbolos nuevos tales como ϵ . Se puede decir que entre Frege y Peano sentaron las bases de lo que hoy se conoce como *lógica de primer orden*. La meta a la que aspiraban Russell y Whitehead era aún más pretenciosa pues querían deducir absolutamente todas las matemáticas a partir de la lógica. Para evitar las paradojas o, al menos, algunas clases de paradojas, desarrollaron lo que se conoce como *teoría de tipos*, cuya exposición se recoge en su insigne obra, los *Principia Mathematica*.

Por otro lado, los formalistas no podían aceptar la radicalidad de las medidas adoptadas por los intuicionistas. Creían que era preciso conservar toda la matemática clásica pues, pese a todo, había demostrado su valía en innumerables ocasiones. Para lograrlo, los formalistas centraron su atención en el aspecto diametralmente opuesto al significado, la *semántica*, de las expresiones matemáticas, esto es, en la *sintaxis*. El formalismo fue iniciado por el que era, junto con Henri Poincaré, el matemático más famoso del momento, David Hilbert. En su opinión, las nociones sintácticas constituyan la única parte de las matemáticas que estaba libre de situaciones problemáticas, concretamente, la lógica elemental y cierta parte de la aritmética.

La idea principal del formalismo radicaba en el hecho de que, si bien numerosos conceptos matemáticos adolecían de ambigüedades que acababan por conducir a paradojas, es cierto que las nociones matemáticas siempre habían sido expresadas mediante *palabras* de algún lenguaje, bien sirviéndose de un lenguaje natural o de uno simbólico. El formalismo advirtió que las palabras no son más que una colección de *símbolos* (repárese en que esta idea originó la definición actual de *palabra* que se acepta hoy en día en el ámbito de las matemáticas ([Definición 1](#))) y que estos estaban libres de ambigüedad pues su comprensión es inmediata una vez que han sido reconocidos. De esta forma, el entendimiento de los símbolos resulta independiente de la semántica.

de las palabras. Por lo tanto, los formalistas pretendieron concebir las palabras como meras sucesiones de símbolos libres de significado. Más aún, puesto que las oraciones son sucesiones de palabras, también estas habrían de ser tratadas como secuencias de símbolos. La razón de esta postura se debe a que la sintaxis siempre goza del rigor necesario para no estar afecta de imprecisiones. Finalmente, incluso una prueba deductiva podría ser tratada como una secuencia finita de constructos del lenguaje completamente libre de vaguedades y de las consiguientes paradojas.

1.3. EL PROGRAMA DE HILBERT.

El *programa de Hilbert* constituyó un intento de reconstruir las matemáticas utilizando sistemas axiomáticos formales que no originen paradojas. Para poder comprenderlo con exactitud es preciso asimilar antes los problemas que trataban de solventarse. Tales problemas son los conocidos como *problemas de la fundamentación de la matemática* y su abordaje reveló la importancia del concepto de *algoritmo* provocando asimismo el nacimiento de la *teoría de la computación*.

El primero de los problemas que pretendían tratarse era el de la *consistencia*. Supongamos que T es una teoría y que $\alpha \in T$ tal que tanto α como $\neg\alpha$ son deducibles en T . Inmediatamente se sigue que la fórmula contradictoria $\alpha \wedge \neg\alpha$ es deducible. En tal caso, se dice que la teoría T es *inconsistente*. En el caso contrario, se dice que la teoría T es *consistente*. Puesto que una teoría inconsistente carece de interés debido a que permite deducir cualquier fórmula, lo sensato es pretender alcanzar una teoría que sea consistente. Por ello, también se dice que una teoría consistente es aquella para la que existe una fórmula que no es un teorema.

Otro de los problemas era el de la *completitud sintáctica*. Supongamos que T es una teoría consistente y sea $\alpha \in T$. Puesto que es consistente, α y $\neg\alpha$ no pueden ser simultáneamente deducibles en T . Pero nada impide en principio que ninguna de las dos sea deducible. En ese caso, se dice que α es *independiente* de T . Así, cuando al menos una de entre α y $\neg\alpha$ es deducible, se dice que la teoría T es *sintácticamente completa*.

El siguiente problema que se plateaba, y que es el que más ataña a las pretensiones de este ensayo, era el de la *decidibilidad*. Puesto que estamos entendiendo las teorías como las clausuras deductivas de conjuntos de axiomas, si una teoría es consistente y sintácticamente completa, a la fuerza habrá de ser también decidible. Volveremos sobre ello más adelante. Por su parte, la deducción de α o $\neg\alpha$ puede ser realmente compleja. Lo deseable sería que existiese un procedimiento finito, un algoritmo, que nos dijese si una fórmula α es deducible en T . A esta situación en la que uno se cuestiona la deducibilidad de α es lo que se conoce como *problema de decisión*. Cuando existe un método que permite responder efectivamente a todos los problemas de decisión que se pueden plantear en una teoría T se dice que T es *decidable*.

El último de los problemas fue el de la *completitud semántica*. Sea T una teoría

consistente. Se dice que una teoría es *correcta* si se verifica que

$$\vdash_T \alpha \implies \models_T \alpha \quad ^4 \tag{1.1}$$

para cualquier α . Notemos que esto es lo mínimo que le pedimos a una teoría, pues nunca aceptaríamos que pudiésemos probar algo que no sea cierto. El recíproco de (1.1) afirma que si una proposición es verdadera, entonces es deducible en la teoría, lo cual se formaliza como

$$\models_T \alpha \implies \vdash_T \alpha. \tag{1.2}$$

Cuando una teoría verifica (1.2) se dice que es *semánticamente completa*. La completitud semántica quizás sea la propiedad más importante de cuantas podemos exigirle a una teoría pues afirma que aquello que verdadero en una teoría también ha de ser un teorema de la misma.

Ahora que hemos expuesto los problemas de la fundamentación de la matemática estamos en condiciones de presentar en qué consistía el *programa de Hilbert*. Señalemos que la presentación que vamos a realizar está traducida a términos actuales.

Definición 4. (*Programa de Hilbert*)

El programa de Hilbert consistía en una empresa matemática que pretendía:

1. encontrar un sistema axiomático formal H capaz de deducir todos los teoremas de la matemática;
2. demostrar que H es consistente;
3. demostrar que H es semánticamente completo;
4. encontrar un algoritmo que resuelva los problemas de decisión de H .

El último de los propósitos descritos en la definición anterior es lo que Hilbert denominó *Entscheidungsproblem*. Él creía que si se lograba completar su programa, todas las matemáticas podrían desarrollarse mecánicamente pues cualquier proposición podría ser expresada mediante una fórmula α de H (existencia de H), α sería una verdad de la teoría si, y sólo si, fuese un teorema (completitud semántica). Como, además, solo una de α y $\neg\alpha$ sería un teorema (consistencia), bastaría acudir al algoritmo que el

⁴El formalismo introduce esta notación la cual refleja lo siguiente: $\vdash_T \alpha$ indica que α es un teorema de la teoría T ; $\models_T \alpha$ expresa que la fórmula α es verdadera en todo modelo de la teoría T .

Entscheidungsproblem dice que debería existir para dirimir cuál de ambas gozaría de tal categoría.

Inmediatamente después de que Hilbert propusiera su programa, los investigadores se pusieron manos a la obra con objeto de satisfacerlo. Si bien es cierto que los resultados relativos a 1. y 4. resultaban alentadores, pocos años después Gödel, al amparo de sus *teoremas de incompletitud*, acabaría por truncar el sueño de Hilbert tan solo unos días después de que este pronunciase su célebre "*debemos saber y sabremos*" [4]. Pero, parafraseando a Michael Ende, esa es otra historia que debe ser contada en otra ocasión.

Volviendo sobre el *Entscheidungsproblem* cabe señalar que su propósito no es otro que el de diseñar un método capaz de decidir si una fórmula arbitraria α es deducible en **H**. Como apuntamos en la sección anterior, los formalistas concebían las pruebas como meras secuencias finitas de símbolos, desarrolladas de acuerdo a unas determinadas reglas sintácticas. Por ello, un primer procedimiento podría ser el siguiente.

Procedimiento 1 : Búsqueda de una deducción de α .

Entrada: Una fórmula α .

Salida: **cierto** si se encuentra una prueba para α en **H**.

- 1: Generar una secuencia finita de símbolos, x .
 - 2: **mientras** x no es una prueba de α **hacer**
 - 3: Generar una nueva secuencia de símbolos $y \rightarrow x$.
 - 4: **si** x es una prueba de α . **entonces**
 - 5: **devolver cierto**
 - 6: **fin si**
 - 7: **fin mientras**
 - 8: **devolver cierto**
-

Debemos reparar que si la fórmula α es deducible en **H**, entonces el **Procedimiento 1** parece ser una forma razonable de encontrar efectivamente una prueba para la misma. Sin embargo, si α no resulta ser deducible, entonces el procedimiento previo no terminaría nunca pues continuaría indefinidamente generando secuencias de símbolos. Por otra parte, si sabemos *a priori* que la teoría **H** es sintácticamente completa, entonces si ocurriese que α no fuese deducible, si que habría de serlo en cambio $\neg\alpha$. Por lo tanto, el método anterior se puede mejorar de manera que termine siempre.

Procedimiento 2 : Búsqueda de una deducción de α o de $\neg\alpha$.

Entrada: Una fórmula α .

Salida: **cierto** si se encuentra una prueba para α en H , **falso** si se encuentra una prueba para $\neg\alpha$ en H .

- 1: Generar una secuencia finita de símbolos, x .
 - 2: **mientras** x no es una prueba de α **hacer**
 - 3: Generar una nueva secuencia de símbolos $y \longrightarrow x$.
 - 4: **si** x es una prueba de α . **entonces**
 - 5: **devolver cierto**
 - 6: **fin si**
 - 7: **si** x es una prueba de $\neg\alpha$. **entonces**
 - 8: **devolver falso**
 - 9: **fin si**
 - 10: **fin mientras**
 - 11: **devolver cierto**
-

Puesto que el **Procedimiento 2** termina siempre para una fórmula α cualquiera, podemos afirmar lo siguiente.

Resultado 1.

Si una teoría es consistente y sintácticamente completa, entonces es decidable.

El esquema al que nos arroja el **Procedimiento 2** se puede resumir en el hecho de que, planteado un problema de decisión, el método nos devuelve siempre una respuesta **cierto/falso**. Si ahora identificamos tales respuestas con 1/0 y reparamos en que la fórmula α puede ser codificada en binario, esto es, como una secuencia de 0's y 1's, es claro que el procedimiento puede ser reformulado en términos de funciones booleanas. Concretamente,

$$\begin{aligned} \psi : \{0, 1\}^n &\longrightarrow \{0, 1\} \\ \llcorner \alpha \lrcorner &\longrightarrow \psi(\llcorner \alpha \lrcorner) \end{aligned} \tag{1.3}$$

Pese al tono halagüeño de esta sección, como consecuencia de los *teoremas de incompletitud* de Gödel, no va ser posible desarrollar estos algoritmos de manera que siempre produzcan una salida. Fueron Church y Turing los que, de manera independiente, repararon en ello.

1.4. MODELOS COMPUTACIONALES.

Es posible que llegados a este punto aún puedan parecernos ambiguos conceptos tales como *algoritmo*, *función computable* o, simplemente, *computar*. No debemos

alarmarnos pues ni si quiera los matemáticos que hemos ido mencionando lograron durante mucho tiempo ponerse de acuerdo. Lo que ocurrió es que tales conceptos se fueron forjando paulatinamente conforme se iban desarrollando los acontecimientos. Más aun, numerosos matemáticos fueron proponiendo alternativas radicalmente diferentes a fin de concretar tales conceptos. Como ya hemos visto, en virtud del **Resultado 1**, si una teoría es consistente y sintácticamente completa, entonces existirá un procedimiento que permita resolver los distintos problemas de decisión que puedan plantearse. Por otra parte, como ya hemos dejado entrever, Gödel truncó la esperanza por hallar una teoría completa y consistente, pero esto no significaba que necesariamente no pudiese existir un procedimiento de decisión. Por ello, los matemáticos siguieron trabajando en el *Entscheidungsproblem* pero, puesto que los problemas de decisión van ligados a la idea de *algoritmo*, era preciso concretar qué habría de entenderse por tal. De esta manera, surgieron los distintos modelos computacionales como cada una de las distintas alternativas propuestas, las cuales a la poste resultarían ser equivalentes, dando lugar a lo que se conoce como la *tesis de Church-Turing*.

Como hemos insinuado, el fin último de un modelo computacional es esencialmente el de caracterizar las nociones de *algoritmo* y *computación*. Algunos intentos apuntaban en la dirección que introdujimos en la [Sección 1.1](#), esto es, plantearse qué se entiende cuando se computa el valor de una determinada función $f : \mathfrak{A} \rightarrow \mathfrak{B}$. Este proyecto condujo a la *teoría de funciones recursivas* por parte de Kleene⁵.

Se puede decir que una función es *recursiva* cuando, o bien se trata de una función inicial, o bien es una composición de funciones que respeta una serie de reglas de construcción. Las *funciones iniciales* propuestas son:

- $\zeta(n) = 0, \forall n \in \mathbb{N};$
- $\sigma(n) = n + 1, \forall n \in \mathbb{N};$
- $\pi_i^k(n_1, \dots, n_k) = n_i, \forall (n_1, \dots, n_k) \in \mathbb{N}^k.$

Por su parte, las *reglas de construcción* admitidas son:

- *Composición.* Dadas sendas funciones $g : \mathbb{N}^m \rightarrow \mathbb{N}$ y $h : \mathbb{N}^k \rightarrow \mathbb{N}$, se define la composición de ambas como $f : \mathbb{N}^k \rightarrow \mathbb{N}$ según

$$f(n_1, \dots, n_k) := g(h_1(n_1, \dots, n_k), \dots, h_m(n_1, \dots, n_k)).$$

- *Recursión primitiva.* Dadas sendas funciones $g : \mathbb{N}^k \rightarrow \mathbb{N}$ y $h : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$, se define la recursión primitiva, para $m \geq 0$, de ambas como $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ según

$$f(n_1, \dots, n_k, 0) := g(n_1, \dots, n_k);$$

$$f(n_1, \dots, n_k, m+1) := h(n_1, \dots, n_k, m, f(n_1, \dots, n_k, m)).$$

⁵Realmente, la teoría de funciones recursivas es iniciada por Gödel en la demostración de su segundo teorema de incompletitud. Sin embargo, su definición adolecía de ciertas patologías de cara a utilizarse para fundamentar la noción de función computable, debido a que existían funciones que debían ser computables pero que no resultaban ser recursivas. Kleen eliminó esta deficiencia del modelo de Gödel incluyendo el operador μ .

- *Operador μ .* Dada una función $g : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$, se define la acción del operador μ sobre ella como $f : \mathbb{N}^k \rightarrow \mathbb{N}$ dada por

$$\begin{aligned} f(n_1, \dots, n_k) &:= \mu x g(n_1, \dots, n_k, x) \\ &= \min \{x \in \mathbb{N} : g(n_1, \dots, n_k, x) = 0 \wedge \exists j g(n_1, \dots, n_k, j), \text{ para cada } j = 0, \dots, x\}. \end{aligned}$$

De esta manera, llegamos a nuestro primer modelo computacional.

Definición 5. (*Modelo computacional de Kleene*)

El modelo computacional de Kleene propone la siguiente caracterización:

- *Una función computable es una función recursiva;*
- *computar consiste en calcular el valor de una función recursiva;*
- *un algoritmo es una construcción de una función recursiva.*

Por otra parte, un joven matemático llamado Jacques Herbrand reparó, en 1931, en que las funciones $f : \mathbb{N}^k \rightarrow \mathbb{N}$ podían ser definidas utilizando sistemas de ecuaciones. Gödel desarrolló su idea dando lugar a la *teoría de funciones generales recursivas*. Sin entrar demasiado en detalles, se dice que una función $f : \mathbb{N}^k \rightarrow \mathbb{N}$ es *generalmente recursiva* si existe un sistema de ecuaciones en forma estándar⁶ que garantiza que f está definida en todo su dominio, al que denotamos por $\varepsilon(f)$. Esta definición nos aboca a otro modelo computacional.

Definición 6. (*Modelo computacional de Gödel-Herbrand*)

El modelo computacional de Gödel-Herbrand propone la siguiente caracterización:

- *Una función computable es una función general recursiva;*
- *computar consiste en sustituir todas las ocurrencias de cada variable que aparezca en $\varepsilon(f)$ por el mismo número natural y/o sustituir la ocurrencia de cada función que aparezca en $\varepsilon(f)$ por su valor;*
- *un algoritmo es un sistema de ecuaciones $\varepsilon(f)$ para una cierta f .*

En torno a los años 1931 y 1933, el matemático y lógico americano Alonzo Church había desarrollado ya otro modelo computacional al que denominó λ -cálculo. De hecho,

⁶ Se dice que un sistema de ecuaciones está en forma *estándar* si todas sus ecuaciones son de la forma $f(g_i(\bullet), \dots, g_j(\bullet)) = \dots$

el modelo de Kleene fue introducido para probar que el λ -cálculo y las funciones recursivas coinciden. Dada una función f de $n \in \mathbb{N}$ argumentos, se permite que cada uno de estos sea o un número u otra función con sus propios argumentos. De esta manera, lo que ocurre es que se anidan unas funciones en otras y así es posible describir la propia función f juntas con sus argumentos como una secuencia de símbolos que implícitamente representan el valor de $f(\bullet, \dots, \bullet)$. Las expresiones aceptadas (bien formadas) por el λ -cálculo se denominan λ -términos se definen recurrentemente como

- *Átomo.* Una variable es un λ -término.
- *Abstracción.* Si F es un λ -término y x una variable, entonces $(\lambda x.F)$ es un λ -término.
- *Aplicación.* Si F y G son λ -términos, entonces (FG) es un λ -término.

Esto nos conduce al modelo computacional de Church.

Definición 7. (Modelo computacional de Church)

El modelo computacional de Church propone la siguiente caracterización:

- *Una función computable es una función λ -definible;*
- *computar es transformar un λ -término inicial en otro final;*
- *un algoritmo es un λ -término.*

Otras matemáticos se inspiraron en la manera en que los seres humanos nos comunicamos, esto es, en los lenguajes. La idea radica en que si bien los seres humanos recibimos una secuencia de palabras para después producir otra, la cual debe haber sido procesada de acuerdo a unas determinadas reglas (nótese que estamos pensando en seres humanos con capacidad cognoscitiva plena y sin ningún tipo de trastorno psíquico). En el fondo, esto puede interpretarse como que la secuencia que recibimos constituye la descripción misma de un problema, mientras que la que producimos sería la propuesta de solución al propio problema.

En 1951, un matemático ruso proveniente de afamada estirpe de matemáticos llamado Andrey Markov Jr. describió un modelo computacional que hoy día conocemos como *gramáticas*. Este consiste en una secuencia finita de *producciones*, digamos M , tales como

$$\begin{array}{l} a_1 \longrightarrow b_1 \\ \vdots \\ a_k \longrightarrow b_k \end{array}$$

donde $a_i, b_i, i = 1, \dots, k$, son palabras sobre un determinado alfabeto Σ .

Sea M una gramática de Markov y $n \in \mathbb{N}$ un cierto numero natural fijo. Podemos definir una función, f_M^n como $f_M^n(a_1, \dots, a_n) := b$, siendo b la palabra generada efectivamente por la secuencia de producciones M . Señalemos que nada nos impide codificar las palabras de entrada y salida mediante números. De esta manera, se dice que una función arbitraria, f , es Markov-computable si existe una gramática M tal que $f \equiv f_M^n$.

Definición 8. (*Modelo computacional de Markov*)

El modelo computacional de Markov propone la siguiente caracterización:

- Una función computable es una función Markov-computable;
- computar es ejecutar una gramática;
- un algoritmo es una gramática.

Demos paso al que sin duda es el modelo computacional más conocido. Su fama se debe a que se basa en una idea completamente diferente al resto de los modelos que hemos presentado hasta ahora. En su artículo de 1936, Alan Turing es capaz de abstraer la actividad mecánica que realiza un ser humano cuando resuelve un problema, descomponiéndola a su vez en varias acciones canónicas. Tal actividad puede ser recreada por la denominada *máquina de Turing*⁷.

La máquina de Turing consta de varios componentes. A saber,

- una *unidad de control*, en clara alusión al cerebro humano;
- una *cinta* de longitud potencialmente infinita dividida en *celdas* del mismo tamaño, correspondiente al papel que utilizaría un humano al realizar sus cálculos;
- un *lector* que se puede mover por cada una de las celdas y que es capaz de escribir, borrar y reconocer los caracteres que aparezcan en estas, simulando lo que haría el ojo y la mano, provista de un lápiz, de un humano.

De esta manera, la unidad de control siempre se encuentra en un cierto *estado* de entre todos los posibles estados en los que puede estar, los cuales constituyen un conjunto finito. Dos de estos estados son los que se denominan *inicial* y *final*. También se ha de contar con un *programa* dado por una serie finita de instrucciones incluidas en la unidad de control al que se denomina *programa de Turing*. Se dice que dos máquinas de Turing son distintas cuando tienen implementados distintos programas de Turing.

⁷ Reparemos en que al utilizar el término *máquina*, Turing pretende reflejar que desde el punto de vista del formalismo de Hilbert, no existe nada en la acción de *calcular* que no pueda ser implementado por un dispositivo mecánico. Además, conviene señalar que lo utiliza con anterioridad a la aparición de los primeros ordenadores. De hecho, serán sus propios trabajos los que acaben por constituir una de las piedras angulares de la *informática*.

Antes de ejecutar una máquina de Turing, esta debe encontrarse en el estado inicial y preparada para leer la llamada *palabra de entrada* de un cierto alfabeto Σ que ha sido escrita con anterioridad en una celda de la cinta. Desde su ejecución, una máquina de Turing va realizando de forma mecánica una serie de operaciones según indique el programa. En cada etapa, la máquina lee el símbolo de la celda sobre la que se encuentre el lector y, atendiendo al estado en el que se encuentre, realizará una de las siguientes acciones:

- borrar el símbolo y reemplazarlo por otro;
- mover el lector de la celda actual a una de las adyacentes;
- modificar el estado de la unidad de control.

La máquina se detiene si la unidad de control alcanza el estado final o bien, si no restan instrucciones que realizar. Las palabras de entrada y de salida (si es que la máquina se detiene) pueden codificarse mediante sendas secuencias binarias, lo cual posibilita su representación vía funciones booleanas para así concebir que la máquina trabaja únicamente con valores numéricos. Identificando las palabras $a_1, \dots, a_k, b \in \Sigma$, siendo Σ un determinado alfabeto, con $\lfloor a_1 \rfloor, \dots, \lfloor a_k \rfloor, \lfloor b \rfloor$, podemos concebir la acción de una máquina de Turing, T , como una función $f : \Sigma^* \rightarrow \Sigma^*$ dada por $f_T^k(a_1, \dots, a_k) := b$. Así, dada una función, f , cualquiera, se dice que esta es Turing-computable si existe alguna máquina de Turing tal que $f \equiv f_T^k$.

Definición 9. (Modelo computacional de Turing)

El modelo computacional de Turing propone la siguiente caracterización:

- Una función computable es una función Turing-computable;
- computar es ejecutar programa de Turing en una maquina de Turing;
- un algoritmo es un programa de Turing.

Restan aún otros modelos computacionales como el de Post o el de Kolmogórov-Uspenski, pero hemos decidido no incorporarlos pues realmente no aportan demasiado toda vez que ya se han presentado suficientes modelos. En cambio, si que se ha incorporado un [Apéndice](#) a este trabajo que pretender poner de manifiesto la mayor complejidad de otros modelos computacionales, los cuales resultan laberínticos incluso para el iniciado en matemáticas, frente al modelo de Turing.

1.5. LA MÁQUINA DE TURING.

La máquina de Turing fue concebida inicialmente con objeto de formalizar los conceptos de *algoritmo*, *computación* y *función computable*. El modelo de Turing original

se presentó en su artículo [17], si bien es cierto que él mismo introduciría más tarde diversas variantes que no eran más que generalizaciones del primero. Además, Turing terminaría por demostrar que tales variante no surtían ningún efecto en cuanto al aumento de la capacidad computacional del modelo primigenio.

La máquina de Turing original cuenta con una *unidad de control* que incorpora el *programa de Turing*, una *cinta* dividida en *celdas* idénticas, y un *lector* móvil dispuesto sobre la cinta y conectado a la unidad de control. Asimismo, se imponen los siguientes requisitos:

- La *cinta* se utiliza para escribir y leer las palabras iniciales, intermedias y finales. Además, ha de ser potencialmente infinita en un sentido. En cada *celda* debe aparecer un símbolo perteneciente a un alfabeto, el *alfabeto de la máquina*, $\Delta = \{z_1, z_2, z_3 \dots, z_t, \sqcup\}$, con $t \geq 3$. Se suele requerir que $z_1 = 0, z_2 = 1$, así como que $z_t = \sqcup$. Al símbolo \sqcup se lo conoce como *espacio en blanco*. La *palabra de entrada* debe pertenecer a otro alfabeto, Σ , que ha de ser tal que $\{0, 1\} \subseteq \Sigma \subseteq \Delta - \{\sqcup\}$ y al que llamamos *alfabeto de entrada*. Inicialmente, todas las celdas a excepción de aquellas que contienen la palabra de entrada se encuentran vacías, esto es, muestran el símbolo \sqcup .
- La *unidad de control* se halla siempre en algún *estado* que pertenece al denominado *espacio de estados* $Q = \{q_1, \dots, q_s\}$. Al estado q_1 se lo conoce como *estado inicial*. Por su parte, se dice que algunos de los estados son *finales*. Al conjunto de estados finales se lo denota por $F \subseteq Q$. A los estados pertenecientes a $Q - F$ se los llama *estados no-finales*. Cuanto no resulte relevante el índice del estado, utilizaremos q_{si} y q_{no} para referir a un estado final y a uno no-final cualesquiera, respectivamente.
- La unidad de control incorpora el *programa de Turing*. El programa es capaz de dirigir las distintas componentes de la máquina y es lo que caracteriza a cada una de las máquinas de Turing. Formalmente, un programa de Turing es una función parcial $\delta : Q \times \Delta \longrightarrow Q \times \Delta \times \{\text{Izquierda, Derecha, Permanecer}\}$. Se asume que $\delta(q_{no}, z) \downarrow$ para algunos $z \in \Delta$ y que $\delta(q_{si}, z) \uparrow$ para todo $z \in \Delta$, esto es, que siempre existe una transición desde un estado no-final y ninguna desde un estado final.
- El lector puede moverse a cada una de las celdas de la cinta, leer sus símbolos y modificar el contenido de las mismas.
- Antes de ejecutar una máquina de Turing debe haber acontecido:
 - que la palabra de entrada haya sido escrita al comienzo de la cinta;
 - que el lector se haya situado en la primera celda de la cinta;
 - que la unidad de control se encuentre en el estado inicial.
- La máquina de Turing actúa de manera independiente, paso a paso y conforme al programa de Turing especificado por δ . En concreto, si la máquina de Turing se encuentra en el estado $q_i \in Q$ y lee el símbolo $z_k \in \Delta$, entonces:
 - se detiene, si q_i es un estado final;

- se detiene, si $\delta(q_i, z_k) \uparrow$;
- pasa al estado q_j , escribe z_r y, o mueve el lector a derecha o izquierda, o bien permanece en el mismo sitio, si $\delta(q_i, z_k) = (q_j, z_r, D)$.

El propio Turing presentó determinadas modificaciones sobre el modelo original. De hecho, en el planteamiento primigenio se contemplaba el uso de una única cinta infinita en ambos sentidos y, por tanto, lo que nosotros hemos presentado supone en sí mismo una variación del modelo original. Otras variantes permiten el uso de diversas cintas e, incluso, de cintas multidimensionales. Sin embargo, la capacidad computacional de todas ellas resulta ser la misma (ver [14]). Demos paso a la definición formal de máquina de Turing más aceptada actualmente.

Definición 10. (*Definición formal de la máquina de Turing*)

Se define formalmente la máquina de Turing como una 7-tupla

$$T = (Q, \Sigma, \Delta, \delta, q_1, \sqcup, F),$$

donde $Q, \Sigma, \Delta, \delta, q_1, \sqcup, F$ son los objetos que hemos introducido anteriormente.

Ahora que ya contamos con una definición precisa de la máquina de Turing, cabe preguntarse al respecto de la manera de representar las distintas etapas o pasos que tienen lugar durante la ejecución de la misma. Para ello, se recurre al concepto de *configuración interna* de la máquina.

Definición 11. (*Configuración interna de la máquina de Turing*)

Sea T una determinada máquina de Turing. Sea también w una cierta palabra de entrada para T . La configuración interna de T después de una cantidad finita de etapas es la palabra $uq_i v$ tal que

- q_i es el estado actual;
- $uv \in \Delta^*$ está formada por (1) los símbolos que no son espacios en blanco de la cinta que se encuentran más a la derecha o, en su caso, (2) por el símbolo que se encuentra inmediatamente a la izquierda de la cinta y que es el que se encuentra más a la derecha. Se asume que $v \neq \lambda$ si (1) y que $v = \lambda$ si (2);
- T lee el símbolo más a la izquierda de v si (1), o el símbolo \sqcup si (2).

El modelo de la máquina de Turing que hemos presentado se caracteriza por tener determinada la acción que realizar a cada etapa de manera unívoca. Por ello, a este modelo se lo suele conocer como *máquina de Turing determinista*. Por otra parte,

fueron concebidas otras máquinas que, a diferencia de las deterministas, son capaces de realizar diferentes operaciones en un mismo paso, dando lugar a los consiguientes resultados distintos. Es decir, mientras que el resultado de una máquina determinista queda determinado por la palabra de entrada que se introduzca, una máquina no determinista puede producir distintos resultados a partir de una misma entrada. A este tipo de máquinas no deterministas se las conoce como *máquinas indeterministas*. La introducción del intederminismo resulta ser una herramienta de suma transcendencia debido a que aumenta la potencia computacional del modelo original al permitir resolver de manera inmediata problemas que antes eran altamente costosos. Es un problema sustancial el dirimir si tal aumento de la potencia computacional es auténtico en el sentido de si realmente resuelve problemas que no podían ser resueltos por máquinas deterministas. A esta cuestión se la conoce como *conjetura de Cook* y volveremos sobre ella más adelante.

El programa de una máquina de Turing indeterminista se caracteriza porque sus instrucciones vienen dadas por una función

$$\delta_I : Q \times \Delta \longrightarrow (Q \times \Delta \times \{\text{Izquierda, Derecha, Permanecer}\})^p, \quad (1.4)$$

dada por

$$\delta_I(q_i, z_k) = \left((q_{j_1}, z_{r_1}, D_1), \dots, (q_{j_p}, z_{r_p}, D_p) \right). \quad (1.5)$$

Existen dos alternativas a la hora de concebir la acción de una máquina de Turing indeterminista. La primera de ellas es la que imagina que la máquina es capaz de elegir la opción más conveniente de entre las distintas posibilidades de manera que minimiza el número de etapas requeridas para resolver un problema. Por su parte, la otra figura que la máquina de Turing se ramifica dando lugar a "submáquinas" las cuales llevan a cabo cada una de las distintas alternativas.

Hasta ahora hemos permitido que la máquina de Turing trabaje con un determinado alfabeto Δ compuesto por un número arbitrario de elementos. Sin embargo, las cosas se simplifican notablemente cuando se trabaja con alfabetos de la máquina y de entrada de tamaño reducido. La elección más sencilla resulta ser $\Delta = \{0, 1, \sqcup\}$ y $\Sigma = \{0, 1\}$. Esta manera de proceder no desemboca en una pérdida de generalidad pues cualesquiera que sean los alfabetos originales, estos pueden ser codificados en código binario. Es posible encontrar una demostración de esta equivalencia en [14]. Además, se suele exigir que el conjunto de estados finales conste de un único elemento, esto es, $F = \{q_F\}$. Cuando se lleva a cabo esta simplificación se habla del *modelo de Turing reducido* y de *máquinas de Turing reducidas*. Así, formalmente se dice que una máquina de Turing reducida es una 7-tupla

$$T = (Q, \{0, 1\}, \{0, 1, \sqcup\}, \delta, q_1, \sqcup, \{q_F\}). \quad (1.6)$$

Por otra parte, a menudo es interesante el disponer de manera clara de una representación de una máquina de Turing concreta. Esta ha de ser tal que permita recuperar toda la información relativa a la propia máquina y, por ello, el procedimiento

adecuado resulta ser el de codificar y, en particular, hacerlo en binario. En efecto, lo que se pretende es codificar la función de transición, δ , pues resulta ser el objeto determinante y característico de cada máquina. Así, si $\delta(q_i, z_j) = (q_k, z_l, D_m)$ es una instrucción del programa de Turing de una cierta máquina T , podemos codificarla como

$$I = 0^{(i)} 10^{(j)} 10^{(k)} 10^{(l)} 10^{(m)}, \quad (1.7)$$

donde D_1 = Izquierda, D_2 = Derecha y D_3 = Permanecer.

A partir de las distintas codificaciones de cada una de las instrucciones, se puede obtener el *código de la máquina* tras una especie de concatenación llevada a cabo de la siguiente manera: si I_1, \dots, I_n son las codificaciones de cada una de las n instrucciones, respectivamente, se obtiene el código de la máquina como

$$\llcorner T \lrcorner = 111I_111I_211\dots11I_n111. \quad (1.8)$$

Por su parte, resulta claro que existe una biyección entre los números naturales y sus representaciones en binario. De esta manera, es posible recuperar de manera unívoca el número natural (su expresión decimal) al que representa $\llcorner T \lrcorner$. A tal número natural se lo conoce como *índice* de la máquina T . Debemos reparar en que algunos números naturales no son índices de ninguna máquina de Turing, concretamente, aquellos que no tienen la forma que hemos presentado antes. Como consecuencia de ello, se suele aceptar que tales números naturales que no representan máquinas de Turing resultan ser el índice de una máquina concreta denominada *máquina de Turing vacía* y que se detiene para cualquier palabra de entrada en 0 etapas. Tras aceptar este convenio, podemos afirmar el siguiente resultado.

Resultado 2.

Todo número natural es el índice de una máquina de Turing y solo de una.

En 1936, Alan Turing reparó en que era posible concebir una máquina de Turing que fuese capaz de imitar la acción de cualquier otra máquina de Turing. Esta máquina es la que se conoce como *máquina de Turing universal* e incorpora, no una, sino tres cintas divididas en celdas homogéneas. La primera de ella, la *cinta de entrada* contiene la palabra de entrada la cual, a su vez, está compuesta por el índice de una máquina de Turing cualquiera y una cierta palabra arbitraria. La segunda es la *cinta de trabajo*, la cual inicialmente se encuentra vacía y se utiliza de manera similar a como lo haría una máquina de Turing básica. La tercera se conoce como *cinta auxiliar* y sirve para grabar el estado actual en el que se encuentra la máquina que se está simulando e ir comprobando si este es un estado final de la misma. Debemos ser conscientes de que, en realidad, la máquina de Turing universal no es de una naturaleza distinta a una máquina estándar. Esto supone que el siguiente resultado sea de enorme trascendencia.

Resultado 3.

Existe una máquina de Turing capaz de computar todo aquello que resulta computable por alguna máquina de Turing.

La existencia de la máquina de Turing universal resulta ser verdaderamente relevante pues pone de manifiesto que es factible diseñar un método suficientemente general como para resolver cualquier problema que pueda resolverse. Notemos también que, puesto que la máquina de Turing universal utiliza como datos de entrada el índice de la máquina que va a ser simulada junto con la propia palabra de entrada, no ha de existir en principio diferencia alguna entre lo que son meros datos y lo que son instrucciones. El propio Turing reparó en este hecho estableciendo que la distinción entre ambos viene fijada por la interpretación que se realice. Por otra parte, la prueba al respecto de la existencia de la máquina de Turing universal parecía respaldar la clarividencia que había mostrado un siglo atrás el matemático británico Charles Babbage, quien postuló que sería posible construir una máquina física que fuese capaz de resolver mecánicamente cualquier problema computable.

1.6. FUNCIONES COMPUTABLES Y LENGUAJES.

El modelo de Turing puede ser empleado para computar valores de una determinada función, para generar elementos de un conjunto y para dilucidar si un determinado elemento pertenece a un determinado conjunto o no, entre otras cosas. El último de los casos es que va a acaparar nuestra atención debido a que de manera natural en el contexto que nos ocupa uno ha de interesarse por conocer cuándo una determinada proposición es o no un elemento de un conjunto que es la *teoría*, es decir, cuando es o no un *teorema*.

A continuación vamos a presentar diversas definiciones elementales, así como formalizaremos la noción de *función computable* relativa al modelo de Turing.

Definición 12. (*Resultado, conjunto de parada*)

Sea $T = (Q, \Sigma, \Delta, \delta, q_1, \sqcup, F)$ una determinada máquina de Turing y sea $w \in \Sigma^*$ una cierta palabra de entrada. Se denomina resultado de T y a la función

$$Res_T : \Sigma^* \longrightarrow \{\text{Aceptar}, \text{Rechazar}, \text{No reconocido}\}$$

que viene dada por

$$Res_T(w) = \begin{cases} \text{Aceptar} & \text{si } T \text{ se detiene en un estado final } q_{si}; \\ \text{Rechazar} & \text{si } T \text{ se detiene en un estado no-final } q_{no}; \\ \text{No reconocido} & \text{si } T \text{ no se detiene.} \end{cases}$$

Por su parte, al conjunto

$$\mathbb{P}_T = \{w \in \Sigma^* : \text{Res}_T(w) \in \{\text{Aceptar, Rechazar}\}\}$$

se lo conoce como conjunto de parada de la maquina T.

Las nociones previas nos conducen inexorablemente a la de *palabra aceptada* por una máquina de Turing, así como la de *lenguaje aceptado* por ella. Formalicemos estas nociones.

Definición 13. (*Palabra aceptada, lenguaje aceptado*)

Sea $T = (Q, \Sigma, \Delta, \delta, q_1, \sqcup, F)$ una determinada máquina de Turing. Se dice que una palabra $w \in \Sigma^*$ es aceptada por T si, y solo si, $\text{Res}_T(w) = \text{Aceptar}$. Asimismo, a la contraimagen por la función resultado de Aceptar se lo denomina lenguaje aceptado por T o también, lenguaje propio de T , esto es,

$$\mathbb{L}_T = \text{Res}_T^{-1}(\{\text{Aceptar}\}) = \{w \in \Sigma^* : \text{Res}_T(w) = \text{Aceptar}\}.$$

Al fin estamos en condiciones de formalizar rigurosamente los conceptos de *función computable* y *problema de decisión*. Sin embargo, en aras de la claridad de claridad expositiva, vamos a ceñirnos desde ahora al caso reducido, esto es, $\Sigma = \{0, 1\}$. Como ya mencionamos, esto no supone una pérdida de generalidad pues cualquier alfabeto puede ser codificado sobre Σ . Además, identificaremos los elementos Aceptar, Rechazar del conjunto de llegada de las funciones resultado con $\{0, 1\}$, concretamente, $\text{Aceptar} \equiv 1$ y $\text{Rechazar} \equiv 0$. De esta manera, podría decirse que una función resultado es tal que $\text{Res}_T : \Sigma^* \rightarrow \Sigma \cup \{\text{No reconocido}\}$. Con esto en mente, demos paso a la siguiente definición.

Definición 14. (*Función computable*)

Sea ψ una función booleana. Se dice que ψ es computable si existe una máquina de Turing, T , de manera que

$$\psi(w) = \text{Res}_T(w) \quad \forall w \in \Sigma^*.$$

Desde otra perspectiva, resulta claro que, dada una cierta función booleana, ψ , podemos plantearnos la determinación del lenguaje

$$\mathbb{L}_\psi = \{w \in \Sigma^* : \psi(w) = 1\}. \quad (1.9)$$

Tal determinación efectiva es lo que se conoce como *problema de decisión* y será factible si la función ψ es computable y, en tal caso, se dice que el problema de decisión es *decidible*.

o *computable*. En otro caso, se manifiesta que es *indecidable* o *incomputable*. Dada la estrecha relación entre las funciones booleanas y sus lenguajes asociados como queda patente en (1.9), un problema de decisión puede ser entendido como, dado un lenguaje $\mathbb{L} \subseteq \Sigma^*$, encontrar una máquina de Turing, T , con $\mathbb{P}_T = \Sigma^*$ tal que

$$\text{Res}_T(w) = \chi_{\mathbb{L}}(w) \quad \forall w \in \Sigma^*, \quad (1.10)$$

donde $\chi_{\mathbb{L}}$ es la *función característica* del lenguaje, esto es, $\chi_{\mathbb{L}} : \Sigma^* \rightarrow \Sigma$ tal que

$$\chi_{\mathbb{L}}(w) = \begin{cases} 1 & \text{si } Tw \in \mathbb{L}; \\ 0 & \text{si } w \notin \mathbb{L}. \end{cases} \quad (1.11)$$

Reparemos en que en el caso que nos ocupa, esto es, $\Sigma = \{0, 1\}$, las funciones características resultan ser también funciones booleanas. De esta manera, damos paso a las consiguientes definiciones de computabilidad relativas a lenguajes.

Definición 15. (*Lenguaje decidable, semi-decidible, indecidible*)

Se dice que un lenguaje $\mathbb{L} \subseteq \Sigma^*$ es semi-decidible o recursivamente enumerable si existe una máquina de Turing T tal que

$$\mathbb{L} = \mathbb{L}_T.$$

Si un lenguaje semi-decidible $\mathbb{L} \subseteq \Sigma^*$ es tal que $\Sigma^* - \mathbb{L}$ también es semi-decidible, entonces se dice que es decidable, recursivo o computable. Los lenguajes no decidibles se conocen como indecidibles.

Una vez que hemos establecido todas estas nociones es sensato preguntar qué problemas pueden ser computados. Si atendemos a la implícita conexión entre computabilidad y resolución efectiva de un problema, nuestra intuición nos anima a contestar que no todos los problemas han de ser necesariamente computables. En efecto, el propio Alan Turing demostró en [17] que existe un problema, el conocido *problema de la parada*, que no lo es. Asimismo, existen otras demostraciones que utilizan procedimientos diagonales para poner de manifiesto la existencia de problemas indecidibles. Nosotros finalizaremos este capítulo presentando un argumento que solo requiere pequeñas nociones al respecto de la cardinalidad de un conjunto.

Resultado 4.

Existen infinitos problemas de decisión no computables.

Demostración. Atendiendo a la Definición 1, resulta claro que se puede establecer fácilmente, utilizando la longitud, un morfismo de semigrupos entre $\langle \mathcal{L}^*(\Sigma), \bullet \rangle$ y $\langle \mathbb{N}, + \rangle$. Así, aunque Σ es finito (es un alfabeto), el conjunto de todas las palabras de cualquier

longitud, $\mathcal{L}^*(\Sigma)$, es infinito numerable y, por lo tanto, cada lenguaje tendrá a lo sumo cardinalidad numerable. Además, respecto a la cantidad de lenguajes posibles, esto es, $|\{\mathbb{L} \subseteq \mathcal{L}^*(\Sigma)\}|$, se tiene que

$$|\{\mathbb{L} \subseteq \mathcal{L}^*(\Sigma)\}| = |\mathcal{P}(\mathcal{L}^*(\Sigma))| = 2^{\aleph_0}. \quad (1.12)$$

Asimismo, cada lenguaje define un único problema de decisión, por lo tanto, existe una cantidad infinita no numerable de problemas de decisión. Por otra parte, en virtud del [Resultado 2](#), existe tan solo una cantidad numerable de máquinas de Turing y, consiguientemente, no es posible asociar a cada problema una máquina de Turing que lo resuelva. Así, puesto que la cantidad de problemas computables es numerable mientras que la cantidad de problemas de decisión no lo es, existe una cantidad infinita no numerable de problemas indecidibles.

■

1.7. REFERENCIAS BIBLIOGRÁFICAS.

Debemos reparar en que a lo largo de estas páginas hemos tratado aspectos bastante diversos como son la fundamentación de la matemática y la teoría formal de lenguajes. Describiremos en las líneas posteriores los aspectos más reseñables de la bibliografía que hemos utilizado para la elaboración de este trabajo.

- El libro de Robic ([14]) realiza una exposición bastante clara y accesible a casi todos los aspectos que se han cubierto en esta parte, si bien es cierto que algunas cuestiones pueden resultar densas o inaccesibles para el lector en matemáticas. Pormenorizadamente, los capítulos 2, 3 y 4 explican en profundidad el nacimiento de la teoría de la computación. Por su parte, el capítulo sexto expone el modelo de la máquina de Turing. Finalmente, aunque nosotros hemos pasado de soslayo sobre los problemas indecidibles, los capítulos 8 y 9 están dedicados a ahondar en ellos.
- Por su parte, no podemos dejar de mencionar las obras de Kolmogórov y Dragalin ([7]-[8]), las cuales constituyen una vía excepcional para los matemáticos que deseen iniciarse en el estudio de la lógica en general y de la computabilidad en particular.
- Asimismo, el celebre artículo de Alan Turing ([17]), aunque tiene partes más técnicas, incorpora una cuidadosa descripción de los planteamientos primordiales que resulta claramente clarificadora.
- También merecen ser mencionados los apuntes de la asignatura *Lógica e informática*, integrada en el máster que nos atañe, elaborados por el profesor José Pedro Úbeda, los cuales presentan las nociones más específicas con absoluto rigor.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

2

UNA INTRODUCCIÓN A LA *teoría de la complejidad.*

"Si la gente no cree que las matemáticas son sencillas,
es solo porque no se da cuenta de lo complicada que es la vida."
— John von Neumann.

Resumen: En primer lugar, se presenta el propósito de la *teoría de la complejidad*. A continuación, se presentan los conceptos más importantes de la misma así como las clases de complejidad más relevantes. Por último, se expone la *conjetura de Cook* y se realizan distintos comentarios al respecto.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

2.1. ¿QUÉ ES LA COMPLEJIDAD?

La *complejidad* resulta ser una noción con la que estamos habituados a lidiar en nuestro día a día. Desde nuestra etapa escolar sabemos lo que es que un profesor nos plantea en un examen un problema demasiado difícil, así como lo costoso de aprender a tocar un instrumento musical de forma que las notas encajen de esa manera que han de hacerlo para no perturbar el sosiego de nuestros oídos. También sabemos que es más fácil sumar dos números que multiplicarlos y que, en ambos casos, es preferible y más fiable utilizar la calculadora. Al fin y al cabo, la inmensa mayoría de la población adulta cree firmemente que la ardua e, incluso, cruel tarea de utilizar papel y lápiz resulta ser absolutamente defectible. En efecto, si bien es cierto que es conveniente saber realizar ciertos procedimientos de manera mecánica, no debemos olvidar que esto termina por ser una actividad irreflexiva, una mera acción no deliberada sin pensamiento consciente. Sin embargo, este tipo de conocimiento puede ser de utilidad cuando uno pretende, por ejemplo, conducir un coche a gran velocidad, evitando reflexionar y actuando de manera mecánica para salvaguardar su integridad personal. Pero este planteamiento carece de sentido cuando se lleva al ámbito de lo matemático pues, después de todo, la probabilidad de que un polinomio atente contra nosotros es escasa. Por ello, la gente no se equivoca cuando cree que calcular mentalmente o a mano las operaciones que se van requiriendo es una empresa baldía toda vez que siempre se dispone de una calculadora a mano.

De esta manera, la atención ha de dirigirse hacia otro tipo de cuestiones, concretamente, hacia aquellas para las que no existe un método que permita resolverlas. En un ejercicio de autorreferencialidad, el proponer una clasificación de los problemas según su complejidad resulta ser una de tales cuestiones. En un primer acercamiento al asunto, dispondremos enseguida que tal clasificación ha de respetar nuestra intuición considerando los problemas que cuentan con un método que permite resolverlos como más "fáciles" que aquellos que no lo hacen pues, por muy tedioso que resulte un método, basta con ser paciente y seguir sus instrucciones con recelo para dar con la solución buscada. Sin embargo, también sabemos gracias a la experiencia que existen problemas que se resuelven *pensando* sin que medie ningún tipo de procedimiento. Acaba de entrar en escena lo que de conoce como *indeterminismo*, esto es, nuestra clasificación ha de ser suficientemente general como abarcar todos los problemas sin prestar atención a un método concreto de resolución. En efecto, no estamos interesados en cuantificar el coste de un determinado método, es decir, en medir cuán farragoso es sino en analizar una propiedad de los propios problemas.

Volvamos sobre la dificultad del problema de multiplicar dos números frente a la de sumarlos. Aunque probablemente todo el mundo concedería que, efectivamente, es más complejo realizar una multiplicación, debemos preguntarnos la razón subyacente a esta creencia. Quizás se deba a que aprendemos a sumar antes de multiplicar y por ende asimilamos que el saber sumar es un requisito para poder multiplicar. Pero, aunque razonemos de esta manera, seguimos sin poder demostrar que multiplicar es más complicado. Cualquiera podría aducir que multiplicar lleva más tiempo que sumar, pero esto se debe a que está pensando en el método tradicional de multiplicación. Esto, en primer lugar, resulta ser un planteamiento ingenuo pues manifiesta que se confunde lo que

es propiamente un problema de lo que es un método para resolverlo. Además, hoy en día sabemos, por ejemplo, que existen algoritmos que multiplican números grandes mucho más deprisa que el método tradicional. Así que la pregunta que debemos plantearnos para catalogar la dificultad de los problemas es si todos y cada uno de los métodos que pueden ser utilizados para realizar multiplicaciones requieren de más tiempo que el mejor de los que se utilizan para sumar. Para responder a esta cuestión se recurre habitualmente a un argumento de imposibilidad, es decir, que no es posible encontrar un algoritmo que realice multiplicaciones más eficiente que el utilizado para sumar. De este tipo de cuestiones se ocupa la llamada *teoría de la complejidad*.

2.2. ALGUNAS NOCIONES DE LA COMPLEJIDAD.

Como hemos anunciado en la sección previa, la teoría de la complejidad se ocupa de proponer una taxonomía para los problemas conforme a la eficiencia con la que pueden ser resueltos. La eficiencia de un método se suele medir a partir de los recursos que han de utilizarse para llevar a cabo una tarea. En el ámbito de la computación los recursos atañen a dos aspectos: el espacio y el tiempo. El primero de ellos está relacionado con la cantidad de memoria de almacenamiento requerida por un método. Podemos pensar que, utilizando lápiz y papel, necesitamos de varios folios para solucionar un problema, mientras que el tiempo simplemente refleja la duración del proceso de cómputo. Si se nos provee además de una goma de borrar, podremos reutilizar los folios de los que disponíamos, optimizando el coste de espacio. Sin embargo, no disponemos de una suerte de goma para el tiempo y, por ello, consideramos el coste temporal más importante que el espacial. Por ello, el recurso que se suele estudiar habitualmente es el tiempo y así lo haremos nosotros.

En primer lugar, esperamos que el lector convenga con nosotros a partir de lo expuesto en la [Sección 1.4](#) que el modelo computacional más manejable es el de la máquina de Turing. Por ello, será el que utilicemos como marco común de referencia para la catalogación de los problemas. Asimismo, toda máquina de Turing no trivial comienza por leer la palabra de entrada antes de proseguir con su computación. Por ello, resulta claro que el coste temporal debe depender de la palabra de entrada que se contempla en cada caso. Esto nos conduce a la siguiente definición.

Definición 16. (*Tiempo de cómputo para una palabra*)

Sea T una máquina de Turing con alfabeto de entrada Σ y sea $w \in \mathbb{P}_T$ una cierta palabra de entrada. Se define el tiempo de cómputo de T para w como la función $t_T : \mathbb{P}_T \rightarrow \mathbb{N}$, que asigna a cada w en número de pasos requeridos por T hasta que se detiene.

La definición anterior solo alude a una palabra de entrada concreta y, además, estamos de acuerdo en que la longitud de la palabra de entrada influye sobre el tiempo de cómputo

de una máquina. Asimismo, usualmente se sigue una doctrina *ad cautelam* y, por tanto, la complejidad de una máquina estará relacionada con el caso más desfavorable de todos los posibles. Esto nos lleva a la definición siguiente.

Definición 17. (*Función de complejidad temporal*)

Sea T una máquina de Turing con alfabeto de entrada Σ y sea $w \in \mathbb{P}_T$ una cierta palabra de entrada. Se define la función de complejidad temporal como

$$T_T : \mathbb{N} \longrightarrow \mathbb{N}$$

dada por

$$T_T(n) = \max_{|w| \leq n} \{t_T(w)\}.$$

En algunas ocasiones, considerar el caso más desfavorable puede distorsionar la realidad del problema. En consecuencia, ocasionalmente se prefiere trabajar con la complejidad media. Nosotros no trataremos esta noción. Vamos a presentar ahora otro de los conceptos propios de la teoría de la complejidad.

Definición 18. (*Función temporalmente computable*)

Se dice que una función $f : \mathbb{N} \longrightarrow \mathbb{N}$ es temporalmente computable si existe una máquina de Turing, T , con alfabeto $\Sigma = \{1\}$ que la computa de manera que $\mathbb{P}_T = \Sigma^*$, $f(n) \geq n$ y $T_T(n) = O(f(n))$.

Cabe realizar varios comentarios al respecto de la definición anterior. En primer lugar, la elección del alfabeto $\Sigma = \{1\}$ supone que $\Sigma^* = \mathbb{N}$ sin más que realizar la identificación $1^{(n)} = n$ ¹. Por otra parte, la condición $f(n) \geq n$ garantiza que la máquina de Turing considerada es capaz de, al menos, leer la palabra de entrada al completo. En cuanto a $T_T(n) = O(f(n))$, cabe señalar que lo que quiere decir es que T_T está acotada superiormente por f cuando $n \rightarrow \infty$, esto es, que T_T es inferior a f para valores de n suficientemente grandes². Finalmente, conviene indicar que, entre otras, las funciones polinómicas resultan ser temporalmente computables.

Por otra parte, en el Capítulo 1 señalábamos la robustez del modelo de Turing en el sentido de que la capacidad computacional del mismo no se ve alterada por el alfabeto que utilice cada máquina pues era suficiente realizar las codificaciones pertinentes. Por supuesto que esto se mantiene en el ámbito de la teoría de la complejidad y, por lo tanto, no ahondaremos en ello. Se puede encontrar una demostración en [10].

¹Esta identificación es habitual cuando se trabaja con máquinas de Turing como puede comprobarse en el Apéndice.

²A esta notación se la conoce como *notación de Landau*.

2.3. CLASES DE COMPLEJIDAD.

Una vez que ya hemos introducido todas las herramientas que precisamos, estamos en disposición de presentar, al menos, las clases de complejidad más importantes. Se dice que una *clase de complejidad* es un conjunto de problemas de decisión que pueden ser computados, según el modelo de Turing, utilizando una cantidad de recursos delimitada por una misma cota. La primera clase que presentaremos serán las que solo permiten máquinas de Turing deterministas.

Definición 19. (Clases DTIME)

Dada una función temporalmente computable $f : \mathbb{N} \rightarrow \mathbb{N}$. Se define la clase **DTIME**(f) como

$$\mathbf{DTIME}(f) = \{\mathbb{L} \subseteq \Sigma^* : \exists T \text{ con } \text{Res}_T = \chi_{\mathbb{L}} \wedge T_T \in O(f)\},$$

donde T es una máquina de Turing determinista.

Si se permite que la máquina de Turing sea indeterminista, entonces de forma similar se definen las clases de complejidad **NTIME**.

Definición 20. (Clases NTIME)

Dada una función temporalmente computable $f : \mathbb{N} \rightarrow \mathbb{N}$. Se define la clase **NTIME**(f) como

$$\mathbf{NTIME}(f) = \{\mathbb{L} \subseteq \Sigma^* : \exists T_I \text{ con } \text{Res}_{T_I} = \chi_{\mathbb{L}} \wedge T_{T_I} \in O(f)\},$$

donde T_I es una máquina de Turing indeterminista.

Resulta claro a partir de la definición anterior que si $f_1(n) \leq f_2(n)$ para todo $n \in \mathbb{N}$, entonces $\mathbf{DTIME}(f_1) \subseteq \mathbf{DTIME}(f_2)$. Similarmente ocurre con las clases **NTIME**, esto es, si $f_1(n) \leq f_2(n)$ para todo $n \in \mathbb{N}$, entonces $\mathbf{NTIME}(f_1) \subseteq \mathbf{NTIME}(f_2)$. Asimismo, debemos tener presente que las máquinas de Turing deterministas son un caso particular de las indeterministas y, por tanto, dada una función temporalmente computable cualquiera, f , se cumple que $\mathbf{DTIME}(f) \subseteq \mathbf{NTIME}(f)$.

Las clases **DTIME** y **NTIME** nos van a permitir definir otras clases que clasifiquen los problemas de acuerdo a su complejidad de una forma que respete nuestra intuición.

Definición 21. (Clase P)

Se define la clase de complejidad **P** como

$$\mathbf{P} = \bigcup_{j=1}^{\infty} \mathbf{DTIME}(n^j).$$

Definición 22. (Clase NP)

Se define la clase de complejidad **NP** como

$$NP = \bigcup_{j=1}^{\infty} NTIME(n^j).$$

Volviendo sobre el problema de multiplicar dos números enteros de n cifras, cabe señalar que si se utiliza el método clásico, el que todos conocemos, se requieren $2n^2$ operaciones. En otras palabras, una máquina de Turing con alfabeto $\{0, 1, \dots, 9\}$ que reciba una palabra de entrada de longitud $2n$ ha de realizar $2n^2$ operaciones básicas para producir una salida que sea, efectivamente, el producto de dos números enteros. Por lo tanto, el algoritmo tradicional para realizar multiplicaciones pertenece a la clase **DTIME** (n^2) y, por tanto, a la clase **P**.

Por su parte, un ejemplo de problema de **NP** es de la *factorización de enteros*, esto es, dados $a, b, m \in \mathbb{Z}$, decidir si existe un número primo $p \in [a, b]$ tal que $p|m$. Este ejemplo resulta muy útil para ilustrar lo que ocurre en realidad con los problemas de **NP**. En la Sección 1.5 presentamos el modelo de la máquina de Turing indeterminista y adelantamos que suponía un aumento de la capacidad computacional. Debemos reparar en que solo las máquinas de Turing deterministas representan algoritmos tradicionales pues en las indeterministas las instrucciones que se han de seguir no quedan determinadas con unicidad a partir de la palabra de entrada. Si uno se ha enfrentado a un problema cualquiera, conoce que es holgadamente más sencillo el verificar si un objeto constituye una solución al mismo que el tratar de obtenerla partiendo de cero. Esto se debe a que el proceso de *verificación* es un mero proceso mecánico ajeno a cualquier atisbo de talento. Con esto en mente, podremos admitir que el indeterminismo aspira a ser una especie de formalización del talento. Exacto, podría decirse que una máquina de Turing indeterminista es más talentosa que una determinista y, por ello, cabe esperar que resuelva los problemas más eficientemente. Así, si en el problema de factorización de enteros, un ser superior con capacidad *adivinatoria* de una cierta naturaleza que no discutiremos aquí nos hace entrega de otro entero, digamos, q , el cual resulta ser tal que $q|p$, y nos insta a verificarlo mediante, por ejemplo, la división euclídea, el problema habrá pasado de estar en **NP** a estar en **P**. De esta forma, puede decirse que la clase **NP** es la clase **P** dotada de un especial talento adivinatorio. Lo que acabamos de plantear es una de vías que habitualmente se utilizan para comprender lo que supone la clase **NP**.

Es posible que el párrafo anterior resulte esperanzador pues ha aparecido una palabra que resulta tan agradable como es "talento", pero debemos ser conscientes de que tan solo estamos tratando con un modelo. Lo que si que es cierto es que los algoritmos, en el sentido estricto, son realmente necesarios para, al menos, verificar si contamos con la solución de un problema. A pesar de lo arduo de formalizar ideas tan intuitivas como con las que estamos lidiando, se ha convenido que los problemas tratables, esto es, los que pueden ser afrontados mediante algoritmos razonables, son exactamente aquellos para los cuales existe una máquina de Turing capaz de resolverlos con un coste temporal

de orden polinómico respecto a la longitud de la entrada, es decir, precisamente los de la clase **P**. Esto es lo que se conoce como la *tesis de Cobham* y supone que la clase **P** sea especialmente importante, quizás más aún que **NP**, al erigirla como el conjunto de los problemas tratables.

La *tesis de Cobham* cuenta con el respaldo de la robustez del modelo Turing, esto es, del hecho de que se respete la clase **P** aun cuando se permitan generalizaciones del modelo de Turing (máquina universal, distintos alfabetos,...). Sin embargo, es cierto que considerar tratable problemas que se encuentren en $\text{DTIME}(n^{1000000000})$ es cuanto menos acrobático debido a que ni en varias vidas el ordenador más potente con el que contamos hoy en día terminaría su computación. Asimismo, otra de las críticas a la tesis concierne precisamente al encumbramiento del determinismo en detrimento de las posibilidades de otros modelos como el cuántico. En efecto, el problema de la factorización de enteros puede ser resuelto mediante el *algoritmo de Shor* si se admite lo que se conoce como máquina de Turing cuántica. Sin embargo, este modelo no parece ser físicamente realizable, al menos, en la actualidad.

Por otra parte, es claro que $\mathbf{P} \subseteq \mathbf{NP}$ pues cada clase **DTIME** está contenida en su correspondiente clase **NTIME**. De esta manera y tras las consideraciones previas, la intuición nos dice que deberán existir algunos problemas de **NP** que sean verdaderamente difíciles, esto es, que no puedan ser tratables. Esta idea se formaliza mediante la siguiente definición.

Definición 23. (*Problema reducido*)

Sean \mathbb{L}_1 y \mathbb{L}_2 dos lenguajes asociados a sendos problemas de decisión. Si existe una aplicación $R : \Sigma^* \rightarrow \Sigma^*$ tal que

$$w \in \mathbb{L}_1 \iff R(w) \in \mathbb{L}_2,$$

se dice que un problema es reducible al otro y que es "al menos tan difícil" como él.

Lo habitual es tratar con ciertas reducciones como son la de Levin o la de Karp. Inmiserirse en los detalles de las mismas queda fuera de los propósitos de este trabajo (ver, por ejemplo, [10]). Sin embargo, si que debemos comentar que, dada una clase de complejidad, se dice que un problema es *duro* cuando todas sus reducciones siguen perteneciendo a la misma clase. De esta forma, se puede hablar de *clases duras* que serán aquellas formadas por los problemas verdaderamente difíciles de una clase dada. Asimismo, se habla de problemas *completos* en una cierta clase cuando todos los problemas de la propia clase son reducibles a él. De esta manera, los problemas completos captan en esencia todo el potencial de la clase y basta con que uno solo de ellos pertenezca a otra clase para que todos ellos lo hagan. En tal caso, se dice que ambas clases *colapsan*. Por su parte, resulta interesante la clase **NP**-dura, pues gran parte de problemas que contiene aparecen en casi cualquier ámbito y, en particular, en el de la lógica. Existen diversas vías para afrontar estos problemas que van desde la *fuerza bruta*, esto es, tratar de

verificar todas las soluciones posibles, hasta tratar de encontrar un algoritmo eficiente, lo que equivaldría a demostrar que el problema pertenece a la clase **P**, pasando por modificar el problema en otro equivalente que resulte ser fácilmente resoluble³. Por supuesto, siempre contamos con una última posibilidad: tirar la toalla.

Concluyamos esta sección llamando la atención sobre el hecho de que, tras introducirnos someramente en el ámbito de la teoría de la complejidad de la manera en que lo hacen los matemáticos, hemos acabado por confundir, pese a haber realizado varias llamadas de atención, los conceptos de *problema* y *algoritmo*. Hemos admitido en el párrafo anterior la posibilidad de que un problema verdaderamente difícil pueda ser reducido a uno fácil si se encuentra un algoritmo eficiente que lo resuelva. Entonces, debemos cuestionarnos hasta qué punto el problema realmente era difícil. En efecto, la clasificación de los problemas en base a su complejidad depende de los avances que vayan realizando los matemáticos y no es más que una rémora del anhelo por una concepción platónica de la matemática el tratar de defender una clasificación *a priori* de los problemas conforme a su complejidad.

2.4. ¿P = NP?

Desde que en 1971, tras tratar de demostrar inútilmente que $\mathbf{P} \subsetneq \mathbf{NP}$ y postular que $\mathbf{P} = \mathbf{NP}$, la conjetura lanzada por el matemático neoyorquino Stephen Cook no ha hecho más que ganar relevancia. Tanto es así que el Clay Mathematics Institute, ubicado en Cambridge, Massachusetts, ofrece desde el año 2000 un millón de dólares a aquel que sea capaz de dar una respuesta a su conjetura. En efecto, la conjetura de Cook ha sido catalogada como uno de los siete problemas del milenio. Sin embargo, ya anunciamos aquí que, de acuerdo a la opinión de la inmensa mayoría de los expertos en computación, el problema no será resuelto en este siglo.

El esquema de la demostración de que $\mathbf{P} = \mathbf{NP}$ habría de ser el siguiente: Encontrar un problema **NP**-duro que esté en **P**, esto es, hallar una máquina de Turing determinista que lo resuelva en tiempo polinómico respecto de la entrada; como el problema es duro, entonces cualquier otro problema de **NP** podría ser reducido a él y entonces tendríamos una máquina de Turing determinista que resuelve cualquier problema de **NP** en tiempo polinómico pero esto es tanto como decir que las clases **P** y **NP** colapsan. En otras palabras, lo que queremos transmitir es que si la conjetura fuese cierta, entonces dispondríamos de un algoritmo capaz de resolver hasta los problemas muy difíciles.

En base a lo anterior, la mayor parte de los artículos escritos por matemáticos

³Para que esto sea válido ha de cumplirse que lo que se conoce como *dependencia continua de los datos*, esto es, que si los datos (el problema) varían ligeramente, se espera que la solución varíe de la misma manera. Esta forma de proceder es habitual en el tratamiento de ecuaciones en derivadas parciales. Por ejemplo, no es posible hallar una solución analítica para la ecuación de Black-Schöles con determinadas condiciones pero, en cambio, el problema puede ser reformulado posibilitando que la solución original pueda ser aproximada con el grado de exactitud que se deseé.

defienden que no debe creerse que la conjetura es cierta pues supondría, entre otras cosas, que todas las enfermedades conocidas pudiesen ser perfectamente diagnosticadas, que todos los problemas de optimización con los que lidian día a día las empresas pudiesen ser resueltos, así como que la inteligencia artificial podría ser llevada al máximo exponente, permitiendo la creación de máquinas capaces de imitar procesos cognitivos humanos. Seguramente, también pudiesen realizarse predicciones fiables (no probabilísticas) sobre todo cuanto puede ser modelado mediante ecuaciones diferenciales, por ejemplo, el clima, los modelos sociológicos, ciertos aspectos de la economía, y un largo etcétera. Como contrapartida, deberíamos estar dispuestos a renunciar a los sistemas criptográficos que utilizamos en la actualidad, pues están basados en el problema de la factorización de enteros, y por ende, a la intimidad de nuestras comunicaciones.

Por su parte, sería también posible decidir el lenguaje formado por todos los teoremas, de una cierta longitud cualquiera, de una determinada teoría. Esto no es más que una versión finita del *Entscheidungsproblem* y podría tener consecuencias relacionadas con la *omnisciencia lógica*. En efecto, si la conjetura fuese cierta, entonces podríamos conocer automáticamente (en un tiempo polinómico) todas y cada una de las consecuencias lógicas de los axiomas, los cuales podrían ser simples hechos contrastados por la experiencia. Esto resulta inconcebible incluso para el propio Turing quien en [17] defiende que la aceptación de que tan pronto como somos conscientes de un hecho pasamos a ser conscientes de todas sus consecuencias es una asunción muy útil en diversas circunstancias pero falsa. Por su parte, resultaría que la incomputabilidad del *Entscheidungsproblem* no sería tan desalentadora pues los que realmente acaparan nuestro interés son los teoremas que pueden ser demostrados mediante una prueba relativamente tratable en lo que a términos de su longitud se refiere.

Parece ser que la mayor parte de los argumentos mostrados por la comunidad matemática para desestimar la veracidad de la conjetura de Cook están promovidos por el pavor que suscita el pensar que los problemas muy "difíciles" puedan ser resueltos por máquinas. Es más, aun cuando hubiésemos podido probar la falsedad de la conjetura de Cook, seguirían apareciendo interrogantes dignos de tratar. Las máquinas han demostrado en numerosas ocasiones que son más fiables que los humanos en diversos aspectos pero, por ejemplo, no son capaces de conducir vehículos de manera que se garantice la no ocurrencia de accidentes. Recientemente han aparecido en los medios de comunicación noticias tendenciosas calificando de fracaso el que un vehículo no pilotado por un humano haya sufrido un accidente como si los vehículos llevados por personas no los sufriesen. En efecto, cabe preguntarse en primer lugar si los humanos somos capaces de resolver problemas **NP**-duros. De ser así, si la conjetura fuese falsa, podríamos pensar que tenemos un buen argumento para afirmar que los humanos no pueden ser simulados por máquinas de Turing deterministas. En cambio, la experiencia nos dice que, salvo en contadas ocasiones, no existe razón para creer que los humanos pueden resolver tal clase de problemas. Basta atender a que frecuentemente no somos siquiera capaces de resolver correctamente problemas de **P** tales como certificar si un número dado arbitrario es primo.

En cualquier caso, lo que es cierto es que se viene manifestando continuamente un deseo por adaptar la realidad a los modelos matemáticos olvidando que tales modelos

no son mas que eso: modelos. Es inherente a las entretelas de la modelación el asumir determinadas hipótesis a fin de crear un modelo consistente con la siguiente condición: cuantas mas hipótesis se incluyan, menos realistas serán los modelos. Aun cuando esto es aceptado en cualquier rama de la ingeniería y de la física, por alguna razón, parece no aceptarse en el ámbito de la computación. Si reparamos en que todo lo que hemos tratado no es más que un modelo, entonces no cabe alarmarse por el hecho de que la conjetura de Cook pudiese ser verdadera. Efectivamente tendría consecuencias sobre el modelo que es la teoría de la computación y seguramente también ocasionaría efectos sobre determinados aspectos de nuestra vida diaria, pero nunca sobre nuestra condición de seres necesarios para la existencia de los problemas y de las máquinas.

2.5. REFERENCIAS BIBLIOGRÁFICAS.

Introducirse en la teoría de la complejidad es una tarea complicada para cualquiera. Además, este trabajo ha pretendido presentar las nociones fundamentales de manera que resultasen inteligibles incluso a aquellos desprovistos de una vasta formación matemática. Por ello, algunas de las obras que hemos utilizado pueden resultar demasiado densas si no se dispone de un amplio bagaje en el manejo simbólico.

- Los artículos de Pardo ([10]-[11]) realizan una clara exposición de las nociones técnicas de la teoría de la complejidad. En realidad, profundiza muchísimo más de lo que nosotros lo hemos hecho en este trabajo, presentando una ingente cantidad de resultados. Por su parte, quizás adolezcan de una carencia de apuntes filosóficos.
- Por su parte, el artículo de Aaronso ([1]) suple perfectamente la carencia de los artículos a los que nos acabamos de referir. En efecto, Aaronson presenta una amplia variedad de ideas originales al respecto de las consecuencias filosóficas de la complejidad computacional.
- Asimismo, el artículo de Cobham ([2]) también recoge ciertos apuntes que podríamos catalogar como filosóficos al respecto de las distintas clases de complejidad.
- Finalmente, las obra de Davis y Weyuker ([3]) y de Pudlak ([13]) constituyen sendos excelentes manuales que cubren holgadamente todos los aspecto técnicos que hemos tratado.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías Mexico

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

COMENTARIOS FINALES.

Como hemos visto a lo largo del trabajo, durante el siglo XX se propusieron diversos modelos computacionales de naturalezas radicalmente diferentes para afrontar el problema de modelizar lo que se entiende por resolver un problema. Así, pronto se planteó la cuestión al respecto de cuál de todos los modelos propuestos era el correcto y si es que alguno habría de merecer tal reconocimiento realmente. Puesto que todos ellos cumplían ciertas propiedades que se habían considerado necesarias (*efectividad* y *completitud*), los interrogantes anteriores fueron remplazados por la cuestión de cuál de todos era el más natural. Por supuesto, la naturalidad resulta ser una propiedad totalmente subjetiva pero, sin embargo, es cierto que propició la aparición de ciertos resultados que, al contrario de lo que suele acontecer en matemáticas, apaciguaron el debate en beneficio del consenso.

En 1934, Kleene intentó demostrar que toda función que se pudiese concebir como computable resultaría ser λ -definible. Tan solo unos meses después, Church conjeturó que las funciones computables eran exactamente aquellas funciones que eran λ -definibles, esto es, que las nociones intuitivas de algoritmo y computación eran capturadas con total precisión por su modelo computacional. A esta suposición se la conoce habitualmente como *tesis de Church*. Aproximadamente dos años después, Turing realizó una conjetura análoga que involucraba al modelo computacional que él mismo había propuesto. Como cabía esperar, a su conjetura se la denominó como *tesis de Turing*.

En 1937, Turing logró demostrar que su modelo computacional y el de Church eran equivalentes en el sentido de que cualquier computación que pudiese ser realizada por uno también podría ser llevada a cabo por el otro. Puesto que el modelo de Turing resultaba ser notablemente menos engorroso, pronto fue aceptado por la comunidad matemática como el idóneo para asir las nociones intuitivas de algoritmo y computación. Esto es lo que se dió en llamar *tesis de Church-Turing*, aunque recientemente el término ha evolucionado en *tesis de la computación* ([14]). Poco a poco se fue demostrando que, en realidad, todos los modelos antes mencionados resultan ser equivalentes en el mismo sentido.

Todo lo anterior nos debe hacer caer en la cuenta de que, independientemente del entorno en el que se conciban los problemas, sus características intrínsecas más esenciales resultan ser similares. En mi opinión, esto se debe a que los problemas, incluso cuando puedan presentarse de las más dispares maneras, dada nuestra condición de seres cognoscitivos necesarios para la existencia de los mismos, deben conservar alguna semejanza subyacente a la hora de ser comprendidos pues, al fin y al cabo, quienes lo hacemos somos todos igualmente humanos. De esta manera, podría utilizarse la tesis de

Church-Turing para aducir que los problemas carecen de una naturaleza objetiva propia pues, de ser así, parece improbable que encajen de la manera en que lo hacen en todos y cada uno de los modelos sugeridos por los distintos matemáticos.

Es un hecho que toda la teoría de la computación surge a partir del inmenso desapego de los matemáticos por las paradojas. Actualmente, algunos investigadores entre los que cabe destacar a Graham Priest, apuntan que posiblemente se deban aceptar las paradojas como objetos inherentes a la propia existencia humana y a sus consiguientes límites. Es un hecho que convivimos con las limitaciones, basta observar que solo disponemos de un tiempo limitado de vida. En esta línea, Priest manifiesta que las paradojas aparecen cuando sobrepasamos los límites del pensamiento y, puesto que las distintas clases de complejidad que hemos tratado parecen pretender lidiar con ciertos límites en cuanto a aquello que podemos resolver, podríamos pensar que, al cuestionarnos si $P = NP$, hemos caído en una paradoja justo al trascender precisamente el límite que atañe a la complejidad. Asimismo, en mi opinión la comunidad matemática ha pecado de pesimista, agorera e insidiosa al defender que si la conjetura de Cook fuese cierta, entonces nos veríamos sobrepasados por la capacidad de las máquinas. Debemos recordar el carácter contingente de las mismas y, en consecuencia, reconocer que la capacidad computacional de las máquinas es precisamente la nuestra, pues no constituirían sino una herramienta desarrollada por humanos para obtener soluciones complejas.

Por otra parte, se debe tener presente que hemos ido renunciando a la generalidad de los problemas. Hemos seguido la reducción de problema, problema matemático, problema de decisión previa codificación pertinente, y sin embargo, hemos pretendido extrapolar las consecuencias del modelo como si tal reducción no se hubiese realizado. Quizás se deba a la especial idiosincrasia de las matemáticas pero deberíamos, al menos, reflexionar sobre si hemos olvidado la esencia original de los problemas. El nacimiento de la teoría que hemos expuesto estaba ligado a la fundamentación de ciertos conceptos y, como consecuencia del formalismo, esos conceptos abstractos se han distorsionado en un compendio de fórmulas extrañas, perdiendo de vista que tales fórmulas no aspiran a ser más que eso, símbolos formales.

No queremos decir que la teoría de la computación sea inútil. Ni mucho menos. Gracias al desapego de los matemáticos por las paradojas poseemos hoy en día avances tecnológicos que eran impensables años atrás. Pero si que sería adecuado que la comunidad matemática fuese consciente del paradigma filosófico del cual surgió gran parte de los conceptos que se estudian hoy en día. Es posible que la especialización y la prisa a la que nos somete la sociedad constituyan graves impedimentos de cara a realizar tales divagaciones filosóficas, pero debemos promoverlo pues si es cierto que el *programa de Hilbert* no pudo sobreponerse a la fuerza de las contradicciones, no es menos cierto que supuso una suerte de Ítaca para la comunidad matemática y que, aún cuando nunca atracase en sus aguas, fue el propio viaje lo que realmente resultó enriquecedor *per se*. Por ello, no resulta descabellado plantearse si sería propicio plantear una suerte de *programa de Hilbert* que abrazase las paradojas y que las integrase como parte del propio pensamiento humano. Quien sabe lo que podría deparar ese otro viaje.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

A

APÉNDICE.

A.1. SUMAR EN DISTINTOS MODELOS COMPUTACIONALES.

El presente apéndice pretende constituir una suerte de argumento visual que ponga de manifiesto la simplicidad del modelo de Turing frente al resto. Lo que en ningún caso es nuestro objetivo es el de presentar una guía introductoria a los otros modelos computacionales. Para ello, existen obras en la literatura al respecto tales como [6].

MODELO DE KLEENE.

Vamos a tratar de construir la función suma usual

$$S(n, m) := n + m.$$

Para computar $S(n, m)$ deberemos haber computado anteriormente $S(n, m - 1)$. Iterando esta idea, es claro que lo primero que se ha de computar es $S(n, 0)$. De esta manera, para conseguir construir S vamos a requerir de ciertas funciones intermedias, concretamente, $\pi_1^1(x)$, $\pi_3^3(x_1, x_2, x_3)$, $\sigma(x)$, $g_1(x_1, x_2, x_3)$ y $f_4(x_1, x_2)$. El objetivo es llegar a que $S \equiv f_5$. Para ello hay que seguir las reglas de formación relativas a este modelo.

- $f_1 \equiv \pi_1^1(n);$
- $f_2 \equiv \pi_3^3(n, m, p);$
- $f_3 \equiv \sigma(n);$

- $f_4(n, m, p)$ es la composición de f_3 y f_4 ;
- $f_5(n, m)$ es la composición de f_1 y f_4 .

Notemos que

$$f_5(n, m) = f_4(n, m - 1, f_5(n, m - 1)) = f_5(n, m - 1) + 1 = \dots = f_5(n, 0) + m = \pi_1^1(n) + m = n + m.$$

Por lo tanto, es cierto que $f_5 \equiv S$ y así la función suma resulta ser computable para el modelo de Kleene.

MODELO DE CHURCH.

Veamos que la función suma usual puede definirse en el λ -cálculo mediante un λ -término. Antes, vamos a presentar las dos reglas de transformación aceptadas en el λ -cálculo:

1. α -conversion. Se denota por \rightarrow_α y sirve para renombrar una determinada variable.
2. β -contraccion. Se denota por \rightarrow_β y permite transformar $(\lambda x.M)N$ en un λ -término haciendo $(\lambda x.M)N \equiv M[x := N]$.

Lo que se pretendemos es definir la función suma usual $S(n, m) := n + m$ como un λ -término $S \equiv \lambda abfx.af(bfx)$. Para ello, debemos verificar que, para cuales quiera que sean n y m , se tiene $Sc_n c_m \rightarrow_\beta \dots \rightarrow_\beta c_{n+m}$. Veámoslo.

$$\begin{aligned} Sc_n c_m &\equiv (Sc_n)c_m \equiv ((\lambda abfx.af(bfx))c_n)c_m \rightarrow_\beta (\lambda bfx.c_nf(bfx))c_m \rightarrow_\beta \\ &\lambda fx.c_nf(c_mfx) \equiv \lambda fx.(\lambda fx.f^n x)(f((\lambda fx.f^m x)fx) \rightarrow_\beta \lambda fx.(\lambda x.f^n x)((\lambda fx.f^m x)fx) \rightarrow_\beta \\ &\lambda fx.(\lambda x.f^n x)((\lambda x.f^m x)x) \rightarrow_\beta \lambda fx.(\lambda x f^n x)(f^m x) \rightarrow_\beta \lambda fx.f^n f^m x \equiv \lambda fx.f^{n+m} x \equiv \\ &c_{n+m}, \text{ como queríamos probar.} \end{aligned}$$

MODELO DE TURING.

Vamos a construir ahora una máquina de Turing que transforme la palabra de entrada $1^{(n)}01^{(m)}$ en $1^{(n+m)}$. Utilizando esta codificación, la máquina sera capaz de computar la función suma usual.

Formalmente, la máquina ha de ser $T = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, \sqcup\}, \delta, q_1, \sqcup, \{q_3\})$, donde la función de transición viene dada por

- $\delta(q_1, 1) = (q_2, \sqcup, Derecha);$
- $\delta(q_2, 1) = (q_2, 1, Derecha);$
- $\delta(q_2, 0) = (q_3, 1, Permanecer);$

- $\delta(q_1, 0) = (q_3, \sqcup, Permanecer)$.

Supongamos que $1^{(n)}01^{(m)}$ es la palabra de entrada y que n es no nulo. La secuencia de configuraciones internas de la máquina sería la siguiente $q_11^{(n)}01^{(m)} \rightarrow q_21^{(n-1)}01^{(m)} \rightarrow 1q_21^{(n-2)}01^{(m)} \rightarrow \dots \rightarrow 1^{(n-1)}q_201^{(m)} \rightarrow 1^{(n-1)}q_31^{(m+1)}$. Como el estado q_3 es final, la máquina se detiene arrojando la salida $1^{(n-2)}1^{(m)} = 1^{(n+m)}$. Si $n = 0$ y $m \neq 0$, al introducir la palabra de entrada $1^{(n)}01^{(m)}$, se ejecuta una única instrucción con resultado $1^{(m)}$. En cualquier caso, la máquina produce el resultado deseado y, por tanto, la función suma usual resulta ser Turing-computable.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

BIBLIOGRAFÍA.

- [1] Aaronson, S. (2011). "Why philosophers should care about computational complexity". <http://www.scottaaronson.com/blog/?p=735>.
- [2] Cobham, A. (1964). "The intrinsic computational difficulty of functions". *Proceedings of the 1964 international congress for logic, methodology and philosophy of science*, pp. 24-30.
- [3] Davis, M.D. & Weyuker, E.J. (1983). *Computability, complexity and languages*. San Diego, Academic Press.
- [4] Gray, J.J. (2006). *El reto de Hilbert*. Crítica.
- [5] Hintikka, J. (1962). *Knowledge and belief*. Cornell University Press.
- [6] Kleene, S.C. (1936). λ -definability and recursiveness. *Duke Mathematical Journal*.
- [7] Kolmogórov, A.N., Dragalin, A. G. (2013). *Lógica matemática. Introducción a la lógica matemática*. Traducción del ruso por Carlos Daniel Navarro Hernández y Juan Enrique Palomino Pérez. Hayka libros.
- [8] Kolmogórov, A.N., Dragalin, A. G. (2013). *Lógica matemática. Capítulos complementarios*. Traducción del ruso por Carlos Daniel Navarro Hernández y Juan Enrique Palomino Pérez. Hayka libros.
- [9] Martin, J. C. (2003). *Lenguajes formales y teoría de la computación*. McGraw-Hill.
- [10] Pardo, L.M. (2012). "La conjetura de Cook (¿P=NP?). Parte I: Lo básico". *La gaceta de la RSME*, Vol. 15, No. 1, pp. 117-147.
- [11] Pardo, L.M. (2012). "La conjetura de Cook (¿P=NP?). Parte II: Probabilidad, interactividad y comprobación probabilística de las demostraciones". *La gaceta de la RSME*, Vol. 15, No. 2, pp. 303-333.
- [12] Priest, G. (2002). *Beyond the limits of thought*. Oxford University Press.
- [13] Pudlak, P. (2013). *Logical foundations of mathematical and computational complexity*. Springer.
- [14] Robic, B. (2015). *The foundations of computability theory*. Springer.
- [15] Salomaa, A. (1973). *Formal languages*. Nueva York, Academic Press.
- [16] Shor, P.W. (1997). "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer". *SIAM J. Comput*, Vol. 26, No. 5, pp. 1484-1509.

- [17] Turing, A.M. (1936). "On computable numbers, with an application to the Entscheidungsproblem". *Proceedings, London Mathematical Society*, Vol. 2, No. 42, pp. 230-265.

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

**Esta guía fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados a
Esta guía de estudio, fue vendida por www.gulaceneval.com.mx Todos los Derechos Reservados.**

FEDICT:_GUÍA_MEX - Editorial Guías México

Lógica y Filosofía de la Ciencia

Máster Interuniversitario

Declaración de integridad intelectual

Trabajo Fin Máster

Curso 2016-2017

Título del trabajo: Panorámica de la teoría computacional: Desarrollo y problemas abiertos.

1. Sé que copiar es una forma de deshonestidad académica.
2. He leído el documento sobre cómo ser intelectualmente íntegro, estoy familiarizado con sus contenidos y he evitado todas las formas de plagio allí recogidas.
3. Cuando utilizo las palabras de otros, lo indico mediante el uso de comillas.
4. He referenciado todas las citas e igualmente el resto de ideas tomadas de otros.
5. No he plagiado mi propio trabajo.
6. No permitiré a otros que plagien mi trabajo.

Fecha: 10 de julio de 2017.

Firma:



Lectura 19. Algoritmos y Estructuras de Datos

Algoritmos y Estructuras de Datos

Bottazzi, Cristian. cristian.bottazzi@gmail.com,
Costarelli, Santiago. santi.costarelli@gmail.com,
D'Elía, Jorge. jdelia@intec.unl.edu.ar,
Dalcin, Lisandro. dalcinl@gmail.com,
Galizzi, Diego. dgalizzi@gmail.com,
Giménez, Juan Marcelo. jmarcelogimenez@gmail.com,
Olivera, José. joseolivera123@gmail.com,
Novara, Pablo. zaskar_84@yahoo.com.ar,
Paz, Rodrigo. rodrigo.r.paz@gmail.com,
Prigioni, Juan. jdprigioni@gmail.com,
Pucheta, Martín. martinpucheta@gmail.com,
Rojas Fredini, Pablo Sebastián. floyd.the.rabbit@gmail.com,
Storti, Mario. mario.storti@gmail.com,

www: <http://www.cimec.org.ar/aed>
Facultad de Ingeniería y Ciencias Hídricas
Universidad Nacional del Litoral <http://fich.unl.edu.ar>
Centro de Investigación de Métodos Computacionales
<http://www.cimec.org.ar>

(Document version: aed-3.1-12-gc28b6c4c)
(Date: Thu Aug 17 16:54:16 2017 -0300)

Índice

1. Diseño y análisis de algoritmos	9
1.1. Conceptos básicos de algoritmos	9
1.1.1. Ejemplo: Sincronización de acceso a objetos en cálculo distribuido	10
1.1.2. Introducción básica a grafos	11
1.1.3. Planteo del problema mediante grafos	11
1.1.4. Algoritmo de búsqueda exhaustiva	12
1.1.5. Generación de las coloraciones	13
1.1.6. Crecimiento del tiempo de ejecución	15
1.1.7. Búsqueda exhaustiva mejorada	16
1.1.8. Algoritmo heurístico ávido	19
1.1.9. Descripción del algoritmo heurístico en seudo-código	21
1.1.10. Crecimiento del tiempo de ejecución para el algoritmo ávido	26
1.1.11. Conclusión del ejemplo	27
1.2. Tipos abstractos de datos	27
1.2.1. Operaciones abstractas y características del TAD CONJUNTO	28
1.2.2. Interfaz del TAD CONJUNTO	28
1.2.3. Implementación del TAD CONJUNTO	30
1.3. Tiempo de ejecución de un programa	30
1.3.1. Notación asintótica	32
1.3.2. Invariancia ante constantes multiplicativas	33
1.3.3. Invariancia de la tasa de crecimiento ante valores en un conjunto finito de puntos	33
1.3.4. Transitividad	33
1.3.5. Regla de la suma	34
1.3.6. Regla del producto	34
1.3.7. Funciones típicas utilizadas en la notación asintótica	34
1.3.8. Equivalencia	36
1.3.9. La función factorial	36
1.3.10. Determinación experimental de la tasa de crecimiento	37
1.3.11. Otros recursos computacionales	38
1.3.12. Tiempos de ejecución no-polinomiales	39
1.3.13. Problemas P y NP	39
1.3.14. Varios parámetros en el problema	40
1.4. Conteo de operaciones para el cálculo del tiempo de ejecución	40

1.4.1. Bloques if	40
1.4.2. Lazos	41
1.4.3. Suma de potencias	45
1.4.4. Llamadas a rutinas	45
1.4.5. Llamadas recursivas	46
2. Tipos de datos abstractos fundamentales	48
2.1. El TAD Lista	48
2.1.1. Descripción matemática de las listas	49
2.1.2. Operaciones abstractas sobre listas	49
2.1.3. Una interfaz simple para listas	50
2.1.4. Funciones que retornan referencias	52
2.1.5. Ejemplos de uso de la interfaz básica	54
2.1.6. Implementación de listas por arreglos	58
2.1.6.1. Eficiencia de la implementación por arreglos	63
2.1.7. Implementación mediante celdas enlazadas por punteros	64
2.1.7.1. El tipo posición	65
2.1.7.2. Celda de encabezamiento	66
2.1.7.3. Las posiciones begin() y end()	68
2.1.7.4. Detalles de implementación	69
2.1.8. Implementación mediante celdas enlazadas por cursores	70
2.1.8.1. Cómo conviven varias celdas en un mismo espacio	72
2.1.8.2. Gestión de celdas	73
2.1.8.3. Analogía entre punteros y cursores	73
2.1.9. Tiempos de ejecución de los métodos en las diferentes implementaciones.	76
2.1.10. Interfaz STL	77
2.1.10.1. Ventajas de la interfaz STL	77
2.1.10.2. Ejemplo de uso	78
2.1.10.2.1. Uso de templates y clases anidadas	78
2.1.10.2.2. Operadores de incremento prefijo y postfijo:	78
2.1.10.3. Detalles de implementación	79
2.1.10.4. Listas doblemente enlazadas	82
2.2. El TAD pila	82
2.2.1. Una calculadora RPN con una pila	83
2.2.2. Operaciones abstractas sobre pilas	84
2.2.3. Interfaz para pila	84
2.2.4. Implementación de una calculadora RPN	85
2.2.5. Implementación de pilas mediante listas	88
2.2.6. La pila como un adaptador	89
2.2.7. Interfaz STL	90
2.3. El TAD cola	90
2.3.1. Intercalación de vectores ordenados	91
2.3.1.1. Ordenamiento por inserción	91
2.3.1.2. Tiempo de ejecución	93

2.3.1.3. Particularidades al estar las secuencias pares e impares ordenadas	93
2.3.1.4. Algoritmo de intercalación con una cola auxiliar	94
2.3.2. Operaciones abstractas sobre colas	95
2.3.3. Interfaz para cola	95
2.3.4. Implementación del algoritmo de intercalación de vectores	96
2.3.4.1. Tiempo de ejecución	97
2.4. El TAD correspondencia	97
2.4.1. Interfaz simple para correspondencias	100
2.4.2. Implementación de correspondencias mediante contenedores lineales	102
2.4.3. Implementación mediante contenedores lineales ordenados	103
2.4.3.1. Implementación mediante listas ordenadas	105
2.4.3.2. Interfaz compatible con STL	106
2.4.3.3. Tiempos de ejecución para listas ordenadas	109
2.4.3.4. Implementación mediante vectores ordenados	110
2.4.3.5. Tiempos de ejecución para vectores ordenados	112
2.4.4. Definición de una relación de orden	113
3. Árboles	114
3.1. Nomenclatura básica de árboles	114
3.1.0.0.1. Altura de un nodo	116
3.1.0.0.2. Profundidad de un nodo. Nivel.	116
3.1.0.0.3. Nodos hermanos	116
3.2. Orden de los nodos	116
3.2.1. Particionamiento del conjunto de nodos	117
3.2.2. Listado de los nodos de un árbol	119
3.2.2.1. Orden previo	119
3.2.2.2. Orden posterior	119
3.2.2.3. Orden posterior y la notación polaca invertida	120
3.2.3. Notación Lisp para árboles	121
3.2.4. Reconstrucción del árbol a partir de sus órdenes	122
3.3. Operaciones con árboles	124
3.3.1. Algoritmos para listar nodos	124
3.3.2. Inserción en árboles	125
3.3.2.1. Algoritmo para copiar árboles	126
3.3.3. Supresión en árboles	128
3.3.4. Operaciones básicas sobre el tipo árbol	129
3.4. Interfaz básica para árboles	129
3.4.1. Listados en orden previo y posterior y notación Lisp	132
3.4.2. Funciones auxiliares para recursión y sobrecarga de funciones	133
3.4.3. Algoritmos de copia	133
3.4.4. Algoritmo de poda	133
3.5. Implementación de la interfaz básica por punteros	134
3.5.1. El tipo iterator	134
3.5.2. Las clases cell e iterator.t	136

3.5.3. La clase tree	139
3.6. Interfaz avanzada	141
3.6.1. Ejemplo de uso de la interfaz avanzada	145
3.7. Tiempos de ejecución	148
3.8. Árboles binarios	148
3.8.1. Listados en orden simétrico	149
3.8.2. Notación Lisp	149
3.8.3. Árbol binario lleno	150
3.8.4. Operaciones básicas sobre árboles binarios	150
3.8.5. Interfaces e implementaciones	151
3.8.5.1. Interfaz básica	151
3.8.5.2. Ejemplo de uso. Predicados de igualdad y espejo	151
3.8.5.3. Ejemplo de uso. Hacer espejo “in place”	153
3.8.5.4. Implementación con celdas enlazadas por punteros	154
3.8.5.5. Interfaz avanzada	159
3.8.5.6. Ejemplo de uso. El algoritmo apply y principios de programación funcional.	160
3.8.5.7. Implementación de la interfaz avanzada	161
3.8.6. Árboles de Huffman	165
3.8.6.1. Condición de prefijos	166
3.8.6.2. Representación de códigos como árboles de Huffman	166
3.8.6.3. Códigos redundantes	167
3.8.6.4. Tabla de códigos óptima. Algoritmo de búsqueda exhaustiva	168
3.8.6.4.1. Generación de los árboles	169
3.8.6.4.2. Agregando un condimento de programación funcional	171
3.8.6.4.3. El algoritmo de combinación	173
3.8.6.4.4. Función auxiliar que calcula la longitud media	174
3.8.6.4.5. Uso de comb y codeLEN	176
3.8.6.5. El algoritmo de Huffman	176
3.8.6.6. Implementación del algoritmo	178
3.8.6.7. Un programa de compresión de archivos	181
4. Conjuntos	191
4.1. Introducción a los conjuntos	191
4.1.1. Notación de conjuntos	191
4.1.2. Interfaz básica para conjuntos	192
4.1.3. Análisis de flujo de datos	193
4.2. Implementación por vectores de bits	198
4.2.1. Conjuntos universales que no son rangos contiguos de enteros	199
4.2.2. Descripción del código	200
4.3. Implementación con listas	202
4.3.0.1. Similaridad entre los TAD conjunto y correspondencia	202
4.3.0.2. Algoritmo lineal para las operaciones binarias	202
4.3.0.3. Descripción de la implementación	204
4.3.0.4. Tiempos de ejecución	207

4.4. Interfaz avanzada para conjuntos	207
4.5. El diccionario	210
4.5.1. La estructura tabla de dispersión	210
4.5.2. Tablas de dispersión abiertas	211
4.5.2.1. Detalles de implementación	212
4.5.2.2. Tiempos de ejecución	214
4.5.3. Funciones de dispersión	215
4.5.4. Tablas de dispersión cerradas	216
4.5.4.1. Costo de la inserción exitosa	217
4.5.4.2. Costo de la inserción no exitosa	219
4.5.4.3. Costo de la búsqueda	220
4.5.4.4. Supresión de elementos	220
4.5.4.5. Costo de las funciones cuando hay supresión	221
4.5.4.6. Reinserción de la tabla	221
4.5.4.7. Costo de las operaciones con supresión	222
4.5.4.8. Estrategias de redispersión	223
4.5.4.9. Detalles de implementación	224
4.6. Conjuntos con árboles binarios de búsqueda	227
4.6.1. Representación como lista ordenada de los valores	227
4.6.2. Verificar la condición de ABB	228
4.6.3. Mínimo y máximo	229
4.6.4. Buscar un elemento	229
4.6.5. Costo de mínimo y máximo	230
4.6.6. Operación de inserción	232
4.6.7. Operación de borrado	233
4.6.8. Recorrido en el árbol	234
4.6.9. Operaciones binarias	235
4.6.10. Detalles de implementación	235
4.6.11. Tiempos de ejecución	240
4.6.12. Balanceo del árbol	240
5. Ordenamiento	241
5.1. Introducción	241
5.1.1. Relaciones de orden débiles	241
5.1.2. Signatura de las relaciones de orden. Predicados binarios.	242
5.1.3. Relaciones de orden inducidas por composición	245
5.1.4. Estabilidad	246
5.1.5. Primeras estimaciones de eficiencia	246
5.1.6. Algoritmos de ordenamiento en las STL	246
5.2. Métodos de ordenamiento lentos	247
5.2.1. El método de la burbuja	247
5.2.2. El método de inserción	248
5.2.3. El método de selección	249
5.2.4. Comparación de los métodos lentos	250

5.2.5. Estabilidad	250
5.3. Ordenamiento indirecto	251
5.3.1. Minimizar la llamada a funciones	253
5.4. El método de ordenamiento rápido, quick-sort	253
5.4.1. Tiempo de ejecución. Casos extremos	255
5.4.2. Elección del pivote	256
5.4.3. Tiempo de ejecución. Caso promedio.	258
5.4.4. Dispersión de los tiempos de ejecución	260
5.4.5. Elección aleatoria del pivote	261
5.4.6. El algoritmo de partición	261
5.4.7. Tiempo de ejecución del algoritmo de particionamiento	262
5.4.8. Búsqueda del pivote por la mediana	263
5.4.9. Implementación de quick-sort	264
5.4.10. Estabilidad	265
5.4.11. El algoritmo de intercambio (swap)	265
5.4.12. Tiempo de ejecución del quick-sort estable	269
5.5. Ordenamiento por montículos	270
5.5.1. El montículo	271
5.5.2. Propiedades	272
5.5.3. Inserción	273
5.5.4. Costo de la inserción	274
5.5.5. Eliminar el mínimo. Re-heap.	274
5.5.6. Costo de re-heap	276
5.5.7. Implementación in-place	276
5.5.8. El procedimiento make-heap	277
5.5.9. Implementación	279
5.5.10. Propiedades del ordenamiento por montículo	280
5.6. Ordenamiento por fusión	280
5.6.1. Implementación	282
5.6.2. Estabilidad	283
5.6.3. Versión estable de split	283
5.6.4. Merge-sort para vectores	284
5.6.5. Ordenamiento externo	286
5.7. Comparación de algunas implementaciones de algoritmos de ordenamiento	287
6. GNU Free Documentation License	289

Sobre este libro:

Este libro corresponde al curso *Algoritmos y Estructura de Datos* que se dicta en la currícula de *Ingeniería Informática y Analista en Informática Aplicada* de la *Facultad de Ingeniería y Ciencias Hídricas* (<http://www.fich.unl.edu.ar>) de la *Universidad Nacional del Litoral* (<http://www.unl.edu.ar>).

Página web del curso: La página web del curso es <http://www.cimec.org.ar/aed>. En esa página funciona un wiki, listas de correo y un repositorio de archivos donde se puede bajar la mayor parte del código que figura en el libro. Este libro se puede bajar en formato PDF de esa página también.

Licencia de uso: This book is Copyright (c) 2004-2016, Bottazzi, Cristian; Costarelli, Santiago; D'Elía, Jorge; Dalcin, Lisandro; Galizzi, Diego; Giménez, Juan Marcelo; Olivera, José; Novara, Pablo; Paz, Rodrigo; Prigioni, Juan; Pucheta, Martín; Rojas Fredini, Pablo Sebastián; Storti, Mario; Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included below in the section entitled "GNU Free Documentation License".

Utilitarios usados: Todo este libro ha sido escrito con utilitarios de software libre, de acuerdo a los lineamientos de la *Free Software Foundation/GNU Project* (<http://www.gnu.org>). La mayoría de los utilitarios corresponden a un sistema **Fedora release 20 (Heisenbug) Kernel 3.16.6-200.fc20.x86_64**.

- El libro ha sido escrito en \LaTeX y convertido a PDF con **pdflatex**. El libro está completamente interreferenciado usando las utilidades propias de \LaTeX y el paquete **hyperref**.
- Muchos de los ejemplos con un matiz matemáticos han sido parcialmente implementados en *Octave* (<http://www.octave.org>). También muchos de los gráficos.
- Los ejemplos en C++ han sido desarrollados y probados con el compilador **GCC 4.8.3 20140911 (Red Hat 4.8.3-7) (GCC)** (<http://gcc.gnu.org>) y con la ayuda de **GNU Make 3.82** <http://www.gnu.org/software/make/make.html>.
- Las figuras han sido generadas con *Inkscape 0.48.5 r10040 (Jul 22 2014)* (<http://inkscape.sourceforge.net/>) y *Xfig 3.2.4-21.1* (<http://www.xfig.org/>).
- El libro ha sido escrito en forma colaborativa por los autores usando Git 1.9.3 (<http://git-scm.com/>).

Errores Al final del libro hay un capítulo dedicado a reportar los errores que se van detectando y corrigiendo. Este capítulo se irá publicando en la página web por separado, de manera que si Ud. posee una versión anterior del libro puede bajar las erratas y corregir su versión.

Si encuentra algún error en el libro le agradecemos reportarlo a cualquiera de los autores, indicando la versión del libro, tal como aparece en la portada.

Capítulo 1

Diseño y análisis de algoritmos

1.1. Conceptos básicos de algoritmos

No existe una regla precisa para escribir un programa que resuelva un dado problema práctico. Al menos por ahora escribir programas es en gran medida un arte. Sin embargo con el tiempo se han desarrollado un variedad de conceptos que ayudan a desarrollar estrategias para resolver problemas y comparar *a priori* la eficiencia de las mismas.

Por ejemplo supongamos que queremos resolver el “*Problema del Agente Viajero*” (TSP, por “*Traveling Salesman Problem*”) el cual consiste en encontrar el orden en que se debe recorrer un cierto número de ciudades (esto es, una serie de puntos en el plano) en forma de tener un recorrido mínimo. Este problema surge en una variedad de aplicaciones prácticas, por ejemplo encontrar caminos mínimos para recorridos de distribución de productos o resolver el problema de “*la vuelta del caballo en el tablero de ajedrez*”, es decir, encontrar un camino para el caballo que recorra toda las casillas del tablero pasando una sola vez por cada casilla. Existe una estrategia (trivial) que consiste en evaluar todos los caminos posibles. Pero esta estrategia de “*búsqueda exhaustiva*” tiene un gran defecto, el costo computacional crece de tal manera con el número de ciudades que deja de ser aplicable a partir de una cantidad relativamente pequeña. Otra estrategia “*heurística*” se basa en buscar un camino que, si bien no es el óptimo (el de menor recorrido sobre todos los posibles) puede ser relativamente bueno en la mayoría de los casos prácticos. Por ejemplo, empezar en una ciudad e ir a la más cercana que no haya sido aún visitada hasta recorrerlas todas.

Una forma abstracta de plantear una estrategia es en la forma de un “*algoritmo*”, es decir una secuencia de instrucciones cada una de las cuales representa una tarea bien definida y puede ser llevada a cabo en una cantidad finita de tiempo y con un número finito de recursos computacionales. Un requerimiento fundamental es que el algoritmo debe terminar en un número finito de pasos, de esta manera él mismo puede ser usado como una instrucción en otro algoritmo más complejo.

Entonces, comparando diferentes algoritmos para el TSP entre sí, podemos plantear las siguientes preguntas

- ¿Da el algoritmo la solución óptima?
- Si el algoritmo es iterativo, ¿converge?
- ¿Cómo crece el esfuerzo computacional a medida que el número de ciudades crece?

1.1.1. Ejemplo: Sincronización de acceso a objetos en cálculo distribuido

Consideremos un sistema de procesamiento con varios procesadores que acceden a un área de memoria compartida. En memoria hay una serie de objetos O_0, O_1, \dots, O_{n-1} , con $n = 10$ y una serie de tareas a realizar T_0, T_1, \dots, T_{m-1} con $m = 12$. Cada tarea debe modificar un cierto subconjunto de los objetos, según la siguiente tabla

- T_0 modifica O_0, O_1 y O_3 .
- T_1 modifica O_4 y O_5 .
- T_2 modifica O_4 .
- T_3 modifica O_2 y O_6 .
- T_4 modifica O_1 y O_4 .
- T_5 modifica O_4 y O_7 .
- T_6 modifica O_0, O_2, O_3 y O_6 .
- T_7 modifica O_1, O_7, O_8 .
- T_8 modifica O_5, O_7 y O_9 .
- T_9 modifica O_3 .
- T_{10} modifica O_6, O_8 y O_9 .
- T_{11} modifica O_9 .

Las tareas pueden realizarse en cualquier orden, pero dos tareas *no pueden ejecutarse al mismo tiempo si acceden al mismo objeto*, ya que los cambios hechos por una de ellas puede interferir con los cambios hechos por la otra. Debe entonces desarrollarse un sistema que sincronice entre sí la ejecución de las diferentes tareas.

Una forma trivial de sincronización es ejecutar cada una de las tareas en forma secuencial. Primero la tarea T_0 luego la T_1 y así siguiendo hasta la T_{11} , de esta forma nos aseguramos que no hay conflictos en el acceso a los objetos. Sin embargo, esta solución puede estar muy lejos de ser óptima en cuanto al tiempo de ejecución ya que por ejemplo T_0 y T_1 pueden ejecutarse al mismo tiempo en diferentes procesadores ya que modifican diferentes objetos. Si asumimos, para simplificar, que todas las tareas llevan el mismo tiempo de ejecución τ , entonces la versión trivial consume una cantidad de tiempo $m\tau$, mientras que ejecutando las tareas T_0 y T_1 al mismo tiempo reducimos el tiempo de ejecución a $(m - 1)\tau$.

El algoritmo a desarrollar debe “*particionar*” las tareas en una serie de p “*etapas*” E_0, \dots, E_p . Las etapas son simplemente subconjuntos de las tareas y la partición debe satisfacer las siguientes restricciones

- Cada tarea debe estar en una y sólo una etapa. (De lo contrario la tarea no se realizaría o se realizaría más de una vez, lo cual es redundante. En el lenguaje de la teoría de conjuntos, estamos diciendo que debemos particionar el conjunto de etapas en un cierto número de subconjuntos “*disjuntos*”.)
- Las tareas a ejecutarse en una dada etapa no deben acceder al mismo objeto.

Una partición “*admisible*” es aquella que satisface todas estas condiciones. El objetivo es *determinar aquella partición admisible que tiene el mínimo número de etapas*.

Este problema es muy común, ya que se plantea siempre que hay un número de tareas a hacer y conflictos entre esas tareas, por ejemplo sincronizar una serie de tareas con maquinaria a realizar en una industria, evitando conflictos en el uso del instrumental o maquinaria, es decir no agendar dos tareas para realizar simultáneamente si van a usar el microscopio electrónico.

1.1.2. Introducción básica a grafos

El problema se puede plantear usando una estructura matemática conocida como “grafo”. La base del grafo es un conjunto finito V de puntos llamados “vértices”. La estructura del grafo está dada por las conexiones entre los vértices. Si dos vértices están conectados se dibuja una línea que va desde un vértice al otro. Estas conexiones se llaman “aristas” (“edges”) del grafo. Los vértices pueden identificarse con un número de 0 a $n_v - 1$ donde n_v es el número total de vértices. También es usual representarlos gráficamente con un letra a, b, c, \dots encerrada en un círculo o usar cualquier etiqueta única relativa al problema.

Desde el punto de vista de la teoría de conjuntos un grafo es un subconjunto del conjunto G de pares de vértices. Un par de vértices está en el grafo si existe una arista que los conecta. También puede representarse como una matriz A simétrica de tamaño $n_v \times n_v$ con 0's y 1's. Si hay una arista entre el vértice i y el j entonces el elemento A_{ij} es uno, y sino es cero. Además, si existe una arista entre dos vértices i y j entonces decimos que i es “adyacente” a j .

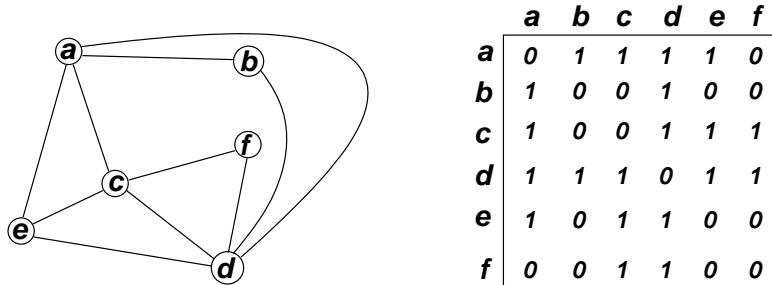


Figura 1.1: Representación gráfica y matricial del grafo G

En la figura 1.1 vemos un grafo con 6 vértices etiquetados de a a f , representado gráficamente y como una matriz de 0's y 1's. El mismo grafo representado como pares de elementos es

$$G = \{\{a, b\}, \{a, c\}, \{a, d\}, \{a, e\}, \{b, d\}, \{c, d\}, \{c, e\}, \{c, f\}, \{d, e\}, \{d, f\}, \} \quad (1.1)$$

Para este ejemplo usaremos “grafos no orientados”, es decir que si el vértice i está conectado con el j entonces el j está conectado con el i . También existen “grafos orientados” donde las aristas se representan por flechas.

Se puede también agregar un peso (un número real) a los vértices o aristas del grafo. Este peso puede representar, por ejemplo, un costo computacional.

1.1.3. Planteo del problema mediante grafos

Podemos plantear el problema dibujando un grafo donde los vértices corresponden a las tareas y dibujaremos una arista entre dos tareas si son incompatibles entre sí (modifican el mismo objeto). En este caso el grafo resulta ser como muestra la figura 1.2.

La buena noticia es que nuestro problema de particionar el grafo ha sido muy estudiado en la teoría de grafos y se llama el problema de “colorear” el grafo, es decir se representan gráficamente las etapas asignándole colores a los vértices del grafo. La mala noticia es que se ha encontrado que obtener el coloreado óptimo (es decir el coloreado admisible con la menor cantidad de colores posibles) resulta ser un problema

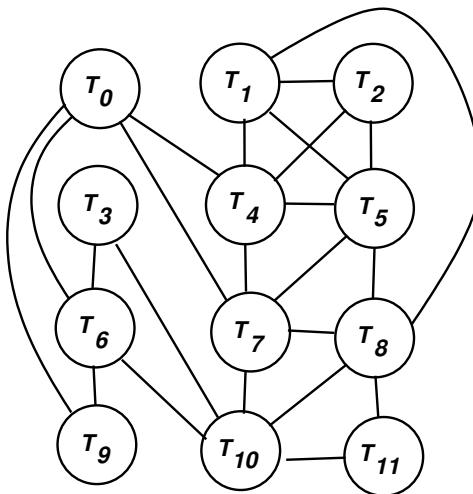


Figura 1.2: Representación del problema mediante un grafo.

extremadamente costoso en cuanto a tiempo de cálculo. (Se dice que es “NP”. Explicaremos esto en la sección §1.3.12.)

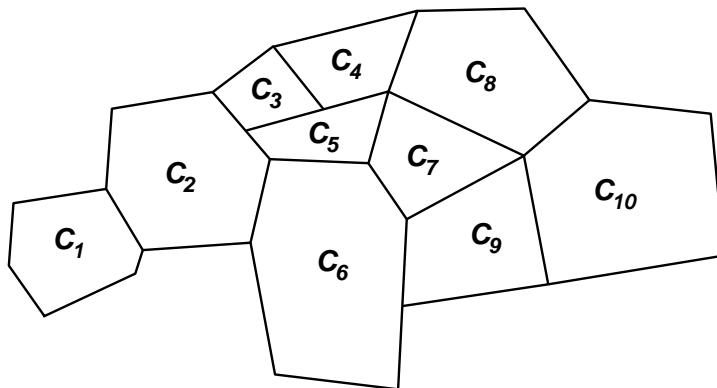


Figura 1.3: Coloración de mapas.

El término “colorear grafos” viene de un problema que también se puede poner en términos de colorear grafos y es el de colorear países en un mapa. Consideremos un mapa como el de la figura 1.3. Debemos asignar a cada país un color, de manera que países limítrofes (esto es, que comparten una porción de frontera de medida no nula) tengan diferentes colores y, por supuesto, debemos tratar de usar el mínimo número de colores posibles. El problema puede ponerse en términos de grafos, poniendo vértices en los países (C_j , $j = 1..10$) y uniendo con aristas aquellos países que son limítrofes (ver figura 1.4).

1.1.4. Algoritmo de búsqueda exhaustiva

Consideremos primero un algoritmo de “búsqueda exhaustiva” es decir, probar si el grafo se puede colorear con 1 solo color (esto sólo es posible si no hay ninguna arista en el grafo). Si esto es posible el problema

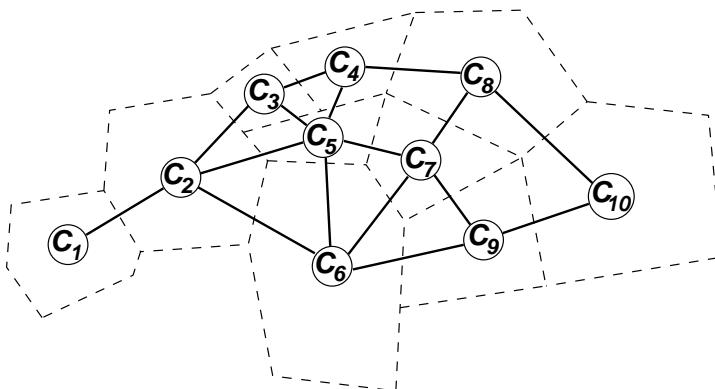


Figura 1.4: Grafo correspondiente al mapa de la figura 1.3.

está resuelto (no puede haber coloraciones con menos de un color). Si no es posible entonces generamos todas las coloraciones con 2 colores, para cada una de ellas verificamos si satisface las restricciones o no, es decir si es admisible. Si lo es, el problema está resuelto: encontramos una coloración admisible con dos colores y ya verificamos que con 1 solo color no es posible. Si no encontramos ninguna coloración admisible de 2 colores entonces probamos con las de 3 colores y así sucesivamente. Si encontramos una coloración de n_c colores entonces será óptima, ya que previamente verificamos para cada número de colores entre 1 y $n_c - 1$ que no había ninguna coloración admisible.

Ahora tratando de resolver las respuestas planteadas en la sección §1.1, vemos que el algoritmo propuesto si da la solución óptima. Por otra parte podemos ver fácilmente que sí termina en un número finito de pasos ya que a lo sumo puede haber $n_c = n_v$ colores, es decir la coloración que consiste en asignar a cada vértice un color diferente es siempre admisible.

1.1.5. Generación de las coloraciones

En realidad todavía falta resolver un punto del algoritmo y es cómo generar todas las coloraciones posibles de n_c colores. Además esta parte del algoritmo debe ser ejecutable en un número finito de pasos así que trataremos de evaluar cuantas coloraciones $N(n_c, n_v)$ hay para n_v vértices con n_c colores. Notemos primero que el procedimiento para generar las coloraciones es independiente de la estructura del grafo (es decir de las aristas), sólo depende de cuantos vértices hay en el grafo y del número de colores que pueden tener las coloraciones.

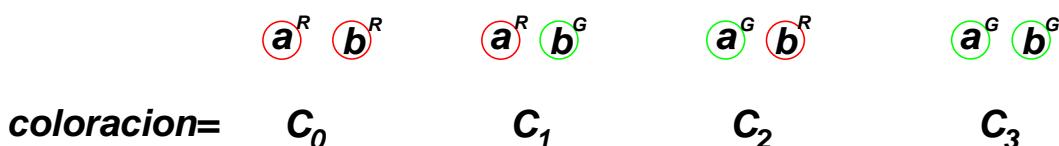


Figura 1.5: Posibles coloraciones de dos vértices con dos colores

Para $n_c = 1$ es trivial, hay una sola coloración donde todos los vértices tienen el mismo color, es decir $N(n_c = 1, n_v) = 1$ para cualquier n_v .

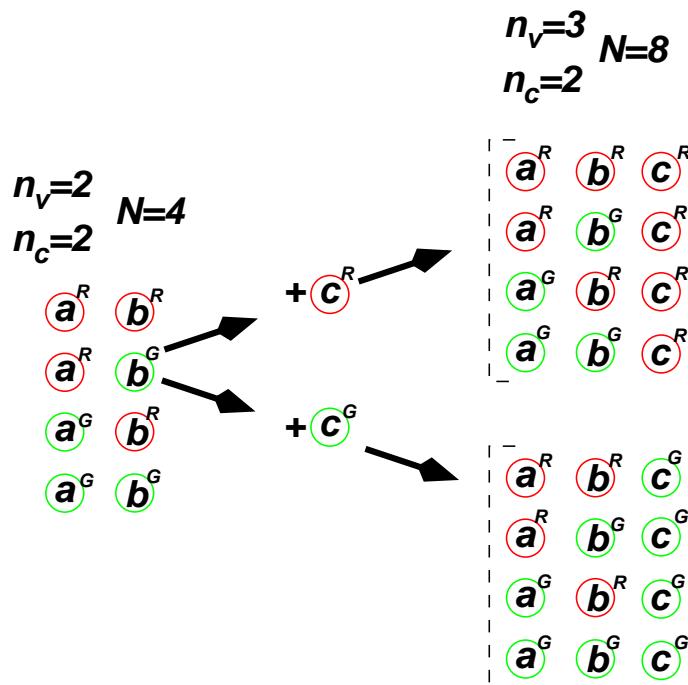


Figura 1.6: Las coloraciones de 3 vértices y dos colores se pueden obtener de las de 2 vértices.

Consideremos ahora las coloraciones de $n_c = 2$ colores, digamos rojo y verde. Si hay un sólo vértice en el grafo, entonces hay sólo dos coloraciones posibles: que el vértice sea rojo o verde. Si hay dos vértices, entonces podemos tener 4 coloraciones rojo-rojo, rojo-verde, verde-rojo y verde-verde, es decir $N(2, 2) = 4$ (ver figura 1.5). *Nota: Para que los gráficos con colores sean entendibles en impresión blanco y negro hemos agregado una pequeña letra arriba del vértice indicando el color*. Las coloraciones de 3 vértices a, b, c y dos colores las podemos generar a partir de las de 2 vértices, combinando cada una de las 4 coloraciones para los vértices a y b con un posible color para c (ver figura 1.6, de manera que tenemos

$$N(2, 3) = 2 N(2, 2) \quad (1.2)$$

Recursivamente, para cualquier $n_c, n_v \geq 1$, tenemos que

$$\begin{aligned} N(n_c, n_v) &= n_c N(n_c, n_v - 1) \\ &= n_c^2 N(n_c, n_v - 2) \\ &\vdots \\ &= n_c^{n_v-1} N(n_c, 1) \end{aligned} \quad (1.3)$$

Pero el número de coloraciones para un sólo vértice con n_c colores es n_c , de manera que

$$N(n_c, n_v) = n_c^{n_v} \quad (1.4)$$

Esto cierra con la última pregunta, ya que vemos que el número de pasos para cada uno de los colores es finito, y hay a lo sumo n_v colores de manera que el número total de posibles coloraciones a verificar es finito.

Notar de paso que esta forma de contar las coloraciones es también “*constructiva*”, da un procedimiento para generar todas las coloraciones, si uno estuviera decidido a implementar la estrategia de búsqueda exhaustiva.

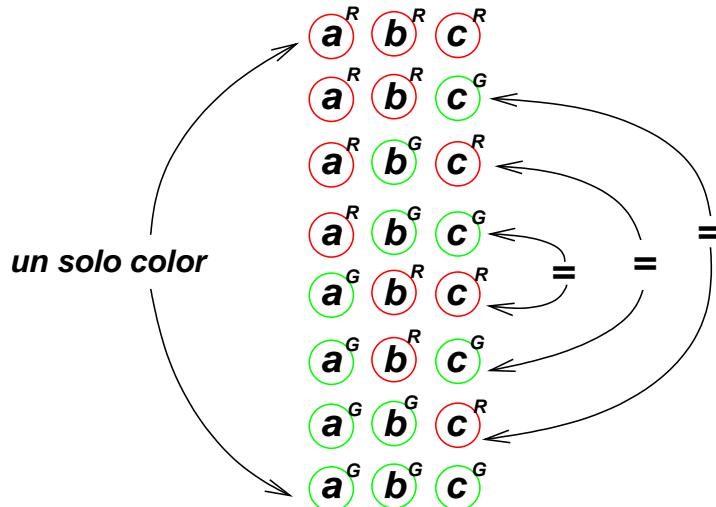


Figura 1.7: Las 8 posibles coloraciones de un grafo de 3 vértices con dos colores

Notemos que en realidad el conteo de coloraciones (1.4) incluye todas las coloraciones de n_c o menos colores. Por ejemplo si $n_v = 3$ y $n_c = 2$ entonces las $n_c^{n_v} = 2^3 = 8$ coloraciones son las que se pueden ver en la figura 1.7. En realidad hay dos (la primera y la última) que tienen un sólo color y las 6 restantes sólo hay 3 esencialmente diferentes, ya que son equivalentes entre sí. La segunda y la séptima son equivalentes entre sí de manera que una de ellas es admisible si y solo si la otra lo es. De las $3^3 = 27$ coloraciones posibles para un grafo de 3 vértices con 3 colores (o menos) en realidad 3 corresponden a un sólo color, 18 a dos colores y 6 con 3 colores (ver figura 1.8). Las 6 coloraciones de un sólo color son variantes de la única posible coloración de un color. Las 18 de dos colores son variantes de las 3 únicas coloraciones de dos colores y las 6 de 3 colores son variantes de la única coloración posible de 3 colores. O sea que de las 27 coloraciones posibles en realidad sólo debemos evaluar 5.

1.1.6. Crecimiento del tiempo de ejecución

No consideremos, por ahora, la eliminación de coloraciones redundantes (si quisieramos eliminarlas deberíamos generar un algoritmo para generar sólo las esencialmente diferentes) y consideremos que para aplicar la estrategia exhaustiva al problema de coloración de un grafo debemos evaluar $N = n_v^{n_v}$ coloraciones. Esto corresponde al peor caso de que el grafo necesite el número máximo de $n_c = n_v$ colores.

Para verificar si una coloración dada es admisible debemos realizar un cierto número de operaciones. Por ejemplo si almacenamos el grafo en forma matricial, podemos ir recorriendo las aristas del grafo y verificar que los dos vértices conectados tengan colores diferentes. Si el color es el mismo pasamos a la siguiente arista. Si recorremos todas las aristas y la coloración es admisible, entonces hemos encontrado la solución óptima al problema. En el peor de los casos el número de operaciones necesario para verificar una dada coloración es igual al número de aristas y a lo sumo el número de aristas es $n_v \cdot n_v$ (el número de elementos

en la forma matricial del grafo) de manera que para verificar todas las coloraciones necesitamos verificar

$$N_{\text{be}} = n_v^2 n_v^{n_v} = n_v^{n_v+2} \quad (1.5)$$

aristas. Asumiendo que el tiempo de verificar una arista es constante, este es el orden del número de operaciones a realizar.

El crecimiento de la función $n_v^{n_v}$ con el número de vértices es tan rápido que hasta puede generar asombro. Consideremos el tiempo que tarda una computadora personal típica en evaluar todas las posibilidades para $n_v = 20$ vértices. Tomando un procesador de 2.4 GHz (un procesador típico al momento de escribir este apunte) y asumiendo que podemos escribir un programa tan eficiente que puede evaluar una arista por cada ciclo del procesador (en la práctica esto es imposible y al menos necesitaremos unas decenas de ciclos para evaluar una coloración) el tiempo en años necesario para evaluar todas las coloraciones es de

$$T = \frac{20^{22}}{2.4 \times 10^9 \cdot 3600 \cdot 24 \cdot 365} = 5.54 \times 10^{11} \text{ años} \quad (1.6)$$

Esto es unas 40 veces la edad del universo (estimada en 15.000.000.000 de años).

Algo que debe quedar en claro es que el problema no está en la velocidad de las computadoras, sino en la estrategia de búsqueda exhaustiva. Incluso haciendo uso de las más sofisticadas técnicas de procesamiento actuales los tiempos no bajarían lo suficiente. Por ejemplo usando uno de los “clusters” de procesadores más grandes existentes actualmente (con más de mil procesadores, ver <http://www.top500.org>) sólo podríamos bajar el tiempo de cálculo al orden de los millones de años.

Otra forma de ver el problema es preguntarse cuál es el máximo número de vértices que se puede resolver en un determinado tiempo, digamos una hora de cálculo. La respuesta es que ya con $n_v = 15$ se tienen tiempos de más de 5 horas.

En la sección siguiente veremos que si bien la eliminación de las coloraciones redundantes puede reducir significativamente el número de coloraciones a evaluar, el crecimiento de la función sigue siendo similar y no permite pasar de unas cuantas decenas de vértices.

1.1.7. Búsqueda exhaustiva mejorada

Para reducir el número de coloraciones a evaluar podemos tratar de evaluar sólo las coloraciones esencialmente diferentes. No entraremos en el detalle de cómo generar las coloraciones esencialmente diferentes, pero sí las contaremos para evaluar si el número baja lo suficiente como para hacer viable esta estrategia.

Llamaremos entonces $N_d(n_c, n_v)$ al número de coloraciones esencialmente diferentes para n_v vértices y n_c colores. Observando la figura 1.7 vemos que el número total de coloraciones para 3 vértices con 2 o menos colores es

$$\begin{aligned} 2^3 &= 2 + 6 \\ \left\{ \begin{array}{l} \text{Número de} \\ \text{coloraciones con} \\ n_c = 2 \text{ o menos} \end{array} \right\} &= \left\{ \begin{array}{l} \text{Número de} \\ \text{coloraciones con} \\ \text{exactamente} \\ n_c = 1 \end{array} \right\} + \left\{ \begin{array}{l} \text{Número de} \\ \text{coloraciones con} \\ \text{exactamente} \\ n_c = 2 \end{array} \right\} \end{aligned} \quad (1.7)$$

A su vez el número de coloraciones con $n_c = 1$, que es 2, es igual al número de coloraciones esencialmente diferentes $N_d(1, 3) = 1$ que es, por las posibilidades de elegir un color de entre 2 que es 2. También el

**esencialmente
 diferentes**

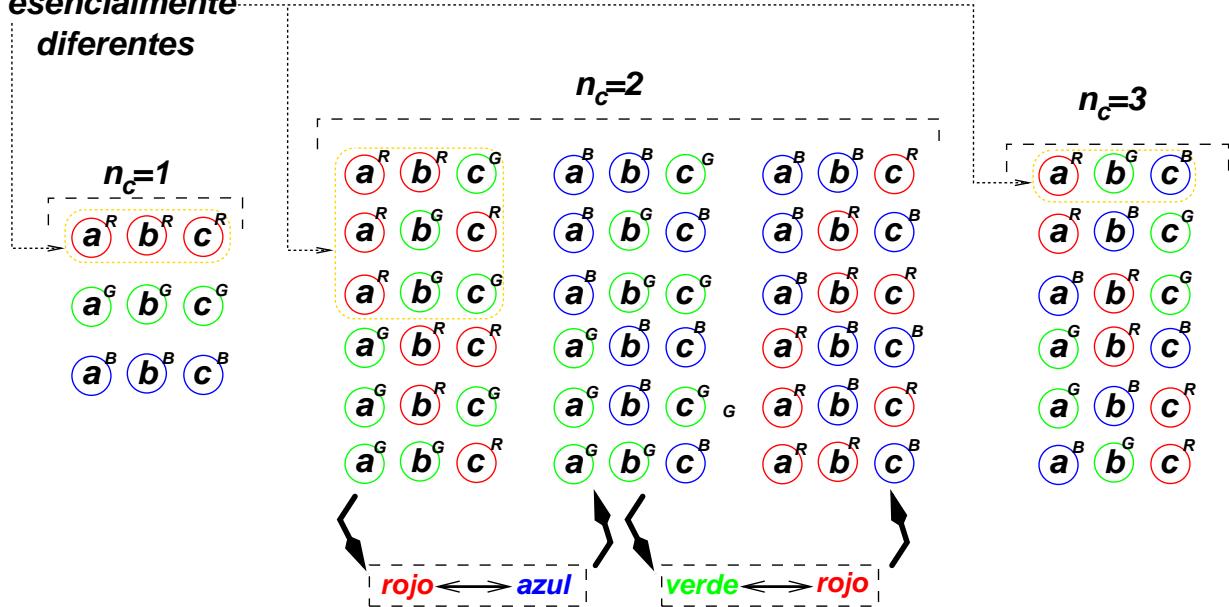


Figura 1.8: Todas las 27 coloraciones de 3 o menos colores son en realidad combinaciones de 5 esencialmente diferentes.

número de coloraciones con exactamente 2 colores es igual al número de coloraciones esencialmente diferentes $N_d(2, 3)$ que es 3, por el número de posibles maneras de elegir 2 colores de dos, que es 2 (rojo-verde, verde-rojo). En general, puede verse que la cantidad posible de elegir k colores de n_c es

$$\left\{ \begin{array}{l} \text{Número de formas} \\ \text{de elegir } k \text{ colores} \\ \text{de } n_c \end{array} \right\} = \frac{n_c!}{(n_c - k)!} \quad (1.8)$$

de manera que tenemos

$$\begin{aligned} 2^3 &= 2 \cdot 1 + 2 \cdot 3 \\ 2^3 &= \frac{2!}{1!} \cdot N_d(1, 3) + \frac{2!}{0!} \cdot N_d(2, 3) \end{aligned} \quad (1.9)$$

Supongamos que queremos calcular las coloraciones esencialmente diferentes para $n_v = 5$ colores,

entonces plantemos las relaciones

$$\begin{aligned}
 1^5 &= \frac{1!}{0!} N_d(1, 5) \\
 2^5 &= \frac{2!}{1!} N_d(1, 5) + \frac{2!}{0!} N_d(2, 5) \\
 3^5 &= \frac{3!}{2!} N_d(1, 5) + \frac{3!}{1!} N_d(2, 5) + \frac{3!}{0!} N_d(3, 5) \\
 4^5 &= \frac{4!}{3!} N_d(1, 5) + \frac{4!}{2!} N_d(2, 5) + \frac{4!}{1!} N_d(3, 5) + \frac{4!}{0!} N_d(4, 5) \\
 5^5 &= \frac{5!}{4!} N_d(1, 5) + \frac{5!}{3!} N_d(2, 5) + \frac{5!}{2!} N_d(3, 5) + \frac{5!}{1!} N_d(4, 5) + \frac{5!}{0!} N_d(5, 5)
 \end{aligned} \tag{1.10}$$

o sea

$$\begin{aligned}
 1 &= N_d(1, v) \\
 32 &= 2 N_d(1, 5) + 2 N_d(2, 5) \\
 243 &= 3 N_d(1, 5) + 6 N_d(2, 5) + 6 N_d(3, 5) \\
 1024 &= 4 N_d(1, 5) + 12 N_d(2, 5) + 24 N_d(3, 5) + 24 N_d(4, 5) \\
 3125 &= 5 N_d(1, 5) + 20 N_d(2, 5) + 60 N_d(3, 5) + 120 N_d(4, 5) + 120 N_d(5, 5)
 \end{aligned} \tag{1.11}$$

Notemos que de la segunda ecuación puede despejarse fácilmente $N_d(2, 5)$ que resulta ser 15. De la tercera se puede despejar $N_d(3, 5)$ ya que conocemos $N_d(1, 5)$ y $N_d(2, 5)$ y resulta ser $N_d(3, 5) = 25$ y así siguiendo resulta ser

$$\begin{aligned}
 N_d(1, 5) &= 1 \\
 N_d(2, 5) &= 15 \\
 N_d(3, 5) &= 25 \\
 N_d(4, 5) &= 10 \\
 N_d(5, 5) &= 1
 \end{aligned} \tag{1.12}$$

de manera que el número total de coloraciones esencialmente diferentes es

$$N_d(1, 5) + N_d(2, 5) + N_d(3, 5) + N_d(4, 5) + N_d(5, 5) = 1 + 15 + 25 + 10 + 1 = 52 \tag{1.13}$$

Es muy fácil escribir un programa (en C++, por ejemplo) para encontrar el número total de coloraciones esencialmente diferentes para un dado número de vértices, obteniéndose una tabla como la 1.1

A primera vista se observa que eliminando las coloraciones redundantes se obtiene una gran reducción en el número de coloraciones a evaluar. Tomando una serie de valores crecientes de n_v y calculando el número de coloraciones diferentes como en la tabla 1.1 se puede ver que éste crece como

$$N_d(n_v) = \sum_{n_c=1}^{n_v} N_d(n_c, n_v) \approx n_v^{n_v/2} \tag{1.14}$$

El número de aristas a verificar, contando n_v^2 aristas por coloración es de

$$N_{\text{bem}} \approx n_v^{n_v/2} n_v^2 = n_v^{n_v/2+2} \tag{1.15}$$

n_v	coloraciones	coloraciones diferentes
1	1	1
2	4	2
3	27	5
4	256	15
5	3125	52
6	46656	203
7	823543	877

Tabla 1.1: Número de coloraciones para un grafo con n_v vértices. Se indica el número de coloraciones total y también el de aquellas que son esencialmente diferentes entre sí.

Sin embargo, si bien esto significa una gran mejora con respecto a $n_v^{n_v}$, el tiempo para colorear un grafo de 20 vértices se reduce del tiempo calculado en (1.6) a sólo 99 días. Está claro que todavía resulta ser excesivo para un uso práctico.

Una implementación en C++ del algoritmo de búsqueda exhaustiva puede encontrarse en el código que se distribuye con este apunte en [aedsrc/colgraf.cpp](#). La coloración óptima del grafo se encuentra después de hacer 1.429.561 evaluaciones en 0.4 secs. Notar que el número de evaluaciones baja notablemente con respecto a la estimación $n_v^{n_v+2} \approx 9 \times 10^{12}$ ya que se encuentra una coloración admisible para $n_c = 4$ con lo cual no es necesario llegar hasta $n_c = n_v$. De todas formas incluso si tomáramos como cota inferior para evaluar los tiempos de ejecución el caso en que debieramos evaluar al menos todas las coloraciones de 2 colores, entonces tendríamos al menos un tiempo de ejecución que crece como 2^{n_v} evaluaciones. Incluso con este “piso” para el número de evaluaciones, el tiempo de cálculo sería de una hora para $n_v = 33$.

1.1.8. Algoritmo heurístico ávido

Una estrategia diferente a la de búsqueda exhaustiva es la de buscar una solución que, si bien no es la óptima (es decir, la mejor de todas), sea aceptablemente buena, y se pueda obtener en un tiempo razonable. Si se quiere, ésta es una estrategia que uno utiliza todo el tiempo: si tenemos que comprar una licuadora y nos proponemos comprar la más barata, no recorremos absolutamente todos los bazares y supermercados de todo el planeta y revisamos todos las marcas posibles, sino que, dentro del tiempo que aceptamos dedicar a esta búsqueda, verificamos el costo de los artículos dentro de algunos comercios y marcas que consideramos los más representativos.

Un algoritmo que produce una solución razonablemente buena haciendo hipótesis “razonables” se llama “heurístico”. Del diccionario: *heurístico: una regla o conjunto de reglas para incrementar la posibilidad de resolver un dado problema*.

Un posible algoritmo heurístico para colorear grafos es el siguiente algoritmo “ávido”. Primero tratamos de colorear tantos vértices como podamos con el primer color, luego con el segundo color y así siguiendo hasta colorearlos todos. La operación de colorear con un dado color puede resumirse como sigue

- Seleccionar algún vértice no coloreado y asignarle el nuevo color.
- Recorrer la lista de vértices no colorados. Para cada vértice no coloreado determinar si está conectado

(esto es, posee algún vértice en común) con un vértice del nuevo color.

Esta aproximación es llamada ávida ya que asigna colores tan rápido como lo puede hacer, sin tener en cuenta las posibles consecuencias negativas de tal acción. Si estuviéramos escribiendo un programa para jugar al ajedrez, entonces una estrategia ávida, sería evaluar todas las posibles jugadas y elegir la que da la mejor ventaja material. En realidad no se puede catalogar a los algoritmos como ávidos en forma absoluta, sino que se debe hacer en forma comparativa: hay algoritmos más ávidos que otros. En general cuanto más ávido es un algoritmo más simple es y más rápido es en cuanto a avanzar para resolver el problema, pero por otra parte explora en menor medida el espacio de búsqueda y por lo tanto puede dar una solución peor que otro menos ávida. Volviendo al ejemplo del ajedrez, un programa que, además de evaluar la ganancia material de la jugada a realizar, evalúe las posibles consecuencias de la siguiente jugada del oponente requerirá mayor tiempo pero a largo plazo producirá mejores resultados.

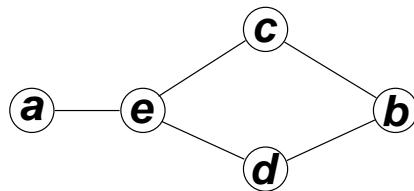


Figura 1.9: Ejemplo de un grafo simple para aplicar el algoritmo ávido.

Si consideramos la coloración de un grafo como el mostrado en la figura 1.9 entonces empezando por el color rojo le asignaríamos el rojo al vértice a . Posteriormente recorreríamos los vértices no coloreados, que a esta altura son $\{b, c, d, e\}$ y les asignamos el color rojo si esto es posible. Podemos asignárselo a b pero no a c y d , ya que están conectados a b ni tampoco a e ya que está conectado a a . Pasamos al siguiente color, digamos verde. La lista de vértices no coloreados es, a esta altura $\{c, d, e\}$. Le asignamos verde a c y luego también a d , pero no podemos asignárselo a e ya que está conectado a c y d . El proceso finaliza asignándole el siguiente color (digamos azul) al último vértice sin colorear e . El grafo coloreado con esta estrategia se muestra en la figura 1.10.



Figura 1.10: Izquierda: El grafo de la figura 1.9 coloreado con la estrategia ávida. Derecha: Coloración óptima.

El algoritmo encuentra una solución con tres colores, sin embargo se puede encontrar una solución con dos colores, como puede verse en la misma figura. Esta última es óptima ya que una mejor debería tener sólo un color, pero esto es imposible ya que entonces no podría haber ninguna arista en el grafo. Este ejemplo ilustra perfectamente que si bien el algoritmo ávido da una solución razonable, ésta puede no ser la óptima.

Notemos también que la coloración producida por el algoritmo ávido depende del orden en el que se recorren los vértices. En el caso previo, si recorriéramos los nodos en el orden $\{a, e, c, d, b\}$, obtendríamos la coloración óptima.

En el caso del grafo original mostrado en la figura 1.2, el grafo coloreado resultante con este algoritmo resulta tener 4 colores y se muestra en la figura 1.11.

Notemos que cada vez que se agrega un nuevo color, por lo menos a un nodo se le asignará ese color, de manera que el algoritmo usa a lo sumo n_v colores. Esto demuestra también que el algoritmo en su totalidad termina en un número finito de pasos.

Una implementación en C++ del algoritmo ávido puede encontrarse en el código que se distribuye con este apunte en [aedsrc/colgraf.cpp](#).

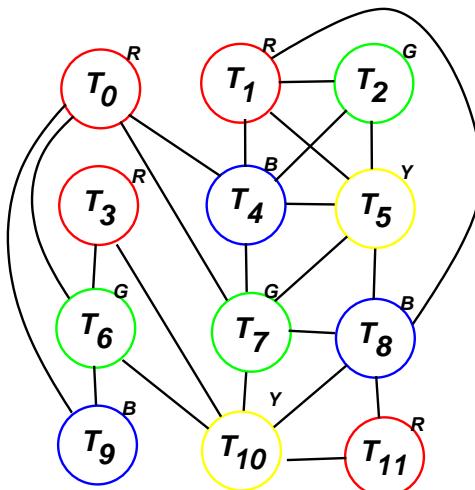


Figura 1.11: Grafo coloreado.

1.1.9. Descripción del algoritmo heurístico en seudo-código

Una vez que tenemos una versión abstracta (matemática) del modelo o algoritmo podemos empezar a implementarlo para llegar a un programa real que resuelve el problema. Este proceso puede llevarse a cabo en varias etapas empezando por una descripción muy general en forma de sentencias vagas, llamado “*seudo-código*”, como “*elegir un vértice no coloreado*”. A veces es común incluir este seudo-código en forma de comentarios seguidos por puntos suspensivos que indican que falta completar esa parte del programa.

Lo ideal es que estas sentencias sean suficientemente claras como para no dejar dudas de cual es la tarea a realizar, pero también lo suficientemente generales como para no tener que entrar en detalles y poder diseñar rápidamente una versión básica del código. Luego en un paso de “*refinamiento*” posterior estas sentencias en seudo-código son refinadas en tareas más pequeñas, las cuales pueden ser descriptas parte en líneas de seudo-código ellas mismas y parte en sentencias válidas del lenguaje, hasta que finalmente terminamos con un código que puede ser compilado y linkeditado en un programa.

Tomemos como ejemplo el algoritmo heurístico descrito previamente en §1.1.8. La rutina **greedyC** mostrada en el código 1.1 (archivo **greedy.cpp**) toma como argumentos un grafo **G**, el conjunto de vértices no coloreados hasta el momento **no_col** y determina un conjunto de nodos **nuevo_color** los cuales pueden ser coloreados con el nuevo color. Los vértices son identificados por un entero de 0 a **nv-1**. La rutina además mantiene una tabla **tabla_color** donde finalmente quedarán los colores de cada vértice. El argumento de

entrada **color** indica el color (un entero empezando desde 0 e incrementándose de a 1 en cada llamada a **greedyc**) con el cual se está coloreando en esta llamada a **greedyc**. Notar que esta rutina será llamada posteriormente dentro de un lazo sobre los colores hasta colorear todos los vértices. Los conjuntos son representados con el template **set<...>** de la librerías STL (por “Standard Template Library”, hoy parte del C++ estándar). Falta todavía implementar la clase **graph**. El lazo de las líneas 9-17 es un lazo típico para recorrer un **set<>**.

```
1 void greedyc(graph &G, set<int> &no_col,
2                 set<int> &nuevo_color,
3                 vector<int> &tabla_color, int color) {
4     // Asigna a 'nuevo_color' un conjunto de vértices
5     // de 'G' a los cuales puede darse el mismo nuevo color
6     // sin entrar en conflicto con los ya coloreados
7     nuevo_color.clear();
8     set<int>::iterator q;
9     for (q=no_col.begin(); q!=no_col.end(); q++) {
10         if /* *q no es adyacente a
11             ningún vértice en 'nuevo_color' ... */ {
12             // marcar a '*q' como coloreado
13             // ...
14             // agregar '*q' a 'nuevo_color'
15             // ...
16         }
17     }
18 }
```

Código 1.1: Rutina para la coloración de un grafo. Determina el conjunto de vértices que pueden ser coloreados con un nuevo color sin entrar en conflicto. Versión inicial. [Archivo: *greedy.cpp*]

Haciendo un breve repaso de los *iterators* de STL, el iterator **q**, declarado en la línea 8 actúa como un puntero a **int**, de hecho ***q** es una referencia a un **int**. Al empezar el lazo apunta al primer elemento del **no_col** y en cada iteración del lazo pasa a otro elemento, hasta que cuando se han acabado todos toma el valor **no_col.end()**, con lo cual finaliza al lazo. Dentro del lazo faltan implementar 3 porciones de código. La condición del **if** de las líneas 10-11, el código para marcar al vértice como coloreado en la línea 13 y para agregarlo a **nuevo_color** en la línea 15.

Vamos ahora a refinar el algoritmo anterior, expandiendo ahora más aún la expresión condicional del **if**. Para verificar si ***q** es adyacente a algún vértice de **nuevo_color** debemos recorrer todos los nodos de **nuevo_color** y verificar si hay alguna arista entre los mismos y ***q**. Para esto hacemos un lazo, definiendo una variable **adyacente** (ver código 1.2, archivo *greedy2.cpp*). Al llegar al comienzo del condicional en la línea 10 la variable **adyacente** tiene el valor apropiado. Notar que si se detecta que uno de los vértices de **nuevo_color** es adyacente a ***q**, entonces no es necesario seguir con el lazo, por eso el **break** de la línea 15. Además hemos implementado las líneas 13 y líneas 15 del código 1.1, resultando en las líneas líneas 20 y líneas 22 del código 1.2. La línea 20 simplemente registra el nuevo color asignado a la tabla **tabla_color** y la línea 22 inserta el vértice que se termina de colorear ***q** al conjunto **nuevo_color**. Notar que deberíamos eliminar los vértices que coloreamos de **no_col** pero esto no lo podemos hacer dentro del lazo de las líneas 9-24 del código 1.2, ya que dentro del mismo se itera sobre **no_col**. Modificar **no_col**

convertiría en inválido el iterador `q` y por lo tanto después no se podría aplicar el operador `++` a `q` en la línea 9.

```

1 void greedyc(graph &G, set<int> &no_col,
2             set<int> &nuevo_color,
3             vector<int> &tabla_color, int color) {
4     // Asigna a 'nuevo_color' un conjunto de vértices
5     // de 'G' a los cuales puede darse el mismo nuevo color
6     // sin entrar en conflicto con los ya coloreados
7     nuevo_color.clear();
8     set<int>::iterator q,w;
9     for (q=no_col.begin(); q!=no_col.end(); q++) {
10         int adyacente=0;
11         for (w=nuevo_color.begin();
12              w!=nuevo_color.end(); w++) {
13             if /* *w' es adyacente a '*q' . . . */) {
14                 adyacente = 1;
15                 break;
16             }
17         }
18         if (!adyacente) {
19             // marcar a '*q' como coloreado
20             tabla_color[*q] = color;
21             // agregar '*q' a 'nuevo_color'
22             nuevo_color.insert(*q);
23         }
24     }
25 }
```

Código 1.2: Rutina para la coloración de un grafo. Versión refinada. [Archivo: greedy2.cpp]

Para refinar el condicional de la línea 13 necesitamos definir la clase `grafo`, para ello utilizaremos una representación muy simple, útil para grafos pequeños basada en mantener una tabla de unos y ceros como fue descripto en §1.1.2. Como el grafo no es orientado, la matriz es simétrica ($A_{jk} = A_{kj}$) de manera que sólo usaremos la parte triangular inferior de la misma. La matriz será almacenada por filas en un arreglo `vector<int>` `g` de manera que el elemento A_{jk} estará en la posición `g[nv * j + k]`. La clase `grafo` se puede observar en el código 1.3. Los elementos de la matriz se acceden a través de una función miembro `edge(j,k)` que retorna una referencia al elemento correspondiente de la matriz. Notar que si $j < k$, entonces se retorna en realidad el elemento simétrico A_{kj} en la parte triangular inferior. Como `edge()` retorna una referencia al elemento correspondiente, puede usarse tanto para insertar aristas en el grafo

`G.edge(j,k) = 1;`

como para consultar si un dado par de vértices es adyacente o no

```

if (!G.edge(j,k)) {
    // no están conectados
    // ...
}
```

```
1 class graph {
2 private:
3     const int nv;
4     vector<int> g;
5 public:
6     // Constructor a partir del numero de vertices
7     graph(int nv_a) : nv(nv_a) { g.resize(nv*nv, 0); }
8     // Este metodo permite acceder a una arista tanto para
9     // agregar la arista ('g.edge(i,j)=1') como para
10    // consultar un valor particular de la
11    // arista. ('adyacente = g.edge(i,j)')
12    int &edge(int j, int k) {
13        if (k<=j) return g[nv*j+k];
14        else return g[nv*k+j];
15    }
16};
```

Código 1.3: Clase básica de grafo. [Archivo: graph.cpp]

La versión final de la rutina **greedyc** puede observarse en el código 1.4.

```
1 void greedyc(graph &G, set<int> &no_col,
2             set<int> &nuevo_color,
3             vector<int> &tabla_color, int color) {
4     // Asigna a 'nuevo_color' un conjunto de vertices
5     // de 'G' a los cuales puede darse el mismo nuevo color
6     // sin entrar en conflicto con los ya coloreados
7     nuevo_color.clear();
8     set<int>::iterator q,w;
9     for (q=no_col.begin(); q!=no_col.end(); q++) {
10         int adyacente=0;
11         for (w=nuevo_color.begin();
12             w!=nuevo_color.end(); w++) {
13             if (G.edge(*q,*w)) {
14                 adyacente = 1;
15                 break;
16             }
17         }
18         if (!adyacente) {
19             // marcar a '*q' como coloreado
20             tabla_color[*q] = color;
21             // agregar '*q' a 'nuevo_color'
22             nuevo_color.insert(*q);
23         }
24     }
25 }
```

Código 1.4: Versión final de la rutina [Archivo: greedy3.cpp]

Ahora falta definir el código exterior que iterará los colores, llamando a **greedyC**. Un primer esbozo puede observarse en el código 1.5. La rutina **greedy** toma como argumentos de entrada el grafo a colorear **G**, el número de vértices **nv** y devuelve la coloración en **tabla_color**. Internamente inicializa el conjunto de vértices no coloreados insertando todos los vértices del grafo en la línea 8. A continuación entra en un lazo infinito, del cual sólo saldrá cuando todos los vértices estén coloreados, y por lo tanto **no_col** sea vacío, lo cual todavía debemos implementar en la línea 19. (Notemos que es válido utilizar un lazo infinito ya que hemos garantizado que el algoritmo se ejecuta a lo sumo un número finito de veces, más precisamente a lo sumo n_v veces.) Dentro del lazo, se determina el conjunto de vértices al cual se asignará el nuevo color llamando a **greedyC(...)** (línea 13). Luego debemos sacar los vértices asignados al nuevo color de **no_col** y, después de verificar la condición de fin del algoritmo, incrementar el número de color.

```

1 void greedy(graph &G, int nv,
2             vector<int> &tabla_color) {
3     int color=0;
4     set<int> nuevo_color, no_col;
5     set<int>::iterator q;
6     // Inicialmente ponemos todos los vertices en
7     // 'no_col'
8     for (int k=0; k<nv; k++) no_col.insert(k);
9     while (1) {
10         // Determina a cuales vertices podemos asignar el
11         // nuevo color
12         greedyC(G,no_col,nuevo_color,
13                   tabla_color,color);
14         // Saca los vertices que se acaban de colorear
15         // ('nuevo_color') de 'no_col'
16         // ...
17         // Detecta el fin del algoritmo cuando ya no hay
18         // mas vertices para colorear.
19         // ...
20         color++;
21     }
22 }
```

Código 1.5: Algoritmo de coloración. Se van agregando nuevos colores llamando a **greedyC** [Archivo: **greedy4.cpp**]

En el código 1.6 vemos la versión refinada definitiva. La eliminación de los elementos de **nuevo_color** de **no_col** se realiza recorriéndolos y usando la función **erase** de **set<>**. La detección de si **no_col** está vacío o no se realiza usando la función **size()**. Esta retorna el número de elementos en el conjunto, de manera que si retorna cero, entonces el conjunto está vacío.

```

1 void greedy(graph &G, int nv,
2             vector<int> &tabla_color) {
3     int color=0;
4     set<int> nuevo_color, no_col;
5     set<int>::iterator q;
6     // Inicialmente ponemos todos los vertices en 'no_col'
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```

7   for (int k=0; k<nv; k++) no_col.insert(k);
8   while (1) {
9     // Determina a cuales vértices podemos asignar
10    // el nuevo color
11    greedyc(G,no_col,nuevo_color,tabla_color,color);
12    // Saca los vértices que se acaban de colorear
13    // ('nuevo_color') de 'no_col'
14    for (q=nuevo_color.begin();
15        q!=nuevo_color.end(); q++)
16      no_col.erase(*q);
17    // Detecta el fin del algoritmo cuando ya no hay
18    // mas vértices para colorear.
19    if (!no_col.size()) return;
20    color++;
21  }
22 }
```

Código 1.6: Algoritmo de coloración. Versión final. [Archivo: *greedy5.cpp*]

El código **greedyf.cpp** contiene una versión completa del código. En el programa principal se definen dos grafos pequeños (correspondientes a las figuras 1.9 y 1.2) para probar el algoritmo y también incluye la posibilidad de generar grafos aleatorios con un número de aristas prefijado. Como para darse una idea de las posibilidades prácticas de este algoritmo, es capaz de colorear un grafo aleatorio de 5000 vértices y 6.25×10^6 aristas (la mitad del total posible $n_v(n_v - 1)/2$ en 7 segs, en un procesador de características similares a las mencionadas en §1.1.6.

1.1.10. Crecimiento del tiempo de ejecución para el algoritmo ávido

Si bien el cálculo de tiempos de ejecución será desarrollado más adelante, en la sección §1.3, podemos rápidamente tener una idea de como se comporta en función del número de vértices. Consideremos el número de operaciones que realiza la rutina **greedyc** (ver código 1.4). El lazo de las líneas 9-17 se ejecuta a lo sumo n_v veces, ya que **no_col** puede tener a lo sumo n_v elementos. Dentro del lazo hay una serie de llamados a funciones (los métodos **begin()**, **end()** e **insert()** de la clase **set**) y el operador incremento de la clase **set::iterator**). Por ahora no sabemos como están implementadas estas operaciones, pero de la documentación de las STL se puede deducir que estas operaciones se hacen en tiempo constante, es decir que no crecen con n_v (En realidad no es así. Crecen como $\log n_v$, pero por simplicidad asumiremos tiempo constante, y las conclusiones serían básicamente las mismas si incluyéramos este crecimiento). Por lo tanto, fuera del lazo de las líneas 12-17, todas las otras instrucciones consumen un número de operaciones constante. El lazo de las líneas 12-17 se ejecuta a lo sumo n_v veces y dentro de él sólo se realizan un número constante de operaciones (la llamada a **G.edge(...)** en nuestra implementación es simplemente un acceso a un miembro de **vector** el cual, también de acuerdo a la documentación de STL se hace en tiempo constante). De manera que el tiempo de ejecución de **greedyc** es a lo sumo n_v^2 operaciones. Por otra parte, en el código 1.6 tenemos el lazo de las líneas 8-21 el cual se ejecuta un número máximo de n_v veces. En cada ejecución del lazo, la llamada a **greedyc** consume a lo sumo n_v^2 operaciones. En el resto del bloque tenemos todas líneas que consumen un número constante de operaciones, menos el lazo de las líneas 15-16, el cual se ejecuta a lo sumo n_v veces y consume en cada iteración a lo sumo un número constante de operaciones.

De manera que todo el bloque de las líneas 8-21 consume a lo sumo n_v^3 operaciones. Lo cual significa una dramática reducción con respecto a las estimaciones para los algoritmos de búsqueda exhaustiva (1.5) y búsqueda exhaustiva mejorada (1.15).

1.1.11. Conclusión del ejemplo

En toda esta sección §1.1.1 hemos visto un ejemplo en el cual resolvemos un problema planteando un modelo matemático abstracto (en este caso las estructuras *grafo* y *conjunto*). Inicialmente el algoritmo es expresado informalmente en términos de operaciones abstractas sobre estas estructuras. Posteriormente se genera un primer esbozo del algoritmo con una mezcla de sentencias escritas en C++ (u otro lenguaje) y seudo-código, el cual es refinado en una serie de etapas hasta llegar a un programa que se puede compilar, linkeditar y ejecutar.

1.2. Tipos abstractos de datos

Una vez que se ha elegido el algoritmo, la implementación puede hacerse usando las estructuras más simples, comunes en casi todos los lenguajes de programación: escalares, arreglos y matrices. Sin embargo algunos problemas se pueden plantear en forma más simple o eficiente en términos de estructuras informáticas más complejas, como listas, pilas, colas, árboles, grafos, conjuntos. Por ejemplo, el TSP se plantea naturalmente en términos de un grafo donde los vértices son las ciudades y las aristas los caminos que van de una ciudad a otra. Estas estructuras están incorporadas en muchos lenguajes de programación o bien pueden obtenerse de librerías. El uso de estas estructuras tiene una serie de ventajas

- Se ahorra tiempo de programación ya que no es necesario codificar.
- Estas implementaciones suelen ser eficientes y robustas.
- Se separan dos capas de código bien diferentes, por una parte el algoritmo que escribe el programador, y por otro las rutinas de acceso a las diferentes estructuras.
- Existen estimaciones bastante uniformes de los tiempos de ejecución de las diferentes operaciones.
- Las funciones asociadas a cada estructura son relativamente independientes del lenguaje o la implementación en particular. Así, una vez que se plantea un algoritmo en términos de operaciones sobre una tal estructura es fácil implementarlo en una variedad de lenguajes con una performance similar.

Un “*Tipo Abstracto de Datos*” (TAD) es la descripción matemática de un objeto abstracto, definido por las operaciones que actúan sobre el mismo. Cuando usamos una estructura compleja como un conjunto, lista o pila podemos separar tres niveles de abstracción diferente, ejemplificados en la figura 1.12, a saber las “*operaciones abstractas*” sobre el TAD, la “*interfaz*” concreta de una implementación y finalmente la “*implementación*” de esa interfaz.

Tomemos por ejemplo el TAD CONJUNTO utilizado en el ejemplo de la sección §1.1.1 . Las siguientes son las operaciones abstractas que podemos querer realizar sobre un conjunto

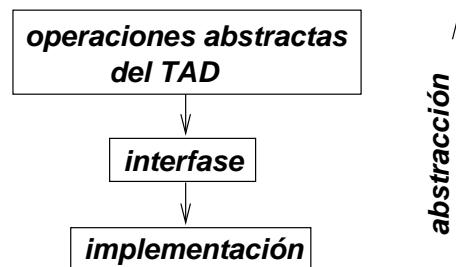


Figura 1.12: Descripción de lo diferentes niveles de abstracción en la definición de un TAD

1.2.1. Operaciones abstractas y características del TAD CONJUNTO

- Contiene elementos, los cuales deben ser diferentes entre sí.
- No existe un orden particular entre los elementos del conjunto.
- Se pueden insertar o eliminar elementos del mismo.
- Dado un elemento se puede preguntar si está dentro del conjunto o no.
- Se pueden hacer las operaciones binarias bien conocidas entre conjuntos a saber, unión, intersección y diferencia.

1.2.2. Interfaz del TAD CONJUNTO

La “*interfaz*” es el conjunto de operaciones (con una sintaxis definida) que producen las operaciones del TAD. Por supuesto depende del lenguaje a utilizar, si bien algunas veces es también común que una librería pueda ser usada desde diferentes lenguajes y trate de mantener la interfaz entre esos diferentes lenguajes.

Por ejemplo la implementación del TAD CONJUNTO en la librería STL es (en forma muy simplificada) la siguiente,

```
1 template<class T>
2 class set {
3 public:
4     class iterator { /* ... */ };
5     void insert(T x);
6     void erase(iterator p);
7     void erase(T x);
8     iterator find(T x);
9     iterator begin();
10    iterator end();
11};
```

Código 1.7: *Interfaz de la clase set*<> [Archivo: stl-set.cpp]

Esta interfaz tiene la desventaja de que no provee directamente las operaciones mencionadas previamente pero encaja perfectamente dentro de la interfaz general de los otros “*contenedores*” de STL. Recordemos que en STL los elementos de los contenedores, como en este caso **set**, se acceden a través de “*iteradores*” (“*iterators*”). En la siguiente descripción **s** es un conjunto, **x** un elemento y **p** un iterador

- **s.insert(x)** inserta un elemento en el conjunto. Si el elemento ya estaba en el conjunto **s** queda inalterado.
- **p=s.find(x)** devuelve el iterador para el elemento **x**. Si **x** no está en **s** entonces devuelve un iterador especial **end()**. En consecuencia, la expresión lógica para saber si un elemento está en **s** es

```
if(s.find(x)==s.end()) {
    // `x` no está en `s`
    // ...
}
```

- **s.erase(p)** elimina el elemento que está en el iterador **p** en **s**. **s.erase(x)** elimina el elemento **x** (si está en el conjunto).
- La unión de dos conjuntos, por ejemplo $C = A \cup B$ podemos lograrla insertando los elementos de A y B en C :

```
set A,B,C;
// Pone elementos en A y B
// ...
C.insert(A.begin(),A.end());
C.insert(B.begin(),B.end());
```

Normalmente en C/C++ la interfaz está definida en los headers de las respectivas clases.

- Todas las operaciones binarias con conjuntos se pueden realizar con algoritmos genéricos definidos en el header **algorithm**, usando el adaptador **inserter**:

```
template<class Container, class Iter>
insert_iterator<Container>
inserter(Container& C, Iter i);
```

Típicamente:

- $C = A \cup B$

```
set_union(a.begin(),a.end(),b.begin(),b.end(),
          inserter(c,c.begin()));
```
- $C = A - B$

```
set_difference(a.begin(), a.end(), b.begin(), b.end(),
               inserter(c, c.begin()));

•  $C = A \cap B$ 

set_intersection(a.begin(), a.end(), b.begin(), b.end(),
                  inserter(c, c.begin()));
```

1.2.3. Implementación del TAD CONJUNTO

Finalmente la “*implementación*” de estas funciones, es decir el código específico que implementa cada una de las funciones declaradas en la interfaz.

Como regla general podemos decir que un programador que quiere usar una interfaz abstracta como el TAD CONJUNTO, debería tratar de elaborar primero un algoritmo abstracto basándose en las operaciones abstractas sobre el mismo. Luego, al momento de escribir su código debe usar la interfaz específica para traducir su algoritmo abstracto en un código compilable. En el caso del TAD CONJUNTO veremos más adelante que internamente éste puede estar implementado de varias formas, a saber con listas o árboles, por ejemplo. En general, el código que escribe no debería depender *nunca* de los detalles de la implementación particular que está usando.

1.3. Tiempo de ejecución de un programa

La eficiencia de un código va en forma inversa con la cantidad de recursos que consume, principalmente tiempo de CPU y memoria. A veces en programación la eficiencia se contrapone con la sencillez y legibilidad de un código. Sin embargo en ciertas aplicaciones la eficiencia es un factor importante que no podemos dejar de tener en cuenta. Por ejemplo, si escribimos un programa para buscar un nombre en una agenda personal de 200 registros, entonces probablemente la eficiencia no es la mayor preocupación. Pero si escribimos un algoritmo para un motor de búsqueda en un número de entradas $> 10^9$, como es común en las aplicaciones para buscadores en Internet hoy en día, entonces la eficiencia probablemente pase a ser un concepto fundamental. Para tal volumen de datos, pasar de un algoritmo $O(n \log n)$ a uno $O(n^{1.3})$ puede ser fatal.

Más importante que saber escribir programas eficientemente es saber *cuándo* y *dónde* preocuparse por la eficiencia. Antes que nada, un programa está compuesto en general por varios componentes o módulos. No tiene sentido preocuparse por la eficiencia de un dado módulo si este representa un 5 % del tiempo total de cálculo. En un tal módulo tal vez sea mejor preocuparse por la robustez y sencillez de programación que por la eficiencia.

El tiempo de ejecución de un programa (para fijar ideas, pensemos por ejemplo en un programa que ordena de menor a mayor una serie de números enteros) depende de una variedad de factores, entre los cuales

- *La eficiencia del compilador y las opciones de optimización que pasamos al mismo en tiempo de compilación.*
- *El tipo de instrucciones y la velocidad del procesador donde se ejecutan las instrucciones compiladas.*
- *Los datos del programa.* En el ejemplo, la cantidad de números y su distribución estadística: ¿son todos iguales?, ¿están ya ordenados o casi ordenados?

- La “complejidad algorítmica” del algoritmo subyacente. En el ejemplo de ordenamiento, el lector ya sabrá que hay algoritmos para los cuales el número de instrucciones crece como n^2 , donde n es la longitud de la lista a ordenar, mientras que algoritmos como el de “ordenamiento rápido” (“quicksort”) crece como $n \log n$. En el problema de coloración estudiado en §1.1.1 el algoritmo de búsqueda exhaustiva crece como $n_v^{n_v}$, donde n_v es el número de vértices del grafo contra n_v^3 para el algoritmo ávido descripto en §1.1.8.

En este libro nos concentraremos en los dos últimos puntos de esta lista.

```

1 int search(int l,int *a,int n) {
2     int j;
3     for (j=0; j<n; j++)
4         if (a[j]==l) break;
5     return j;
6 }
```

Código 1.8: Rutina simple para buscar un elemento l en un arreglo $a[]$ de longitud n [Archivo: search.cpp]

En muchos casos, el tiempo de ejecución depende no tanto del conjunto de datos específicos, sino de alguna “medida” del tamaño de los datos. Por ejemplo sumar un arreglo de n números no depende de los números en sí mismos sino de la longitud n del arreglo. Denotando por $T(n)$ el tiempo de ejecución

$$T(n) = cn \quad (1.16)$$

donde c es una constante que representa el tiempo necesario para sumar un elemento. En otros muchos casos, si bien el tiempo de ejecución sí depende de los datos específicos, *en promedio* sólo depende del tamaño de los datos. Por ejemplo, si buscamos la ubicación de un elemento l en un arreglo a , simplemente recorriendo el arreglo desde el comienzo hasta el fin (ver código 1.8) hasta encontrar el elemento, entonces el tiempo de ejecución dependerá fuertemente de la ubicación del elemento dentro del arreglo. El tiempo de ejecución es proporcional a la posición j del elemento dentro del vector tal que $a_j = l$, tomando $j = n$ si el elemento no está. El mejor caso es cuando el elemento está al principio del arreglo, mientras que el peor es cuando está al final o cuando no está. Pero “*en promedio*” (asumiendo que la distribución de elementos es aleatoria) el elemento buscado estará en la zona media del arreglo, de manera que una ecuación como la (1.16) será válida (en promedio). Cuando sea necesario llamaremos $T_{\text{prom}}(n)$ al promedio de los tiempos de ejecución de un dado algoritmo sobre un “ensamble” de posibles entradas y por $T_{\text{peor}}(n)$ el peor de todos sobre el ensamble. Entonces, para el caso de buscar la numeración de un arreglo tenemos

$$\begin{aligned} T(n) &= cj \\ T_{\text{peor}}(n) &= cn \\ T_{\text{prom}}(n) &= c \frac{n}{2} \end{aligned} \quad (1.17)$$

En estas expresiones c puede tomarse como el tiempo necesario para ejecutar una vez el cuerpo del lazo en la rutina **search(...)**. Notar que esta constante c , si la medimos en segundos, puede depender fuertemente de los ítems considerados en los dos primeros puntos de la lista anterior. Por eso, preferimos dejar la

constante sin especificar en forma absoluta, es decir que de alguna forma estamos evaluando el tiempo de ejecución en términos de “*unidades de trabajo*”, donde una unidad de trabajo c es el tiempo necesario para ejecutar una vez el lazo.

En general, determinar analíticamente el tiempo de ejecución de un algoritmo puede ser una tarea intelectual ardua. Muchas veces, encontrar el $T_{\text{peor}}(n)$ es una tarea relativamente más fácil. Determinar el $T_{\text{prom}}(n)$ puede a veces ser más fácil y otras veces más difícil.

1.3.1. Notación asintótica

Para poder obtener una rápida comparación entre diferentes algoritmos usaremos la “*notación asintótica*” $O(\dots)$. Por ejemplo, decimos que el tiempo de ejecución de un programa es $T(n) = O(n^2)$ (se lee “ $T(n)$ es orden n^2 ”) si existen constantes $c, n_0 > 0$ tales que para

$$T(n) \leq cn^2, \quad \text{para } n \geq n_0 \quad (1.18)$$

La idea es que no nos interesa como se comporta la función $T(n)$ para valores de n pequeños sino sólo la tendencia para $n \rightarrow \infty$.

Ejemplo 1.1: Sea $T(n) = (n + 1)^2$, entonces si graficamos $T(n)$ en función de n (ver figura 1.13) junto con la función $2n^2$ vemos que la relación $T(n) \leq 2n^2$ es cierta para valores de $3 \leq n \leq 10$. Para ver que esta relación es válida para *todos* los valores de n tales que $n \geq 3$, entonces debemos recurrir a un poco de álgebra. Tenemos que, como

$$n \geq 3, \quad (1.19)$$

entonces

$$\begin{aligned} n - 1 &\geq 2, \\ (n - 1)^2 &\geq 4, \\ n^2 - 2n + 1 &\geq 4, \\ n^2 &\geq 3 + 2n, \\ 3 + 2n + n^2 &\leq 2n^2. \end{aligned} \quad (1.20)$$

Pero

$$3 + 2n + n^2 = (n + 1)^2 + 2, \quad (1.21)$$

y por lo tanto

$$(n + 1)^2 \leq (n + 1)^2 + 2 \leq 2n^2, \quad (1.22)$$

que es la relación buscada. Por lo tanto $T(n) = O(n^2)$ con $c = 2$ y $n_0 = 3$.

La notación $O(\dots)$ puede usarse con otras funciones, es decir $O(n^3), O(2^n), O(\log n)$. En general decimos que $T(n) = O(f(n))$ (se lee “ $T(n)$ es orden $f(n)$ ”) si existen constantes $c, n_0 > 0$ tales que

$$T(n) \leq cf(n), \quad \text{para } n \geq n_0 \quad (1.23)$$

Se suele llamar a $f(n)$ la “*tasa de crecimiento*” de $T(n)$ (también “*velocidad de crecimiento*” o “*complejidad algorítmica*”). De la definición de $O(\dots)$ pueden hacerse las siguientes observaciones.

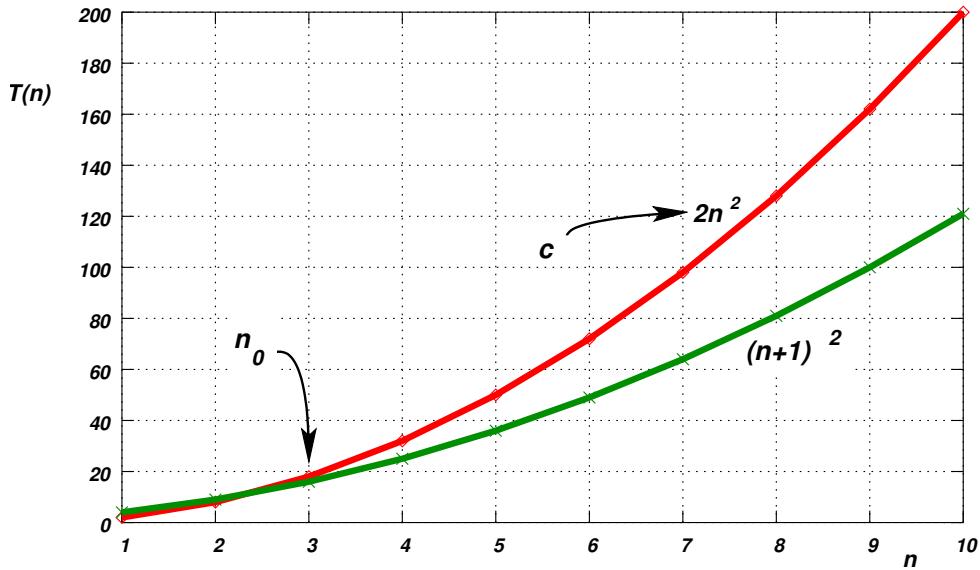


Figura 1.13: $T(n) = (n + 1)^2$ es $O(n^2)$

1.3.2. Invariancia ante constantes multiplicativas

Podemos ver que la definición de la tasa de crecimiento es invariante ante constantes multiplicativas, es decir

$$T(n) = O(cf(n)) \implies T(n) = O(f(n)) \quad (1.24)$$

Por ejemplo, si $T(n) = O(2n^3)$ entonces también es $O(n^3)$.

Demostración: Esto puede verse fácilmente ya que si $T(n) = O(2n^3)$, entonces existen c y n_0 tales que $T(n) \leq c2n^3$ para $n \geq n_0$, pero entonces también podemos decir que $T(n) \leq c'n^3$ para $n \geq n_0$, con $c' = 2c$.

1.3.3. Invariancia de la tasa de crecimiento ante valores en un conjunto finito de puntos

Es decir si

$$T_1(n) = \begin{cases} 100 & ; \text{ para } n < 10, \\ (n+1)^2 & ; \text{ para } n \geq 10, \end{cases} \quad (1.25)$$

entonces vemos que $T_1(n)$ coincide con la función $T(n) = (n+1)^2$ estudiada en el ejemplo 1.1. Por lo tanto, como sólo difieren en un número finito de puntos (los valores de $n < 10$) las dos son equivalentes y por lo tanto $T_1(n) = O(n^2)$ también.

Demostración: Esto puede verse ya que si $T(n) < 2n^2$ para $n \geq 3$ (como se vio en el ejemplo citado), entonces $T_1(n) < 2n^2$ para $n > n'_0 = 10$.

1.3.4. Transitividad

La propiedad $O(\cdot)$ es transitiva, es decir si $T(n) = O(f(n))$ y $f(n) = O(g(n))$ entonces $T(n) = O(g(n))$.

Demostración: si $T(n) \leq cf(n)$ para $n \geq n_0$ y $f(n) \leq c'g(n)$ para $n \geq n'_0$, entonces $T(n) \leq c''g(n)$ para $n \geq n''_0$, donde $c'' = cc'$ y $n''_0 = \max(n_0, n'_0)$. En cierta forma, $O(\dots)$ representa una relación de orden entre las funciones (como “ $<$ ” entre los números reales).

1.3.5. Regla de la suma

Si $f(n) = O(g(n))$, y a, b son constantes positivas, entonces $a f(n) + b g(n) = O(g(n))$. Es decir, si en una expresión tenemos una serie de términos, sólo queda el “mayor” de ellos (en el sentido de $O(\dots)$). Así por ejemplo, si $T(n) = 2n^3 + 3n^5$, entonces puede verse fácilmente que $n^3 = O(n^5)$ por lo tanto, $T(n) = O(n^5)$.

Demostración: Si $f(n) \leq cg(n)$ para $n \geq n_0$ entonces $a f(n) + b g(n) \leq c'g(n)$ para $n \geq n_0$ con $c' = ac + b$.

Nota: Pero la regla de la suma debe aplicarse un número constante de veces, si no, por ejemplo, consideremos una expresión como

$$T(n) = n^2 = n + n + \dots + n. \quad (1.26)$$

Aplicando repetidamente la regla de la suma, podríamos llegar a la conclusión que $T(n) = O(n)$, lo cual es ciertamente falso.

1.3.6. Regla del producto

Si $T_1(n) = O(f_1(n))$ y $T_2(n) = O(f_2(n))$ entonces $T_1(n)T_2(n) = O(f_1(n)f_2(n))$.

Demostración: Si $T_1(n) \leq c_1f_1(n)$ para $n \geq n_{01}$ y $T_2(n) \leq c_2f_2(n)$ para $n \geq n_{02}$ entonces $T_1(n)T_2(n) \leq f_1(n)f_2(n)$ para $n > n_0 = \max(n_{01}, n_{02})$ con $c = c_1c_2$.

1.3.7. Funciones típicas utilizadas en la notación asintótica

Cualquier función puede ser utilizada como tasa de crecimiento, pero las más usuales son, en orden de crecimiento

$$1 < \log n < \sqrt{n} < n < n^2 < \dots < n^p < 2^n < 3^n < \dots < n! < n^n \quad (1.27)$$

- La función logaritmo crece menos que cualquier potencia n^α con $\alpha > 1$.
- Los logaritmos en diferente base son equivalentes entre sí por la bien conocida relación

$$\log_b n = \log_b a \log_a n, \quad (1.28)$$

de manera que en muchas expresiones con logaritmos no es importante la base utilizada. En computación científica es muy común que aparezcan expresiones con logaritmos en base 2.

- En (1.27) “ $<$ ” representa $O(\dots)$. Es decir, $1 = O(\log n)$, $\log n = O(\sqrt{n})$, ...
- 1 representa las funciones constantes.
- Las potencias n^α (con $\alpha > 0$) se comparan entre sí según sus exponentes, es decir $n^\alpha = O(n^\beta)$ si $\alpha \leq \beta$. Por ejemplo, $n^2 = O(n^3)$, $n^{1/2} = O(n^{2/3})$.

- Las exponenciales a^n (con $a > 1$) se comparan según su base, es decir que $a^n = O(b^n)$ si $a \leq b$. Por ejemplo $2^n = O(3^n)$. En los problemas de computación científica es muy común que aparezcan expresiones con base $a = 2$.

Ejemplo 1.2: Notar que los valores de c y n_0 no son absolutos. En el caso del ejemplo 1.1 podríamos demostrar, con un poco más de dificultad, que $T(n) > 1.1n^2$ para $n \geq 21$.

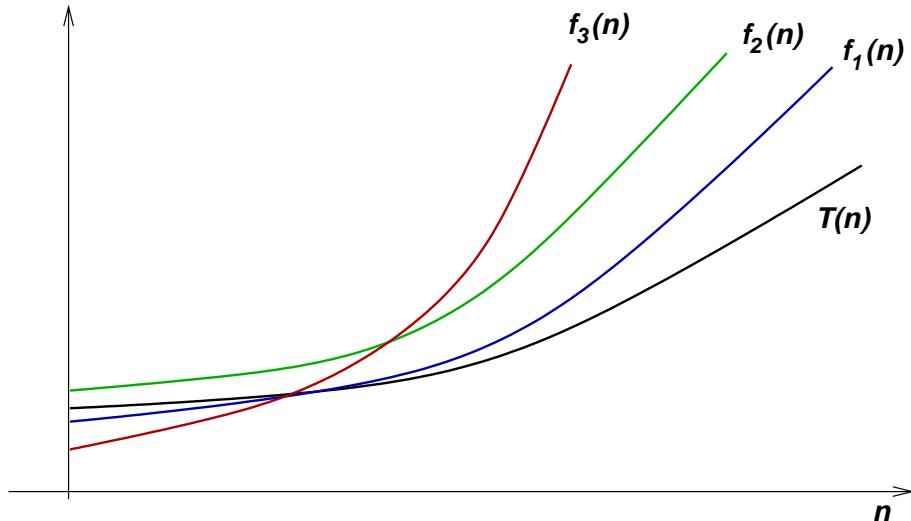


Figura 1.14: Decir que $T(n) = O(f_1(n))$ es “más fuerte” que $T(n) = O(f_2(n))$ o $T(n) = O(f_3(n))$

Ejemplo 1.3: Para $T(n) = (n+1)^2$ entonces también podemos decir que $T(n) = O(n^3)$ ya que, como vimos antes $T(n) = O(n^2)$ y $n^2 = O(n^3)$ de manera que por la transitividad (ver §1.3.4) $T(n) = O(n^3)$, pero decir que $T(n) = O(n^2)$ es una aseveración “más fuerte” del tiempo de ejecución ya que $n^2 < n^3$. En general (ver figura 1.14) si podemos tomar varias funciones $f_1(n)$, $f_2(n)$, $f_3(n)$ entonces debemos tomar la “menor” de todas. Así, si bien podemos decir que $T(n) = O(n^\alpha)$ para cualquier $\alpha \geq 2$ la más fuerte de todas es para $T(n) = O(n^2)$. Por otra parte puede verse que $T(n) \neq O(n^\alpha)$ para $\alpha < 2$. Razonemos por el absurdo. Si, por ejemplo, fuera cierto que $T(n) = O(n)$ entonces deberían existir $c, n_0 > 0$ tales que $T(n) = (n+1)^2 \leq cn$ para $n \geq n_0$. Pero entonces

$$\frac{(n+1)^2}{n} \leq c, \quad \text{para } n \geq n_0 \quad (1.29)$$

lo cual es ciertamente falso ya que

$$\frac{(n+1)^2}{n} \rightarrow \infty, \quad \text{para } n \rightarrow \infty. \quad (1.30)$$

Esta demostración puede extenderse fácilmente para cualquier $\alpha < 2$.

1.3.8. Equivalencia

Si dos funciones f y g satisfacen que $f(n) = O(g(n))$ y $g(n) = O(f(n))$ entonces decimos que “sus tasas de crecimiento son equivalentes” lo cual denotamos por

$$f \sim g \quad (1.31)$$

Así, en el ejemplo anterior 1.3 se puede demostrar ver que $n^2 = O((n+1)^2)$ por lo que

$$T(n) = (n+1)^2 \sim n^2 \quad (1.32)$$

1.3.9. La función factorial

Una función que aparece muy comúnmente en problemas combinatorios es la función factorial $n!$ (ver §1.1.7). Para poder comparar esta función con otras para grandes valores de n es conveniente usar la “aproximación de Stirling”

$$n! \sim \sqrt{2\pi} n^{n+1/2} e^{-n} \quad (1.33)$$

Gracias a esta aproximación es fácil ver que

$$n! = O(n^n) \quad (1.34)$$

y

$$a^n = O(n!), \quad (1.35)$$

lo cual justifica la ubicación del factorial en la tabla (1.3.7).

Demostración: La primera relación (1.34) se desprende fácilmente de aplicar la aproximación de Stirling y del hecho que $n^{1/2}e^{-n} \rightarrow 0$ para $n \rightarrow \infty$.

La segunda relación (1.35) se deduce de

$$n^{n+1/2} e^{-n} = a^n \left(\frac{n}{ae} \right)^n n^{1/2} \quad (1.36)$$

Entonces para $n \geq n_0 = \max(ae, 1)$

$$\left(\frac{n}{ae} \right)^n \geq 1 \quad (1.37)$$

y

$$n^{1/2} \geq n_0^{1/2} \quad (1.38)$$

con lo cual

$$n^{n+1/2} e^{-n} \geq n_0^{1/2} a^n \quad (1.39)$$

y por lo tanto

$$a^n \leq n_0^{-1/2} n^{n+1/2} e^{-n} \quad (1.40)$$

entonces

$$a^n = O(n^{n+1/2} e^{-n}) = O(n!) \quad (1.41)$$

con $c = n_0^{-1/2}$.

Ejemplo 1.4: Una de las ventajas de la notación asintótica es la gran simplificación que se obtiene en las expresiones para los tiempos de ejecución de los programas. Por ejemplo, si

$$T(n) = (3n^3 + 2n^2 + 6)n^5 + 2^n + 16n! \quad (1.42)$$

entonces vemos que, aplicando la regla de la suma, la expresión entre paréntesis puede estimarse como

$$(3n^3 + 2n^2 + 6) = O(n^3) \quad (1.43)$$

Aplicando la regla del producto, todo el primer término se puede estimar como

$$(3n^3 + 2n^2 + 6)n^5 = O(n^8) \quad (1.44)$$

Finalmente, usando la tabla (1.3.7) vemos que el término que gobierna es el último de manera que

$$T(n) = O(n!) \quad (1.45)$$

Muchas veces los diferentes términos que aparecen en la expresión para el tiempo de ejecución corresponde a diferentes partes del programa, de manera que, como ganancia adicional, la notación asintótica nos indica cuáles son las partes del programa que requieren más tiempo de cálculo y, por lo tanto, deben ser eventualmente optimizadas.

1.3.10. Determinación experimental de la tasa de crecimiento

A veces es difícil determinar en forma analítica la tasa de crecimiento del tiempo de ejecución de un algoritmo. En tales casos puede ser útil determinarla aunque sea en forma “experimental” es decir corriendo el programa para una serie de valores de n , tomar los tiempos de ejecución y a partir de estos datos obtener la tasa de crecimiento. Por ejemplo, para el algoritmo heurístico de la sección §1.1.8, si no pudiéramos encontrar el orden de convergencia, entonces ejecutamos el programa con una serie de valores de n obteniendo los valores de la tabla 1.2.

n	$T(n)$ [segundos]
300	0.2
600	1.2
1000	4.8
1500	14.5
3000	104.0

Tabla 1.2: Tiempo de ejecución del algoritmo ávido de la sección §1.1.8.

Graficando los valores en ejes logarítmicos (es decir, graficar $\log T(n)$ en función de $\log n$) obtenemos un gráfico como el de la figura 1.15. La utilidad de tales ejes es que las funciones de tipo potencia $\propto n^\alpha$ resultan ser rectas cuya pendiente es proporcional a α . Además curvas que difieren en una constante multiplicativa resultan ser simplemente desplazadas según la dirección vertical. Pero entonces si un programa tiene un comportamiento $T(n) = O(n^\alpha)$ pero no conocemos α , basta con graficar su tiempo de ejecución en ejes logarítmicos junto con varias funciones n^α y buscar cuál de ellas es paralela a la de $T(n)$. En el ejemplo

Figura 1.15: Determinación experimental del tiempo de ejecución del algoritmo ávido de coloración de la sección §1.1.8

de la figura vemos que $T(n)$ resulta ser perfectamente paralela a n^3 confirmando nuestra estimación de la sección §1.1.10.

En casos donde el tiempo de ejecución es exponencial, es decir $\sim a^n$ pueden ser preferibles ejes semi-logarítmicos, es decir, graficar $\log T(n)$ en función de n (y no en función de $\log n$ como en los logarítmicos) ya que en estos gráficos las exponenciales son rectas, con pendiente $\log a$.

Si no tenemos idea de que tipo de tasa de crecimiento puede tener un programa, podemos proceder en forma incremental. Primero probar con un gráfico logarítmico, si la curva resulta ser una recta, entonces es una potencia y determinando la pendiente de la recta determinamos completamente la tasa de crecimiento. Si la función no aparece como una recta y tiende a acelerarse cada vez más, de manera que crece más que cualquier potencia, entonces podemos probar con las exponenciales, determinando eventualmente la pendiente correspondiente. Finalmente, si crece todavía más que las exponenciales, entonces puede ser del tipo $n!$ o n^n . En este caso pueden intentarse una serie de procedimientos para refinar la estimación, pero debemos notar que en muchos casos basta con notar que la tasa de crecimiento es mayor que cualquier exponencial para calificar al algoritmo.

1.3.11. Otros recursos computacionales

Las técnicas desarrolladas en esta sección pueden aplicarse a cualquier otro tipo de recurso computacional, no sólo el tiempo de ejecución. Otro recurso muy importante es la memoria total requerida. Veremos, por ejemplo, en el capítulo sobre algoritmos de ordenamiento que el algoritmo de ordenamiento por “intercalamiento” (*merge-sort*) tiene un tiempo de ejecución $O(n \log n)$ en el caso promedio, pero llega a ser $O(n^2)$ en el peor caso. Pero existe una versión modificada que es siempre $O(n \log n)$, sin embargo la versión modificada requiere una memoria adicional que es a lo sumo $O(\sqrt{n})$.

1.3.12. Tiempos de ejecución no-polinomiales

Se dice que un algoritmo tiene tiempo “*polinomial*” (“*P*”, para abreviar), si es $T(n) = O(n^\alpha)$ para algún α . Por contraposición, aquellos algoritmos que tienen tiempo de ejecución mayor que cualquier polinomio (funciones exponenciales a^n , $n!$, n^n) se les llama “*no polinomiales*”. Por ejemplo, en el caso del problema de coloración de grafos discutido en la sección §1.1.1, el algoritmo exhaustivo descripto en §1.1.4 resulta ser no polinomial mientras que el descripto en la sección §1.1.8 es *P*. Esta nomenclatura ha sido originada por la gran diferencia entre la velocidad de crecimiento entre los algoritmos polinomiales y no polinomiales. En muchos casos, determinar que un algoritmo es no polinomial es la razón para descartar el algoritmo y buscar otro tipo de solución.

1.3.13. Problemas *P* y *NP*

Si bien para ciertos problemas (como el de colorear grafos) no se conocen algoritmos con tiempo de ejecución polinomial, es muy difícil demostrar que realmente es así, es decir que para ese problema no existe ningún algoritmo de complejidad polinomial.

Para ser más precisos hay que introducir una serie de conceptos nuevos. Por empezar, cuando se habla de tiempos de ejecución se refiere a instrucciones realizadas en una “*máquina de Turing*”, que es una abstracción de la computadora más simple posible, con un juego de instrucciones reducido. Una “*máquina de Turing no determinística*” es una máquina de Turing que en cada paso puede *invocar* un cierto número de instrucciones, y no una sola instrucción como es el caso de la máquina de Turing determinística. En cierta forma, es como si una máquina de Turing no-determinística pudiera invocar otras series de máquinas de Turing, de manera que en vez de tener un “*camino de cómputo*”, como es usual en una computadora secuencial, tenemos un “*árbol de cómputo*”. Un problema es “*NP*” si tiene un tiempo de ejecución polinomial en una máquina de Turing no determinística (*NP* significa aquí *non-deterministic polynomial*, y no *non-polynomial*). La pregunta del millón de dólares (literalmente!, ver <http://www.claymath.org>) es si existen problemas en *NP* para los cuales no existe un algoritmo polinomial.

Una forma de simplificar las cosas es demostrar que un problema se “*reduce*” a otro. Por ejemplo, el problema de hallar la mediana de un conjunto de N números (ver §5.4.2) (es decir el número tal que existen $N/2$ números menores o iguales que él y otros tantos $N/2$ mayores o iguales) se reduce a poner los objetos en un vector y ordenarlo, ya que una vez ordenado basta con tomar el elemento de la posición media. De manera que si tenemos un cierto algoritmo con una complejidad algorítmica para el ordenamiento, automáticamente tenemos una cota superior para el problema de la mediana. Se dice que un problema es “*NP-completo*” (*NPC*) si cualquier problema de *NP* se puede reducir a ese problema. Esto quiere decir, que los problemas de *NPC* son los candidatos a tener la más alta complejidad algorítmica de *NP*. Se ha demostrado que varios problemas pertenecen a *NPC*, entre ellos el *Problema del Agente Viajero* (1.1). Si se puede demostrar que algún problema de *NPC* tiene complejidad algorítmica no-polinomial (y por lo tanto todos los problemas de *NPC*) entonces $P \neq NP$. Por otra parte, si se encuentra algún algoritmo de tiempo polinomial para un problema de *NPC* entonces todos los problemas de *NPC* (y por lo tanto de *NP*) serán *P*, es decir $P = NP$. De aquí la famosa forma de poner la pregunta del millón que es: ¿“*Es P=NP?*”?

1.3.14. Varios parámetros en el problema

No siempre hay un sólo parámetro n que determina el tamaño del problema. Volviendo al ejemplo de la coloración de grafos, en realidad el tiempo de ejecución depende no sólo del número de vértices, sino también del número de aristas n_e , es decir $T(n_v, n_e)$. Algunos algoritmos pueden ser más apropiados cuando el número de aristas es relativamente bajo (grafos “*ralos*”) pero ser peor cuando el número de aristas es alto (grafos “*denses*”). En estos casos podemos aplicar las técnicas de esta sección considerando uno de los parámetros fijos y (digamos n_e) y considerar la tasa de crecimiento en función del otro parámetro n_v y viceversa. Otras veces es conveniente definir un parámetro adimensional como la “*tasa de ralitud*” (“*sparsity ratio*”)

$$s = \frac{n_e}{n_v(n_v - 1)/2} \quad (1.46)$$

y considerar el crecimiento en función de n_v a tasa de ralitud constante. (Notar que $s = 1$ para un grafo completamente *conectado*, $s = 0$ para un grafo completamente *desconectado*).

Por supuesto el análisis de los tiempos de ejecución se hace mucho más complejo cuantos más parámetros se consideran.

1.4. Conteo de operaciones para el cálculo del tiempo de ejecución

Comenzaremos por asumir que no hay llamadas recursivas en el programa. (Ni cadenas recursivas, es decir, **sub1()** llama a **sub()** y **sub2()** llama a **sub1()**). Entonces, debe haber rutinas que no llaman a otras rutinas. Comenzaremos por calcular el tiempo de ejecución de éstas. La regla básica para calcular el tiempo de ejecución de un programa es ir *desde los lazos o construcciones más internas hacia las más externas*. Se comienza asignando un costo computacional a las sentencias básicas. A partir de esto se puede calcular el costo de un bloque, sumando los tiempos de cada sentencia. Lo mismo se aplica para funciones y otras construcciones sintácticas que iremos analizando a continuación.

1.4.1. Bloques if

Para evaluar el tiempo de un bloque **if**

```
if(<cond>) {
    <body>
}
```

podemos o bien considerar el peor caso, asumiendo que **<body>** se ejecuta siempre

$$T_{\text{peor}} = T_{\text{cond}} + T_{\text{body}} \quad (1.47)$$

o, en el caso promedio, calcular la probabilidad P de que **<cond>** sea verdadero. En ese caso

$$T_{\text{prom}} = T_{\text{cond}} + P T_{\text{body}} \quad (1.48)$$

Notar que T_{cond} no está afectado por P ya que la condición se evalúa siempre. En el caso de que tenga un bloque **else**, entonces

```
if(<cond>) {
    <body-true>
} else {
    <body-false>
}
```

podemos considerar,

$$\begin{aligned} T_{\text{peor}} &= T_{\text{cond}} + \max(T_{\text{body-true}}, T_{\text{body-false}}) \\ &\leq T_{\text{cond}} + T_{\text{body-true}} + T_{\text{body-false}} \\ T_{\text{prom}} &= T_{\text{cond}} + P T_{\text{body-true}} + (1 - P) T_{\text{body-false}} \end{aligned} \quad (1.49)$$

Las dos cotas para T_{peor} son válidas, la que usa “max” es más precisa.

1.4.2. Lazos

El caso más simple es cuando el lazo se ejecuta un número fijo de veces, y el cuerpo del lazo tiene un tiempo de ejecución constante,

```
for (i=0; i<N; i++) {
    <body>
}
```

donde $T_{\text{body}} = \text{constante}$. Entonces

$$T = T_{\text{ini}} + N(T_{\text{body}} + T_{\text{inc}} + T_{\text{stop}}) \quad (1.50)$$

donde

- T_{ini} es el tiempo de ejecución de la parte de “*inicialización*” del lazo, en este caso **i=0**,
- T_{inc} es el tiempo de ejecución de la parte de “*incremento*” del contador del lazo, en este caso, **i++** y
- T_{stop} es el tiempo de ejecución de la parte de “*detención*” del contador del lazo, en este caso, **i<N**.

En el caso más general, cuando T_{body} no es constante, debemos evaluar explícitamente la suma de todas las contribuciones,

$$T = T_{\text{ini}} + \sum_{i=0}^{N-1} (T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}}). \quad (1.51)$$

Algunas veces es difícil calcular una expresión analítica para tales sumas. Si podemos determinar una cierta tasa de crecimiento para todos los términos

$$T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}} = O(f(n)), \quad \text{para todo } i \quad (1.52)$$

entonces,

$$T \leq N \max_{i=1}^{N-1} (T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}}) = O(Nf(n)) \quad (1.53)$$

Más difícil aún es el caso en que el número de veces que se ejecuta el lazo no se conoce a priori, por ejemplo un lazo **while** como el siguiente

```
while (<cond>) {  
    <body>  
}
```

En este caso debemos determinar también el número de veces que se ejecutará el lazo.

Ejemplo 1.5: Calcularemos el tiempo de ejecución del algoritmo de ordenamiento por el “método de la burbuja” (“bubble-sort”). Si bien a esta altura no es necesario saber exactamente como funciona el método, daremos una breve descripción del mismo. La función `bubble_sort(...)` toma como argumento un vector de enteros y los ordena de menor a mayor. En la ejecución del lazo de las líneas 6–16 para `j=0` el menor elemento de todos es insertado en `a[0]` mediante una serie de intercambios. A partir de ahí `a[0]` no es tocado más. Para `j=1` el mismo procedimiento es aplicado al rango de índices que va desde `j=1` hasta `j=n-1`, donde `n` es el número de elementos en el vector, de manera que después de la ejecución del lazo para `j=1` el segundo elemento menor es insertado en `a[1]` y así siguiendo hasta que todos los elementos terminan en la posición que les corresponde en el elemento ordenado.

```
1 void bubble_sort(vector<int> &a) {  
2     int n = a.size();  
3     // Lazo externo. En cada ejecucion de este lazo  
4     // el elemento j-esimo menor elemento llega a la  
5     // posicion 'a[j]'  
6     for (int j=0; j<n-1; j++) {  
7         // Lazo interno. Los elementos consecutivos se  
8         // van comparando y eventualmente son intercambiados.  
9         for (int k=n-1; k>j; k--) {  
10             if (a[k-1] > a[k]) {  
11                 int tmp = a[k-1];  
12                 a[k-1] = a[k];  
13                 a[k]=tmp;  
14             }  
15         }  
16     }  
17 }
```

Código 1.9: Algoritmo de clasificación por el método de la burbuja. [Archivo: `bubble.cpp`]

En el lazo interno (líneas 9–15) se realizan una serie de intercambios de manera de llevar el menor elemento del rango `k=j` a `k=n-1` a su posición. En cada ejecución del lazo se ejecuta en forma condicional el cuerpo del bloque `if` (líneas 11–13). Primero debemos encontrar el número de operaciones que se realiza en el cuerpo del bloque `if`, luego sumarlos para obtener el tiempo del lazo interno y finalmente sumarlo para obtener el del lazo externo.

El bloque de las líneas 11–13 requiere un número finito de operaciones (asignaciones de enteros, operaciones de adición/sustracción con enteros, referenciación de elementos de vectores). Llamaremos c_0 al número total de operaciones. Por supuesto lo importante aquí es que c_0 no depende de la longitud del vector n . Con respecto al condicional (líneas 10–14) tenemos que, si llamamos c_1 al número de operaciones necesarias para evaluar la condición `a[k-1]>a[k]`, entonces el tiempo es $c_0 + c_1$ cuando la condición da verdadera y c_1 cuando da falsa. Como de todas formas ambas expresiones son constantes ($O(1)$), podemos

tomar el criterio de estimar el peor caso, es decir que siempre se ejecute, de manera que el tiempo de ejecución de las líneas (líneas 10–14) es $c_0 + c_1$, constante. Pasando al lazo interno (líneas 9–15) éste se ejecuta desde $k = n - 1$ hasta $k = j + 1$, o sea un total de $(n - 1) - (j + 1) + 1 = n - j - 1$ veces. Tanto el cuerpo del lazo, como el incremento y condición de detención (línea 9) consumen un número constante de operaciones. Llamando c_2 a este número de operaciones tenemos que

$$T_{\text{líneas 9-15}} = c_3 + (n - j - 1)c_2 \quad (1.54)$$

donde c_3 es el número de operaciones en la inicialización del lazo (**k=n-1**). Para el lazo externo (líneas 6–16) tenemos

$$\begin{aligned} T_{\text{ini}} &= c_4, \\ T_{\text{stop}} &= c_5, \\ T_{\text{inc}} &= c_6, \\ T_{\text{body},j} &= c_3 + (n - j - 1)c_2. \end{aligned} \quad (1.55)$$

Lamentablemente el cuerpo del lazo no es constante de manera que no podemos aplicar (1.50), sino que debemos escribir explícitamente una suma

$$T(\text{líneas 6-16}) = c_4 + \sum_{j=0}^{n-2} (c_5 + c_6 + c_3 + (n - j - 1)c_2) \quad (1.56)$$

Los términos c_3 , c_5 y c_6 dentro de la suma son constantes, de manera que podemos poner

$$T(\text{líneas 6-16}) = c_4 + (n - 1)(c_3 + c_5 + c_6) + c_2 \sum_{j=0}^{n-2} (n - j - 1) \quad (1.57)$$

Ahora debemos hallar una expresión para la sumatoria. Notemos que

$$\sum_{j=0}^{n-2} (n - j - 1) = (n - 1) + (n - 2) + \dots + 1 = \left(\sum_{j=1}^n j \right) - n. \quad (1.58)$$

Consideremos entonces la expresión

$$\sum_{j=1}^n j \quad (1.59)$$

Gráficamente, podemos representar esta suma por una serie de columnas de cuadrados de lado unitario. En la primera columna tenemos un sólo cuadrado, en la segunda 2, hasta que en la última tenemos n . Para $n = 4$ tenemos una situación como en la figura 1.16. La suma (1.59) representa el área sombreada de la figura, pero ésta puede calcularse usando la construcción de la izquierda, donde se ve que el área es igual a la mitad inferior del cuadrado, de área $n^2/2$ más la suma de n áreas de triángulos de área $1/2$, por lo tanto

$$\sum_{j=1}^n j = \frac{n^2}{2} + \frac{n}{2} = \frac{n(n+1)}{2} \quad (1.60)$$

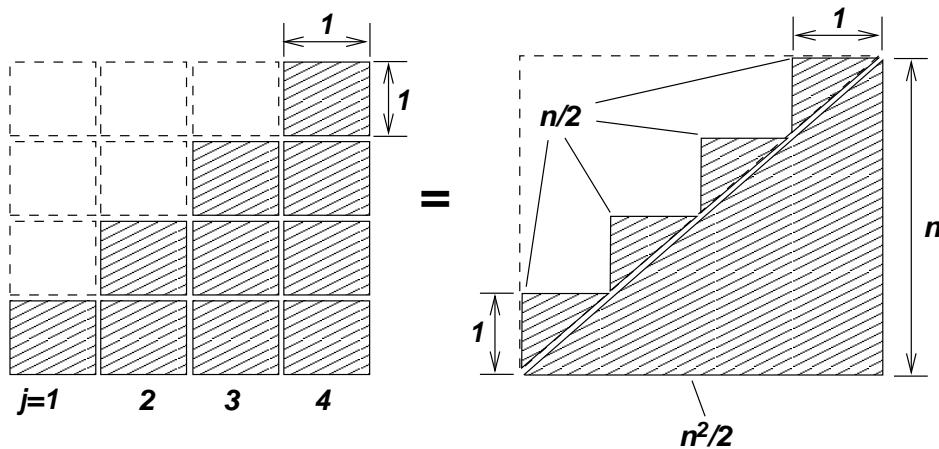


Figura 1.16: Cálculo gráfico de la suma en (1.59)

Podemos verificar la validez de esta fórmula para valores particulares de n , por ejemplo para $n = 4$ da 10, lo cual coincide con la suma deseada ($1+2+3+4$).

Volviendo a (1.57), vemos entonces que, usando (1.58) y (1.60)

$$\begin{aligned} T(\text{líneas 6–16}) &= c_4 + (n-1)(c_5 + c_6) + c_2 \left(\frac{n(n+1)}{2} - n \right) \\ &= c_4 + (n-1)(c_5 + c_6) + c_2 \frac{n(n-1)}{2} \end{aligned} \quad (1.61)$$

Finalmente, notemos que el tiempo total de la función `bubble_sort()` es igual al del lazo externo más un número constante de operaciones (la llamada a `vector<...>::size()` es de tiempo constante, de acuerdo a la documentación de STL). Llamando c_7 a las operaciones adicionales, incluyendo la llamada a la función, tenemos

$$T(\text{bubble_sort}) = c_4 + c_7 + (n-1)(c_5 + c_6) + c_2 \frac{n(n-1)}{2} \quad (1.62)$$

Ahora vamos a simplificar esta expresión utilizando los conceptos de notación asintótica. Primero note mos que

$$T(\text{bubble_sort}) \leq (c_4 + c_7) + n(c_5 + c_6) + c_2 \frac{n^2}{2} \quad (1.63)$$

y que los tres términos involucrados son $O(1)$, $O(n)$ y $O(n^2)$ respectivamente. De manera que, aplicando la regla de la suma, tenemos que

$$T(\text{bubble_sort}) = O(n^2) \quad (1.64)$$

Si bien, estos cálculos pueden parecer tediosos y engorrosos al principio, poco a poco el programador se va acostumbrando a hacerlos mentalmente y con experiencia, se puede hallar la tasa de crecimiento para funciones como `bubble_sort()` simplemente por inspección.

1.4.3. Suma de potencias

Sumas como (1.60) ocurren frecuentemente en los algoritmos con lazos anidados. Una forma alternativa de entender esta expresión es aproximar la suma por una integral (pensando a j como una variable continua)

$$\sum_{j=1}^n j \approx \int_0^n j \, dj = \frac{n^2}{2} = O(n^2). \quad (1.65)$$

De esta forma se puede llegar a expresiones asintóticas para potencias más elevadas

$$\sum_{j=1}^n j^2 \approx \int_0^n j^2 \, dj = \frac{n^3}{3} = O(n^3) \quad (1.66)$$

y, en general

$$\sum_{j=1}^n j^p \approx \int_0^n j^p \, dj = \frac{n^{p+1}}{p+1} = O(n^{p+1}) \quad (1.67)$$

1.4.4. Llamadas a rutinas

Una vez calculados los tiempos de ejecución de las rutinas que no llaman a otras rutinas (llamemos al conjunto de tales rutinas S_0), podemos calcular el tiempo de ejecución de aquellas rutinas que sólo llaman a las rutinas de S_0 (llamemos a este conjunto S_1), asignando a las líneas con llamadas a rutinas de S_0 de acuerdo con el tiempo de ejecución previamente calculado, como si fuera una instrucción más del lenguaje.

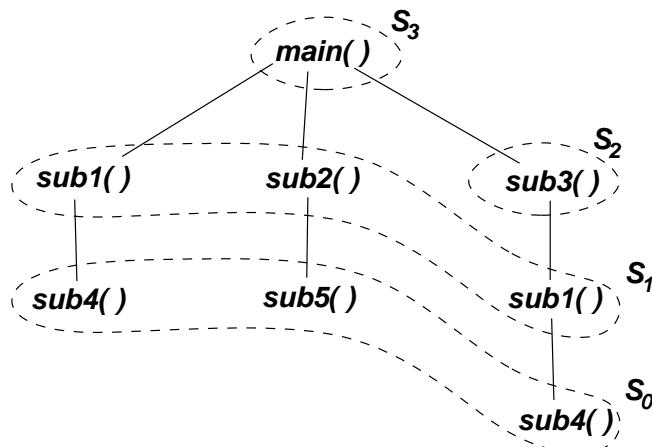


Figura 1.17: Ejemplo de árbol de llamadas de un programa y como calcular los tiempos de ejecución

Por ejemplo, si un programa tiene un árbol de llamadas como el de la figura 1.17, entonces podemos empezar por calcular los tiempos de ejecución de las rutinas `sub4()` y `sub5()`, luego los de `sub1()` y `sub2()`, luego el de `sub3()` y finalmente el de `main()`.

1.4.5. Llamadas recursivas

Si hay llamadas recursivas, entonces el principio anterior no puede aplicarse. Una forma de evaluar el tiempo de ejecución en tales casos es llegar a una expresión recursiva para el tiempo de ejecución mismo.

```

1 int bsearch2(vector<int> &a, int k, int j1, int j2) {
2     if (j1==j2-1) {
3         if (a[j1]==k) return j1;
4         else return j2;
5     } else {
6         int p = (j1+j2)/2;
7         if (k<a[p]) return bsearch2(a,k,j1,p);
8         else return bsearch2(a,k,p,j2);
9     }
10 }
11
12 int bsearch(vector<int> &a, int k) {
13     int n = a.size();
14     if (k<a[0]) return 0;
15     else return bsearch2(a,k,0,n);
16 }
```

Código 1.10: Algoritmo de búsqueda binaria. [Archivo: *bsearch.cpp*]

Ejemplo 1.6: Consideremos el algoritmo de “*búsqueda binaria*” (“*binary search*”) que permite encontrar un valor dentro de un vector ordenado. En el código 1.10 vemos una implementación típica. La rutina **bsearch()** toma como argumentos un **vector<int>** (que se asume que está ordenado de menor a mayor y sin valores repetidos) y un entero *k* y retorna la primera posición *j* dentro del arreglo tal que $k \leq a_j$. Si *k* es mayor que todos los elementos de *a*, entonces debe retornar *n*, como si en la posición *n*, (que está fuera del arreglo) hubiera un ∞ . La rutina utiliza una rutina auxiliar **bsearch2(a,k,j1,j2)** la cual busca el elemento en un rango $[j_1, j_2)$ (que significa $j_1 \leq j < j_2$). Este rango debe ser un rango válido en *a*, es decir $j_1 < j_2$, $0 \leq j_1 < n$, $1 \leq j_2 \leq n$ y $a[j_1] \leq k < a[j_2]$. Notar que *j₂* puede tomar la posición “*ficticia*” *n*, pero *j₁* no.

La rutina **bsearch()** determina primero un rango válido inicial. Si $k \geq a_0$, entonces $[0, n)$ es un rango válido y llama a **bsearch()** mientras que si no la posición *j* = 0 es la solución al problema.

La rutina **bsearch2** opera recursivamente calculando un punto medio *p* y llamando nuevamente a **bsearch2()**, ya sea con el intervalo $[j_1, p)$ o $[p, j_2)$. En cada paso el tamaño del rango se reduce en un factor cercano a 2, de manera que en un cierto número de pasos el tamaño del intervalo se reduce a 1, en cuyo caso termina la recursión.

Consideremos ahora el tiempo de ejecución de la función **bsearch2()** como función del número de elementos $m = j_2 - j_1$ en el intervalo. Si la condición de la línea 2 da verdadero entonces $m = 1$ y el tiempo es una constante *c*. Caso contrario, se realiza un número constante de operaciones *d* más una llamada a **bsearch2()** (en la línea 7 ó la 8) con un rango de longitud menor. Por simplicidad asumiremos que *m* es una potencia de 2, de manera que puede verse que el nuevo intervalo es de longitud $m/2$. Resumiendo

$$T(m) = \begin{cases} c & ; \text{ si } m = 1; \\ d + T(m/2) & ; \text{ si } m > 1; \end{cases} \quad (1.68)$$

Ahora, aplicando recursivamente esta expresión, tenemos que

$$\begin{aligned} T(2) &= d + T(1) = d + c \\ T(4) &= d + T(2) = 2d + c \\ T(8) &= d + T(4) = 3d + c \\ &\vdots \\ T(2^p) &= d + T(2^{p-1}) = pd + c \end{aligned} \tag{1.69}$$

como $p = \log_2 m$, vemos que

$$T(m) = d \log_2 m + c = O(\log_2 m) \tag{1.70}$$

Notar que el algoritmo más simple, consistente en recorrer el vector hasta el primer elemento mayor que k sería $O(n)$ en el peor caso y $O(n/2)$ en promedio (siempre que el elemento este en el vector), ya que en promedio encontrará al elemento en la parte media del mismo. El reducir el tiempo de ejecución de $O(n)$ a $O(\log n)$ es, por supuesto, la gran ventaja del algoritmo de búsqueda binaria.

Capítulo 2

Tipos de datos abstractos fundamentales

En este capítulo se estudiarán varios tipos de datos básicos. Para cada uno de estos TAD se discutirán en el siguiente orden

1. Sus operaciones abstractas.
2. Una interfaz básica en C++ y ejemplos de uso con esta interfaz.
3. Una o más implementaciones de esa interfaz, discutiendo las ventajas y desventajas de cada una, tiempos de ejecución ...
4. Una interfaz más avanzada, compatible con las STL, usando templates, sobrecarga de operadores y clases anidadas y ejemplos de uso de esta interfaz.
5. Una implementación de esta interfaz.

La razón de estudiar primero la interfaz básica (sin templates, sobrecarga de operadores ni clases anidadas) es que estos ítems pueden ser demasiado complejos de entender, en cuanto a sintaxis, para un programador principiante, y en este libro el énfasis está puesto en el uso de la interfaz, el concepto de TAD y la comprensión de los tiempos de ejecución de los diferentes algoritmos, y no en sutilezas sintácticas del C++. De todas formas, en las fases 4 y 5 se muestra una interfaz compatible con la STL, ejemplos de uso e implementación, ya que pretendemos que este libro sirva también para aprender a usar correcta y eficientemente las STL.

2.1. El TAD Lista

Las listas constituyen una de las estructuras lineales más flexibles, porque pueden crecer y acortarse según se requiera, insertando o suprimiendo elementos tanto en los extremos como en cualquier otra posición de la lista. Por supuesto esto también puede hacerse con vectores, pero en las implementaciones más comunes estas operaciones son $O(n)$ para los vectores, mientras que son $O(1)$ para las listas. El poder de las listas es tal que la familia de lenguajes derivados del Lisp, que hoy en día cuenta con el Lisp, Common Lisp y Scheme, entre otros, el lenguaje mismo está basado en la lista (“Lisp” viene de “list processing”).

2.1.1. Descripción matemática de las listas

Desde el punto de vista abstracto, una lista es una secuencia de cero o más elementos de un tipo determinado, que en general llamaremos **elem_t**, por ejemplo **int** o **double**. A menudo representamos una lista en forma impresa como una sucesión de elementos entre paréntesis, separados por comas

$$L = (a_0, a_1, \dots, a_{n-1}) \quad (2.1)$$

donde $n \geq 0$ es el número de elementos de la lista y cada a_i es de tipo **elem_t**. Si $n \geq 1$ entonces decimos que a_0 es el primer elemento y a_{n-1} es el último elemento de la lista. Si $n = 0$ decimos que la lista “está vacía”. Se dice que n es la “*longitud*” de la lista.

Una propiedad importante de la lista es que sus elementos están ordenados en forma lineal, es decir, para cada elemento a_i existe un sucesor a_{i+1} (si $i < n - 1$) y un predecesor a_{i-1} (si $i > 0$). Este orden es parte de la lista, es decir, dos listas son iguales si tienen los mismos elementos y *en el mismo orden*. Por ejemplo, las siguientes listas son distintas

$$(1, 3, 7, 4) \neq (3, 7, 4, 1) \quad (2.2)$$

Mientras que en el caso de conjuntos, éstos serían iguales. Otra diferencia con los conjuntos es que puede haber elementos repetidos en una lista, mientras que en un conjunto no.

Decimos que el elemento a_i “está en la posición i ”. También introducimos la noción de una posición ficticia n que “está fuera de la lista”. A las posiciones en el rango $0 \leq i \leq n - 1$ las llamamos “derefenciables” ya que pertenecen a un objeto real, y por lo tanto podemos obtener una referencia a ese objeto. Notar que, a medida que se vayan insertando o eliminando elementos de la lista la posición ficticia n va variando, de manera que convendrá tener un método de la clase **end()** que retorne esta posición.

2.1.2. Operaciones abstractas sobre listas

Consideremos un operación típica sobre las listas que consiste en eliminar todos los elementos duplicados de la misma. El algoritmo más simple consiste en un doble lazo, en el cual el lazo externo sobre i va desde el comienzo hasta el último elemento de la lista. Para cada elemento i el lazo interno recorre desde $i + 1$ hasta el último elemento, eliminando los elementos iguales a i . Notar que no hace falta revisar los elementos anteriores a i (es decir, los elementos j con $j < i$), ya que, por construcción, todos los elementos de 0 hasta i son distintos.

Este problema sugiere las siguientes operaciones abstractas

- Dada una posición i , “*insertar*” el elemento x en esa posición, por ejemplo

$$\begin{aligned} L &= (1, 3, 7) \\ &\text{inserta } 5 \text{ en la posición } 2 \\ &\rightarrow L = (1, 3, 5, 7) \end{aligned} \quad (2.3)$$

Notar que el elemento 7, que estaba en la posición 2, se desplaza hacia el fondo, y termina en la posición 3. Notar que es válido insertar en cualquier posición dereferenciable, o en la posición ficticia **end()**

- Dada una posición i , “suprimir” el elemento que se encuentra en la misma. Por ejemplo,

$$\begin{aligned} L &= (1, 3, 5, 7) \\ \text{suprime elemento en la posición 2} \\ \rightarrow L &= (1, 3, 7) \end{aligned} \tag{2.4}$$

Notar que esta operación es, en cierta forma, la inversa de insertar. Si, como en el ejemplo anterior, insertamos un elemento en la posición i y después suprimimos en esa misma posición, entonces la lista queda inalterada. Notar que sólo es válido suprimir en las posiciones dereferenciables.

Si representáramos las posiciones como enteros, entonces avanzar la posición podría efectuarse con la sencilla operación de enteros $i \leftarrow i + 1$, pero es deseable pensar en las posiciones como entidades abstractas, no necesariamente enteros y por lo tanto para las cuales no necesariamente es válido hacer operaciones de enteros. Esto lo sugiere la experiencia previa de cursos básicos de programación donde se ha visto que las listas se representan por celdas encadenadas por punteros. En estos casos, puede ser deseable representar a las posiciones como punteros a las celdas. De manera que asumiremos que las posiciones son objetos abstractos. Consideramos entonces las operaciones abstractas:

- Acceder al elemento en la posición p , tanto para modificar el valor ($a_p \leftarrow x$) como para acceder al valor ($x \leftarrow a_p$).
- Avanzar una posición, es decir dada una posición p correspondiente al elemento a_i , retornar la posición q correspondiente al elemento a_{i+1} . (Como mencionamos previamente, no es necesariamente $q = p + 1$, o más aún, pueden no estar definidas estas operaciones aritméticas sobre las posiciones p y q .)
- Retornar la primera posición de la lista, es decir la correspondiente al elemento a_0 .
- Retornar la posición ficticia al final de la lista, es decir la correspondiente a n .

2.1.3. Una interfaz simple para listas

Definiremos ahora una interfaz apropiada en C++. Primero observemos que, como las posiciones no serán necesariamente enteros, enmascararemos el concepto de posición en una clase **iterator_t**. El nombre está tomado de las STL, agregándole el sufijo **_t** para hacer énfasis en que es un tipo. En adelante hablaremos indiferentemente de posiciones o iterators. La interfaz puede observarse en el código 2.1.

Primero declaramos la clase **iterator_t** de la cual no damos mayores detalles. Luego la clase **list**, de la cual sólo mostramos algunos de sus métodos públicos.

```

1 class iterator_t { /* . . . */ };
2
3 class list {
4 private:
5     // ...
6 public:
7     // ...
8     iterator_t insert(iterator_t p, elem_t x);
9     iterator_t erase(iterator_t p);

```

```
10 elem_t & retrieve(iterator_t p);
11 iterator_t next(iterator_t p);
12 iterator_t begin();
13 iterator_t end();
14 }
```

Código 2.1: Interfaz básica para listas. [Archivo: listbas.cpp]

- **insert**: inserta el elemento **x** en la posición **p**, devolviendo una posición **q** al elemento insertado. Todas las posiciones de **p** en adelante (incluyendo **p**) pasan a ser inválidas, por eso la función devuelve a **q**, la nueva posición insertada, ya que la anterior **p** es inválida. Es válido insertar en cualquier posición dereferenciable o no dereferenciable, es decir que es válido insertar también en la posición ficticia.
- **erase**: elimina el elemento en la posición **p**, devolviendo una posición **q** al elemento que previamente estaba en la posición siguiente a **p**. Todas las posiciones de **p** en adelante (incluyendo **p**) pasan a ser inválidas. Sólo es válido suprimir en las posiciones dereferenciables de la lista.
- **retrieve**: “recupera” el elemento en la posición **p**, devolviendo una *referencia* al mismo, de manera que es válido hacer tanto **x = L.retrieve(p)** como **L.retrieve(p)=x**. Se puede aplicar a cualquier posición **p** dereferenciable y no modifica a la lista. Notar que retorna una *referencia* al elemento correspondiente de manera que este puede ser cambiado, es decir, puede ser usado como un “*valor assignable*” (“left hand side value”).
- **next**: dada una posición dereferenciable **p**, devuelve la posición del siguiente elemento. Si **p** es la última posición dereferenciable, entonces devuelve la posición ficticia. No modifica la lista.
- **begin**: devuelve la posición del primer elemento de la lista.
- **end**: devuelve la posición ficticia (no dereferenciable), después del final de la lista.

Algunas observaciones con respecto a estas funciones son

- **Posiciones inválidas**: Un elemento a tener en cuenta es que las funciones que modifican la lista como **insert** and **erase**, convierten en inválidas algunas de las posiciones de la lista, normalmente desde el punto de inserción/supresión en adelante, incluyendo **end()**. Por ejemplo,

```
1 iterator_t p,q,r;
2 list L;
3 elem_t x,y,z;
4 //...
5 // p es una posicion dereferenciable
6 q = L.next(p);
7 r = L.end();
8 L.erase(p);
9 x = *p;           // incorrecto
10 y = *q;          // incorrecto
11 L.insert(r,z); // incorrecto
```

ya que **p, q, r** ya no son válidos (están después de la posición borrada **p**). La forma correcta de escribir el código anterior es

```
1 iterator_t p,q,r;
2 list L;
3 elem_t x,y,z;
4 //...
5 // p es una posición dereferenciable
6 p = L.erase(p);
7 x = *p;           // correcto
8 q = L.next(p);
9 y = *q;           // correcto
10 r = L.end();
11 L.insert(r,z); // correcto
```

- **Las posiciones sólo se acceden a través de funciones de la clase:** Las únicas operaciones válidas con posiciones son

▷ **Asignar:**

```
p = L.begin();
q = L.end();
```

▷ **Avanzar:**

```
q = L.next(p);
```

▷ **Acceder al elemento:**

```
x = L.retrieve(p);
L.retrieve(q) = y;
```

▷ **Copiar:**

```
q = p;
```

▷ **Comparar:** Notar que sólo se puede comparar por igualdad o desigualdad, no por operadores de comparación, como < ó >.

```
q == p
r != L.end();
```

2.1.4. Funciones que retornan referencias

A veces es útil escribir funciones que dan acceso a ciertos componentes internos de estructuras complejas, permitiendo cambiar su valor. Supongamos que queremos escribir una función **int min(int *v,int n)** que retorna el mínimo de los valores de un vector de enteros **v** de longitud **n**, pero además queremos dar la posibilidad al usuario de la función de *cambiar el valor interno* correspondiente al mínimo. Una posibilidad es retornar por un argumento adicional el índice **j** correspondiente al mínimo. Posteriormente para modificar el valor podemos hacer **v[j]=<nuevo-valor>**. El siguiente fragmento de código modifica el valor del mínimo haciendo que valga el doble de su valor anterior.

```
1 int min(int *v, int n, int *jmin);
2 ...
3
4 int jmin;
5 int m = min(v, n, &jmin);
6 v[jmin] = 2*v[jmin];
```

Sin embargo, si **min** operara sobre estructuras más complejas sería deseable que retornara directamente un objeto modificable, es decir que pudiéramos hacer

```
1 min(v, n) = 2*min(v, n);
```

```
1 int *min(int *v, int n) {
2     int x = v[0];
3     int jmin = 0;
4     for (int k=1; k<n; k++) {
5         if (v[k]<x) {
6             jmin = k;
7             x = v[jmin];
8         }
9     }
10    return &v[jmin];
11 }
12
13 void print(int *v, int n) {
14     cout << "Vector: (" ;
15     for (int j=0; j<n; j++) cout << v[j] << " ";
16     cout << "), valor minimo: " << *min(v, n) << endl;
17 }
18
19 int main() {
20     int v[] = {6,5,1,4,2,3};
21     int n = 6;
22
23     print(v, n);
24     for (int j=0; j<6; j++) {
25         *min(v, n) = 2* (*min(v, n));
26         print(v, n);
27     }
28 }
```

Código 2.2: Ejemplo de función que retorna un puntero a un elemento interno, de manera de poder modificarlo. [Archivo: ptrexa.cpp]

Esto es posible de hacer en C, si modificamos **min** de manera que *retorne un puntero* al elemento mínimo. El código 2.2 muestra una posible implementación. A continuación se muestra la salida del programa.

```
1 [mstorti@spider aedsrc]$ ptrexa
2 Vector: (6 5 1 4 2 3 ), valor minimo: 1
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```

3 Vector: (6 5 2 4 2 3 ), valor minimo: 2
4 Vector: (6 5 4 4 2 3 ), valor minimo: 2
5 Vector: (6 5 4 4 4 3 ), valor minimo: 3
6 Vector: (6 5 4 4 4 6 ), valor minimo: 4
7 Vector: (6 5 8 4 4 6 ), valor minimo: 4
8 Vector: (6 5 8 8 4 6 ), valor minimo: 4
9 [mstorti@spider aedsrc]$

```

```

1 int &min(int *v, int n) {
2     int x = v[0];
3     int jmin = 0;
4     for (int k=1; k<n; k++) {
5         if (v[k]<x) {
6             jmin = k;
7             x = v[jmin];
8         }
9     }
10    return v[jmin];
11 }
12
13 void print(int *v, int n) {
14     cout << "Vector: (" ;
15     for (int j=0; j<n; j++) cout << v[j] << " ";
16     cout << "), valor minimo: " << min(v,n) << endl;
17 }
18
19 int main() {
20     int v[] = {6,5,1,4,2,3};
21     int n = 6;
22
23     print(v,n);
24     for (int j=0; j<6; j++) {
25         min(v,n) = 2*min(v,n);
26         print(v,n);
27     }
28 }

```

Código 2.3: Ejemplo de función que retorna una referencia a un elemento interno, de manera de poder modificarlo. [Archivo: refexa.cpp]

C++ permite retornar directamente referencias (`int &`, por ejemplo) a los elementos, de manera que no hace falta despues dereferenciarlos como en la línea 25. El mismo program usando referencias puede verse en el código 2.3. El método `val=retrieve(p)` en la interfaz presentada para listas es un ejemplo. Esta técnica es usada frecuentemente en las STL.

2.1.5. Ejemplos de uso de la interfaz básica

```

1 void purge(list &L) {
2     iterator_t p,q;
3     p = L.begin();
4     while (p!=L.end()) {
5         q = L.next(p);
6         while (q!=L.end()) {
7             if (L.retrieve(p)==L.retrieve(q)) {
8                 q = L.erase(q);
9             } else {
10                q = L.next(q);
11            }
12        }
13        p = L.next(p);
14    }
15 }
16
17 int main() {
18     list L;
19     const int M=10;
20     for (int j=0; j<2*M; j++)
21         L.insert(L.end(),rand()%M);
22     cout << "Lista antes de purgar: " << endl;
23     print(L);
24     cout << "Purga lista... " << endl;
25     purge(L);
26     cout << "Lista despues de purgar: " << endl;
27     print(L);
28 }
```

Código 2.4: Eliminar elementos repetidos de una lista [Archivo: purge.cpp]

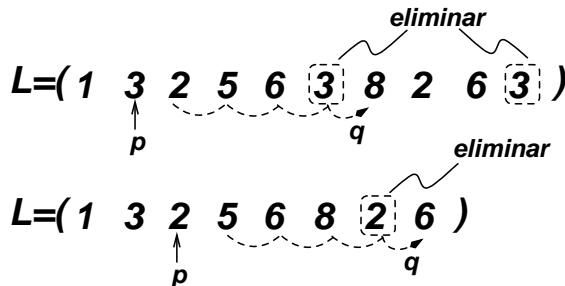


Figura 2.1: Proceso de eliminación de elementos en purge

Ejemplo 2.1: Eliminar elementos duplicados de una lista. Volviendo al problema descripto en §2.1.1 de eliminar los elementos repetidos de una lista, consideremos el código [código 2.4](#). Como describimos previamente, el algoritmo tiene dos lazos anidados. En el lazo exterior una posición p recorre todas las posiciones de la lista. En el lazo interior otra posición q recorre las posiciones más allá de p eliminando los elementos iguales a los que están en p .

El código es muy cuidadoso en cuanto a usar siempre posiciones válidas. Por ejemplo la operación `L.retrieve(q)` de la línea 7 está garantizado que no fallará. Asumiendo que `p` es una posición válida dereferenciable a la altura de la línea 5, entonces en esa línea `q` recibe una posición válida, dereferenciable o no. Si la posición asignada a `q` es `end()`, entonces la condición del `while` fallará y `retrieve` no se ejecutará. A su vez, `p` es dereferenciable en la línea 5 ya que en la línea 3 se le asignó una posición válida (`L.begin()` es siempre válida, y también es dereferenciable a menos que la lista este vacía). Pero al llegar a la línea 5 ha pasado el test de la línea precedente, de manera que seguramente es dereferenciable. Luego, `p` sólo es modificada en la línea 13. Notar que a esa altura no es trivial decir si `p` es válida o no, ya que eventualmente pueden haberse hecho operaciones de eliminación en la línea 8. Sin embargo, todas estas operaciones se realizan sobre posiciones que están más allá de `p`, de manera que, efectivamente `p` llega a la línea 13 siendo una posición válida (de hecho dereferenciable). Al avanzar `p` una posición en línea 13 puede llegar a tomar el valor `end()`, pero en ese caso el lazo terminará, ya que fallará la condición de la línea 4.

En el `main()` (líneas 17–28) se realiza una verificación del funcionamiento de `purge()`. Primero se declara una variable `L` de tipo `list` (asumiendo que el tipo elemento es entero, es decir `elem_t=int`). Por supuesto esto involucra todas las tareas de inicialización necesarias, que estarán dentro del constructor de la clase, lo cual se discutirá más adelante en las diferentes implementaciones de `list`. Para un cierto valor `M` se agregan $2*M$ elementos generados aleatoriamente entre `0` y `M-1`. (La función `irand(int n)` retorna elementos en forma aleatoria entre `0` y `n - 1`). La lista se imprime por consola con la función `print()`, es purgada y luego vuelta a imprimir. Por brevedad, no entraremos aquí en el código de `print()` e `irand()`. Una salida típica es la siguiente

```

1 [mstorti@minerva aedsrc]$ purge
2 Lista antes de purgar:
3 8 3 7 7 9 1 3 7 2 5 4 6 3 5 9 9 6 7 1 6
4 Purga lista...
5 Lista despues de purgar:
6 8 3 7 9 1 2 5 4 6
7 [mstorti@minerva aedsrc]$

```

$$\begin{array}{c} L1=\{ 1,2,3, \quad 3,2,5, \quad 4,1, \quad 6, \quad 8,3\} \\ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\ L2=\{ \quad 6, \quad 10, \quad 5, \quad 6, \quad 11\} \end{array}$$

Figura 2.2: Agrupar subsecuencias sumando.

Ejemplo 2.2: *Consigna:* Dadas dos listas de enteros positivos `L1` y `L2` escribir una función `bool check_sum(list &L1, list &L2);` que retorna verdadero si los elementos de `L1` pueden agruparse (sumando secuencias de elementos contiguos) de manera de obtener los elementos de `L2` sin alterar el orden de los elementos. Por ejemplo, en el caso de la figura 2.2 `check_sum(L1, L2)` debe retornar verdadero ya que los agrupamientos mostrados reducen la lista `L1` a la `L2`.

Solución Proponemos el siguiente algoritmo. Tenemos dos posiciones `p, q` en `L1` y `L2`, respectivamente, y un acumulador `suma` que contiene la suma de un cierto número de elementos de `L1`. En un lazo infinito vamos avanzando los punteros `p, q` de manera que los elementos que están antes de ellos verifican las siguientes condiciones (ver figura 2.3):

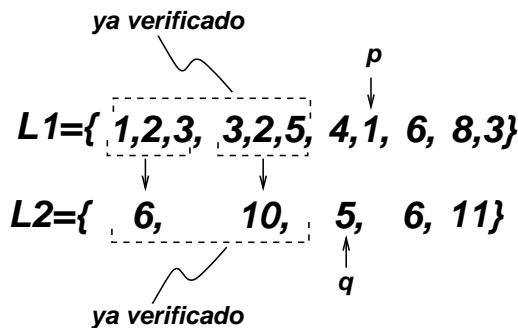


Figura 2.3: Estado parcial en el algoritmo check-sum

- Cada elemento de **L2** antes de **p** se corresponde con una serie de elementos de **L1**, sin dejar huecos, salvo eventualmente un resto, al final de **L1**.
- La suma del resto de los elementos de **L1** coincide con el valor de **suma**.

Inicialmente **p,q** están en los comienzos de las listas y **suma=0**, que ciertamente cumple con las condiciones anteriores. Después de una serie de pasos, se puede llegar a un estado como el mostrado en la figura 2.3. Tenemos (**p->1, q->5, suma=4**).

Para avanzar las posiciones comparamos el valor actual de **suma** con **L.retrieve(q)** y seguimos las siguientes reglas,

1. **Avanza q:** Si **suma==L2.retrieve(q)** entonces ya hemos detectado un grupo de **L1** que coincide con un elemento de **L2**. Ponemos **suma** en 0, y avanzamos **q**.
2. **Avanza p:** Si **suma<L2.retrieve(q)** entonces podemos avanzar **p** acumulando **L2.retrieve(p)** en **suma**.
3. **Falla:** Si **suma>L2.retrieve(q)** entonces las listas no son compatibles, hay que retornar falso.

Mientras tanto, en todo momento antes de avanzar una posición hay que verificar de mantener la validez de las mismas. El lazo termina cuando alguna de las listas se termina. El programa debe retornar verdadero si al salir del lazo ambas posiciones están al final de sus respectivas listas y **suma==0**.

Partiendo del estado de la figura 2.3, tenemos los siguientes pasos

- (**p->1, q->5, suma=4**): avanza **p**, **suma=5**
- (**p->6, q->5, suma=5**): avanza **q**, **suma=0**
- (**p->6, q->6, suma=0**): avanza **p**, **suma=6**
- (**p->8, q->6, suma=6**): avanza **q**, **suma=0**
- (**p->8, q->11, suma=0**): avanza **p**, **suma=8**
- (**p->3, q->11, suma=8**): avanza **p**, **suma=11**
- (**p->end(), q->11, suma=11**): avanza **q**, **suma=0**
- (**p->end(), q->end(), suma=0**): Sale del lazo.

```

1 bool check_sum(list &L1, list &L2) {
2     iterator_t p,q;
```

```

3  p = L1.begin();
4  q = L2.begin();
5  int suma = 0;
6  while (true) {
7      if (q==L2.end()) break;
8      else if (suma==L2.retrieve(q)) {
9          suma=0;
10         q = L2.next(q);
11     }
12     else if (p==L1.end()) break;
13     else if (suma<L2.retrieve(q)) {
14         suma += L1.retrieve(p);
15         p = L1.next(p);
16     }
17     else return false;
18 }
19 return suma==0 && p==L1.end() && q==L2.end();
20 }
```

Código 2.5: Verifica que L2 proviene de L1 sumando elementos consecutivos. [Archivo: check-sum.cpp]

El código correspondiente se muestra en el código 2.5. Después de inicializar **p, q, suma** se entra en el lazo de las líneas 6–18 donde se avanza **p, q**. Los tres casos listados más arriba coinciden con tres de las entradas en el **if**. Notar que antes de recuperar el elemento en **q** en la línea 8 hemos verificado que la posición es dereferenciable porque pasó el test de la línea precedente. **q** es avanzado en la línea 10 con lo cual uno podría pensar que el **retrieve** que se hace en la línea 13 podría fallar si en ese avance **q** llego a **end()**. Pero esto no puede ser así, ya que si se ejecuta la línea 10, entonces el condicional de la línea 13 sólo puede hacerse en otra ejecución del lazo del **while** ya que ambas líneas están en ramas diferentes del mismo **if**.

2.1.6. Implementación de listas por arreglos

A continuación veremos algunas posibles implementaciones de listas. Probablemente la representación de listas más simple de entender es mediante arreglos. En esta representación los valores son almacenados en celdas contiguas de un arreglo, como se muestra en la figura 2.4. Las posiciones se representan simplemente mediante enteros (recordemos que, en general, esto no es así para otras implementaciones). El principal problema de esta representación es que, para insertar un elemento en una posición intermedia de la lista requiere mover todos los elementos que le suceden una posición hacia el final (ver 2.5). Igualmente, para borrar un elemento hay que desplazar todos los elementos que suceden una posición hacia el comienzo para “rellenar” el hueco dejado por el elemento eliminado.

```

1  typedef int iterator_t;
2
3  class list {
4  private:
5      static int MAX_SIZE;
6      elem_t *elems;
```

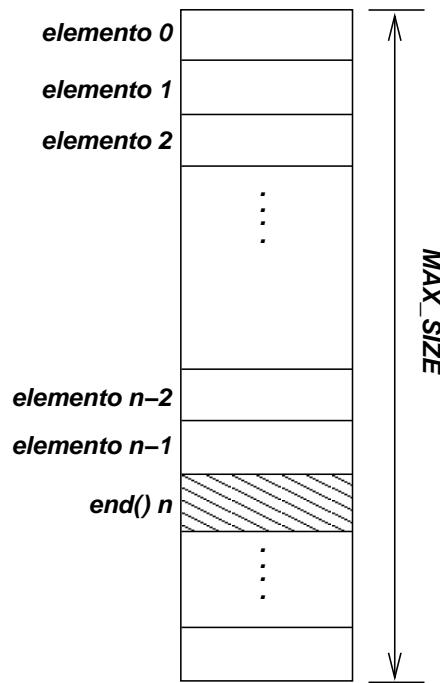


Figura 2.4: Representación de listas mediante arreglos

```

7   int size;
8 public:
9   list();
10  ~list();
11  iterator_t insert(iterator_t p, elem_t j);
12  iterator_t erase(iterator_t p);
13  iterator_t erase(iterator_t p, iterator_t q);
14  void clear();
15  iterator_t begin();
16  iterator_t end();
17  void print();
18  iterator_t next(iterator_t p);
19  iterator_t prev(iterator_t p);
20  elem_t & retrieve(iterator_t p);
21 };

```

Código 2.6: Declaraciones para listas implementadas por arreglos. [Archivo: `lista.h`]

```

1 #include <iostream>
2 #include <aedsrc/lista.h>
3 #include <cstdlib>
4
5 using namespace std;

```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

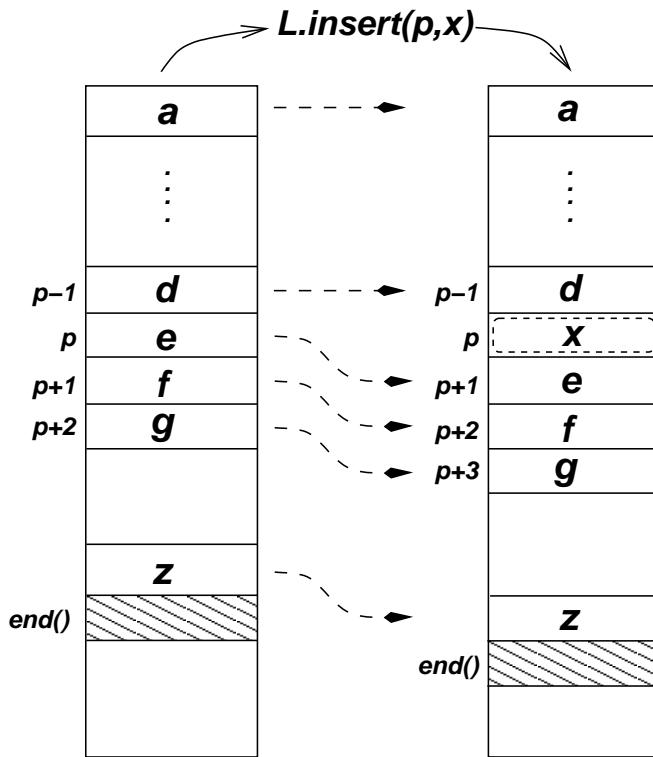


Figura 2.5: Inserción de un elemento en la representación de listas por arreglos.

```

6 using namespace aed;
7
8 int list::MAX_SIZE=100;
9
10 list::list() : elems(new elem_t[MAX_SIZE]),
11     size(0) { }
12
13 list::~list() { delete[] elems; }
14
15 elem_t &list::retrieve(iterator_t p) {
16     if (p<0 || p>=size) {
17         cout << "p: mala posicion.\n";
18         abort();
19     }
20     return elems[p];
21 }
22
23
24 iterator_t list::begin() { return 0; }
25
26 iterator_t list::end() { return size; }
27
28 iterator_t list::next(iterator_t p) {

```

```
29 if (p<0 || p>=size) {
30     cout << "p: mala posicion.\n";
31     abort();
32 }
33 return p+1;
34 }
35
36 iterator_t list::prev(iterator_t p) {
37     if (p<=0 || p>size) {
38         cout << "p: mala posicion.\n";
39         abort();
40     }
41     return p-1;
42 }
43
44 iterator_t list::insert(iterator_t p, elem_t k) {
45     if (size>=MAX_SIZE) {
46         cout << "La lista esta llena.\n";
47         abort();
48     }
49     if (p<0 || p>size) {
50         cout << "Insertando en posicion invalida.\n";
51         abort();
52     }
53     for (int j=size; j>p; j--) elems[j] = elems[j-1];
54     elems[p] = k;
55     size++;
56     return p;
57 }
58
59 iterator_t list::erase(iterator_t p) {
60     if (p<0 || p>=size) {
61         cout << "p: posicion invalida.\n";
62         abort();
63     }
64     for (int j=p; j<size-1; j++) elems[j] = elems[j+1];
65     size--;
66     return p;
67 }
68
69 iterator_t list::erase(iterator_t p, iterator_t q) {
70     if (p<0 || p>=size) {
71         cout << "p: posicion invalida.\n";
72         abort();
73     }
74     if (q<0 || q>size) {
75         cout << "q: posicion invalida.\n";
76         abort();
77     }
78     if (p>q) {
79         cout << "p debe estar antes de q\n";
80         abort();
81 }
```

```

82 if (p==q) return p;
83 int shift = q-p;
84 for (int j=p; j<size-shift; j++)
85     elems[j] = elems[j+shift];
86 size -= shift;
87 return p;
88 }
89
90 void list::clear() { erase(begin(),end()); }
91
92 void list::print() {
93     iterator_t p = begin();
94     while (p!=end()) {
95         cout << retrieve(p) << " ";
96         p = next(p);
97     }
98     cout << endl;
99 }
```

Código 2.7: Implementación de funciones para listas implementadas por arreglos. [Archivo: lista.cpp]

En esta implementación, el header de la clase puede ser como se muestra en el código 2.6. Los detalles de la implementación de los diferentes métodos está en el código 2.7. El tipo `iterator_t` es igual al tipo entero (`int`) y por lo tanto, en vez de declarar una clase, hacemos la equivalencia vía un `typedef`. Los únicos campos datos en la clase `list` son un puntero a enteros `elems` y un entero `size` que mantiene la longitud de la lista. Por simplicidad haremos que el vector subyacente `elems` tenga siempre el mismo tamaño. Esta cantidad está guardada en la variable estática de la clase `MAX_SIZE`. Recordemos que cuando una variable es declarada estática dentro de una clase, podemos pensar que en realidad es una constante dentro de la clase, es decir que no hay una copia de ella en cada instancia de la clase (es decir, en cada objeto). Estos miembros de la clase, que son datos, están en la parte privada, ya que forman parte de la implementación de la clase, y no de la interfaz, de manera que un usuario de la clase no debe tener acceso a ellos.

Los métodos públicos de la clase están en las líneas 9–20. Además de los descriptos en la sección §2.1.3 (código 2.1) hemos agregado el constructor y el destructor y algunos métodos que son variantes de `erase()` como el `erase(p,q)` de un rango (línea 13) y `clear()` que equivale a `erase(begin(),end())`, es decir que borra todos los elementos de la lista. También hemos introducido `prev()` que es similar a `next()` pero retorna el antecesor, no el sucesor y un método básico de impresión `print()`.

En la implementación (código 2.7), vemos que el constructor inicializa las variables `size` y aloca el vector `elems` con `new[]`. El destructor desaloca el espacio utilizado con `delete[]`. Notemos que la inicialización se realiza en la “lista de inicialización” del constructor. Los métodos `retrieve`, `next` y `prev` son triviales, simplemente retornan el elemento correspondiente del vector o incrementan apropiadamente la posición, usando aritmética de enteros. Notar que se verifica primero que la posición sea válida para la operación correspondiente. En `retrieve()`, después de verificar que la posición es válida para insertar (notar que en el test de la línea 49 da error si `p==size`), el elemento es retornado usando la indexación normal de arreglos a través de `[]`. El método `insert()`, después de verificar la validez de la posición (notar que `p==size` no da error en este caso), corre todos los elementos después de `p` en la línea 53, inserta el elemento, e incrementa el contador `size`. `erase()` es similar.

`erase(p, q)` verifica primero la validez del rango a eliminar. Ambas posiciones deben ser válidas, incluyendo `end()` y `p` debe preceder a `q` (también pueden ser iguales, en cuyo caso `erase(p, q)` no hace nada). `shift` es el número de elementos a eliminar, y por lo tanto también el desplazamiento que debe aplicarse a cada elemento posterior a `q`. Notar que uno podría pensar en implementar `erase(p, q)` en forma “genérica”

```

1 iterator_t list::erase(iterator_t p, iterator_t q) {
2     while (p!=q) p = erase(p); // Ooops! q puede no ser válido...
3     return p;
4 }
```

Este código es genérico, ya que en principio sería válido para cualquier implementación de listas que siga la interfaz código 2.1. Sin embargo, hay un error en esta versión: después de ejecutar el primer `erase(p)` la posición `q` deja de ser válida. Además, en esta implementación con arreglos hay razones de eficiencia para no hacerlo en forma genérica (esto se verá en detalle luego). `clear()` simplemente asigna a `size` 0, de esta forma es $O(1)$. La alternativa “genérica” (`erase(begin(), end())`), sería $O(n)$.

Otro ejemplo de código genérico es `purge`. Por supuesto, la mayoría de las funciones sobre listas que no pertenecen a la clase, son en principio genéricas, ya que sólo acceden a la clase a través de la interfaz pública y, por lo tanto, pueden usar cualquier otra implementación. Sin embargo, el término genérico se aplica preferentemente a operaciones bien definidas, de utilidad general, como `purge()` o `sort()` (que ordena los elementos de menor a mayor). Estas funciones, a veces son candidatas a pertenecer a la clase. `print()` es genérica y podríamos copiar su código tal cual e insertarlo en cualquier otra implementación de listas. Pero todavía sería mejor evitar esta duplicación de código, usando la noción de polimorfismo y herencia de clases.

2.1.6.1. Eficiencia de la implementación por arreglos

La implementación de listas por arreglos tiene varias desventajas. Una es la rigidez del almacenamiento. Si en algún momento se insertan más de `MAX_SIZE` elementos, se produce un error y el programa se detiene. Esto puede remediarselfe realocando el arreglo `elems`, copiando los elementos en el nuevo arreglo y liberando el anterior. Sin embargo, estas operaciones pueden tener un impacto en el tiempo de ejecución si la realocación se hace muy seguido.

Pero el principal inconveniente de esta implementación se refiere a los tiempos de ejecución de `insert(p, x)` y `erase(p)`. Ambos requieren un número de instrucciones que es proporcional al número de ejecuciones de los lazos correspondientes, es decir, proporcional al número de elementos que deben moverse. El mejor caso es cuando se inserta un elemento en `end()` o se elimina el último elemento de la lista. En este caso el lazo no se ejecuta ninguna vez. El peor caso es cuando se inserta o elimina en la posición `begin()`. En ese caso ambas operaciones son $O(n)$, donde n es el número de elementos en la lista. El caso promedio, depende de la probabilidad P_j de que al elemento a eliminar esté en la `j`

$$T_{\text{prom}}(n) = \sum_{j=0}^{n-1} P_j T(j) \quad (2.5)$$

Si asumimos que la probabilidad del elemento a eliminar es la misma para todos los elementos ($P_j = 1/n$),

y tenemos en cuenta que $T(j) = n - j - 1$ entonces el tiempo promedio está dado por

$$\begin{aligned}
 T_{\text{prom}}(n) &= \frac{1}{n} \sum_{j=0}^{n-1} n - j - 1 \\
 &= \frac{1}{n} ((n-1) + (n-2) + \cdots + 1 + 0) \\
 &= \frac{1}{n} \frac{(n-1)n}{2} = \frac{n-1}{2} \approx \frac{n}{2} = O(n)
 \end{aligned} \tag{2.6}$$

Como se espera que `insert(p, x)` y `erase(p)` sean dos de las rutinas más usadas sobre listas, es deseable encontrar otras representaciones que disminuyan estos tiempos, idealmente a $O(1)$.

Los tiempos de ejecución de las otras rutinas es $O(1)$ (salvo `erase(p, q)` y `print()`).

2.1.7. Implementación mediante celdas enlazadas por punteros

```

1  class cell;
2  typedef cell *iterator_t;
3
4  class list {
5  private:
6      cell *first, *last;
7  public:
8      list();
9      ~list();
10     iterator_t insert(iterator_t p, elem_t j);
11     iterator_t erase(iterator_t p);
12     iterator_t erase(iterator_t p, iterator_t q);
13     void clear();
14     iterator_t begin();
15     iterator_t end();
16     void print();
17     void printd();
18     iterator_t next(iterator_t p);
19     iterator_t prev(iterator_t p);
20     elem_t & retrieve(iterator_t p);
21     int size();
22 };
23
24 class cell {
25     friend class list;
26     elem_t elem;
27     cell *next;
28     cell() : next(NULL) {}
29 };

```

Código 2.8: Implementación de listas mediante celdas enlazadas por punteros. Declaraciones. [Archivo: `listp.h`]

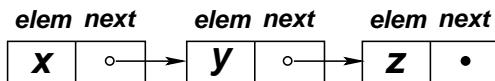


Figura 2.6: Celdas enlazadas por punteros

La implementación mediante celdas enlazadas por punteros es probablemente la más conocida y más usada. Una posible interfaz puede verse en código 2.8. La lista está compuesta de una serie de celdas de tipo **cell** que constan de un campo **elem** de tipo **elem_t** y un campo **next** de tipo **cell ***. Las celdas se van encadenando unas a otras por el campo **next** (ver figura 2.6). Teniendo un puntero a una de las celdas, es fácil seguir la cadena de enlaces y recorrer todas las celdas siguientes. El fin de la lista se detecta manteniendo en el campo **next** de la última celda un puntero nulo (**NULL**). Está garantizado que **NULL** es un puntero inválido (**new** o **malloc()** no retornarán nunca **NULL** a menos que fallen). (En el gráfico representamos al **NULL** por un pequeño círculo lleno.)

Notemos que es imposible recorrer la celda en el sentido contrario. Por ejemplo, si tenemos un puntero a la celda que contiene a **z**, entonces no es posible saber cual es la celda cuyo campo **next** apunta a ella, es decir, la celda que contiene **y**. Las celdas normalmente son alocadas con **new** y liberadas con **delete**, es decir están en el área de almacenamiento dinámico del programa (el “*free store*” o “*heap*”). Esto hace que se deba tener especial cuidado en liberar la memoria alocada, de lo contrario se producen pérdidas de memoria (“*memory leaks*”).

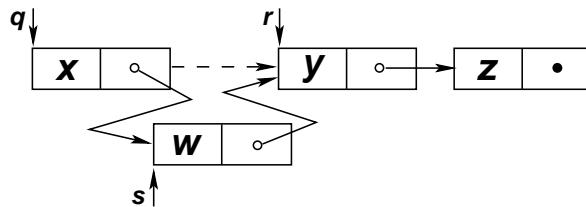


Figura 2.7: Operaciones de enlace necesarias para insertar un nuevo elemento en la lista.

2.1.7.1. El tipo posición

Debemos definir ahora que tipo será, en esta implementación, una posición, es decir el tipo **iterator_t**. La elección natural parece ser **cell ***, ya que si tenemos un puntero a la celda tenemos acceso al contenido. Es decir, parece natural elegir como posición, el puntero a la celda que contiene el dato. Sin embargo, consideremos el proceso de insertar un elemento en la lista (ver figura 2.7). Originalmente tenemos los elementos **L=(...,x,y,z,...)** y queremos insertar un elemento **w** en la posición de **y** es decir **L=(...,x,w,y,z,...)**. Sean **q** y **r** los punteros a las celdas que contienen a **x** e **y**. Las operaciones a realizar son

```

1  s = new cell;
2  s->elem = w;
3  s->next = r;
4  q->next = s;
    
```

Notar que para realizar las operaciones necesitamos el puntero **q** a la celda anterior. Es decir, si definimos como posición el puntero a la celda que contiene el dato, entonces la posición correspondiente a **y** es **r**, y para insertar a **w** en la posición de **y** debemos hacer **L.insert(r,w)**. Pero entonces **insert(...)** no podrá hacer las operaciones indicadas arriba ya que no hay forma de conseguir el puntero a la posición anterior **q**. La solución es *definir como posición el puntero a la celda anterior a la que contiene el dato*. De esta forma, la posición que corresponde a **y** es **q** (antes de insertar **w**) y está claro que podemos realizar las operaciones de enlace. Decimos que las posiciones están “adelantadas” con respecto a los elementos. Notar que después de la inserción la posición de **y** pasa a ser el puntero a la nueva celda **s**, esto ilustra el concepto de que después de insertar un elemento las posiciones posteriores a la de inserción (inclusive) son inválidas.

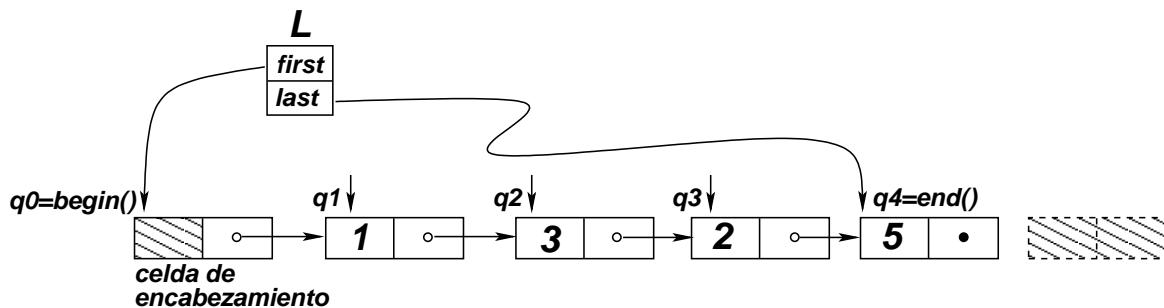


Figura 2.8: Lista enlazada por punteros.

2.1.7.2. Celda de encabezamiento

La lista en sí puede estar representada por un campo **cell *first** que es un puntero a la primera celda. Recordemos que una vez que tenemos un puntero a la primera celda podemos recorrer todas las siguientes. Pero el hecho de introducir un adelanto en las posiciones trae aparejado un problema. ¿Cuál es la posición del primer elemento de la lista? ¿A qué apunta **first** cuando la celda está vacía? Estos problemas se resuelven si introducimos una “celda de encabezamiento”, es decir una celda que no contiene dato y tal que **first** apunta a ella. Entonces, por ejemplo, si nuestra lista contiene a los elementos **L=(1, 3, 2, 5)**, la representación por punteros sería como se muestra en la figura 2.8. Si **q0-q4** son punteros a las 5 celdas de la lista (incluyendo la de encabezamiento), entonces el elemento 1 está en la posición **q0** de manera que, por ejemplo

- **L.retrieve(q0)** retornará 1.
- **L.retrieve(q1)** retornará 3.
- **L.retrieve(q2)** retornará 2.
- **L.retrieve(q3)** retornará 5.
- **L.retrieve(q4)** dará error ya que corresponde a la posición de la celda ficticia (representada en línea de trazos en la figura).

```

1 list::list() : first(new cell), last(first) {
2   first->next = NULL;
3 }
```

```

4
5 list::~list() { clear(); delete first; }
6
7 elem_t &list::retrieve(iterator_t p) {
8     return p->next->elem;
9 }
10
11 iterator_t list::next(iterator_t p) {
12     return p->next;
13 }
14
15 iterator_t list::prev(iterator_t p) {
16     iterator_t q = first;
17     while (q->next != p) q = q->next;
18     return q;
19 }
20
21 iterator_t
22 list::insert(iterator_t p, elem_t k) {
23     iterator_t q = p->next;
24     iterator_t c = new cell;
25     p->next = c;
26     c->next = q;
27     c->elem = k;
28     if (q==NULL) last = c;
29     return p;
30 }
31
32 iterator_t list::begin() { return first; }
33
34 iterator_t list::end() { return last; }
35
36 iterator_t list::erase(iterator_t p) {
37     if (p->next==last) last = p;
38     iterator_t q = p->next;
39     p->next = q->next;
40     delete q;
41     return p;
42 }
43
44 iterator_t list::erase(iterator_t p, iterator_t q) {
45     if (p==q) return p;
46     iterator_t s, r = p->next;
47     p->next = q->next;
48     if (!p->next) last = p;
49     while (r!=q->next) {
50         s = r->next;
51         delete r;
52         r = s;
53     }
54     return p;
55 }
56

```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```

57 void list::clear() { erase(begin(),end()); }
58
59 void list::print() {
60     iterator_t p = begin();
61     while (p!=end()) {
62         cout << retrieve(p) << " ";
63         p = next(p);
64     }
65     cout << endl;
66 }
67
68 void list::printfd() {
69     cout << "h(" << first << ")" << endl;
70     iterator_t c = first->next;
71     int j=0;
72     while (c!=NULL) {
73         cout << j++ << "(" << c << ") :" << c->elem << endl;
74         c = c->next;
75     }
76 }
77
78 int list::size() {
79     int sz = 0;
80     iterator_t p = begin();
81     while (p!=end()) {
82         sz++;
83         p = next(p);
84     }
85     return sz;
86 }
```

Código 2.9: Implementación de listas mediante celdas enlazadas por punteros. Implementación de los métodos de la clase. [Archivo: listp.cpp]

2.1.7.3. Las posiciones begin() y end()

begin() es la posición correspondiente al primer elemento, por lo tanto un puntero a la celda anterior, es decir la celda de encabezamiento. Por otra parte, **end()** es una posición ficticia *después* del último elemento (marcada con línea de trazos en la figura). Su posición es un puntero a la celda anterior, es decir la celda que contiene el último elemento (**q4** en la figura).

Notar que **begin()** no cambia nunca durante la vida de la lista ya que inserciones o supresiones, incluso en el comienzo de la lista no modifican la celda de encabezamiento. Por otra parte **end()** sí cambia cuando hay una inserción o supresión al final de la lista. Esto significa que al momento de implementar **end()** debemos comenzar desde **begin()** y recorrer toda los enlaces hasta llegar a la última celda. Por ejemplo:

```

1 iterator_t list::end() {
2     cell *q = first;
3     while (q->next) q = q->next;
```

```
4     return q;  
5 }
```

Pero el tiempo de ejecución de esta operación es $O(n)$, y **end()** es usada frecuentemente en los lazos para detectar el fin de la lista (ver por ejemplo los códigos 2.4 y 2.5), con lo cual *debe* tener costo $O(1)$.

La solución es mantener en la declaración de la lista un puntero a la última celda, actualizándolo convenientemente cuando es cambiado (esto sólo puede ocurrir en **insert()** y **erase()**).

2.1.7.4. Detalles de implementación

Figura 2.9: Una lista vacía.

- El constructor aloca la celda de encabezamiento y asigna su puntero a **first**. Inicialmente la celda está vacía y por lo tanto **last=first** (**begin()=end()**). También debe inicializar el terminador (de la única celda) a **NULL**.
- El destructor llama a **clear()** (que está implementada en términos de **erase(p,q)**, la veremos después) y finalmente libera la celda de encabezamiento.
- **retrieve(p)** retorna el elemento en el campo **elem**, teniendo en cuenta previamente el adelanto en las posiciones.
- **next()** simplemente avanza un posición usando el campo **next**. Notar que no hay colisión entre la función **next()** y el campo **next** ya que *pertenecen a clases diferentes* (el campo **next** pertenece a la clase **cell**).
- **begin()** y **end()** retornan los campos **first** y **last**, respectivamente.
- **insert()** y **erase()** realizan los enlaces ya mencionados en §2.1.7.1 (ver figura 2.10) y actualizan, de ser necesario, **last**.
- **erase(p,q)** es implementado haciendo una operación de enlace de punteros descripta en la figura 2.12. Luego es necesario liberar todas las celdas que están en el rango a eliminar. Recordar que **erase(p,q)** debe eliminar las celdas desde **p** hasta **q**, excluyendo a **q**. Como **w** está en la posición **p** y **z** en la posición **q**, esto quiere decir que hay que eliminar las celdas que contienen a los elementos **w** a **y**.

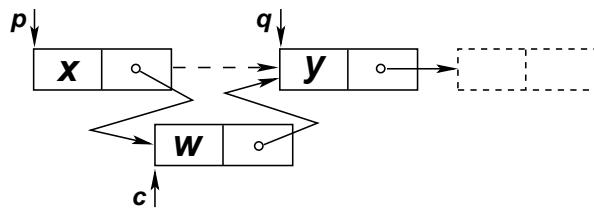


Figura 2.10: Operaciones de punteros para insert

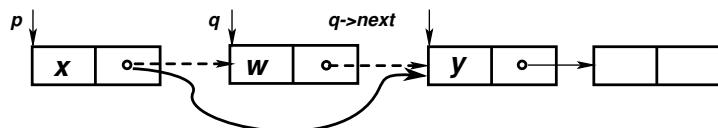


Figura 2.11: Operaciones de punteros para erase

- **prev()** es implementada mediante un lazo, se recorre la lista desde el comienzo hasta encontrar la celda anterior a la de la posición **p**. Esto es $O(n)$ y por lo tanto debe ser *evitada en lo posible*. (Si **prev()** debe ser usada frecuentemente, entonces puede considerarse en usar una lista *dblemente enlazada*).

2.1.8. Implementación mediante celdas enlazadas por cursores

En la implementación de celdas enlazadas por punteros, los datos son guardados en celdas que son alojadas dinámicamente en el área de almacenamiento dinámico del programa. Una implementación similar consiste en usar celdas enlazadas por cursores, es decir celdas indexadas por punteros dentro de un gran arreglo de celdas. Este arreglo de celdas puede ser una variable global o un objeto estático de la clase, de manera que muchas listas pueden convivir en el mismo espacio de celdas. Puede verse que esta implementación es equivalente a la de punteros (más adelante daremos una tabla que permite “*traducir*” las operaciones con punteros a operaciones con celdas y viceversa) y tiene ventajas y desventajas con respecto a aquella. Entre las ventajas tenemos que,

- La gestión de celdas puede llegar a ser más eficiente que la del sistema (en tiempo y memoria).
- Cuando se alojan dinámicamente con **new** y **delete** muchos objetos pequeños (como las celdas) el área de la memoria ocupada queda muy fragmentada. Esto impide la alocación de objetos grandes, incluso después de eliminar gran parte de estos objetos pequeños. Usando cursores, todas las celdas

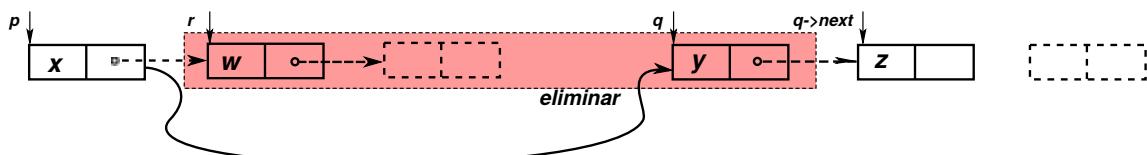


Figura 2.12: Operaciones de punteros para erase(**p**, **q**)

viven en un gran espacio de celdas, de manera que no se mezclan con el resto de los objetos del programa.

- En lenguajes donde no existe la alocación dinámica de memoria, el uso de cursores reemplaza al de punteros.

Esto puede ser de interés sobre todo para el manejo de grandes cantidades de celdas relativamente pequeñas. Además, el uso de cursores es interesante en sí mismo, independientemente de las ventajas o desventajas, ya que permite entender mejor el funcionamiento de los punteros.

Entre las desventajas que tienen los cursores podemos citar que,

- Hay que reservar de entrada un gran espacio de celdas. Si este espacio es pequeño, corremos riesgo de que el espacio se llene y el programa aborte por la imposibilidad de alocar nuevas celdas dentro del espacio. Si es muy grande, estaremos alocando memoria que en la práctica no será usada. (Esto se puede resolver parcialmente, realocando el espacio de celdas, de manera que pueda crecer o reducirse.)
- Listas de elementos del mismo tipo comparten el mismo espacio de celdas, pero si son de diferentes tipos se debe generar un espacio celdas por cada tipo. Esto puede agravar más aún las desventajas mencionadas en el punto anterior.

```
1 class list;
2     typedef int iterator_t;
3
4 class cell {
5     friend class list;
6     elem_t elem;
7     iterator_t next;
8     cell();
9 };
10
11 class list {
12     private:
13     friend class cell;
14     static iterator_t NULL_CELL;
15     static int CELL_SPACE_SIZE;
16     static cell *cell_space;
17     static iterator_t top_free_cell;
18     iterator_t new_cell();
19     void delete_cell(iterator_t c);
20     iterator_t first, last;
21     void cell_space_init();
```

Código 2.10: Implementación de listas por cursores. Declaraciones. [Archivo: listc.h]

Un posible juego de declaraciones puede observarse en el código 2.10. Las celdas son como en el caso de los punteros, pero ahora el campo `next` es de tipo entero, así como las posiciones (`iterator_t`). Las

celdas viven en el arreglo **cell_space**, que es un arreglo estándar de elementos de tipo **cell**. Este arreglo podría declararse global (es decir fuera de la clase), pero es más prolífico incluirlo en la clase. Sin embargo, para evitar que cada lista tenga su espacio de celdas, lo declaramos **static**, de esta forma actúa como si fuera global, pero dentro de la clase. También declaramos **static** el tamaño del arreglo **CELL_SPACE_SIZE**. Así como con punteros existe el puntero inválido **NULL**, declaramos un cursor inválido **NULL_CELL**. Como nuestras celdas están indexadas dentro de un arreglo estándar de C, los índices pueden ir entre 0 y **CELL_SPACE_SIZE-1**, de manera que podemos elegir **NULL_CELL** como -1.

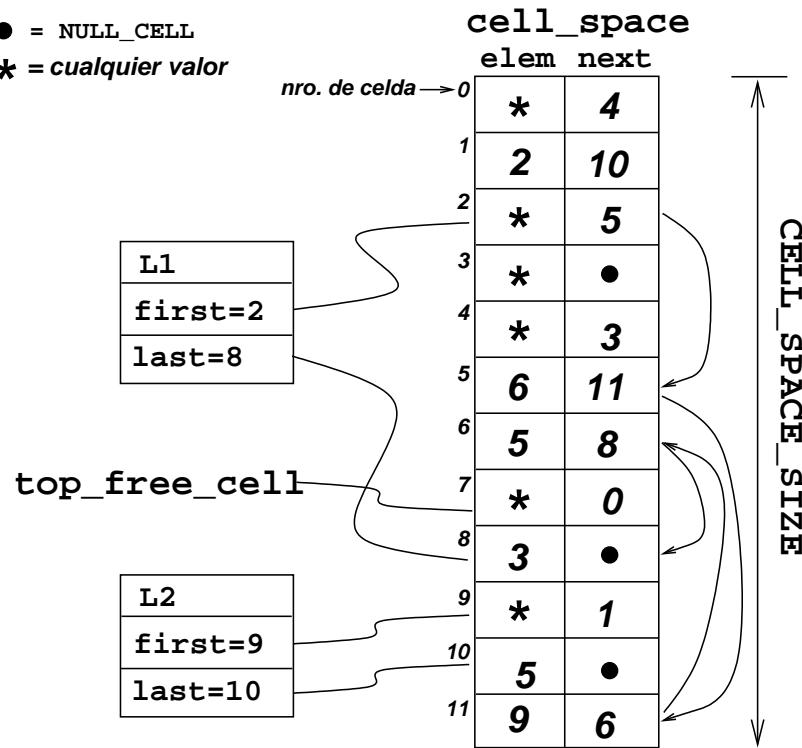


Figura 2.13: Lista enlazada por cursos.

2.1.8.1. Cómo conviven varias celdas en un mismo espacio

Las listas consistirán entonces en una serie de celdas dentro del arreglo, con una celda de encabezamiento y terminadas por una celda cuyo campo **next** posee el cursor inválido **NULL_CELL**. Por ejemplo, en la figura 2.13 vemos una situación típica, el espacio de celdas **cell_space** tiene **CELL_SPACE_SIZE=12** celdas. En ese espacio conviven 2 listas **L1=(6,9,5,3)** y **L2=(2,5)**. La lista **L1** ocupa 5 celdas incluyendo la de encabezamiento que en este caso es la celda 2. Como el dato en las celdas de encabezamiento es irrelevante ponemos un “*”. El campo **next** de la celda de encabezamiento apunta a la primera celda que en este caso es la 5. La celda 5 contiene el primer elemento (que es un 6) en el campo **elem** y el campo **next** apunta a la siguiente celda (que es la 11). Los enlaces para las celdas de la lista **L1** se muestran con flechas a la derecha del arreglo. La última celda (la 8) contiene en el campo **next** el cursor inválido **NULL_CELL**, representado en el

dibujo por un pequeño círculo negro. La lista L2 contiene 3 celdas, incluyendo la de encabezamiento, a saber las celdas 9, 1 y 10. Las 4 celdas restantes (7,0,4 y 3) están libres.

2.1.8.2. Gestión de celdas

Debemos generar un sistema de gestión de celdas, similar a como los operadores **new** y **delete** operan sobre el *heap*. Primero debemos mantener una lista de cuales celdas están alojadas y cuales no. Para esto construimos una lista de celdas libres enlazadas mediante el campo **next** ya que para las celdas libres este campo no es utilizado de todas formas. El cursor **top_free_cell** (que también es estático) apunta a la primera celda libre. El campo **next** de la *i*-ésima celda libre apunta a la *i* + 1-ésima celda libre, mientras que la última celda libre contiene un cursor inválido **NULL_CELL** en el campo **next**. Esta disposición es completamente equivalente a la de las listas normales, sólo que no hace falta un cursor a la última celda ni necesita el concepto de posición, sino que sólo se accede a través de uno de los extremos de la lista, apuntando por **top_free_cell**. (En realidad se trata de una “*pila*”, pero esto lo veremos en una sección posterior).

Las rutinas **c=new_cell()** y **delete_cell(c)** definen una interfaz abstracta (dentro de la clase **list**) para la gestión de celdas, la primera devuelve una nueva celda libre y la segunda libera una celda utilizada, en forma equivalente a los operadores **new** y **delete** para las celdas enlazadas por punteros.

Antes de hacer cualquier operación sobre una lista, debemos asegurarnos que el espacio de celdas esté correctamente inicializado. Esto se hace dentro de la función **cell_space_init()**, la cual aloca el espacio celdas e inserta todas las celdas en la lista de celdas libres. Esto se realiza en el lazo de las líneas 16–17 en el cual se enlaza la celda **i** con la **i+1**, mientras que en la última celda **CELL_SPACE_SIZE-1** se inserta el cursor inválido.

Para que **cell_space_init()** sea llamado automáticamente antes de cualquier operación con las listas, lo incluimos en el constructor de la clase lista (línea 9). Podemos verificar si el espacio ya fue inicializado por el valor del puntero **cell_space** ya que si el espacio no fue inicializado entonces este puntero es nulo (ver línea 3). El **if** de la línea 9 hace esta verificación.

2.1.8.3. Analogía entre punteros y cursores

Hemos mencionado que hay un gran parecido entre la implementación con punteros y cursores. De hecho casi todos los métodos de la clase se pueden implementar para cursores simplemente aplicando a los métodos de la implementación por punteros (código 2.9) las transformaciones listadas en la tabla 2.1). En la tabla se usa el nombre genérico de “*direcciones*” a los punteros o cursores.

```
1 cell::cell() : next(list::NULL_CELL) {}
2
3 cell *list::cell_space = NULL;
4 int list::CELL_SPACE_SIZE = 100;
5 iterator_t list::NULL_CELL = -1;
6 iterator_t list::top_free_cell = list::NULL_CELL;
7
8 list::list() {
9     if (!cell_space) cell_space_init();
10    first = last = new_cell();
11    cell_space[first].next = NULL_CELL;
12 }
```

	Punteros	Curosres
Área de almacenamiento	<i>heap</i>	<i>cell_space</i>
Tipo usado para las direcciones de las celdas (<i>iterator_t</i>)	<i>cell* c</i>	<i>int c</i>
Dereferenciación de direcciones (dirección → celda)	<i>*c</i>	<i>cell_space[c]</i>
Dato de una celda dada su dirección <i>c</i>	<i>c->elem</i>	<i>cell_space[c].elem</i>
Enlace de una celda (campo <i>next</i>) dada su dirección <i>c</i>	<i>c->next</i>	<i>cell_space[c].next</i>
Alocar una celda	<i>c = new cell;</i>	<i>c = new_cell();</i>
Liberar una celda	<i>delete cell;</i>	<i>delete_cell(c);</i>
Dirección inválida	<i>NULL</i>	<i>NULL_CELL</i>

Tabla 2.1: Tabla de equivalencia entre punteros y cursores.

```

13
14 void list::cell_space_init() {
15   cell_space = new cell[CELL_SPACE_SIZE];
16   for (int j=0; j<CELL_SPACE_SIZE-1; j++)
17     cell_space[j].next = j+1;
18   cell_space[CELL_SPACE_SIZE-1].next = NULL_CELL;
19   top_free_cell = 0;
20 }
21
22 iterator_t list::new_cell() {
23   iterator_t top = top_free_cell;
24   if (top==NULL_CELL) {
25     cout << "No hay mas celdas \n";
26     abort();
27   }
28   top_free_cell = cell_space[top_free_cell].next;
29   return top;
30 }
31
32 void list::delete_cell(iterator_t c) {
33   cell_space[c].next = top_free_cell;
34   top_free_cell = c;
35 }
36
37 list::~list() { clear(); }
38
39 elem_t &list::retrieve(iterator_t p) {
40   iterator_t q= cell_space[p].next;
41   return cell_space[q].elem;
42 }
43
44 iterator_t list::next(iterator_t p) {
45   return cell_space[p].next;

```

```

46 }
47
48 iterator_t list::prev(iterator_t p) {
49   iterator_t q = first;
50   while (cell_space[q].next != p)
51     q = cell_space[q].next;
52   return q;
53 }
54
55 iterator_t list::insert(iterator_t p, elem_t k) {
56   iterator_t q = cell_space[p].next;
57   iterator_t c = new_cell();
58   cell_space[p].next = c;
59   cell_space[c].next = q;
60   cell_space[c].elem = k;
61   if (q==NULL_CELL) last = c;
62   return p;
63 }
64
65 iterator_t list::begin() { return first; }
66
67 iterator_t list::end() { return last; }
68
69 iterator_t list::erase(iterator_t p) {
70   if (cell_space[p].next == last) last = p;
71   iterator_t q = cell_space[p].next;
72   cell_space[p].next = cell_space[q].next;
73   delete_cell(q);
74   return p;
75 }
76
77 iterator_t list::erase(iterator_t p, iterator_t q) {
78   if (p==q) return p;
79   iterator_t s, r = cell_space[p].next;
80   cell_space[p].next = cell_space[q].next;
81   if (cell_space[p].next == NULL_CELL) last = p;
82   while (r!=cell_space[q].next) {
83     s = cell_space[r].next;
84     delete_cell(r);
85     r = s;
86   }
87   return p;
88 }
89
90 void list::clear() { erase(begin(),end()); }
91
92 void list::print() {
93   iterator_t p = begin();
94   while (p!=end()) {
95     cout << retrieve(p) << " ";
96     p = next(p);
97   }
98   cout << endl;

```

```

99 }
100
101 void list::printd() {
102     cout << "h(" << first << ")" << endl;
103     iterator_t c = cell_space[first].next;
104     int j=0;
105     while (c!=NULL_CELL) {
106         cout << j++ << "(" << c << ") :" << cell_space[c].elem << endl;
107         c = next(c);
108     }
109 }
```

Código 2.11: Implementación de listas por cursores. Implementación de los métodos. [Archivo: listc.cpp]

Por ejemplo, el método `next()` en la implementación por punteros simplemente retorna `p->next`. Según la tabla, esto se traduce en retornar `cell_space[c].next` que es lo que precisamente hace el método correspondiente en la implementación por cursores.

2.1.9. Tiempos de ejecución de los métodos en las diferentes implementaciones.

Método	Arreglos	Punteros / cursores
<code>insert(p,x)</code> , <code>erase(p)</code>	$O(n)$ [$T = c(n - j)$]	$O(1)$
<code>erase(p,q)</code>	$O(n)$ [$T = c(n - k)$]	$O(n)$ [$T = c(k - j)$]
<code>clear()</code>	$O(1)$	$O(n)$
<code>begin()</code> , <code>end()</code> , <code>next()</code> , <code>retrieve()</code>	$O(1)$	$O(1)$
<code>prev(p)</code>	$O(1)$	$O(n)$ [$T = cj$]

Tabla 2.2: Tiempos de ejecución de los métodos del TAD lista en las diferentes implementaciones. j, k son las posiciones enteras correspondientes a p, q

Consideremos ahora los tiempos de ejecución de las operaciones del TAD lista en sus diferentes implementaciones. Los tiempos de implementación de la implementación por punteros y cursores son los mismos, ya que sólo difieren en cómo acceden a las celdas, pero de todas formas las operaciones involucradas son $O(1)$, en ambos casos, de manera que la comparación es entre punteros/cursos y arreglos. Las operaciones `begin()`, `end()`, `next()`, `retrieve()` son $O(1)$ en ambos casos. La diferencia más importante es, como ya hemos mencionado, en las operaciones `insert(p,x)` y `erase(p)`. En la implementación por arreglos se debe mover todos los elementos que están después de la posición p (los lazos de las líneas 64 y 53, código 2.7), o sea que es $O(n - j)$ donde j es la posición (como número entero) de la posición abstracta p , mientras que para punteros/cursos es $O(1)$. `erase(p,q)` debe hacer el `delete` (o `delete_cell()`) de todas las celdas en el rango `[p,q]` de manera que requiere $O(k - j)$ operaciones, donde k, j son las posiciones enteras correspondientes a p, q . Por otra parte, en la implementación por arreglos sólo debe moverse los elementos en el rango `[q,end()]` (esto es $n - k$ elementos) $k - j$ posiciones hacia el comienzo. Pero el mover cada elemento en un arreglo es tiempo constante, independientemente de cuantas posiciones se mueve, de manera que la operación es $O(n - k)$. En el límite, la función `clear()` es $O(n)$ para punteros/cursos y

$O(1)$ para arreglos. Por otra parte, la función `prev(p)` es $O(j)$ para punteros/cursores, ya que involucra ir al comienzo de la lista y recorrer todas las celdas hasta encontrar la `p` (los lazos de las líneas 17 en el código 2.9 y la línea 50 en el código 2.11). Esto involucra un tiempo $O(j)$, mientras que en el caso de la implementación por arreglos debe retornar `p-1` ya que las posiciones son enteras, y por lo tanto es $O(1)$. Los tiempos de ejecución están listados en la Tabla 2.2. (Nota: Para `insert` por arreglos se indica en la tabla $O(n)$, que corresponde al peor caso que es cuando `p` está al principio de la lista y entre corchetes se indica $T = c(n - j)$ que es el número de operaciones dependiendo de j . La constante c es el tiempo promedio para hacer una de las operaciones. Lo mismo ocurre para otras funciones.)

Comparando globalmente las implementaciones, vemos que la implementación por arreglos es más competitiva en `prev()`, `clear()`. Pero `prev()` decididamente es una operación para la cual no están diseñadas las listas simplemente enlazadas y normalmente `clear()` debería ser menos usada que `insert()` o `erase()`, por lo cual raramente se usan arreglos para representar listas. Por otra parte la diferencia para `erase(p, q)` puede ser a favor de punteros/cursores (cuando se borran pequeños intervalos en la mitad de la lista) o a favor de los arreglos (cuando se borran grandes regiones cerca del final).

2.1.10. Interfaz STL

2.1.10.1. Ventajas de la interfaz STL

Uno de los principales inconvenientes de la interfaz definida hasta ahora (ver código 2.1) es que asocia el tipo lista a una lista de un tipo de elemento dado. Es decir, normalmente un juego de declaraciones como el citado debe ser precedido de una serie de asignaciones de tipo, como por ejemplo

```
1 typedef elem_t int;
```

si se quieren manipular listas de enteros. Por otra parte, si se desean manipular listas de dos tipos diferentes, por ejemplo enteros y dobles, entonces se debe duplicar el código (las declaraciones y las implementaciones) definiendo un juego de tipos para cada tipo de dato, por ejemplo `list_int` y `iterator_int_t` para enteros y `list_double` y `iterator_double_t`. Pero esto, por supuesto, no es deseable ya que lleva a una duplicación de código completamente innecesaria.

La forma correcta de evitar esto en C++ es mediante el uso de “*templates*”. Los templates permiten definir clases o funciones parametrizadas por un tipo (u otros objetos también). Por ejemplo, podemos definir la clase `list<class T>` de manera que luego el usuario puede declarar simplemente

```
1 list<int> lista_1;
2 list<double> lista_2;
```

Esta es la forma en que los diferentes contenedores están declarados en las STL.

En esta sección veremos como generalizar nuestras clases con el uso de templates, es más modificaremos nuestra interfaz de manera que sea totalmente “*compatible con las STL*”, es decir que si un código funciona llamando a nuestros contenedores, también funciona con el contenedor correspondiente de las STL. Una diferencia puramente sintáctica con respecto a las STL es el uso de la sobrecarga de operadores para las funciones `next()` y `prev()`. Notemos que, si las posiciones fueran enteros, como en el caso de los arreglos, entonces podríamos hacer los reemplazos

```
p = next(p); → p++
p = prev(p); → p--
```

Las STL extienden el alcance de los operadores `++` y `--` para los objetos de tipo posición usando sobre-carga de operadores y lo mismo haremos en nuestra implementación. También el operador `*p` que permite dereferenciar punteros será usado para recuperar el dato asociado con la posición, como lo hace la función `retrieve(p)` en la interfaz básica código 2.1.

Finalmente, otra diferencia sintáctica entre las STL y nuestra versión básica es la clase de posiciones. Si usamos templates, deberemos tener un template separado para la clase de iteradores, por ejemplo `iterator<int>`. Pero cuando tengamos otros contenedores como conjuntos (`set`) y correspondencias (`map`), cada uno tendrá su clase de posiciones y para evitar la colisión, podemos agregarle el tipo de contenedor al nombre de la clase, por ejemplo, `list_iterator<int>`, `set_iterator<int>` o `map_iterator<int>`. Esto puede evitarse haciendo que la clase `iterator` sea una “clase anidada” (“nested class”) de su contenedor, es decir que la declaración de la clase `iterator` está dentro de la clase del contenedor correspondiente. De esta manera, el contenedor actúa como un `namespace` para la clase del contenedor y así pasa a llamarse `list<int>::iterator`, `set<int>::iterator`...

2.1.10.2. Ejemplo de uso

```
1 bool check_sum(list<int> &L1, list<int> &L2) {
2     list<int>::iterator p,q;
3     p = L1.begin();
4     q = L2.begin();
5     int suma = 0;
6     while (true) {
7         if (q==L2.end()) break;
8         else if (suma==*q) { suma=0; q++; }
9         else if (p==L1.end()) break;
10        else if (suma<*q) suma += *p++;
11        else return false;
12    }
13    return suma==0 && p==L1.end() && q==L2.end();
14 }
```

Código 2.12: El procedimiento `check-sum`, con la sintaxis de la librería STL. [Archivo: `check-sum-stl.cpp`]

Con la nueva sintaxis (idéntica a la de los contenedores de STL) el ejemplo del procedimiento `bool check-sum(L1,L2)` descrito en la sección § 2.1.5 puede escribirse como en el código 2.12.

2.1.10.2.1. Uso de templates y clases anidadas El uso de templates permite definir contenedores en base al tipo de elemento en forma genérica. En el ejemplo usamos listas de enteros mediante la expresión `list<int>`. Como las posiciones están declaradas *dentro* de la clase del contenedor deben declararse con el scope (“alcance”) correspondiente, es decir `list<int>::iterator`.

2.1.10.2.2. Operadores de incremento prefijo y postfijo: Recordemos que los operadores de incremento “prefijo” (`++p`) y “postfijo” (`p++`) tienen el mismo “efecto colateral” (“side effect”) que es incrementar la

variable **p** pero tienen diferente “*valor de retorno*” a saber el valor incrementado en el caso del postfijo y el valor no incrementado para el prefijo. Es decir

- **q = p++;** es equivalente a **q = p; p = p.next();**, mientras que
- **q = ++p;** es equivalente a **p = p.next(); q = p;** .

Por ejemplo en la línea 10, incrementamos la variable **suma** con el elemento de la posición **p** *antes* de incrementar **p**.

2.1.10.3. Detalles de implementación

```

1 #ifndef AED_LIST_H
2 #define AED_LIST_H
3
4 #include <cstddef>
5 #include <iostream>
6
7 namespace aed {
8
9     template<class T>
10    class list {
11    public:
12        class iterator;
13    private:
14        class cell {
15            friend class list;
16            friend class iterator;
17            T t;
18            cell *next;
19            cell() : next(NULL) {}
20        };
21        cell *first, *last;
22    public:
23        class iterator {
24    private:
25        friend class list;
26        cell* ptr;
27    public:
28        T & operator*() { return ptr->next->t; }
29        T *operator->() { return &ptr->next->t; }
30        bool operator!=(iterator q) { return ptr!=q.ptr; }
31        bool operator==(iterator q) { return ptr==q.ptr; }
32        iterator(cell *p=NULL) : ptr(p) {}
33        // Prefix:
34        iterator operator++() {
35            ptr = ptr->next;
36            return *this;
37        }
38        // Postfix:
39        iterator operator++(int) {

```

```

40     iterator q = *this;
41     ptr = ptr->next;
42     return q;
43 }
44 };
45
46 list() {
47     first = new cell;
48     last = first;
49 }
50 ~list() { clear(); delete first; }
51 iterator insert(iterator p,T t) {
52     cell *q = p.ptr->next;
53     cell *c = new cell;
54     p.ptr->next = c;
55     c->next = q;
56     c->t = t;
57     if (q==NULL) last = c;
58     return p;
59 }
60 iterator erase(iterator p) {
61     cell *q = p.ptr->next;
62     if (q==last) last = p.ptr;
63     p.ptr->next = q->next;
64     delete q;
65     return p;
66 }
67 iterator erase(iterator p,iterator q) {
68     cell *s, *r = p.ptr->next;
69     p.ptr->next = q.ptr->next;
70     if (!p.ptr->next) last = p.ptr;
71     while (r!=q.ptr->next) {
72         s = r->next;
73         delete r;
74         r = s;
75     }
76     return p;
77 }
78 void clear() { erase(begin(),end()); }
79 iterator begin() { return iterator(first); }
80 iterator end() { return iterator(last); }
81 void print() {
82     iterator p = begin();
83     while (p!=end()) std::cout << *p++ << " ";
84     std::cout << std::endl;
85 }
86 void printd() {
87     std::cout << "h(" << first << ")" << std::endl;
88     cell *c = first->next;
89     int j=0;
90     while (c!=NULL) {
91         std::cout << j++ << "(" << c << " ) :" << c->t << std::endl;
92         c = c->next;
93     }

```

```

94     }
95     int size() {
96         int sz = 0;
97         iterator p = begin();
98         while (p++!=end()) sz++;
99         return sz;
100    }
101   };
102 }
103 }
104 #endif

```

Código 2.13: Implementación de listas por punteros con sintaxis compatible STL. Declaraciones. [Archivo: *list.h*]

En el código 2.13 podemos ver un posible juego de declaraciones para las listas implementadas por punteros con sintaxis compatible STL. Como es la clase que usaremos en los ejemplos hemos incluido todos los detalles de implementación (en los ejemplos anteriores hemos omitido algunos detalles para facilitar la lectura).

- El encabezado **#ifndef AED_LIST_H...** es para evitar la doble inclusión de los headers.
- Hemos incluido un **namespace aed** para evitar la colisión con otras clases con nombres similares que pudieran provenir de otros paquetes. Por lo tanto las clases deben ser en realidad referenciadas como **aed::list<int>** y **aed::list<int>::iterator**. Otra posibilidad es incluir una declaración **using namespace aed;**
- La clase **list** va precedida del calificador **template<class T>** que indica que la clase **T** es un tipo genérico a ser definido en el momento de *instanciar* la clase. Por supuesto, también puede ser un tipo básico como **int** o **double**. Por ejemplo, al declarar **list<int>** el tipo genérico **T** pasa a tomar el valor concreto **int**.
- La clase **cell<T>** ha sido incluida también como clase anidada dentro de la clase **list<T>**. Esto permite (al igual que con **iterator**) tener una clase **cell** para cada tipo de contenedor (lista, pila, cola...) sin necesidad de agregarle un prefijo o sufijo (como en **list_cell**, **stack_cell**, etc...).
- La clase **cell<T>** declara **friend** a las clases **list<T>** e **iterator<T>**. Recordemos que el hecho de declarar la clase en forma anidada dentro de otra no tiene ninguna implicancia en cuanto a la privacidad de sus miembros. Si queremos que **cell<T>** acceda a los miembros privados de **list<T>** y **iterator<T>** entonces debemos declarar a las clases como **friend**.
- Las clases **list<T>**, **cell<T>** e **iterator<T>** son un ejemplo de “*clases fuertemente ligadas*” (“*tightly coupled classes*”), esto es una serie de grupo de clases que están conceptualmente asociadas y probablemente son escritas por el mismo programador. En tal caso es común levantar todas las restricciones entre estas clases con declaraciones **friend**. En este caso sólo es necesario que **cell** declare friend a **iterator** y **list**, y que **iterator** declare friend a **list**.

- La clase **cell<T>** es declarada privada dentro de **list<T>** ya que normalmente no debe ser accedida por el usuario de la clase, el cual sólo accede los valores a través de **iterator<T>**.
- Para poder sobrecargar los operadores de incremento y dereferenciación (**p++**, **++p** y ***p**) debemos declarar a **iterator<T>** como una clase y no como un **typedef** como en la interfaz básica. Esta clase contiene como único miembro un puntero a celda **cell *ptr**. Como ya no es un **typedef** también debemos declarar los operadores de comparación **p!=q** y **p==q**. (Los operadores **<**, **>**, **<=**, y **>=** también podrían ser definidos, pero probablemente serían $O(n)$). Por supuesto estos operadores comparan simplemente los punteros a las celdas correspondientes. Otro inconveniente es que ahora hay que extraer el campo **ptr** cada vez que se hace operaciones sobre los enlaces, por ejemplo la primera línea de **next()**

```
1 iterator_t q = p->next;
```

se convierte en

```
1 cell *q = p.ptr->next;
```

2.1.10.4. Listas doblemente enlazadas

Si es necesario realizar repetidamente la operación **q=L.prev(p)** que retorna la posición **q** anterior a **p** en la lista **L**, entonces probablemente convenga utilizar una “*lista doblemente enlazada*”. En este tipo de listas cada celda tiene dos punteros uno al elemento siguiente y otro al anterior.

```
1 class cell {
2     elem_t elem;
3     cell *next, *prev;
4     cell() : next(NULL), prev(NULL) {}
5 };
```

Una ventaja adicional es que en este tipo de implementación la posición puede implementarse como un “*puntero a la celda que contiene el elemento*” y no a la celda precedente, como en las listas simplemente enlazadas (ver §2.1.7.1). Además las operaciones sobre la lista pasan a ser completamente simétricas en cuanto al principio y al fin de la lista.

Notemos también que en este caso al eliminar un elemento en la posición **p**, ésta deja de ser válida efectivamente, ya que la celda a la que apunta desaparece. En la versión simplemente enlazada, en cambio, la celda a la que apunta la posición sigue existiendo, con lo cual la posición en principio sigue siendo válida. Es decir, por ejemplo en el código código 2.4, la línea 8 puede ser reemplazada por **L.erase(q)** sin actualizar **q**. Sin embargo, por una cuestión de uniformidad conviene mantener la convención de reasignar siempre la posición, de manera que el código siga valiendo tanto para listas simple como doblemente enlazadas.

2.2. El TAD pila

Básicamente es una lista en la cual todas las operaciones de inserción y borrado se producen en uno de los extremos de la lista. Un ejemplo gráfico es una pila de libros en un cajón. A medida que vamos

recibiendo más libros los ubicamos en la parte superior. En todo momento tenemos acceso sólo al libro que se encuentra sobre el “*tope*” de la pila. Si queremos acceder a algún libro que se encuentra más abajo (digamos en la quinta posición desde el tope) debemos sacar los primeros cuatro libros y ponerlos en algún lugar para poder acceder al mismo. La Pila es el típico ejemplo de la estructura tipo “*LIFO*” (por “*Last In First Out*”, es decir “*el último en entrar es el primero en salir*”).

La pila es un subtipo de la lista, es decir podemos definir todas las operaciones abstractas sobre pila en función de las operaciones sobre lista. Esto motiva la idea de usar “*adaptadores*” es decir capas de código (en la forma de templates de C++) para adaptar cualquiera de las posibles clases de lista (por arreglo, punteros o cursores) a una pila.

Ejemplo 2.3: *Consigna:* Escribir un programa para calcular expresiones aritméticas complejas con números en doble precisión usando “*notación polaca invertida*” (RPN, por “*reverse polish notation*”).

Solución: En las calculadoras con RPN una operación como $2+3$ se introduce en la forma $2\ 3\ +$. Esto lo denotamos así

$$\text{rpn}[2 + 3] = 2, 3, + \quad (2.7)$$

donde hemos separados los elementos a ingresar en la calculadora por comas. En general, para cualquier operador binario (como $+$, $-$, $*$ o $/$) tenemos

$$\text{rpn}[(a) \theta (b)] = \text{rpn}(a), \text{rpn}(b), \theta \quad (2.8)$$

donde a , b son los operandos y θ el operador. Hemos introducido paréntesis alrededor de los operandos a y b ya que (eventualmente) estos pueden ser también expresiones, de manera que, por ejemplo la expresión $(2 + 3) * (4 - 5)$ puede escribirse como

$$\begin{aligned} \text{rpn}[(2 + 3) * (4 - 5)] &= \text{rpn}[2 + 3], \text{rpn}[4 - 5], * \\ &= 2, 3, +, 4, 5, -, * \end{aligned} \quad (2.9)$$

La ventaja de una tal calculadora es que no hace falta ingresar paréntesis, con el inconveniente de que el usuario debe convertir mentalmente la expresión a RPN.

2.2.1. Una calculadora RPN con una pila

La forma de implementar una calculadora RPN es usando una pila. A medida que el usuario entra operandos y operadores se aplican las siguientes reglas

- Si el usuario ingresó un operando, entonces simplemente se almacena en la pila.
- Si ingresó un operador θ se extraen dos operandos del tope de la pila, digamos t el tope de la pila y u el elemento siguiente, se aplica el operador a los dos operandos (en forma invertida) es decir $u \theta t$ y se almacena el resultado en el tope de la pila.

Por ejemplo, para la expresión (2.9) tenemos un seguimiento como el mostrado en la tabla 2.3. que es el resultado correcto. El algoritmo puede extenderse fácilmente a funciones con un número arbitrario de variables (como **exp()**, **cos()** ...), sólo que en ese caso se extrae el número de elementos apropiados de la pila, se le aplica el valor y el resultado es introducido en la misma. De nuevo, notar que en general debe invertirse el orden de los argumentos al sacarlos de la pila. Por ejemplo la función **rem(a,b)** (resto) retorna el resto de dividir **b** en **a**. Si analizamos la expresión **mod(5,3)** (que debe retornar 2) notamos que al momento de aplicar la función **mod**, tenemos en la pila los elementos **3,5** (el top primero), de manera que la función debe aplicarse a los elementos *en orden invertido*.

Ingresa	Tipo	Pila
2	operando	2
3	operando	3,2
+	operador	5
4	operando	4,5
5	operando	5,4,5
-	operador	-1,5
*	operador	-5

Tabla 2.3: Seguimiento del funcionamiento de la calculadora RPN usando una pila. (Los elementos en la pila son enumerados empezando por el tope.)

2.2.2. Operaciones abstractas sobre pilas

El ejemplo anterior sugiere las siguientes operaciones abstractas sobre pilas

- Insertar un elemento en el tope de la pila.
- Obtener el valor del elemento en el tope de la pila.
- Eliminar el elemento del tope.

2.2.3. Interfaz para pila

```

1 elem_t top();
2 void pop();
3 void push(elem_t x);
4 void clear();
5 int size();
6 bool empty();

```

Código 2.14: Interfaz [Archivo: sbas.h]

Una posible interfaz puede observarse en el código 2.14. Consta de sólo tres funciones básicas a saber,

- **top()** devuelve el elemento en el tope de la pila (sin modificarla).
- **pop()** remueve el elemento del tope (sin retornar su valor!).
- **push(x)** inserta el elemento **x** en el tope de la pila.

Esta interfaz es directamente compatible con STL, ya que, a diferencia con las listas, la pila no tiene iterators.

Casi todas las librerías que implementan pilas usan una interfaz similar a esta con muy pequeñas variantes. En algunos casos, por ejemplo, la función que remueve el elemento también devuelve el elemento del tope.

También hemos agregado tres funciones auxiliares más a saber

- `clear()` remueve todos los elementos de la pila.
- `int size()` devuelve el número de elementos en la pila.
- `bool empty()` retorna verdadero si la pila esta vacía, verdadero en caso contrario.

Notar que `empty()` *no modifica* la pila, mucha gente tiende a confundirla con `clear()`.

2.2.4. Implementación de una calculadora RPN

```
1 bool check2(stack &P, double &v1, double&v2) {
2     if (P.size()<2) {
3         cout << "Debe haber al menos 2 elementos en la pila!!\n";
4         return false;
5     } else {
6         v2 = P.top(); P.pop();
7         v1 = P.top(); P.pop();
8         return true;
9     }
10 }
11
12 bool check1(stack &P, double &v1) {
13     if (P.size()<1) {
14         cout << "Debe haber al menos 1 elemento en la pila!!\n";
15         return false;
16     } else {
17         v1 = P.top(); P.pop();
18         return true;
19     }
20 }
21
22 int main() {
23     stack P,Q;
24     const int SIZE=100;
25     char line[SIZE];
26     double v1,v2;
27     // REPL (read, eval print loop)
28     while(true) {
29         // Read
30         cout << "calc> ";
31         assert(line);
32         cin.getline(line,SIZE, '\n');
33         if(!cin) break;
34         // 'Eval' y 'print' dependiendo del caso
35         if (!strcmp(line,"+")) {
36             if (check2(P,v1,v2)) {
37                 P.push(v1+v2);
38                 printf("->%lf\n",P.top());
39             }
40         } else if (!strcmp(line,"-")) {
41             if (check2(P,v1,v2)) {
42                 P.push(v1-v2);
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```
43     printf(">%lf\n",P.top());
44 }
45 } else if (!strcmp(line,"*")) {
46     if (check2(P,v1,v2)) {
47         P.push(v1*v2);
48         printf(">%lf\n",P.top());
49     }
50 } else if (!strcmp(line,"/")) {
51     if (check2(P,v1,v2)) {
52         P.push(v1/v2);
53         printf(">%lf\n",P.top());
54     }
55 } else if (!strcmp(line,"log")) {
56     if (check1(P,v1)) {
57         P.push(log(v1));
58         printf(">%lf\n",P.top());
59     }
60 } else if (!strcmp(line,"exp")) {
61     if (check1(P,v1)) {
62         P.push(exp(v1));
63         printf(">%lf\n",P.top());
64     }
65 } else if (!strcmp(line,"sqrt")) {
66     if (check1(P,v1)) {
67         P.push(sqrt(v1));
68         printf(">%lf\n",P.top());
69     }
70 } else if (!strcmp(line,"atan2")) {
71     if (check2(P,v1,v2)) {
72         P.push(atan2(v1,v2));
73         printf(">%lf\n",P.top());
74     }
75 } else if (!strcmp(line,"c")) {
76     printf("vaciando la pila...\n");
77     P.clear();
78 } else if (!strcmp(line,"p")) {
79     printf("pila: ");
80     while(!P.empty()) {
81         double x = P.top();
82         cout << x << " ";
83         P.pop();
84         Q.push(x);
85     }
86     while(!Q.empty()) {
87         double x = Q.top();
88         Q.pop();
89         P.push(x);
90     }
91     cout << endl;
92 } else if (!strcmp(line,"x")) {
93     "Saliendo de calc!!\n";
94     exit(0);
95 } else {
96     double val;
```

```

97     int nread = sscanf(line, "%lf", &val);
98     if (nread!=1) {
99         printf("Entrada invalida!!: \"%s\"\n", line);
100        continue;
101    } else {
102        P.push(val);
103        printf("<%g\n", val);
104    }
105 }
106 }
107 }
```

Código 2.15: Implementación de una calculadora RPN usando una pila. [Archivo: stackcalc.cpp]

En el código 2.15 vemos una posible implementación de la calculadora usando la interfaz STL de la pila descripta en código 2.14. En el **main()** se declara la pila **P** que es la base de la calculadora. Después de la declaración de algunas variables auxiliares se ingresa en un lazo infinito, que es un ejemplo de lazo “*REPL*” (por “*read, eval, print loop*”), típico en los lenguajes interpretados.

- **read:** Se lee una línea de la consola,
- **eval:** Se evalúa para producir efectos laterales y producir un resultado
- **print:** Se imprime el resultado de la evaluación .

La línea se lee con la función **getline()** del standard input **cin**. Si hay cualquier tipo de error al leer (fin de archivo, por ejemplo) **cin** queda en un estado que retorna **false**, de ahí que basta con verificar **!cin** para ver si la lectura ha sido exitosa. El valor leído queda en el string (de C) **line**. El string tiene un tamaño fijo **SIZE**. También se podría hacer dinámicamente con rutinas más elaboradas y seguras como la **sprintf** o **asprintf** (ver Foundation [b]). Antes de leer la línea se imprime el *prompt* “**calc>** ”.

Después de leer la línea se entra en una secuencia de **if-else** (similar a un **switch**). Si la línea entrada es un operador o función, entonces se extrae el número apropiado de operandos de la pila, se aplica la operación correspondiente y el resultado es ingresado en la pila. Es importante verificar que la pila contenga un número apropiado de valores antes de hacer la operación. Por ejemplo, si el usuario entra **+** entonces debemos verificar que al menos haya 2 operandos en la pila. Esto se hace con la función **check2** que verifica que efectivamente haya dos operandos, los extrae de la pila y los pone en las variables **v1** y **v2**. Si no hay un número apropiado de valores entonces **check2** retorna **false**. Para funciones u operadores unarios usamos la función similar **check1**.

Además de los 4 operadores binarios normales y de algunas funciones comunes, hemos incluido un comando para salir de la calculadora (**x**), para limpiar la pila (**c**) y para imprimir la pila (**p**).

Notar que para imprimir la pila se necesita una pila auxiliar **Q**. Los elementos de la pila se van extrayendo de **P**, se imprimen por consola y se guardan en **Q**. Una vez que **P** está vacía, todos los elementos de **Q** son devueltos a **P**.

Una sesión típica, correspondiente a (2.9), sería así

```

1 [mstorti@spider aedsrc]$ stackcalc
2 calc> 2
```

```

3 <- 2
4 calc> 3
5 <- 3
6 calc> +
7 -> 5.000000
8 calc> 4
9 <- 4
10 calc> 5
11 <- 5
12 calc> -
13 -> -1.000000
14 calc> *
15 -> -5.000000
16 calc> x
17 [mstorti@spider aedsrc]$

```

2.2.5. Implementación de pilas mediante listas

Como ya mencionamos, la pila se puede implementar fácilmente a partir de una lista, *asumiendo que el tope de la pila está en el comienzo de la lista*. **push(x)** y **pop()** se pueden implementar a partir de **insert** y **erase** en el comienzo de la lista. **top()** se puede implementar en base a **retrieve**. La implementación por punteros y cursores es apropiada, ya que todas estas operaciones son $O(1)$. Notar que si se usa el último elemento de la lista como tope, entonces las operaciones de **pop()** pasan a ser $O(n)$, ya que una vez que, asumiendo que contamos con la posición **q** del último elemento de la lista, al hacer un **pop()**, **q** deja de ser válida y para obtener la nueva posición del último elemento de la lista debemos recorrerla desde el principio. La implementación de pilas basada en listas puede observarse en los código 2.16 y código 2.17. Notar que la implementación es muy simple, ya que prácticamente todas las operaciones son transferidas al tipo lista.

Por otra parte, en la implementación de listas por arreglos tanto la inserción como la supresión en el primer elemento son $O(n)$. En cambio, si podría implementarse con una implementación basada en arreglos *si el tope de la pila está al final*.

El tamaño de la pila es guardado en un miembro **int size_m**. Este contador es inicializado a cero en el constructor y después es actualizado durante las operaciones que modifican la longitud de la lista como **push()**, **pop()** y **clear()**.

```

1 class stack : private list {
2     private:
3         int size_m;
4     public:
5         stack();
6         void clear();
7         elem_t& top();
8         void pop();
9         void push(elem_t x);
10        int size();
11        bool empty();
12    };

```

Código 2.16: Pila basada en listas. Declaraciones. [Archivo: stackbas.h]

```
1 stack::stack() : size_m(0) { }
2
3 elem_t& stack::top() {
4     return retrieve(begin());
5 }
6
7 void stack::pop() {
8     erase(begin()); size_m--;
9 }
10
11 void stack::push(elem_t x) {
12     insert(begin(),x); size_m++;
13 }
14
15 void stack::clear() {
16     erase(begin(),end()); size_m = 0;
17 }
18
19 bool stack::empty() {
20     return begin()==end();
21 }
22
23 int stack::size() {
24     return size_m;
25 }
```

Código 2.17: Pila basada en listas. Implementación. [Archivo: stackbas.cpp]

2.2.6. La pila como un adaptador

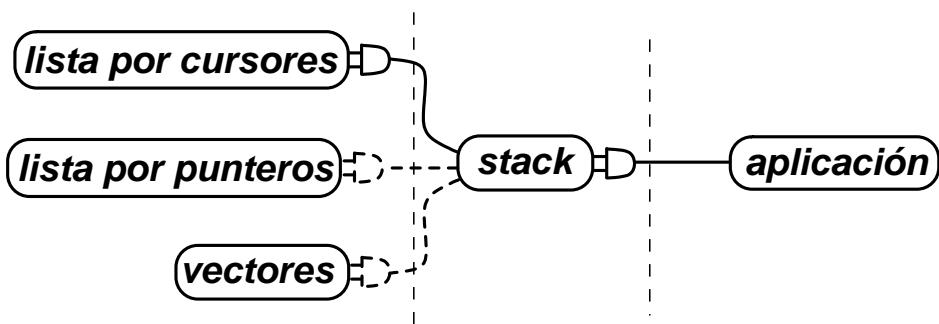


Figura 2.14: La clase pila como un adaptador para varios contenedores básicos de STL

Notar que la pila deriva directamente de la lista, pero con una declaración **private**. De esta forma el usuario de la clase **stack** no puede usar métodos de la clase lista. El hecho de que la pila sea tan simple permite que pueda ser implementada en términos de otros contenedores también, como por ejemplo el contenedor **vector** de STL. De esta forma, podemos pensar a la pila como un “*adaptador*” (“*container adaptor*”), es decir que brinda un subconjunto de la funcionalidad del contenedor original ver figura 2.14. La ventaja de operar sobre el adaptador (en este caso la pila) y no directamente sobre el contenedor básico (en este caso la lista) es que el adaptador puede después conectarse fácilmente a otros contenedores (en la figura representado por enchufes).

2.2.7. Interfaz STL

```
1 #ifndef AED_STACK_H
2 #define AED_STACK_H
3
4 #include <aedsrc/list.h>
5
6 namespace aed {
7
8     template<class T>
9     class stack : private list<T> {
10     private:
11         int size_m;
12     public:
13         stack() : size_m(0) { }
14         void clear() { erase(begin(),end()); size_m = 0; }
15         T &top() { return *begin(); }
16         void pop() { erase(begin()); size_m--; }
17         void push(T x) { insert(begin(),x); size_m++; }
18         int size() { return size_m; }
19         bool empty() { return size_m==0; }
20     };
21 }
22 #endif
```

Código 2.18: Clase pila con templates. [Archivo: stack.h]

Como la pila en si misma no contiene iteradores no hay necesidad de clases anidadas ni sobrecarga de operadores, de manera que la única diferencia con la interfaz STL es el uso de templates. Una interfaz compatible con STL puede observarse en el código 2.18.

2.3. El TAD cola

Por contraposición con la pila, la cola es un contenedor de tipo “*FIFO*” (por “*First In First Out*”, el primero en entrar es el primero en salir). El ejemplo clásico es la cola de la caja en el supermercado. La cola es un objeto muchas veces usado como buffer o pulmón, es decir un contenedor donde almacenar una serie de

objetos que deben ser procesados, manteniendo el orden en el que ingresaron. La cola es también, como la pila, un subtipo de la lista llama también a ser implementado como un adaptador.

2.3.1. Intercalación de vectores ordenados

Ejemplo 2.4: Un problema que normalmente surge dentro de los algoritmos de ordenamiento es el intercalamiento de contenedores ordenados. Por ejemplo, si tenemos dos listas ordenadas **L1** y **L2**, el proceso de intercalamiento consiste en generar una nueva lista **L** ordenada que contiene los elementos en **L1** y **L2** de tal forma que **L** está ordenada, en la forma lo más eficiente posible. Con listas es fácil llegar a un algoritmo $O(n)$ simplemente tomando de las primeras posiciones de ambas listas el menor de los elementos e insertándolo en **L** (ver secciones §4.3.0.2 y §5.6). Además, este algoritmo no requiere memoria adicional, es decir, no necesita alocar nuevas celdas ya que los elementos son agregados a **L** a medida que se eliminan de **L1** y **L2** (Cuando manipula contenedores sin requerir memoria adicional se dice que es “*in place*” (“en el lugar”)). Para vectores el problema podría plantearse así.

Consigna: Sea **a** un arreglo de longitud par, tal que las posiciones pares como las impares están ordenadas entre sí, es decir

$$\begin{aligned} a_0 &\leq a_2 \leq \cdots \leq a_{n-2} \\ a_1 &\leq a_3 \leq \cdots \leq a_{n-1} \end{aligned} \quad (2.10)$$

Escribir un algoritmo que ordena los elementos de **a**.

2.3.1.1. Ordenamiento por inserción

Solución: Consideremos por ejemplo que el arreglo contiene los siguientes elementos.

$$a = \begin{array}{ccccccccccccccccccccc} 10 & 1 & 12 & 3 & 14 & 5 & 16 & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \end{array} \quad (2.11)$$

Verificamos que los elementos en las posiciones pares ($10 \ 12 \ 14 \ 16 \dots$) están ordenados entre sí, como también los que están en las posiciones impares ($1 \ 3 \ 5 \ 7 \dots$). Consideremos primero el algoritmo de “ordenamiento por inserción” (ver código 2.19, los algoritmos de ordenamiento serán estudiados en más detalle en un capítulo posterior).

```

1 void inssort(vector<int> &a) {
2     int n=a.size();
3     for (int j=1; j<n; j++) {
4         int x = a[j];
5         int k = j;
6         while (--k>=0 && x<a[k]) a[k+1] = a[k];
7         a[k+1] = x;
8     }
9 }
```

Código 2.19: Algoritmo de ordenamiento por inserción. [Archivo: *inssort.cpp*]

El cursor j va avanzando desde el comienzo del vector hasta el final. Después de ejecutar el cuerpo del lazo sobre j el rango de elementos $[0, j]$ queda ordenado. (Recordar que $[a, b)$ significa los elementos que

están en las posiciones entre a y b *includiendo a a y excluyendo a b*). Al ejecutar el lazo para un dado j , los elementos a_0, \dots, a_{j-1} están ordenados, de manera que basta con “*insertar*” (de ahí el nombre del método) el elemento a_j en su posición correspondiente. En el lazo sobre k , todos los elementos mayores que a_j son desplazados una posición hacia el fondo y el elemento es insertado en alguna posición $p \leq j$, donde corresponde.

10	1	j	12	3	14	5	16	7	18	9	20	51	22	53	24	55	26	57	28	59
1	p	10	12	3	14	5	16	7	18	9	20	51	22	53	24	55	26	57	28	59
1	10	12	j	3	14	5	16	7	18	9	20	51	22	53	24	55	26	57	28	59
1	10	12	p	3	14	5	16	7	18	9	20	51	22	53	24	55	26	57	28	59
1	10	12	3	j	14	5	16	7	18	9	20	51	22	53	24	55	26	57	28	59
1	3	p	10	12	14	5	16	7	18	9	20	51	22	53	24	55	26	57	28	59
1	3	10	12	14	j	5	16	7	18	9	20	51	22	53	24	55	26	57	28	59
1	3	10	12	14	p	5	16	7	18	9	20	51	22	53	24	55	26	57	28	59
1	3	10	12	14	5	j	16	7	18	9	20	51	22	53	24	55	26	57	28	59
1	5	p	10	12	14	16	7	18	9	20	51	22	53	24	55	26	57	28	59	
1	3	5	10	12	14	16	j	7	18	9	20	51	22	53	24	55	26	57	28	59
1	3	5	10	12	14	16	p	7	18	9	20	51	22	53	24	55	26	57	28	59
1	3	5	10	12	14	16	7	j	18	9	20	51	22	53	24	55	26	57	28	59
1	7	p	10	12	14	16	18	9	20	51	22	53	24	55	26	57	28	59		
1	3	5	7	10	12	14	16	18	j	9	20	51	22	53	24	55	26	57	28	59
1	3	5	7	10	12	14	16	18	p	9	20	51	22	53	24	55	26	57	28	59
1	3	5	7	10	12	14	16	18	9	j	20	51	22	53	24	55	26	57	28	59
1	3	5	7	9	p	10	12	14	16	18	20	51	22	53	24	55	26	57	28	59

(2.12)

En (2.12) vemos un seguimiento de algunas de las operaciones de inserción. Cada par de líneas corresponde a la ejecución para un j dado, la primera línea muestra el estado del vector antes de ejecutar el cuerpo del lazo y la segunda línea muestra el resultado de la operación. En ambos casos se indica con una caja el elemento que es movido desde la posición j a la p . Por ejemplo, para $j = 9$ el elemento $a_j = 9$ debe viajar hasta la posición $p = 4$, lo cual involucra desplazar todos los elementos que previamente estaban en el rango $[4, 9]$ una posición hacia arriba en el vector. Este algoritmo funciona, por supuesto, para cualquier vector, independientemente de si las posiciones pares e impares están ordenadas entre sí, como estamos asumiendo en este ejemplo.

2.3.1.2. Tiempo de ejecución

Consideremos ahora el tiempo de ejecución de este algoritmo. El lazo sobre j se ejecuta $n - 1$ veces, y el lazo interno se ejecuta, en el peor caso $j - 1$ veces, con lo cual el costo del algoritmo es, en el peor caso

$$T_{\text{peor}}(n) = \sum_{j=1}^{n-1} (j - 1) = O(n^2) \quad (2.13)$$

En el mejor caso, el lazo interno no se ejecuta ninguna vez, de manera que sólo cuenta el lazo externo que es $O(n)$.

En general, el tiempo de ejecución del algoritmo dependerá de cuantas posiciones deben ser desplazadas (es decir $p - j$) para cada j

$$T(n) = \sum_{j=1}^{n-1} (p - j) \quad (2.14)$$

o, tomando promedios

$$T_{\text{prom}}(n) = \sum_{j=1}^{n-1} d_j \quad (2.15)$$

donde d_j es el tamaño promedio del rango que se desplaza.

El promedio debe ser tomado sobre un cierto conjunto de posibles vectores. Cuando tomamos vectores completamente desordenados, se puede ver fácilmente que $d_j = j/2$ ya que el elemento a_j no guarda ninguna relación con respecto a los elementos precedentes y en promedio irá a parar a la mitad del rango ordenado, es decir $p = j/2$ y entonces $j - p = j/2$, de manera que

$$T_{\text{prom}}(n) = \sum_{j=1}^{n-1} d_j = \sum_{j=1}^{n-1} \frac{j}{2} = O(n^2) \quad (2.16)$$

2.3.1.3. Particularidades al estar las secuencias pares e impares ordenadas

Como la intercalación de listas ordenadas es $O(n)$ surge la incógnita de si el algoritmo para arreglos puede ser mejorado. Al estar las posiciones pares e impares ordenadas entre sí puede ocurrir que *en promedio* el desplazamiento sea menor, de hecho, generando vectores en forma aleatoria, pero tales que sus posiciones pares e impares estén ordenada se llega a la conclusión que el desplazamiento promedio es $O(\sqrt{n})$, de manera que el algoritmo resulta ser $O(n^{3/2})$. Esto representa una gran ventaja contra el $O(n^2)$ del algoritmo de ordenamiento original.

De todas formas, podemos mejorar más aún esto si tenemos en cuenta que las subsecuencias pares e impares están ordenadas. Por ejemplo consideremos lo que ocurre en el seguimiento (2.12) al mover los elementos 18 y 9 que originalmente estaban en las posiciones $q = 8$ y $q + 1 = 9$. Como vemos, los elementos en las posiciones 0 a $q - 1 = 7$ están ordenados. Notar que el máximo del rango ya ordenado $[0, q]$ es menor que el máximo de estos dos nuevos elementos a_q y a_{q+1} , ya que todos los elementos en $[0, q]$ provienen de elementos en las subsecuencias que estaban *antes* de a_q y a_{q+1} .

$$\max_{j=0}^{q-1} a_j < \max(a_q, a_{q+1}) \quad (2.17)$$

por lo tanto después de insertar los dos nuevos elementos, el mayor (que en este caso es 18) quedará en la posición $q + 1$. El menor ($\min(a_q, a_{q+1}) = 9$) viaja una cierta distancia, hasta la posición $p = 4$. Notar que, por un razonamiento similar, todos los elementos en las posiciones $[q + 2, n)$ deben ser mayores que $\min(a_q, a_{q+1})$, de manera que los elementos en $[0, p)$ no se moverán a partir de esta inserción.

2.3.1.4. Algoritmo de intercalación con una cola auxiliar

```

1 void merge(vector<int> &a) {
2     int n = a.size();
3     // C = cola vacia ...
4     int p=0, q=0, minr, maxr;
5     while (q<n) {
6         // minr = min(a_q,a_{q+1}), maxr = max(a_q,a_{q+1})
7         if (a[q]<=a[q+1]) {
8             minr = a[q];
9             maxr = a[q+1];
10        } else {
11            maxr = a[q];
12            minr = a[q+1];
13        }
14        // Apendizar todos los elementos del frente de la cola menores que
15        // min(a_q,a_{q+1}) al rango [0,p), actualizando eventualmente
16        while ( /* C no esta vacia... */ ) {
17            x = /* primer elemento de C ... */;
18            if (x>minr) break;
19            a[p++] = x;
20            // Saca primer elemento de C ...
21        }
22        a[p++] = minr;
23        a[p++] = minr;
24        // Apendizar 'maxr' al rango [0,p) ...
25        q += 2;
26    }
27    // Apendizar todos los elementos en C menores que
28    // min(a_q,a_{q+1}) al rango [0,p)
29    // ...
30 }
```

Código 2.20: Algoritmo de intercalación con una cola auxiliar [Archivo: mrgarray1.cpp]

El algoritmo se muestra en el código 2.20, manteniendo el rango $[p, q)$ en una cola auxiliar C . En (2.18) vemos el seguimiento correspondiente. Los elementos en la cola C son mostrados encerrados en una caja, en el rango $[p, q)$. Notar que si bien, el número de elementos en la cola entra exactamente en ese rango, en la implementación el estado los elementos en ese rango es irrelevante y los elementos están en una cola

auxiliar.

1	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>10</td></tr></table> ^C	10	12	3	14	5	16	7	18	9	20	51	22	53	24	55	26	57	28	59
10																				
1	3	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>10</td><td>12</td></tr></table> ^C	10	12	14	5	16	7	18	9	20	51	22	53	24	55	26	57	28	59
10	12																			
1	3	5	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>10</td><td>12</td><td>14</td></tr></table> ^C	10	12	14	16	7	18	9	20	51	22	53	24	55	26	57	28	59
10	12	14																		
1	3	5	7	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>10</td><td>12</td><td>14</td><td>16</td></tr></table> ^C	10	12	14	16	18	9	20	51	22	53	24	55	26	57	28	59
10	12	14	16																	
1	3	5	7	9	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>10</td><td>12</td><td>14</td><td>16</td><td>18</td></tr></table> ^C	10	12	14	16	18	20	51	22	53	24	55	26	57	28	59
10	12	14	16	18																
1	3	5	7	9	10	12	14	16	18	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>20</td><td>51</td></tr></table> ^C	20	51	22	53	24	55	26	57	28	59
20	51																			
1	3	5	7	9	10	12	14	16	18	20	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>22</td><td>51</td><td>53</td></tr></table> ^C	22	51	53	24	55	26	57	28	59
22	51	53																		
1	3	5	7	9	10	12	14	16	18	20	22	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>24</td><td>51</td><td>53</td><td>55</td></tr></table> ^C	24	51	53	55	26	57	28	59
24	51	53	55																	
1	3	5	7	9	10	12	14	16	18	20	22	24	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>26</td><td>51</td><td>53</td><td>55</td><td>57</td></tr></table> ^C	26	51	53	55	57	28	59
26	51	53	55	57																
1	3	5	7	9	10	12	14	16	18	20	22	24	26	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>28</td><td>51</td><td>53</td><td>55</td><td>57</td><td>59</td></tr></table> ^C	28	51	53	55	57	59
28	51	53	55	57	59															

(2.18)

Consideremos por ejemplo el paso en el cual se procesan los elementos $a_q = 20$ y $a_{q+1} = 51$ en las posiciones $q = 10$ y $q + 1 = 11$. En ese momento la cola contiene los elementos 10,12,14,16 y 18. Como son todos menores que $\min(a_q, a_{q+1}) = 20$ apendizamos todos al rango $[0, p = 5)$ de manera que queda $p = 10$. Se apendiza también el 20, con lo cual queda $p = 11$ y finalmente se apendiza $\max(a_q, a_{q+1}) = 51$ a la cola. Como en ese momento la cola esta vacía, después de la inserción queda en la cola solo el 51.

2.3.2. Operaciones abstractas sobre colas

Del ejemplo anterior se hacen evidentes las siguientes operaciones

- Obtener el elemento en el frente de la cola
- Eliminar el elemento en el frente de la cola
- Agregar un elemento a la cola

2.3.3. Interfaz para cola

Una versión reducida de la interfaz STL para la cola puede observarse en el código 2.21. Al igual que en el caso de la pila, la cola no tiene posiciones, de manera que no necesita clases anidadas ni sobrecarga de operadores, por lo que el código es sencillo, de manera que no presentamos una versión básica, como hemos hecho con las listas y pilas, sino que presentamos directamente la versión compatible STL.

Las operaciones abstractas descriptas se realizan a través de las funciones **pop()** y **front()**, que operan sobre el principio de la lista, y **push()** que opera sobre el fin de la lista. Todas estas operaciones son $O(1)$. Notar que de elegir lo opuesto (**pop()** y **front()** sobre el fin de la lista y **push()** sobre el principio) entonces la operación **pop()** sería $O(n)$ ya que acceder al último elemento de la lista (no **end()** que es una posición fuera de la lista) es $O(n)$.

También hemos agregado, como en el caso de la pila operaciones estándar `size()` y `empty()`, que también son $O(1)$ y `clear()` que es $O(n)$.

```

1 #ifndef AED_QUEUE_H
2 #define AED_QUEUE_H
3
4 #include <aedsrc/list.h>
5
6 namespace aed {
7
8     template<class T>
9     class queue : private list<T> {
10     private:
11         int size_m;
12     public:
13         queue() : size_m(0) { }
14         void clear() { erase(begin(),end()); size_m = 0; }
15         T &front() { return *begin(); }
16         void pop() { erase(begin()); size_m--; }
17         void push(T x) { insert(end(),x); size_m++; }
18         int size() { return size_m; }
19         bool empty() { return size_m==0; }
20     };
21 }
22 #endif

```

Código 2.21: Interfaz STL para cola [Archivo: `queue.h`]

2.3.4. Implementación del algoritmo de intercalación de vectores

El algoritmo completo, usando la interfaz STL puede observarse en el código 2.22.

```

1 void merge(vector<int> &a) {
2     queue<int> C;
3     int n = a.size();
4     if (n==0) return;
5     if (n % 2) {
6         cout << "debe haber un numero par de elementos en el vector\n";
7         exit(1);
8     }
9     int p=0,q=0, minr, maxr;
10
11    print(a,C,p,q);
12    while (q<n) {
13        if (a[q]<=a[q+1]) {
14            minr = a[q];
15            maxr = a[q+1];
16        } else {
17            maxr = a[q];

```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```

18     minr = a[q+1];
19 }
20 while (!C.empty() && C.front()<=minr) {
21     a[p++] = C.front();
22     C.pop();
23 }
24 a[p++] = minr;
25 C.push(maxr);
26 q += 2;
27 print(a,C,p,q);
28 }
29 while (!C.empty()) {
30     a[p++] = C.front();
31     C.pop();
32 }
33 }
```

Código 2.22: Algoritmo de intercalación con una cola auxiliar. Implementación con la interfaz STL. [Archivo: *mrgarray.cpp*]

2.3.4.1. Tiempo de ejecución

El lazo sobre q se ejecuta $n/2$ veces. Dentro del lazo todas las operaciones son de tiempo constante, salvo los lazos sobre la cola de las líneas 20–23 y 29–32. Las veces que el primer lazo se ejecuta para cada q puede ser completamente variable, pero notar que por cada ejecución de este lazo, un elemento es introducido en el rango $[0, p)$. Lo mismo ocurre para el segundo lazo. Como finalmente todos los elementos terminan en el rango $[0, p)$, el número de veces total que se ejecutan los dos lazos debe ser menor que n . De hecho como para cada ejecución del cuerpo del lazo sobre q se introduce un elemento en $[0, p)$ en la línea 24, el número de veces total que se ejecutan los dos lazos es exactamente igual a $n/2$. De manera que el algoritmo es finalmente $O(n)$.

Sin embargo, el algoritmo no es *in-place*, la memoria adicional está dada por el tamaño de la cola **C**. En el peor caso, **C** puede llegar a tener $n/2$ elementos y en el mejor caso ninguno. En el caso promedio, el tamaño máximo de **C** es tanto como el número de desplazamientos que deben hacerse en el algoritmo de inserción puro, descrito en la sección §2.3.1.3, es decir $O(\sqrt{n})$.

Todo esto esta resumido en la tabla (M es la memoria *adicional* requerida).

2.4. El TAD correspondencia

La “correspondencia” o “memoria asociativa” es un contenedor que almacena la relación entre elementos de un cierto conjunto universal D llamado el “dominio” con elementos de otro conjunto universal llamado el “contradominio” o “rango”. Por ejemplo, la correspondencia \mathcal{M} que va del dominio de los números enteros en sí mismo y transforma un número j en su cuadrado j^2 puede representarse como se muestra en la figura 2.15. Una restricción es que un dado elemento del dominio o bien no debe tener asignado ningún elemento del contradominio o bien debe tener asignado uno solo. Por otra parte, puede ocurrir que a varios

	inssort	merge
$T_{\text{peor}}(n)$	$O(n^2)$	$O(n)$
$T_{\text{prom}}(n)$	$O(n^{3/2})$	$O(n)$
$T_{\text{mejor}}(n)$	$O(n)$	$O(n)$
$M_{\text{peor}}(n)$	$O(n)$	$O(n)$
$M_{\text{prom}}(n)$	$O(n)$	$O(\sqrt{n})$
$M_{\text{mejor}}(n)$	$O(n)$	$O(1)$

Tabla 2.4: Tiempo de ejecución para la intercalación de vectores ordenados. T es tiempo de ejecución, M memoria *adicional* requerida.

elementos del dominio se les asigne un solo elemento del contradominio. En el ejemplo de la figura a los elementos 3 y -3 del dominio les es asignado el mismo elemento 9 del contradominio. A veces también se usa el término “clave” (“key”) (un poco por analogía con las bases de datos) para referirse a un valor del dominio y “valor” para referirse a los elementos del contradominio.

Las correspondencias son representadas en general guardando internamente los pares de valores y poseen algún algoritmo para asignar valores a claves, en forma análoga a como funcionan las bases de datos. Por eso, en el caso del ejemplo previo $j \rightarrow j^2$, es mucho más eficiente representar la correspondencia como una función, ya que es mucho más rápido y no es necesario almacenar todos los valores. Notar que para las representaciones más usuales de enteros, por ejemplo con 32 bits, harían falta varios gigabytes de RAM. El uso de un contenedor tipo correspondencia es útil justamente cuando no es posible calcular el elemento del contradominio a partir del elemento del dominio. Por ejemplo, un tal caso es una correspondencia entre el número de documento de una persona y su nombre. Es imposible de “calcular” el nombre a partir del número de documento, necesariamente hay que almacenarlo internamente en forma de pares de valores.

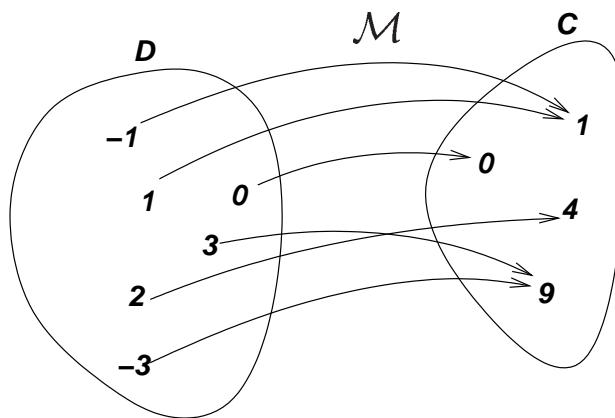


Figura 2.15: Correspondencia entre números enteros $j \rightarrow j^2$

Ejemplo 2.5: *Consigna:* Escribir un programa que memoriza para cada documento de identidad el sueldo de un empleado. Se van ingresando números de documento, si el documento ya tiene un sueldo asignado, entonces esto se reporta por consola, sino el usuario debe entrar un sueldo el cual es asignado a ese número de documento en la tabla. Una posible interacción con el programa puede ser como sigue

```
1 [mstorti@spider aedsrc]$ payroll
2 Ingrese nro. documento > 14203323
3 Ingrese salario mensual: 2000
4 Ingrese nro. documento > 13324435
5 Ingrese salario mensual: 3000
6 Ingrese nro. documento > 13323421
7 Ingrese salario mensual: 2500
8 Ingrese nro. documento > 14203323
9 Doc: 14203323, salario: 2000
10 Ingrese nro. documento > 13323421
11 Doc: 13323421, salario: 2500
12 Ingrese nro. documento > 13242323
13 Ingrese salario mensual: 5000
14 Ingrese nro. documento > 0
15 No se ingresan mas sueldos...
16 [mstorti@spider aedsrc]$
```

Solución: Un posible seudocódigo puede observarse en el código 2.23. El programa entra en un lazo infinito en el cual se ingresa el número de documento y se detiene cuando se ingresa un documento nulo. Reconocemos las siguientes operaciones abstractas necesarias para manipular correspondencias

- Consultar la correspondencia para saber si una dada clave tiene un valor asignado.
- Asignar un valor a una clave.
- Recuperar el valor asignado a una clave.

```
1 // declarar 'tabla_sueldos' como 'map' ...
2 while(1) {
3     cout << "Ingrese nro. documento > ";
4     int doc;
5     double sueldo;
6     cin >> doc;
7     if (!doc) break;
8     if /* No tiene 'doc' sueldo asignado?... */ {
9         cout << "Ingrese sueldo mensual: ";
10        cin >> sueldo;
11        // Asignar 'doc -> sueldo'
12        // ...
13    } else {
14        // Reportar el valor almacenado
15        // en 'tabla_sueldos'
16        // ...
17        cout << "Doc: " << doc << ", sueldo: "
18            << sueldo << endl;
19    }
20 }
21 cout << "No se ingresan mas sueldos... " << endl;
```

Código 2.23: Seudocódigo para construir una tabla que representa la correspondencia número de documento → sueldo. [Archivo: payroll4.cpp]

2.4.1. Interfaz simple para correspondencias

```
1 class iterator_t /* . . . */;
2
3 class map {
4 private:
5 // ...
6 public:
7 iterator_t find(domain_t key);
8 iterator_t insert(domain_t key, range_t val);
9 range_t& retrieve(domain_t key);
10 void erase(iterator_t p);
11 int erase(domain_t key);
12 domain_t key(iterator_t p);
13 range_t& value(iterator_t p);
14 iterator_t begin();
15 iterator_t next(iterator_t p);
16 iterator_t end();
17 void clear();
18 void print();
19 };
```

Código 2.24: Interfaz básica para correspondencias. [Archivo: mapbas.h]

En el código 2.24 vemos una interfaz básica posible para correspondencias. Está basada en la interfaz STL pero, por simplicidad evitamos el uso de clases anidadas para el correspondiente iterator y también evitamos el uso de templates y sobrecarga de operadores. Primero se deben definir (probablemente via **typedef's**) los tipos que corresponden al dominio (**domain_t**) y al contradominio (**range_t**). Una clase **iterator** (cuyos detalles son irrelevantes para la interfaz) representa las posiciones en la correspondencia. Sin embargo, considerar de que, en contraposición con las listas y a semejanza de los conjuntos, no hay un orden definido entre los pares de la correspondencia. Por otra parte en la correspondencia el iterator *itera sobre los pares de valores* que representan la correspondencia.

En lo que sigue **M** es una correspondencia, **p** es un iterator, **k** una clave, **val** un elemento del contradominio (tipo **range_t**). Los métodos de la clase son

- **p = find(k)**: Dada una clave **k** devuelve un iterator *al par correspondiente* (si existe debe ser único). Si **k** no tiene asignado ningún valor, entonces devuelve **end()**.
- **p = insert(k, val)**: asigna a **k** el valor **val**. Si **k** ya tenía asignado un valor, entonces este nuevo valor reemplaza a aquel en la asignación. Si **k** no tenía asignado ningún valor entonces la nueva asignación es definida. Retorna un iterator al par.
- **val = retrieve(k)**: Recupera el valor asignado a **k**. Si **k** no tiene ningún valor asignado, entonces *inserta una asignación de k al valor creado por defecto* para el tipo **range_t** (es decir el que retorna el constructor **range_t()**). *Esto es muy importante y muchas veces es fuente de error.* (Muchos esperan que **retrieve()** de un error en ese caso.) Si queremos recuperar en **val** el valor asignado a **k** *sin insertar accidentalmente una asignación* en el caso que **k** no tenga asignado ningún valor entonces debemos hacer

```
1 if (M.find(k)!=M.end()) val = M.retrieve(k);
```

val=M.retrieve(k) retorna una *referencia* al valor asignado a **k**, de manera que también puede ser usado como miembro izquierdo, es decir, es válido hacer (ver §2.1.4)

```
1 M.retrieve(k) = val;
```

- **k = key(p)** retorna el valor correspondiente a la asignación *apuntada* por **p**.
- **val = value(p)** retorna el valor correspondiente a la asignación *apuntada* por **p**. El valor retornado es una referencia, de manera que también podemos usar **value(p)** como miembro izquierdo (es decir, asignarle un valor como en **value(p)=val**). Notar que, por el contrario, **key(p)** no retorna una referencia.
- **erase(p)**: Elimina la asignación apuntada por **p**. Si queremos eliminar una eventual asignación a la clave **k** entonces debemos hacer

```
1 p = M.find(k);
2 if (p!=M.end()) M.erase(p);
```

- **p = begin()**: Retorna un iterator a la primera asignación (en un orden no especificado).
- **p = end()**: Retorna un iterator a una asignación ficticia después de la última (en un orden no especificado).
- **clear()**: Elimina todas las asignaciones.
- **print()**: Imprime toda la tabla de asignaciones.

Con esta interfaz, el programa 2.23 puede completarse como se ve en código 2.25. Notar que el programa es cuidadoso en cuanto a no crear nuevas asignaciones. El **retrieve** de la línea 16 está garantizado que no generará ninguna asignación involuntaria ya que el test del if garantiza que **doc** ya tiene asignado un valor.

```
1 map sueldo;
2 while(1) {
3     cout << "Ingrese nro. documento > ";
4     int doc;
5     double salario;
6     cin >> doc;
7     if(!doc) break;
8     iterator_t q = sueldo.find(doc);
9     if (q==sueldo.end()) {
10        cout << "Ingrese salario mensual: ";
11        cin >> salario;
12        sueldo.insert(doc,salario);
13        cout << sueldo.size() << " salarios cargados" << endl;
14    } else {
15        cout << "Doc: " << doc << ", salario: "
16            << sueldo.retrieve(doc) << endl;
17    }
18 }
19 cout << "No se ingresan mas sueldos. . ." << endl;
```

Código 2.25: Tabla de sueldos del personal implementado con la interfaz básica de código 2.24. [Archivo: payroll2.cpp]

2.4.2. Implementación de correspondencias mediante contenedores lineales

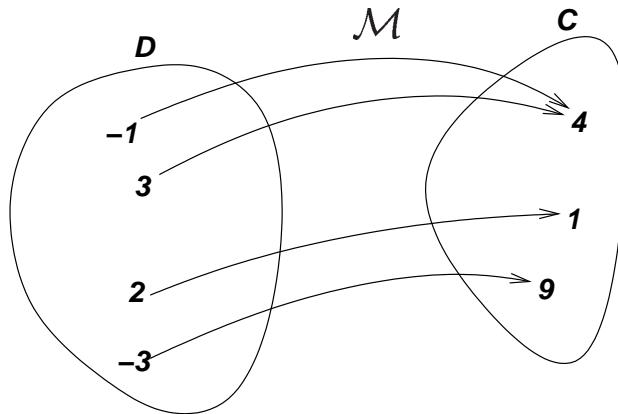


Figura 2.16: Ejemplo de correspondencia.

Tal vez la forma más simple de implementar una correspondencia es guardando en un contenedor todas las asignaciones. Para eso simplemente definimos una clase `elem_t` que simplemente contiene dos campos `first` y `second` con la clave y el valor de la asignación (los nombres de los campos vienen de la clase `pair` de las STL). Estos pares de elementos (asignaciones) se podrían guardar tanto en un `vector<elem_t>` como en una lista (`list<elem_t>`). Por ejemplo, la correspondencia de enteros a enteros que se muestra en la figura 2.15 se puede representar almacenando los siguientes pares en un contenedor:

$$(-1, 4), (3, 4), (2, 1), (-3, 9) \quad (2.19)$$

A las listas y vectores se les llama contenedores lineales, ya que en ellos existe un ordenamiento natural de las posiciones. En lo que sigue discutiremos la implementación del TAD correspondencia basadas en estos contenedores lineales. Más adelante, en otro capítulo, veremos otras implementaciones más eficientes. Cuando hablamos de la implementación con listas asumiremos una implementación de listas basada en punteros o cursores, mientras que para vectores asumiremos arreglos estándar de C++ o el mismo `vector` de STL. En esta sección asumiremos que las asignaciones son insertadas en el contenedor ya sea al principio o en el final del mismo, de manera que el orden entre las diferentes asignaciones es en principio aleatorio. Más adelante discutiremos el caso en que las asignaciones se mantienen ordenadas por la clave. En ese caso las asignaciones aparecerían en el contenedor como en (2.20).

- `p=find(k)` debe recorrer todas las asignaciones y si encuentra una cuyo campo `first` coincide con `k` entonces debe devolver el iterator correspondiente. Si la clave no tiene ninguna asignación, entonces después de recorrer todas las asignaciones, debe devolver `end()`. El peor caso de `find()` es cuando la clave no está asignada, o la asignación está al final del contenedor, en cuyo caso es $O(n)$ ya que debe recorrer todo el contenedor (n es el número de asignaciones en la correspondencia). Si la clave tiene una asignación, entonces el costo es proporcional a la distancia desde la asignación hasta el origen. En el peor caso esto es $O(n)$, como ya mencionamos, mientras que en el mejor caso, que es cuando la asignación está al comienzo del contenedor, es $O(1)$. La distancia media de la asignación es (si las asignaciones se han ingresado en forma aleatoria) la mitad del número de asociaciones y por lo tanto en promedio el costo será $O(n/2)$.

- **insert()** debe llamar inicialmente a **find()**. Si la clave ya tiene un valor asignado, la inserción es $O(1)$ tanto para listas como vectores. En el caso de vectores, notar que esto se debe a que no es necesario insertar una nueva asignación. Si la clave no está asignada entonces el elemento se puede insertar al final para vectores, lo cual también es $O(1)$, y en cualquier lugar para listas.
- Un análisis similar indica que **retrieve(k)** también es equivalente a **find()**.
- Para **erase(p)** sí hay diferencias, la implementación por listas es $O(1)$ mientras que la implementación por vectores es $O(n)$ ya que implica mover todos los elementos que están después de la posición eliminada.
- **clear()** es $O(1)$ para vectores, mientras que para listas es $O(n)$.
- Para las restantes funciones el tiempo de ejecución *en el peor caso* es $O(1)$.

Operación	lista	vector
find(key)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
insert(key, val)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
retrieve(key)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
erase(p)	$O(1)$	$O(1)/O(n)/O(n)$
key, value, begin, end,	$O(1)$	$O(1)$
clear	$O(n)$	$O(1)$

Tabla 2.5: Tiempos de ejecución para operaciones sobre correspondencias con contenedores lineales no ordenadas. n es el número de asignaciones en la correspondencia. (La notación es *mejor/promedio/peor*. Si los tres son iguales se reporta uno sólo.)

Estos resultados se sumarizan en la Tabla 2.5. No mostraremos ninguna implementación de correspondencias con contenedores lineales no ordenados ya que discutiremos a continuación la implementación con contenedores ordenados, que es más eficiente.

2.4.3. Implementación mediante contenedores lineales ordenados

Una posibilidad de reducir el tiempo de ejecución es usar contenedores ordenados, es decir ya sea vectores o listas, pero donde los pares de asignación están ordenados de menor a mayor según la clave. Notar que esto exige que se pueda definir un tal ordenamiento en el conjunto universal del dominio. Si el dominio son los enteros o los reales, entonces se puede usar la relación de orden propia del tipo. Para “strings” (cadenas de caracteres) se puede usar el orden “lexicográfico”. Para otros tipos compuestos, para los cuales no existe una relación de orden se pueden definir relaciones de orden *ad-hoc* (ver §2.4.4). En algunos casos estas relaciones de orden no tienen ningún otro interés que ser usadas en este tipo de representaciones o en otros algoritmos relacionados.

```

1 class map;
2
3 class elem_t {

```

```
4  private:
5    friend class map;
6    domain_t first;
7    range_t second;
8  };
9  // iterator para map va a ser el mismo que para listas.
10 class map {
11 private:
12   list l;
13
14   iterator_t lower_bound(domain_t key) {
15     iterator_t p = l.begin();
16     while (p!=l.end()) {
17       domain_t dom = l.retrieve(p).first;
18       if (dom >= key) return p;
19       p = l.next(p);
20     }
21     return l.end();
22   }
23
24 public:
25   map() { }
26   iterator_t find(domain_t key) {
27     iterator_t p = lower_bound(key);
28     if (p!=l.end() && l.retrieve(p).first == key)
29       return p;
30     else return l.end();
31   }
32   iterator_t insert(domain_t key, range_t val) {
33     iterator_t p = lower_bound(key);
34     if (p==l.end() || l.retrieve(p).first != key) {
35       elem_t elem;
36       elem.first = key;
37       p = l.insert(p,elem);
38     }
39     l.retrieve(p).second = val;
40     return p;
41   }
42   range_t &retrieve(domain_t key) {
43     iterator_t q = find(key);
44     if (q==end()) q=insert(key,range_t());
45     return l.retrieve(q).second;
46   }
47   bool empty() { return l.begin()==l.end(); }
48   void erase(iterator_t p) { l.erase(p); }
49   int erase(domain_t key) {
50     iterator_t p = find(key); int r = 0;
51     if (p!=end()) { l.erase(p); r = 1; }
52     return r;
53   }
54   iterator_t begin() { return l.begin(); }
55   iterator_t end() { return l.end(); }
56   void clear() { l.erase(l.begin(),l.end()); }
```

```

57     int size() { return l.size(); }
58     domain_t key(iterator_t p) {
59         return l.retrieve(p).first;
60     }
61     range_t &value(iterator_t p) {
62         return l.retrieve(p).second;
63     }
64 };

```

Código 2.26: Implementación de correspondencia mediante listas ordenadas. [Archivo: mapl.h]

2.4.3.1. Implementación mediante listas ordenadas

En el caso de representar la correspondencia mediante una lista ordenada, los pares son ordenados de acuerdo con su campo clave. Por ejemplo la correspondencia de la figura 2.16, se representaría por una lista como sigue,

$$\mathcal{M} = ((-3, 9), (-1, 4), (2, 1), (3, 4)). \quad (2.20)$$

Ahora `p=find(k)` no debe necesariamente recorrer toda la lista cuando la clave no está, ya que el algoritmo puede detenerse cuando encuentra una clave *mayor* a la que se busca. Por ejemplo, si hacemos `p=find(0)` en la correspondencia anterior podemos dejar de buscar cuando llegamos al par $(2, 1)$, ya que como los pares están ordenados por clave, todos los pares siguientes deben tener claves mayores que 2, y por lo tanto no pueden ser 0. Sin embargo, al insertar nuevas asignaciones hay que tener en cuenta que no se debe insertar en cualquier posición sino que hay que mantener la lista ordenada. Por ejemplo, si queremos asignar a la clave 0 el valor 7, entonces el par $(0, 7)$ *debe* insertarse entre los pares $(-1, 4)$ y $(2, 1)$, para mantener el orden entre las claves.

Una implementación de la interfaz simplificada mostrada en 2.24 basada en listas ordenadas puede observarse en el código 2.26. Tanto `p=find(key)` como `p=insert(key, val)` se basan en una función auxiliar `p=lower_bound(key)` que retorna la primera posición donde podría insertarse la nueva clave sin violar la condición de ordenamiento sobre el contenedor. Como casos especiales, si todas las claves son mayores que `key` entonces debe retornar `begin()` y si son todas menores, o la correspondencia está vacía, entonces debe retornar `end()`.

La implementación de `p=insert(key, val)` en términos de `p=lower_bound(key)` es simple, `p=lower_bound(key)` retorna un iterator a la asignación correspondiente a `key` (si `key` tiene un valor asignado) o bien un iterator a la posición donde la asignación a `key` debe ser insertada. En el primer caso un nuevo par (de tipo `elem_t`) es construido e insertado usando el método `insert` de listas. Al llegar a la línea 39 la posición `p` apunta al par correspondiente a `key`, independientemente de si este ya existía o si fue creado en el bloque previo.

La implementación de `p=find(key)` en términos de `p=lower_bound(key)` también es simple. Si `p` es `end()` o la asignación correspondiente a `p` contiene *exactamente* la clave `key`, entonces debe retornar `p`. Caso contrario, `p` debe corresponder a una posición dereferenciable, pero cuya clave no es `key` de manera que en este caso no debe retornar `p` sino `end()`.

El método `val=retrieve(key)` busca una asignación para `key`. Notar que, como mencionamos en §2.4.1 si `key` no está asignado entonces *debe generar una asignación*. El valor correspondiente en ese caso es el que retorna el constructor por defecto (`range_t()`).

Notar que `lower_bound()` es declarado `private` en la clase ya que es sólo un algoritmo interno auxiliar para `insert()` y `find()`.

2.4.3.2. Interfaz compatible con STL

```

1 template<typename first_t,typename second_t>
2 class pair {
3 public:
4     first_t first;
5     second_t second;
6 };
7
8 template<typename domain_t,typename range_t>
9 class map {
10 private:
11     typedef pair<domain_t,range_t> pair_t;
12     typedef list<pair_t> list_t;
13     list_t l;
14
15 public:
16     typedef typename list_t::iterator iterator;
17     map();
18     iterator find(domain_t key);
19     range_t & operator[](domain_t key);
20     bool empty();
21     void erase(iterator p);
22     int erase(domain_t key);
23     iterator begin();
24     iterator end();
25     void clear();
26 };

```

Código 2.27: Versión básica de la interfaz STL para correspondencia. [Archivo: `mapstl.h`]

Una versión reducida de la interfaz STL para correspondencia se puede observar en el código 2.27. En el código 2.28 vemos la implementación de esta interfaz mediante listas ordenadas.

```

1 #ifndef AED_MAP_H
2 #define AED_MAP_H
3
4 #include <aedsrc/list.h>
5 #include <iostream>
6
7 using namespace std;
8

```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```
9 namespace aed {
10
11     template<typename first_t,typename second_t>
12     class pair {
13         public:
14             first_t first;
15             second_t second;
16             pair(first_t f=first_t(),second_t s=second_t())
17                 : first(f), second(s) {}
18     };
19
20     // iterator para map va a ser el mismo que para listas.
21     template<typename domain_t,typename range_t>
22     class map {
23
24         private:
25             typedef pair<domain_t,range_t> pair_t;
26             typedef list<pair_t> list_t;
27             list_t l;
28
29         public:
30             typedef typename list_t::iterator iterator;
31
32         private:
33             iterator lower_bound(domain_t key) {
34                 iterator p = l.begin();
35                 while (p!=l.end()) {
36                     domain_t dom = p->first;
37                     if (dom >= key) return p;
38                     p++;
39                 }
40                 return l.end();
41             }
42
43         public:
44             map() { }
45
46             iterator find(domain_t key) {
47                 iterator p = lower_bound(key);
48                 if (p!=l.end() && p->first == key)
49                     return p;
50                 else return l.end();
51             }
52             range_t & operator[](domain_t key) {
53                 iterator q = lower_bound(key);
54                 if (q==end() || q->first!=key)
55                     q = l.insert(q,pair_t(key,range_t()));
56                 return q->second;
57             }
58             bool empty() { return l.begin()==l.end(); }
59             void erase(iterator p) { l.erase(p); }
60             int erase(domain_t key) {
61                 iterator p = find(key);
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```

62     if (p!=end()) {
63         l.erase(p);
64         return 1;
65     } else {
66         return 0;
67     }
68 }
69 iterator begin() { return l.begin(); }
70 iterator end() { return l.end(); }
71 iterator next(iterator p) { return l.next(p); }
72 void clear() { l.erase(l.begin(),l.end()); }
73 };
74 }
75 #endif

```

Código 2.28: Implementación de correspondencia mediante listas ordenadas con la interfaz STL. [Archivo: map.h]

- En vez del tipo **elem_t** se define un template **pair<class first_t, class second_t>**. Este template es usado para **map** y otros contenedores y algoritmos de STL. Los campos **first** y **second** de **pair** son públicos. Esto es un caso muy especial dentro de las STL y la programación orientada a objetos en general ya que en general se *desaconseja permitir el acceso a los campos datos de un objeto*. (La motivación para esto es que **pair<>** es una construcción tan simple que se permite violar la regla.) **pair<>** es una forma muy simple de asociar pares de valores en un único objeto. Otro uso de **pair<>** es para permitir que una función retorne dos valores al mismo tiempo. Esto se logra haciendo que retorne un objeto de tipo **pair<>**.
- La clase **map** es un template de las clases **domain_t** y **range_t**. Los elementos de la lista serán de tipo **pair<domain_t, range_t>**.
- Para simplificar la escritura de la clase, se definen dos tipos internos **pair_t** y **list_t**. Los elementos de la lista serán de tipo **pair_t**. También se define el tipo público **iterator** que es igual al iterator sobre la lista, pero esta definición de tipo permitirá verlo también externamente como **map<domain_t, range_t>::iterator**.
- **val=M.retrieve(key)** se reemplaza sobrecargando el operador **[]**, de manera que con esta interfaz la operación anterior se escribe **val=M[key]**. Recordar que tiene el mismo efecto colateral que **retrieve**: Si **key** no tiene ningún valor asignado, entonces **M[key]** le asigna uno por defecto. Además, igual que **retrieve**, **M[key]** retorna una referencia de manera que es válido usarlo como miembro izquierdo, como en **M[key]=val**.

```

1 map<int,double> sueldo;
2 while(1) {
3     cout << "Ingrese nro. documento > ";
4     int doc;
5     double salario;
6     cin >> doc;
7     if (!doc) break;

```

```

8 map<int,double>::iterator q = sueldo.find(doc);
9 if (q==sueldo.end()) {
10     cout << "Ingrese salario mensual: ";
11     cin >> salario;
12     sueldo[doc]=salario;
13 } else {
14     cout << "Doc: " << doc << ", salario: "
15     << sueldo[doc] << endl;
16 }
17 }
18 cout << "No se ingresan mas sueldos... " << endl;

```

Código 2.29: Tabla de sueldos del personal implementado con la interfaz STL de map (ver código 2.27).
 [Archivo: payroll3.cpp]

El programa implementado en el código 2.25 escrito con esta interface puede observarse en el código 2.29.

2.4.3.3. Tiempos de ejecución para listas ordenadas

Consideremos primero `p=lower_bound(k)` con éxito (es decir, cuando `k` tiene un valor asignado). El costo es proporcional a la distancia con respecto al origen de la posición donde se encuentra la asignación. Valores de `k` bajos quedarán en las primeras posiciones de la lista y los valores altos al fondo de la misma. En promedio un valor puede estar en cualquier posición de la lista, con lo cual tendrá un costo $O(n/2) = O(n)$. El `p=lower_bound(k)` sin éxito (es decir, cuando `k` no tiene un valor asignado) en este caso también se detiene cuando llega a un elemento mayor o igual y, usando el mismo razonamiento, también es $O(n/2) = O(n)$. Consecuentemente, si bien hay una ganancia en el caso de listas ordenadas, el *orden* del tiempo de ejecución es prácticamente el mismo que en el caso de listas no ordenadas.

Operación	lista	vector
<code>find(key)</code>	$O(1)/O(n)/O(n)$	$O(1)/O(\log n)/O(\log n)$
<code>M[key]</code> (no existente)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
<code>M[key]</code> (existente)	$O(1)/O(n)/O(n)$	$O(1)/O(\log n)/O(\log n)$
<code>erase(key)</code>	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
<code>key, value, begin, end,</code>	$O(1)$	$O(1)$
<code>clear</code>	$O(n)$	$O(1)$

Tabla 2.6: Tiempos de ejecución para operaciones sobre correspondencias con contenedores lineales *ordenados*. n es el número de asignaciones en la correspondencia. (La notación es *mejor/promedio/peor*. Si los tres son iguales se reporta uno sólo.)

`find(key)` y `M[key]` están basados en `lower_bound` y tienen el mismo tiempo de ejecución ya que para listas las inserciones son $O(1)$. Los tiempos de ejecución para correspondencias por listas ordenadas se sumarizan en la Tabla 2.6.

2.4.3.4. Implementación mediante vectores ordenados

```
1 #ifndef AED_MAPV_H
2 #define AED_MAPV_H
3
4 #include <iostream>
5 #include <vector>
6
7 using namespace std;
8
9 namespace aed {
10
11     template<typename first_t, typename second_t>
12     class pair {
13     public:
14         first_t first;
15         second_t second;
16         pair() : first(first_t()), second(second_t()) {}
17     };
18
19 // iterator para map va a ser el mismo que para listas.
20     template<typename domain_t, typename range_t>
21     class map {
22
23     public:
24         typedef int iterator;
25
26     private:
27         typedef pair<domain_t,range_t> pair_t;
28         typedef vector<pair_t> vector_t;
29         vector_t v;
30
31         iterator lower_bound(domain_t key) {
32             int p=0, q=v.size(), r;
33             if (!q || v[p].first > key) return 0;
34             while (q-p > 1) {
35                 r = (p+q)/2;
36                 domain_t kr = v[r].first;
37                 if (key > kr) p=r;
38                 else if (key < kr) q=r;
39                 else if (kr==key) return r;
40             }
41             if (v[p].first == key) return p;
42             else return q;
43         }
44
45     public:
46         map() { }
47
48         iterator find(domain_t key) {
49             int p = lower_bound(key);
50             if (p == v.size() || v[p].first == key) return p;
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```

51     else return v.size();
52 }
53 range_t & operator[](domain_t key) {
54     iterator p = lower_bound(key);
55     if (p == v.size() || v[p].first != key) {
56         v.push_back(pair_t());
57         iterator q = v.size();
58         while (--q > p) v[q] = v[q-1];
59         v[p].first = key;
60     }
61     return v[p].second;
62 }
63 int erase(domain_t key) {
64     iterator p = find(key); int r = 0;
65     if (p!=end()) { erase(p); r = 1; }
66     return r;
67 }
68 bool empty() { return v.size()==0; }
69 void erase(iterator p) {
70     iterator q = p;
71     while (q != v.size()) {
72         v[q] = v[q+1];
73         q++;
74     }
75     v.pop_back();
76 }
77 iterator begin() { return 0; }
78 iterator end() { return v.size(); }
79 void clear() { v.clear(); }
80 int size() { return v.size(); }
81 };
82 }
83 #endif

```

Código 2.30: Implementación de correspondencia con vectores ordenados. [Archivo: mapv.h]

La ganancia real de usar contenedores ordenados es en el caso de vectores ya que en ese caso podemos usar el algoritmo de “*búsqueda binaria*” (“*binary search*”) en cuyo caso `p=lower_bound(k)` resulta ser $O(\log n)$ en el peor caso. Una implementación de correspondencias con vectores ordenados puede verse en el código 2.30. El código se basa en la clase `vector` de STL, pero en realidad con modificaciones menores se podrían reemplazar los vectores de STL por arreglos estándar de C. La correspondencia almacena las asignaciones (de tipo `pair_t`) en un `vector<pair_t> v`. Para el tipo `map<>::iterator` usamos directamente el tipo entero.

Veamos en detalle el algoritmo de búsqueda binaria en `lower_bound()`. El algoritmo se basa en ir refinando un rango $[p, q)$ tal que la clave buscada esté garantizado siempre en el rango, es decir $k_p \leq k < k_q$, donde k_p, k_q son las claves en las posiciones p y q respectivamente y k es la clave buscada. Las posiciones en el vector como p, q se representan por enteros comenzando de 0. La posición `end()` en este caso coincide con el entero `size()` (el tamaño del vector). En el caso en que $q = \text{size()}$ entonces asumimos que la clave correspondiente es $k_q = \infty$, es decir más grande que todas las claves posibles.

Si el vector está vacío, entonces `lower_bound` retorna 0 ya que entonces debe insertar en `end()` que en ese caso vale 0. Lo mismo si $k < k_0$, ya que en ese caso la clave va en la primera posición. Si ninguno de estos casos se aplica, entonces el rango $[p, q)$, con $p = 0$ y $m = \mathbf{v.size()}$ es un rango válido, es decir $k_p \leq k < k_q$.

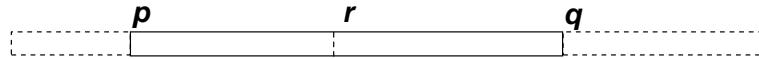


Figura 2.17: Refinando el rango de búsqueda en el algoritmo de búsqueda binaria.

Una vez que tenemos un rango válido $[p, q)$ podemos *refinarlo* (ver figura 2.17) calculando la posición media

$$r = \text{floor}((p + q)/2) \quad (2.21)$$

donde $\text{floor}(x)$ es la parte entera de x ($\text{floor}()$ es parte de la librería `libc`, estándar de C). Esto divide $[p, q)$ en dos subrangos disjuntos $[p, r)$ y $[r, q)$. y comparando la clave `k` con la que está almacenado en la posición r , k_r . Si $k \geq k_r$ entonces el nuevo rango es el $[r, q)$, mientras que si no es el $[p, r)$. Notar que si el rango $[p, q)$ es de longitud par, esto es $n = q - p$ es par, entonces los dos nuevos rangos son iguales, de longitud igual a la mitad $n = (q - p)/2$. Si no, uno es de longitud $n = \text{floor}((q - p)/2)$ y el otro de longitud $n = \text{floor}((q - p)/2) + 1$. Ahora bien, a menos que $n = 1$, esto implica que en cada refinamiento la longitud del rango se reduce estrictamente, de manera que en a lo sumo en n pasos la longitud se reduce a longitud 1, en cuyo caso el algoritmo se detiene. En ese caso o bien la clave buscada esta en `p`, en cuyo caso `lower_bound` retorna `p`, o bien la clave no está y el punto de inserción es `q`.

`p=find(k)` se implementa fácilmente en términos de `lower_bound()`. Básicamente es idéntica al `find()` de listas en el código 2.26. Por otra parte `operator[]` es un poco más complicado, ya que si la búsqueda no es exitosa (es decir, si k no tiene ningún valor asignado) hay que desplazar todas las asignaciones una posición hacia el final para hacer lugar para la nueva asignación (el lazo de las líneas 55–60).

2.4.3.5. Tiempos de ejecución para vectores ordenados

Ahora estimemos mejor el número de refinamientos. Si el número de elementos es inicialmente una potencia de 2, digamos $n = 2^m$, entonces después del primer refinamiento la longitud será 2^{m-1} , después de dos refinamientos 2^{m-2} , hasta que, después de m refinamientos la longitud se reduce a $2^0 = 1$ y el algoritmo se detiene. De manera que el número de refinamientos es $m = \log_2 n$. Puede verse que, si n no es una potencia de dos, entonces el número de refinamientos es $m = \text{floor}(\log_2 n) + 1$. Pero el tiempo de ejecución de `lower_bound` es proporcional al número de veces que se ejecuta el lazo de refinamiento, de manera que el costo de `lower_bound()` es en el peor caso $O(\log n)$. Esto representa una significante reducción con respecto al $O(n)$ que teníamos con las listas. Notar de paso que el algoritmo de búsqueda binaria no se puede aplicar a listas ya que estas no son contenedores de “acceso aleatorio”. Para vectores, acceder al elemento j -ésimo es una operación $O(1)$, mientras que para listas involucra recorrer toda la lista desde el comienzo hasta la posición entera j , lo cual es $O(j)$.

Como `p=find(k)` está basado en `lower_bound()` el costo de éste es también $O(\log n)$. Para `M[k]` tenemos el costo de `lower_bound()` por un lado pero también tenemos el lazo de las líneas 55–60 para desplazar las asignaciones, el cual es $O(n)$. Los resultados se sumarizan en la Tabla 2.6. Para `M[key]`

hemos separado los casos en que la clave era ya existente de cuando no existía. Notar la implementación por vectores ordenados es óptima en el caso de no necesitar eliminar asignaciones o insertar nuevas.

2.4.4. Definición de una relación de orden

Para implementar la correspondencia con contenedores ordenados es necesario contar con una relación de orden en conjunto universal de las claves. Para los tipos numéricos básicos se puede usar el orden usual y para las cadenas de caracteres el orden lexicográfico (alfabético). Para otros conjuntos universales como por ejemplo el conjunto de los pares de enteros la definición de una tal relación de orden puede no ser trivial. Sería natural asumir que

$$(2, 3) < (5, 6) \quad (2.22)$$

ya que cada uno de las componentes del primer par es menor que la del segundo par, pero no sabríamos como comparar $(2, 3)$ con $(5, 1)$. Primero definamos más precisamente qué es una “relación de orden”.

Definición: “ $<$ ” es una relación de orden en el conjunto C si,

1. $<$ es transitiva, es decir, si $a < b$ y $b < c$, entonces $a < c$.
2. Dados dos elementos cualquiera de C , una y sólo una de las siguientes afirmaciones es válida:

- $a < b$,
- $b < a$
- $a = b$.

Una posibilidad sería en comparar ciertas funciones escalares del par, como la suma, o la suma de los cuadrados. Por ejemplo definir que $(a, b) < (c, d)$ si y sólo sí $(a + b)^2 < (c + d)^2$. Una tal definición satisface 1, pero no 2, ya que por ejemplo los pares $(2, 3)$ y $(1, 4)$ no satisfacen ninguna de las tres condiciones.

Notar que una vez que se define un operador $<$, los operadores \leq , $>$ y \geq se pueden definir fácilmente en términos de $<$.

Una posibilidad para el conjunto de pares de enteros es la siguiente $(a, b) < (c, d)$ si $a < c$ o $a = c$ y $b < d$. Notar, que esta definición es equivalente a la definición lexicográfica para pares de letras si usamos el orden alfabético para comparar las letras individuales.

Probemos ahora la transitividad de esta relación. Sean $(a, b) < (c, d)$ y $(c, d) < (e, f)$, entonces hay cuatro posibilidades

- $a < c < e$
- $a < c = e$ y $d < f$
- $a = c < e$ y $b < d$
- $a = c = e$ y $b < d < f$

y es obvio que en cada una de ellas resulta ser $(a, b) < (e, f)$. También es fácil demostrar la condición 2.

Notar que esta relación de orden puede extenderse a cualquier conjunto universal compuesto de pares de conjuntos los cuales individualmente tienen una relación de orden, por ejemplo pares de la forma *(double,entero)* o *(entero,string)*. A su vez, aplicando recursivamente el razonamiento podemos ordenar n -tuplas de elementos que pertenezcan cada uno de ellos a conjuntos ordenados.

Capítulo 3

Arboles

Los árboles son contenedores que permiten organizar un conjunto de objetos en forma jerárquica. Ejemplos típicos son los diagramas de organización de las empresas o instituciones y la estructura de un sistema de archivos en una computadora. Los árboles sirven para representar fórmulas, la descomposición de grandes sistemas en sistemas más pequeños en forma recursiva y aparecen en forma sistemática en muchísimas aplicaciones de la computación científica. Una de las propiedades más llamativas de los árboles es la capacidad de acceder a muchísimos objetos desde un punto de partida o raíz en unos pocos pasos. Por ejemplo, en mi cuenta poseo unos 61,000 archivos organizados en unos 3500 directorios a los cuales puedo acceder con un máximo de 10 cambios de directorio (en promedio unos 5).

Sorprendentemente *no existe un contenedor STL de tipo árbol*, si bien varios de los otros contenedores (como conjuntos y correspondencias) están implementados internamente en términos de árboles. Esto se debe a que en la filosofía de las STL el árbol es considerado o bien como un subtipo del grafo o bien como una entidad demasiado básica para ser utilizada directamente por los usuarios.

3.1. Nomenclatura básica de árboles

Un árbol es una colección de elementos llamados “*nodos*”, uno de los cuales es la “*raíz*”. Existe una relación de parentesco por la cual cada nodo tiene un y sólo un “*padre*”, salvo la raíz que no lo tiene. El nodo es el concepto análogo al de “*posición*” en la lista, es decir un objeto abstracto que representa una posición en el mismo, no directamente relacionado con el “*elemento*” o “*etiqueta*” del nodo. Formalmente, el árbol se puede definir recursivamente de la siguiente forma (ver figura 3.1)

- Un nodo sólo es un árbol
- Si n es un nodo y T_1, T_2, \dots, T_k son árboles con raíces n_1, \dots, n_k entonces podemos construir un nuevo árbol que tiene a n como raíz y donde n_1, \dots, n_k son “*hijos*” de n .

También es conveniente postular la existencia de un “*árbol vacío*” que llamaremos Λ .

Ejemplo 3.1: Consideremos el árbol que representa los archivos en un sistema de archivos. Los nodos del árbol pueden ser directorios o archivos. En el ejemplo de la figura 3.2, la cuenta **anuser/** contiene 3 subdirectorios **docs/**, **programas/** y **juegos/**, los cuales a su vez contienen una serie de archivos. En este caso la relación entre nodos hijos y padres corresponde a la de pertenencia: un nodo a es hijo de otro b , si

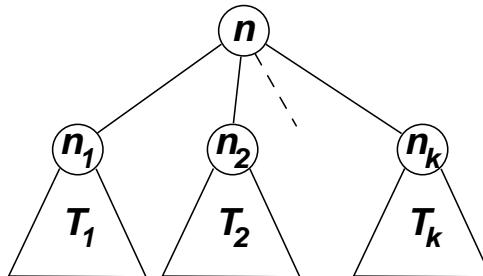


Figura 3.1: Construcción recursiva de un árbol

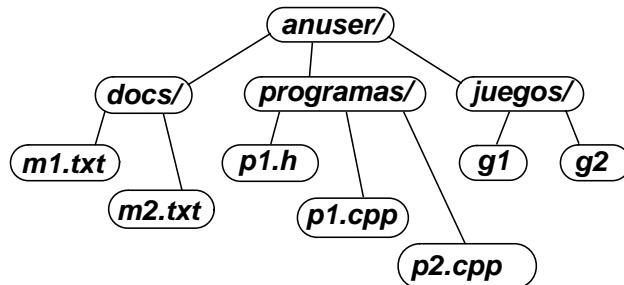


Figura 3.2: Árboles representando un sistema de archivos

el archivo *a* pertenece al directorio *b*. En otras aplicaciones la relación padre/hijo puede querer significar otra cosa.

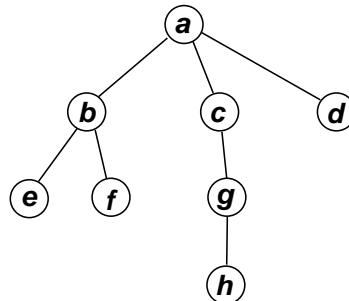


Figura 3.3: Ejemplo simple de árbol

Camino. Si n_1, n_2, \dots, n_k es una secuencia de nodos tales que n_i es padre de n_{i+1} para $i = 1 \dots k - 1$, entonces decimos que esta secuencia de nodos es un “camino” (“path”), de manera que {**anuser**, **docs**, **m2.txt**} es un camino, mientras que {**docs**, **anuser**, **programas**} no. (Coincidientemente en Unix se llama camino a la especificación completa de directorios que va desde el directorio raíz hasta un archivo, por ejemplo el camino correspondiente a **m2.txt** es **/anuser/docs/m2.txt**.) La “longitud” de un camino es igual al número de nodos en el camino menos uno, por ejemplo la longitud del camino {**anuser**, **docs**, **m2.txt**} es

2. Notar que siempre existe un camino de longitud 0 de un nodo a sí mismo.

Descendientes y antecesores. Si existe un camino que va del nodo a al b entonces decimos que a es antecesor de b y b es descendiente de a . Por ejemplo `m1.txt` es descendiente de **anuser** y **juegos** es antecesor de **g2**. Estrictamente hablando, un nodo es antecesor y descendiente de sí mismo ya que existe camino de longitud 0. Para diferenciar este caso trivial, decimos que a es descendiente (antecesor) propio de b si a es descendiente (antecesor) de b , pero $a \neq b$. En el ejemplo de la figura 3.3 a es antecesor propio de c, f y d ,

Hojas. Un nodo que no tiene hijos es una “hoja” del árbol. (Recordemos que, por contraposición el nodo que no tiene padre es único y es la raíz.) En el ejemplo, los nodos e, f, h y d son hojas.

3.1.0.0.1. Altura de un nodo. La altura de un nodo en un árbol es la máxima longitud de un camino que va desde el nodo a una hoja. Por ejemplo, el árbol de la figura la altura del nodo c es 2. La altura del árbol es la altura de la raíz. La altura del árbol del ejemplo es 3. Notar que, para cualquier nodo n

$$\text{altura}(n) = \begin{cases} 0; & \text{si } n \text{ es una hoja} \\ 1 + \max_{s=\text{hijo de } n} \text{altura}(s); & \text{si no lo es.} \end{cases} \quad (3.1)$$

3.1.0.0.2. Profundidad de un nodo. Nivel. La “profundidad” de un nodo es la longitud de único camino que va desde el nodo a la raíz. La profundidad del nodo g en el ejemplo es 2. Un “nivel” en el árbol es el conjunto de todos los nodos que están a una misma profundidad. El nivel de profundidad 2 en el ejemplo consta de los nodos e, f y g .

3.1.0.0.3. Nodos hermanos Se dice que los nodos que tienen un mismo parente son “hermanos” entre sí. Notar que no basta con que dos nodos estén en el mismo nivel para que sean hermanos. Los nodos f y g en el árbol de la figura 3.3 están en el mismo nivel, pero no son hermanos entre sí.

3.2. Orden de los nodos

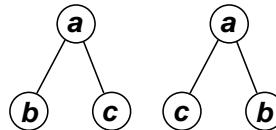


Figura 3.4: Arboles ordenados: el orden de los hijos es importante de manera que los árboles de la figura son diferentes.

En este capítulo, estudiamos árboles para los cuales el *orden* entre los hermanos es relevante. Es decir, los árboles de la figura 3.4 *son diferentes* ya que si bien a tiene los mismos hijos, están en diferente orden. Volviendo a la figura 3.3 decimos que el nodo c está a la derecha de b , o también que c es el hermano

derecho de b . También decimos que b es el “*hijo más a la izquierda*” de a . El orden entre los hermanos se propaga a los hijos, de manera que h está a la derecha de e ya que ambos son descendientes de c y b , respectivamente. A estos árboles se les llama “árboles ordenados orientados” (AOO).

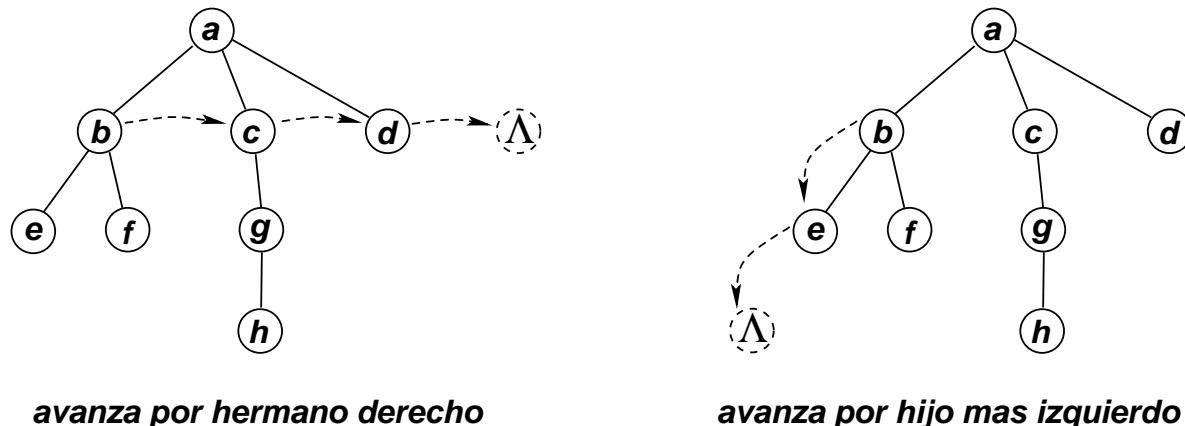


Figura 3.5: Direcciones posibles para avanzar en un árbol.

Podemos pensar al árbol como una lista bidimensional. Así como en las listas se puede avanzar linealmente desde el comienzo hacia el fin, en cada nodo del árbol podemos avanzar en dos direcciones (ver figura 3.5)

- Por el hermano derecho, de esta forma se recorre toda la lista de hermanos de izquierda a derecha.
- Por el hijo más izquierdo, tratando de descender lo más posible en profundidad.

En el primer caso el recorrido termina en el último hermano a la derecha. Por analogía con la posición `end()` en las listas, asumiremos que después del último hermano existe un nodo ficticio no dereferenciable. Igualmente, cuando avanzamos por el hijo más izquierdo, el recorrido termina cuando nos encontramos con una hoja. También asumiremos que el hijo más izquierdo de una hoja es un nodo ficticio no dereferenciable. Notar que, a diferencia de la lista donde hay una sola posición no dereferenciable (la posición `end()`), en el caso de los árboles puede haber más de una posiciones ficticias no dereferenciables, las cuales simbolizaremos con Λ cuando dibujamos el árbol. En la figura 3.6 vemos todas las posibles posiciones ficticias $\Lambda_1, \dots, \Lambda_8$ para el árbol de la figura 3.5. Por ejemplo, el nodo f no tiene hijos, de manera que genera la posición ficticia Λ_2 . Tampoco tiene hermano derecho, de manera que genera la posición ficticia Λ_3 .

3.2.1. Particionamiento del conjunto de nodos

Ahora bien, dados dos nodos cualquiera m y n consideremos sus caminos a la raíz. Si m es descendiente de n entonces el camino de n está incluido en el de m o viceversa. Por ejemplo, el camino de c , que es a, c , está incluido en el de h, a, c, g, h , ya que c es antecesor de h . Si entre m y n no hay relación de descendiente o antecesor, entonces los caminos se deben bifurcar necesariamente en un cierto nivel. *El orden entre m y n es el orden entre los antecesores a ese nivel*. Esto demuestra que, dados dos nodos cualquiera m y n sólo una de las siguientes afirmaciones puede ser cierta

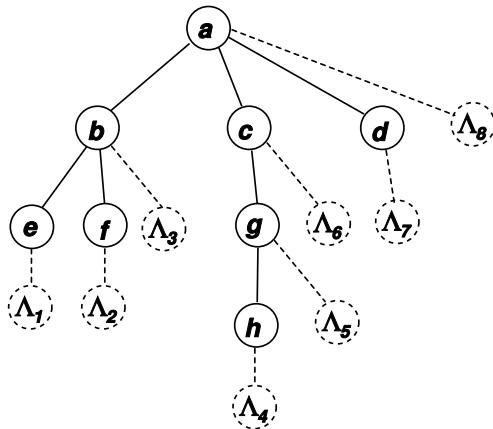


Figura 3.6: Todas las posiciones no dereferenciables de un árbol.

- $m = n$
- m es antecesor propio de n
- n es antecesor propio de m
- m está a la derecha de n
- n está a la derecha de m

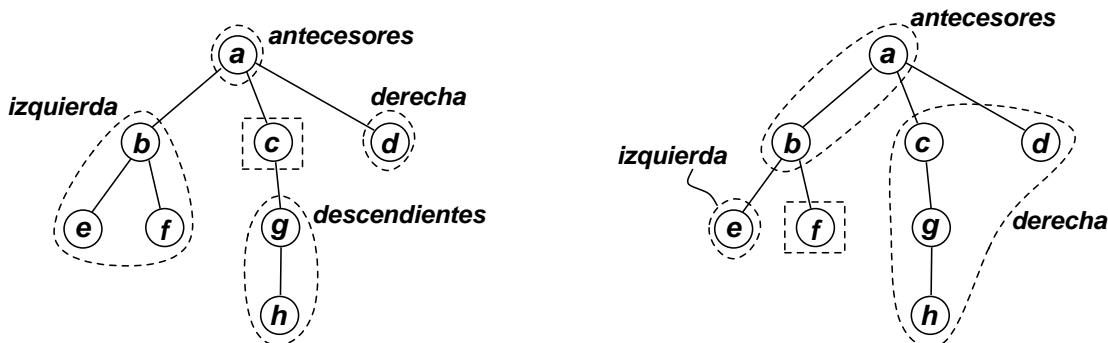


Figura 3.7: Clasificación de los nodos de un árbol con respecto a un nodo. Izquierda: con respecto al nodo c . Derecha: con respecto al nodo f .

Dicho de otra manera, dado un nodo n el conjunto N de todos los nodos del árbol se puede dividir en 5 conjuntos *disjuntos* a saber

$$N = \{n\} \cup \{\text{descendientes}(n)\} \cup \{\text{antecesores}(n)\} \cup \{\text{derecha}(n)\} \cup \{\text{izquierda}(n)\} \quad (3.2)$$

En la figura 3.7 vemos la partición inducida para los nodos c y f . Notar que en el caso del nodo f el conjunto de los descendientes es vacío (\emptyset).

3.2.2. Listado de los nodos de un árbol

3.2.2.1. Orden previo

Existen varias formas de recorrer un árbol listando los nodos del mismo, generando una lista de nodos. Dado un nodo n con hijos n_1, n_2, \dots, n_m , el “listado en orden previo” (“preorder”) del nodo n que denotaremos como $\text{oprev}(n)$ se puede definir recursivamente como sigue

$$\text{oprev}(n) = (n, \text{oprev}(n_1), \text{oprev}(n_2), \dots, \text{oprev}(n_m)) \quad (3.3)$$

Además el orden previo del árbol vacío es la lista vacía: $\text{oprev}(\Lambda) = ()$.

Consideremos por ejemplo el árbol de la figura 3.3. Aplicando recursivamente (3.3) tenemos

$$\begin{aligned} \text{oprev}(a) &= a, \text{oprev}(b), \text{oprev}(c), \text{oprev}(d) \\ &= a, b, \text{oprev}(e), \text{oprev}(f), c, \text{oprev}(g), d \\ &= a, b, \text{oprev}(e), \text{oprev}(f), c, \text{oprev}(g), d \\ &= a, b, e, f, c, g, \text{oprev}(h), d \\ &= a, b, e, f, c, g, h, d \end{aligned} \quad (3.4)$$

Una forma más visual de obtener el listado en orden previo es como se muestra en la figura 3.8. Recorremos el borde del árbol en el sentido contrario a las agujas del reloj, partiendo de un punto imaginario a la izquierda del nodo raíz y terminando en otro a la derecha del mismo, como muestra la línea de puntos. Dado un nodo como el b el camino pasa cerca de él en varios puntos (3 en el caso de b , marcados con pequeños números en el camino). El orden previo consiste en *listar los nodos una sola vez, la primera vez que el camino pasa cerca del árbol*. Así en el caso del nodo b , este se lista al pasar por 1. Queda como ejercicio para el lector verificar el orden resultante coincide con el dado en (3.4).

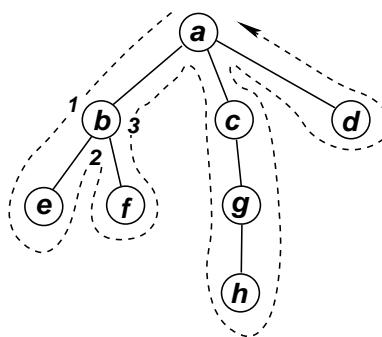


Figura 3.8: Recorrido de los nodos de un árbol en orden previo.

3.2.2.2. Orden posterior

El “orden posterior” (“postorder”) se puede definir en forma análoga al orden previo pero reemplazando (3.3) por

$$\text{opost}(n) = (\text{opost}(n_1), \text{opost}(n_2), \dots, \text{opost}(n_m), n) \quad (3.5)$$

y para el árbol del ejemplo resulta ser

$$\begin{aligned}
 \text{opost}(a) &= \text{opost}(b), \text{opost}(c), \text{opost}(d), a \\
 &= \text{opost}(e), \text{opost}(f), b, \text{opost}(g), c, d, a \\
 &= e, f, b, \text{opost}(h), g, c, d, a \\
 &= e, f, b, h, g, c, d, a
 \end{aligned} \tag{3.6}$$

Visualmente se puede realizar de dos maneras.

- Recorriendo el borde del árbol igual que antes (esto es en sentido contrario a las agujas del reloj), listando el nodo *la última vez que el recorrido pasa por al lado del mismo*. Por ejemplo el nodo *b* sería listado al pasar por el punto 3.
- Recorriendo el borde en el sentido opuesto (es decir en el mismo sentido que las agujas del reloj), y listando los nodos la *primera vez* que el camino pasa cerca de ellos. Una vez que la lista es obtenida, *invertimos la lista*. En el caso de la figura el recorrido en sentido contrario daría (*a, d, c, g, h, b, f, e*). Al invertirlo queda como en (3.6).

Existe otro orden que se llama “simétrico”, pero este sólo tiene sentido en el caso de árboles binarios, así que no será explicado aquí.

3.2.2.3. Orden posterior y la notación polaca invertida

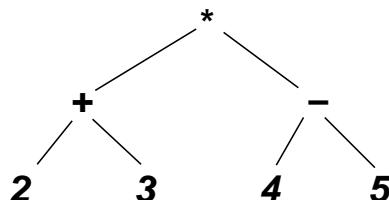


Figura 3.9: Árbol correspondiente a la expresión matemática $(2 + 3) * (4 - 5)$

Las expresiones matemáticas como $(2 + 4) * (4 - 5)$ se pueden poner en forma de árbol como se muestra en la figura 3.9. La regla es

- Para operadores binarios de la forma $a + b$ se pone el operador (+) como padre de los dos operandos (*a* y *b*). Los operandos pueden ser a su vez expresiones. Funciones binarias como *rem(10, 5)* (*rem* es la función resto) se tratan de esta misma forma.
- Operadores unarios (como -3) y funciones (como $\sin(20)$) se escriben poniendo el operando como hijo del operador o función.
- Operadores asociativos con más de dos operandos (como $1 + 3 + 4 + 9$) deben asociarse de a 2 (como en $((1 + 3) + 4) + 9$).

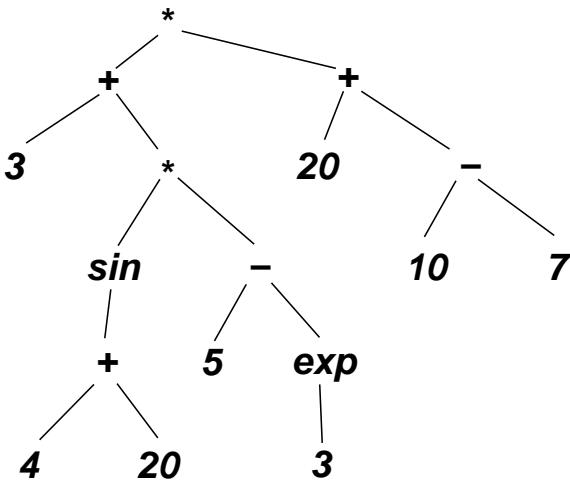


Figura 3.10: Arbol correspondiente a la expresión matemática (3.7)

De esta forma, expresiones complejas como

$$(3 + \sin(4 + 20) * (5 - e^3)) * (20 + 10 - 7) \quad (3.7)$$

pueden ponerse en forma de árbol, como en la figura 3.10.

El listado en orden posterior de este árbol coincide con la notación polaca invertida (RPN) discutida en la sección §2.2.1.

3.2.3. Notación Lisp para árboles

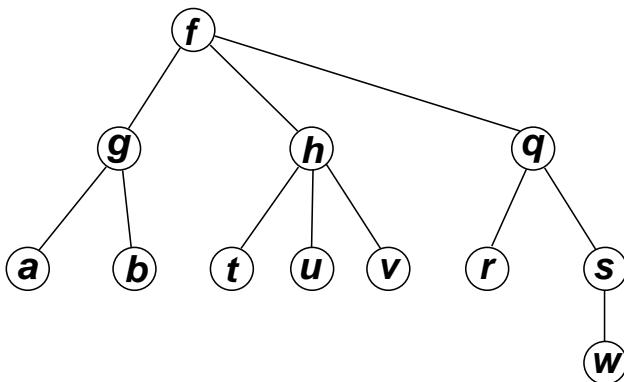


Figura 3.11: Arbol correspondiente a una expresión de llamadas a funciones.

Una expresión matemática compleja que involucra funciones cuyos argumentos son a su vez llamadas a otras funciones puede ponerse en forma de árbol. Por ejemplo, para la expresión

$$f(g(a, b), h(t, u, v), q(r, s(w))) \quad (3.8)$$

corresponde un árbol como el de la figura 3.11. En este caso cada función es un nodo cuyos hijos son los argumentos de la función. En Lisp la llamada a un función $f(x, y, z)$ se escribe de la forma **(f x y z)**, de manera que la llamada anterior se escribiría como

```
1 (f (g a b) (h t u v) (q r (s w)))
```

Para expresiones más complejas como la de (3.7), la forma Lisp para el árbol (figura 3.10) da el código Lisp correspondiente

```
1 (* (+ 3 (* (sin (+ 4 20)) (- 5 (exp 3)))) (+ 20 (- 10 7)))
```

Esta notación puede usarse para representar árboles en forma general, de manera que, por ejemplo, el árbol de la figura 3.3 puede ponerse, en notación Lisp como

```
1 (a (b e f) (c (g h)) d)
```

Notemos que el orden de los nodos es igual al del orden previo. Se puede dar una definición precisa de la notación Lisp como para el caso de los órdenes previo y posterior:

$$\text{lisp}(n) = \begin{cases} \text{si } n \text{ es una hoja: } & n \\ \text{caso contrario: } & (n \text{ lisp}(n_1) \text{ lisp}(n_2) \dots \text{ lisp}(n_m)) \end{cases} \quad (3.9)$$

donde $n_1 \dots n_m$ son los hijos del nodo n .

Es evidente que existe una relación unívoca entre un árbol y su notación Lisp. Los paréntesis dan la estructura adicional que permite establecer la relación unívoca. La utilidad de esta notación es que permite fácilmente escribir árboles en una línea de texto, sin tener que recurrir a un gráfico. Basado en esta notación, es fácil escribir una función que convierta un árbol a una lista y viceversa.

También permite “serializar” un árbol, es decir, convertir una estructura “bidimensional” como es el árbol, en una estructura unidimensional como es una lista. El serializar una estructura compleja permite almacenarla en disco o comunicarla a otro proceso por mensajes.

3.2.4. Reconstrucción del árbol a partir de sus órdenes

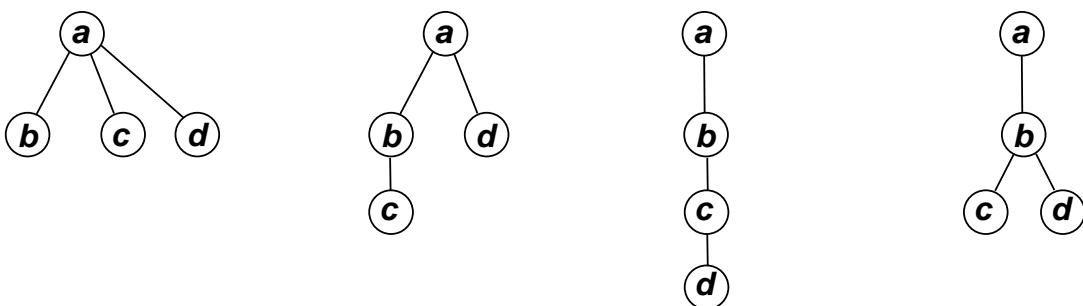


Figura 3.12: Los cuatro árboles de la figura tienen el mismo orden previo (a, b, c, d)

Podemos preguntarnos si podemos reconstruir un árbol a partir de su listado en orden previo. Si tal cosa fuera posible, entonces sería fácil representar árboles en una computadora, almacenando dicha lista. Sin

Figura 3.13: Los cuatro árboles de la figura tienen el mismo orden posterior (b, c, d, a)

embargo puede verse fácilmente que árboles distintos pueden dar el mismo orden previo (ver figura 3.12) o posterior (ver figura 3.13).

Sin embargo, es destacable que, dado el orden previo y posterior de un árbol sí se puede reconstruir el árbol. Primero notemos que (3.3) implica que el orden de los nodos queda así

$$\text{oprev}(n) = (n, n_1, \text{descendientes}(n_1), n_2, \text{descendientes}(n_2), \dots, n_m, \text{descendientes}(n_m)) \quad (3.10)$$

mientras que

$$\text{opost}(n) = (\text{descendientes}(n_1), n_1, \text{descendientes}(n_2), n_2, \dots, \text{descendientes}(n_m), n_m, n). \quad (3.11)$$

Notemos que el primer nodo listado en orden previo es la raíz, y el segundo su primer hijo n_1 . Todos los nodos que están *después* de n_1 en orden previo pero *antes* de n_1 en orden posterior son los descendientes de n_1 . Prestar atención a que el orden en que aparecen los descendientes de un dado nodo en (3.10) puede no coincidir con el que aparecen en (3.11). De esta forma podemos deducir cuales son los descendientes de n_1 . El nodo siguiente, en orden previo, a todos los descendientes de n_1 debe ser el segundo hijo n_2 . Todos los nodos que están después de n_2 en orden previo pero antes de n_2 en orden posterior son descendientes de n_2 . Así siguiendo podemos deducir cuales son los hijos de n y cuales son descendientes de cada uno de ellos.

Ejemplo 3.2: *Consigna:* Encontrar el árbol A tal que

$$\begin{aligned} \text{orden previo} &= (z, w, a, x, y, c, m, t, u, v) \\ \text{orden posterior} &= (w, x, y, a, t, u, v, m, c, z) \end{aligned} \quad (3.12)$$

Solución: De los primeros dos nodos en orden previo se deduce que z debe ser el nodo raíz y w su primer hijo. No hay nodos antes de w en orden posterior de manera que w no tiene hijos. El nodo siguiente a w en orden previo es a que por lo tanto debe ser el segundo hijo de z . Los nodos que están antes de a pero después de w en orden posterior son x e y , de manera que estos son descendientes de a . De la misma forma se deduce que el tercer hijo de z es c y que sus descendientes son m, t, u, v . A esta altura podemos esbozar un dibujo del árbol como se muestra en la figura 3.14. Las líneas de puntos indican que, por ejemplo, sabemos que m, t, u, v son descendientes de c , pero todavía no conocemos la estructura de ese subárbol.

Ahora bien, para hallar la estructura de los descendientes de c volvemos a (3.12), y vemos que

$$\begin{aligned} \text{oprev}(c) &= (c, m, t, u, v) \\ \text{opost}(c) &= (t, u, v, m, c) \end{aligned} \quad (3.13)$$

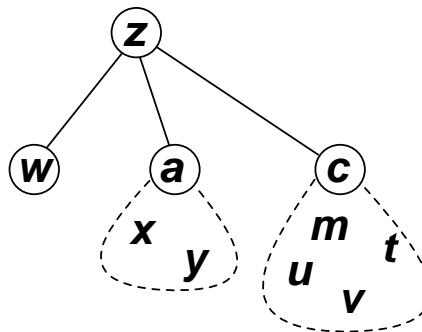


Figura 3.14: Etapa parcial en la reconstrucción del árbol del ejemplo 3.2

de manera que el procedimiento se puede aplicar recursivamente para hallar los hijos de c y sus descendientes y así siguiendo hasta reconstruir todo el árbol. El árbol correspondiente resulta ser, en este caso el de la figura 3.15, o en notación Lisp (`(z w (a x y) (c (m t u v)))`).

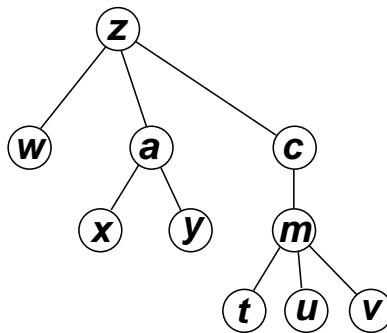


Figura 3.15: Árbol reconstruido a partir de los órdenes previo y posterior especificados en (3.12)

3.3. Operaciones con árboles

3.3.1. Algoritmos para listar nodos

Implementar un algoritmo para recorrer los nodos de un árbol es relativamente simple debido a su naturaleza intrínsecamente recursiva, expresada en (3.3). Un posible algoritmo puede observarse en el código 3.1. Si bien el algoritmo es genérico hemos usado ya algunos conceptos familiares de las STL, por ejemplo las posiciones se representan con una clase **iterator**. Recordar que para árboles se puede llegar al “*fin del contenedor*”, es decir los nodos Λ , en más de un punto del contenedor. El código genera una lista de elementos L con los elementos de T en orden previo.

```
1 void preorder(tree &T, iterator n, list &L) {
2     L.insert(L.end(), /* valor en el nodo 'n'... */);
3     iterator c = /* hijo mas izquierdo de n... */;
```

```
4 while /* 'c' no es 'Lambda'... */ {  
5   preorder(T,c,L);  
6   c /* hermano a la derecha de c... */;  
7 }  
8 }
```

Código 3.1: Algoritmo para recorrer un árbol en orden previo. [Archivo: preorder.cpp]

```
1 void postorder(tree &T, iterator n, list &L) {  
2   iterator c /* hijo mas izquierdo de n... */;  
3   while (c != T.end()) {  
4     postorder(T,c,L);  
5     c /* hermano a la derecha de c... */;  
6   }  
7   L.insert(L.end(), /* valor en el nodo 'n'... */);  
8 }
```

Código 3.2: Algoritmo para recorrer un árbol en orden posterior. [Archivo: postorder.cpp]

```
1 void lisp_print(tree &T, iterator n) {  
2   iterator c /* hijo mas izquierdo de n... */;  
3   if /* 'c' es 'Lambda'... */ {  
4     cout << /* valor en el nodo 'n'... */;  
5   } else {  
6     cout << "(" << /* valor de 'n' ... */;  
7     while /* 'c' no es 'Lambda'... */ {  
8       cout << " ";  
9       lisp_print(T,c);  
10      c /* hermano derecho de c... */;  
11    }  
12    cout << ")";  
13  }  
14 }
```

Código 3.3: Algoritmo para imprimir los datos de un árbol en notación Lisp. [Archivo: lispprint.cpp]

En el código 3.2 se puede ver un código similar para generar la lista con el orden posterior, basada en (3.5). Similarmente, en código 3.3 puede verse la implementación de una rutina que imprime la notación Lisp de un árbol.

3.3.2. Inserción en árboles

Para construir árboles necesitaremos rutinas de inserción supresión de nodos. Como en las listas, las operaciones de inserción toman un elemento y una posición e insertan el elemento en esa posición en el árbol.

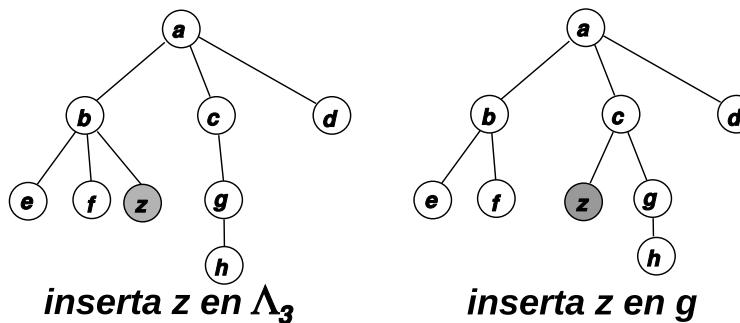


Figura 3.16: Resultado de insertar el elemento z en el árbol de la figura 3.6. *Izquierda:* Inserta z en la posición Λ_3 . *Derecha:* Inserta z en la posición g .

- Cuando insertamos un nodo en una posición Λ entonces simplemente el elemento pasa a generar un nuevo nodo en donde estaba el nodo ficticio Λ . Por ejemplo, el resultado de insertar el elemento z en el nodo Λ_3 de la figura 3.6 se puede observar en la figura 3.16 (izquierda). (*Observación:* En un abuso de notación estamos usando las mismas letras para denotar el contenido del nodo que el nodo en sí.)
- Cuando insertamos un nodo en una posición dereferenciable, entonces simplemente el elemento pasa a generar un nuevo nodo hoja en el lugar en esa posición, tal como operaría la operación de inserción del TAD lista en la lista de hijos. Por ejemplo, consideremos el resultado de insertar el elemento z en la posición g . El padre de g es c y su lista de hijos es (g) . Al insertar z en la posición de g la lista de hijos pasa a ser (z, g) , de manera que z pasa a ser el hijo más izquierdo de c (ver figura 3.16 derecha).
- Así como en listas `insert(p, x)` invalida las posiciones después de p (inclusive), en el caso de árboles, una inserción en el nodo n invalida las posiciones que son descendientes de n y que están a la derecha de n .

3.3.2.1. Algoritmo para copiar árboles

```

1 iterator tree_copy(tree &T, iterator nt,
2                     tree &Q, iterator nq) {
3     nq = /* nodo resultante de insertar el
4           elemento de 'nt' en 'nq' ... */;
5     iterator
6         ct = /* hijo mas izquierdo de 'nt' ...*/,
7         cq = /* hijo mas izquierdo de 'nq' ... */;
8     while (/* 'ct' no es 'Lambda'... */) {
9         cq = tree_copy(T, ct, Q, cq);
10        ct = /* hermano derecho de 'ct'... */;
11        cq = /* hermano derecho de 'cq'... */;
12    }
13    return nq;
14 }
```

Código 3.4: Seudocódigo para copiar un árbol. [Archivo: treecpy.cpp]

Figura 3.17: Algoritmo para copiar árboles.

Con estas operaciones podemos escribir el seudocódigo para una función que copia un árbol (ver código 3.4). Esta función copia el subárbol del nodo **nt** en el árbol **T** en la posición **nq** en el árbol **Q** y devuelve la posición de la raíz del subárbol insertado en **Q** (actualiza el nodo **nq** ya que después de la inserción es inválido). La función es recursiva, como lo son la mayoría de las operaciones no triviales sobre árboles. Consideremos el algoritmo aplicado al árbol de la figura 3.17 a la izquierda. Primero inserta el elemento que está en **nt** en la posición **nq**. Luego va copiando cada uno de los subárboles de los hijos de **nq** como hijos del nodo **nt**. El nodo **ct** itera sobre los hijos de **nt** mientras que **cq** lo hace sobre los hijos de **nq**. Por ejemplo, si consideramos la aplicación del algoritmo a la copia del árbol de la figura 3.17 a la izquierda, concentrémonos en la copia del subárbol del nodo *c* del árbol *T* al *Q*.

Cuando llamamos a **tree_copy(T, nt, Q, nq)**, **nt** es *c* y **nq** es Λ_1 , (mostrado como Q^1 en la figura). La línea 3 copia la raíz del subárbol que en este caso es el nodo *c* insertándolo en Λ_1 . Después de esta línea, el árbol queda como se muestra en la etapa Q^2 . Como en la inserción en listas, la línea actualiza la posición **nq**, la cuál queda apuntando a la posición que contiene a *c*. Luego **ct** y **nt** toman los valores de los hijos más izquierdos, a saber *r* y Λ_2 . Como **ct** no es Λ entonces el algoritmo entra en el lazo y la línea 9 copia todo el subárbol de *r* en Λ_2 , quedando el árbol como en Q^3 . De paso, la línea actualiza el iterador **cq**, de manera que **ct** y **cq** quedan apuntando a los dos nodos *r* en sus respectivos árboles. Notar que en este análisis no consideramos la llamada recursiva a **tree_copy()** sino que simplemente asumimos que estamos analizando la instancia específica de llamada a **tree_copy** donde **nt** es *c* y no aquéllas llamadas generadas por esta instancia. En las líneas 10–11, los iteradores **ct** y **cq** son avanzados, de manera que quedan apuntando a *g* y Λ_2 . En la siguiente ejecución del lazo la línea 9 copiará todo el subárbol de *g* a Λ_3 . El proceso se detiene después de copiar el subárbol de *w* en cuyo caso **ct** obtendrá en la línea 10 un nodo Λ y la función termina, retornando el valor de **nq** actualizado.

```

1 iterator mirror_copy(tree &T, iterator nt,
2                     tree &Q, iterator nq) {
3     nq = /* nodo resultante de insertar
4           el elemento de 'nt' en 'nq' */;
5     iterator
6     ct = /* hijo mas izquierdo de 'nt' ...*/,
7     cq = /* hijo mas izquierdo de 'nq' ...*/;
8     while (/* 'ct' no es 'Lambda'... */) {
9         cq = mirror_copy(T,ct,Q,cq);

```

```
10     ct = /* hermano derecho de 'ct' . . . */;
11 }
12 return nq;
13 }
```

Código 3.5: Seudocódigo para copiar un árbol en espejo. [Archivo: mirrorcpy.cpp]

Con menores modificaciones la función puede copiar un árbol en forma espejada, es decir, de manera que todos los nodos hermanos queden en orden inverso entre sí. Para eso basta con *no avanzar* el iterator **cq** donde se copian los subárboles, es decir eliminar la línea 11. Recordar que si en una lista se van insertando valores en una posición *sin avanzarla*, entonces los elementos quedan ordenados *en forma inversa* a como fueron ingresados. El algoritmo de copia espejo **mirror_copy()** puede observarse en el código 3.5.

Con algunas modificaciones el algoritmo puede ser usado para obtener la copia de un árbol reordenando las hojas en un orden arbitrario, por ejemplo dejándolas ordenadas entre sí.

3.3.3. Supresión en árboles

Al igual que en listas, solo se puede suprimir en posiciones dereferenciables. En el caso de suprimir en un nodo hoja, solo se elimina el nodo. Si el nodo tiene hijos, eliminarlo equivale a eliminar todo el subárbol correspondiente. Como en listas, eliminando un nodo devuelve la posición del hermano derecho que llena el espacio dejado por el nodo eliminado.

```
1 iterator_t prune_odd(tree &T, iterator_t n) {
2     if /*valor de 'n' . . . */ % 2)
3         /* elimina el nodo 'n' y refresca. . . */;
4     else {
5         iterator_t c =
6             /* hijo mas izquierdo de 'n' . . . */;
7         while /*'c' no es 'Lambda' . . . */
8             c = prune_odd(T, c);
9         n = /* hermano derecho de 'n' . . . */;
10    }
11    return n;
12 }
```

Código 3.6: Algoritmo que elimina los nodos de un árbol que son impares, incluyendo todo su subárbol [Archivo: pruneodd.cpp]

Por ejemplo consideremos el algoritmo **prune_odd** (ver código 3.6) que “poda” un árbol, eliminando todos los nodos de un árbol que son impares *incluyendo sus subárboles*. Por ejemplo, si $T=(6 \ (2 \ 3 \ 4) \ (5 \ 8 \ 10))$. Entonces después de aplicar **prune_odd** tenemos $T=(6 \ (2 \ 4))$. Notar que los nodos 8 y 10 han sido eliminados ya que, si bien son pares, pertenecen al subárbol del nodo 5, que es impar. Si el elemento del nodo es impar todo el subárbol del nodo es eliminado en la línea 3, caso contrario los hijos son podados aplicándoles recursivamente la función. Notar que tanto si el valor contenido en **n** es impar como si no, **n**

avanza una posición dentro de **prune_odd**, ya sea al eliminar el nodo en la línea 3 o al avanzar explícitamente en la línea 9.

3.3.4. Operaciones básicas sobre el tipo árbol

Los algoritmos para el listado presentados en las secciones previas, sugieren las siguientes operaciones abstractas sobre árboles

- Dado un nodo (posición o iterator sobre el árbol), obtener su hijo más izquierdo. (Puede retornar una posición Λ).
- Dado un nodo obtener su hermano derecho. (Puede retornar una posición Λ).
- Dada una posición, determinar si es Λ o no.
- Obtener la posición de la raíz del árbol.
- Dado un nodo obtener una referencia al dato contenido en el nodo.
- Dada una posición (dereferenciable o no) y un dato, insertar un nuevo nodo con ese dato en esa posición.
- Borrar un nodo y todo su subárbol correspondiente.

3.4. Interfaz básica para árboles

```
1 class iterator_t {
2     /* . . . */
3 public:
4     iterator_t lchild();
5     iterator_t right();
6 };
7
8 class tree {
9     /* . . . */
10 public:
11     iterator_t begin();
12     iterator_t end();
13     elem_t &retrieve(iterator_t p);
14     iterator_t insert(iterator_t p, elem_t t);
15     iterator_t erase(iterator_t p);
16     void clear();
17     iterator_t splice(iterator_t to, iterator_t from);
18 };
```

Código 3.7: Interfaz básica para árboles. [Archivo: treebas1.h]

Una interfaz básica, parcialmente compatible con la STL puede observarse en el código 3.7. Como con las listas y correspondencias tenemos una clase **iterator_t** que nos permite iterar sobre los nodos del árbol, tanto dereferenciables como no dereferenciables. En lo que sigue **T** es un árbol, **p**, **q** y **r** son nodos (iterators) y **x** es un elemento de tipo **elem_t**.

- **`q = p.lchild()`**: Dada una posición dereferenciable `p` retorna la posición del hijo más izquierdo (“*left-most child*”). La posición retornada puede ser dereferenciable o no.
- **`q = p.right()`**: Dada una posición dereferenciable `p` retorna la posición del hermano derecho. La posición retornada puede ser dereferenciable o no.
- **`T.end()`**: retorna un iterator no dereferenciable.
- **`p=T.begin()`**: retorna la posición del comienzo del árbol, es decir la raíz. Si el árbol está vacío, entonces retorna `end()`.
- **`x = T.retrieve(p)`**: Dada una posición dereferenciable retorna una referencia al valor correspondiente.
- **`q = T.insert(p,x)`**: Inserta un nuevo nodo en la posición `p` conteniendo el elemento `x`. `p` puede ser dereferenciable o no. Retorna la posición del nuevo elemento insertado.
- **`p = T.erase(p)`**: Elimina la posición dereferenciable `p` y todo el subárbol de `p`.
- **`T.clear()`**: Elimina todos los elementos del árbol (equivale a `T.erase(T.begin())`).

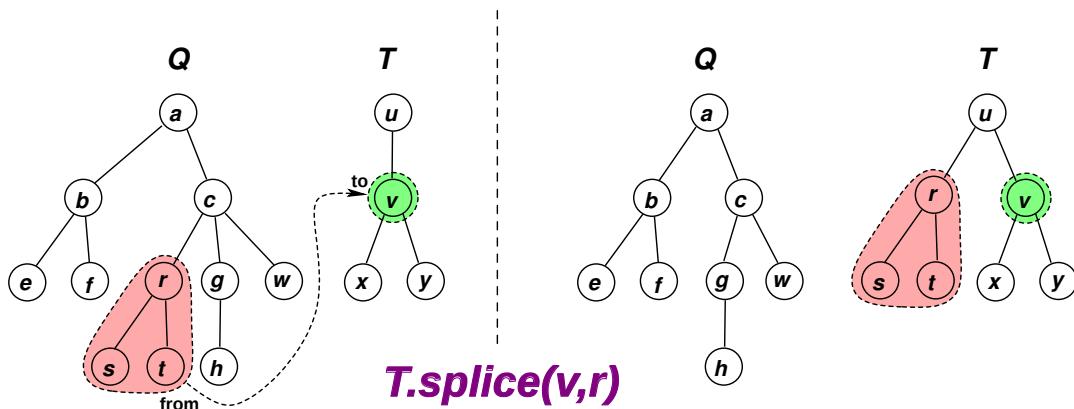


Figura 3.18: Operación splice.

- **`T.splice(to, from)`**: Elimina todo el subárbol del nodo dereferenciable `from` y lo inserta en el nodo (dereferenciable o no) `to`. `to` y `from` no deben tener relación de antecesor o descendiente y pueden estar en diferentes árboles. Por ejemplo, consideremos el ejemplo de la figura 3.18 (izquierda), donde deseamos mover todo el subárbol del nodo `r` en el árbol `T` a la posición del nodo `v` en el árbol `Q`. El resultado es como se muestra en la parte izquierda de la figura y se obtiene con el llamado `T.splice(r,v)`.

```

1 void preorder(tree &T, iterator_t n, list<int> &L) {
2     L.insert(L.end(), T.retrieve(n));
3
4     iterator_t c = n.lchild();
5     while (c!=T.end()) {
6         preorder(T,c,L);
7         c = c.right();
8     }
9 }
10 void preorder(tree &T, list<int> &L) {

```

```
11 if (T.begin()==T.end()) return;
12 preorder(T,T.begin(),L);
13 }
14
15 //---:---<*>---:---<*>---:---<*>---:---<*>
16 void postorder(tree &T,iterator_t n,list<int> &L) {
17 iterator_t c = n.lchild();
18 while (c!=T.end()) {
19 postorder(T,c,L);
20 c = c.right();
21 }
22 L.insert(L.end(),T.retrieve(n));
23 }
24 void postorder(tree &T,list<int> &L) {
25 if (T.begin()==T.end()) return;
26 postorder(T,T.begin(),L);
27 }
28
29 //---:---<*>---:---<*>---:---<*>---:---<*>
30 void lisp_print(tree &T,iterator_t n) {
31 iterator_t c = n.lchild();
32 if (c==T.end()) cout << T.retrieve(n);
33 else {
34 cout << "(" << T.retrieve(n);
35 while (c!=T.end()) {
36 cout << " ";
37 lisp_print(T,c);
38 c = c.right();
39 }
40 cout << ")";
41 }
42 }
43 void lisp_print(tree &T) {
44 if (T.begin()!=T.end()) lisp_print(T,T.begin());
45 }
46
47 //---:---<*>---:---<*>---:---<*>---:---<*>
48 iterator_t tree_copy(tree &T,iterator_t nt,
49 tree &Q,iterator_t nq) {
50 nq = Q.insert(nq,T.retrieve(nt));
51 iterator_t
52 ct = nt.lchild(),
53 cq = nq.lchild();
54 while (ct!=T.end()) {
55 cq = tree_copy(T,ct,Q,cq);
56 ct = ct.right();
57 cq = cq.right();
58 }
59 return nq;
60 }
61
62 void tree_copy(tree &T,tree &Q) {
63 if (T.begin() != T.end())
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
827
828
829
829
830
831
832
833
834
835
836
837
837
838
839
839
840
841
842
843
844
845
846
847
847
848
849
849
850
851
852
853
854
855
856
857
857
858
859
859
860
861
862
863
864
865
866
866
867
868
868
869
869
870
871
872
873
874
875
876
876
877
878
878
879
879
880
881
882
883
884
885
886
886
887
888
888
889
889
890
891
892
893
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
904
905
906
907
907
908
909
909
910
911
912
913
914
915
915
916
917
917
918
918
919
919
920
921
922
923
924
925
925
926
927
927
928
928
929
929
930
931
932
933
934
935
935
936
937
937
938
938
939
939
940
941
942
943
944
944
945
946
946
947
947
948
948
949
949
950
951
952
953
954
954
955
956
956
957
957
958
958
959
959
960
961
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
971
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
981
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1620
1621
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1630
1631
1631
1632
1632
1633
1633
1634
1634
1635
1635
1636
1636
1637
1637
1638
1638
1639
1639
1640
1640

```

```

64     tree_copy(T,T.begin(),Q,Q.begin());
65 }
66
67 //---:---<*>---:---<*>---:---<*>---:---<*>
68 iterator_t mirror_copy(tree &T,iterator_t nt,
69                         tree &Q,iterator_t nq) {
70     nq = Q.insert(nq,T.retrieve(nt));
71     iterator_t
72         ct = nt.lchild(),
73         cq = nq.lchild();
74     while (ct != T.end()) {
75         cq = mirror_copy(T,ct,Q,cq);
76         ct = ct.right();
77     }
78     return nq;
79 }
80
81 void mirror_copy(tree &T,tree &Q) {
82     if (T.begin() != T.end())
83         mirror_copy(T,T.begin(),Q,Q.begin());
84 }
85
86 //---:---<*>---:---<*>---:---<*>---:---<*>
87 iterator_t prune_odd(tree &T,iterator_t n) {
88     if (T.retrieve(n) % 2) n = T.erase(n);
89     else {
90         iterator_t c = n.lchild();
91         while (c != T.end()) c = prune_odd(T,c);
92         n = n.right();
93     }
94     return n;
95 }
96
97 void prune_odd(tree &T) {
98     if (T.begin() != T.end()) prune_odd(T,T.begin());
99 }
```

Código 3.8: Diversos algoritmos sobre árboles con la interfaz básica. [Archivo: treetools.cpp]

Los algoritmos **preorder**, **postorder**, **lisp_print**, **tree_copy**, **mirror_copy** y **prune_odd** descriptos en las secciones previas se encuentran implementados con las funciones de la interfaz básica en el código 3.8.

3.4.1. Listados en orden previo y posterior y notación Lisp

El listado en orden previo es simple. Primero inserta, el elemento del nodo **n** en el fin de la lista **L**. Notar que para obtener el elemento se utiliza el método **retrieve**. Luego se hace que **c** apunte al hijo más izquierdo de **n** y se va aplicando **preorder()** en forma recursiva sobre los hijos **c**. En la línea 7 se actualiza **c** de manera que recorra la lista de hijos de **n**. La función **postorder** es completamente análoga, sólo que el elemento de **n** es agregado *después* de los órdenes posteriores de los hijos.

3.4.2. Funciones auxiliares para recursión y sobrecarga de funciones

En general estas funciones recursivas se escriben utilizando una función auxiliar. En principio uno querría llamar a la función como `preorder(T)` asumiendo que se aplica al nodo raíz de `T`. Pero para después poder aplicarlo en forma recursiva necesitamos agregar un argumento adicional que es un nodo del árbol. Esto lo podríamos hacer usando una función recursiva adicional `preorder_aux(T, n)`. Finalmente, haríamos que `preorder(T)` llame a `preorder_aux`:

```
1 void preorder_aux(tree &T, iterator_t n, list<int> &L) {  
2     /* ... */  
3 }  
4 void preorder(tree &T, list<int> &L) {  
5     preorder_aux(T, T.begin(), L);  
6 }
```

Pero como C++ admite “sobrecarga del nombre de funciones”, no es necesario declarar la función auxiliar con un nombre diferente. Simplemente hay dos funciones `preorder()`, las cuales se diferencian por el número de argumentos.

A veces se dice que la función `preorder(T)` actúa como un “wrapper” (“envoltorio”) para la función `preorder(T, n)`, que es la que hace el trabajo real. `preorder(T)` sólo se encarga de pasarle los parámetros correctos a `preorder(T, n)`. De paso podemos usar el wrapper para realizar algunos chequeos como por ejemplo verificar que el nodo `n` no sea Λ . Si un nodo Λ es pasado a `preorder(T, n)` entonces seguramente se producirá un error al querer dereferenciar `n` en la línea 2.

Notar que Λ puede ser pasado a `preorder(T, n)` sólo si `T` es vacío, ya que una vez que un nodo dereferenciable es pasado a `preorder(T, n)`, el test de la línea 5 se encarga de no dejar nunca pasar un nodo Λ a una instancia inferior de `preorder(T, n)`. Si `T` no es vacío, entonces `T.begin()` es dereferenciable y a partir de ahí nunca llegará a `preorder(T, n)` un nodo Λ . Notar que es mucho más eficiente verificar que el árbol no este vacío en `preorder(T)` que hacerlo en `preorder(T, n)` ya que en el primer caso la verificación se hace una sola vez para todo el árbol.

La rutina `lisp_print` es básicamente equivalente a las de orden previo y orden posterior. Una diferencia es que `lisp_print` no apendiza a una lista, ya que en ese caso habría que tomar alguna convención para representar los paréntesis. `lisp_print()` simplemente imprime por terminal la notación Lisp del árbol.

3.4.3. Algoritmos de copia

Pasemos ahora a estudiar `tree_copy()` y `mirror_copy()`. Notar el llamado a `insert` en la línea 50. Notar que la línea actualiza el valor de `nq` ya que de otra manera quedaría inválido por la inserción. Notar como las líneas 52–53 y 56–57 van manteniendo los iterators `ct` y `cq` sobre posiciones equivalentes en `T` y `Q` respectivamente.

3.4.4. Algoritmo de poda

Notar que si el elemento en `n` es impar, todo el subárbol de `n` es eliminado, y `n` queda apuntando al hermano derecho, por el `erase` de la línea 88, caso contrario, `n` queda apuntando al hermano derecho por el avance explícito de la línea 92.

3.5. Implementación de la interfaz básica por punteros

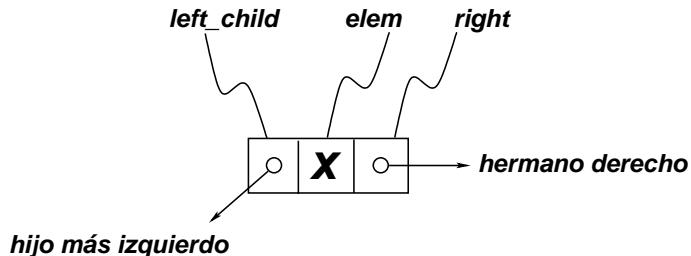


Figura 3.19: Celdas utilizadas en la representación de árboles por punteros.

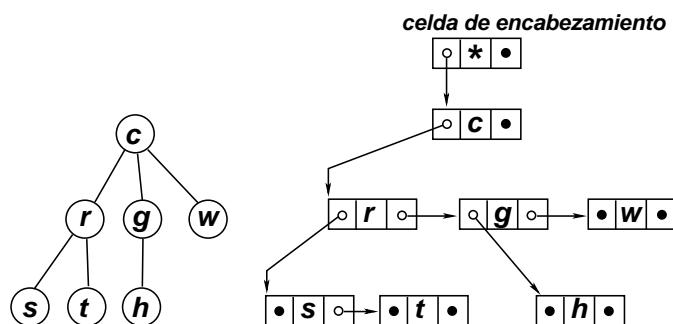


Figura 3.20: Representación de un árbol con celdas enlazadas por punteros.

Así como la implementación de listas por punteros mantiene los datos en celdas enlazadas por un campo **next**, es natural considerar una implementación de árboles en la cual los datos son almacenados en celdas que contienen, además del dato **elem**, un puntero **right** a la celda que corresponde al hermano derecho y otro **left_child** al hijo más izquierdo (ver figura 3.19). En la figura 3.20 vemos un árbol simple y su representación mediante celdas enlazadas.

3.5.1. El tipo iterator

Por analogía con las listas podríamos definir el tipo **iterator_t** como un **typedef a cell ***. Esto bastaría para representar posiciones dereferenciables y posiciones no dereferenciables que provienen de haber aplicado **right()** al último hermano, como la posición Λ_3 en la figura 3.21. Por supuesto habría que mantener el criterio de usar “*posiciones adelantadas*” con respecto al dato. Sin embargo no queda en claro como representar posiciones no dereferenciables como la Λ_1 que provienen de aplicar **lchild()** a una hoja.

Una solución posible consiste en hacer que el tipo **iterator_t** contenga, además de un puntero a la celda que contiene el dato, punteros a celdas que de otra forma serían inaccesibles. Entonces el iterator consiste en *tres punteros a celdas* (ver figura 3.22) a saber,

- Un puntero **ptr** a la celda que contiene el dato. (Este puntero es nulo en el caso de posiciones no dereferenciables).

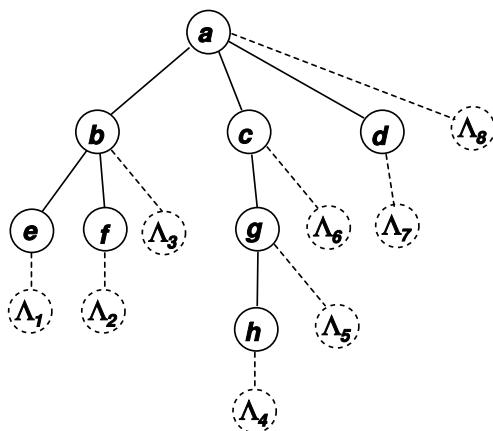


Figura 3.21: Todas las posiciones no dereferenciables de un árbol.

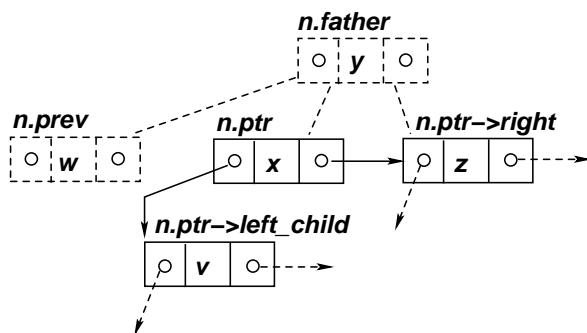


Figura 3.22: Entorno local de un iterator sobre árboles.

- Un puntero **prev** al *hermano izquierdo*.
- Un puntero **father** al *padre*.

Así, por ejemplo a continuación mostramos los juegos de punteros correspondientes a varias posiciones dereferenciables y no dereferenciables en el árbol de la figura 3.6:

- nodo *e*: **ptr=e**, **prev=NULL**, **father=b**.
- nodo *f*: **ptr=f**, **prev=e**, **father=b**.
- nodo Λ_1 : **ptr=NULL**, **prev=NULL**, **father=e**.
- nodo Λ_2 : **ptr=NULL**, **prev=NULL**, **father=f**.
- nodo Λ_3 : **ptr=NULL**, **prev=f**, **father=b**.
- nodo *g*: **ptr=g**, **prev=NULL**, **father=c**.
- nodo Λ_6 : **ptr=NULL**, **prev=g**, **father=c**.

Estos tres punteros tienen la suficiente información como para ubicar a todas las posiciones (dereferenciables o no) del árbol. Notar que todas las posiciones tienen un puntero **father** no nulo, mientras que el puntero **prev** puede o no ser nulo.

Para tener un código más uniforme se introduce una celda de encabezamiento, al igual que con las listas. La raíz del árbol, si existe, es una celda hija de la celda de encabezamiento. Si el árbol está vacío, entonces el iterator correspondiente a la raíz (y que se obtiene llamando a `begin()`) corresponde a `ptr=NULL`, `prev=NULL`, `father=`celda de encabezamiento.

3.5.2. Las clases cell e iterator_t

```

1  class tree;
2  class iterator_t;
3
4 //---:---<*>---:---<*>---:---<*>---:---<*>
5 class cell {
6     friend class tree;
7     friend class iterator_t;
8     elem_t elem;
9     cell *right, *left_child;
10    cell() : right(NULL), left_child(NULL) {}
11 };
12
13 //---:---<*>---:---<*>---:---<*>---:---<*>
14 class iterator_t {
15 private:
16     friend class tree;
17     cell *ptr,*prev,*father;
18     iterator_t(cell *p,cell *prev_a, cell *f_a)
19         : ptr(p), prev(prev_a), father(f_a) {}
20 public:
21     iterator_t(const iterator_t &q) {
22         ptr = q.ptr;
23         prev = q.prev;
24         father = q.father;
25     }
26     bool operator!=(iterator_t q) { return ptr!=q.ptr; }
27     bool operator==(iterator_t q) { return ptr==q.ptr; }
28     iterator_t()
29         : ptr(NULL), prev(NULL), father(NULL) {}
30
31     iterator_t lchild() {
32         return iterator_t(ptr->left_child,NULL,ptr);
33     }
34     iterator_t right() {
35         return iterator_t(ptr->right,ptr,father);
36     }
37 };
38
39 //---:---<*>---:---<*>---:---<*>---:---<*>
40 class tree {
41 private:
42     cell *header;
43     tree(const tree &T) {}

```

```

44 public:
45
46     tree() {
47         header = new cell;
48         header->right = NULL;
49         header->left_child = NULL;
50     }
51     ~tree() { clear(); delete header; }
52
53     elem_t &retrieve(iterator_t p) {
54         return p.ptr->elem;
55     }
56
57     iterator_t insert(iterator_t p, elem_t elem) {
58         assert(!(p.father==header && p.ptr));
59         cell *c = new cell;
60         c->right = p.ptr;
61         c->elem = elem;
62         p.ptr = c;
63         if (p.prev) p.prev->right = c;
64         else p.father->left_child = c;
65         return p;
66     }
67     iterator_t erase(iterator_t p) {
68         if(p==end()) return p;
69         iterator_t c = p.lchild();
70         while (c!=end()) c = erase(c);
71         cell *q = p.ptr;
72         p.ptr = p.ptr->right;
73         if (p.prev) p.prev->right = p.ptr;
74         else p.father->left_child = p.ptr;
75         delete q;
76         return p;
77     }
78
79     iterator_t splice(iterator_t to, iterator_t from) {
80         assert(!(to.father==header && to.ptr));
81         if (from.ptr->right == to.ptr) return from;
82         cell *c = from.ptr;
83
84         if (from.prev) from.prev->right = c->right;
85         else from.father->left_child = c->right;
86
87         c->right = to.ptr;
88         to.ptr = c;
89         if (to.prev) to.prev->right = c;
90         else to.father->left_child = c;
91
92         return to;
93     }
94
95     iterator_t find(elem_t elem) {
96         return find(elem,begin());

```

```

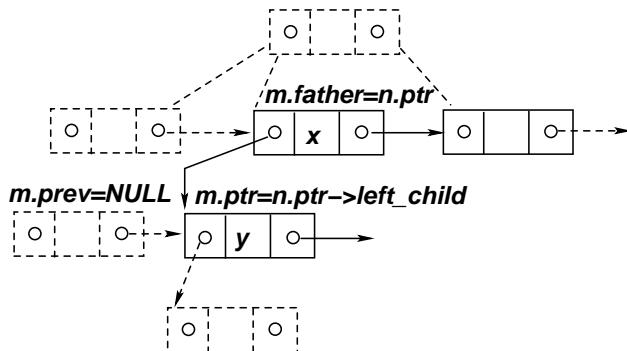
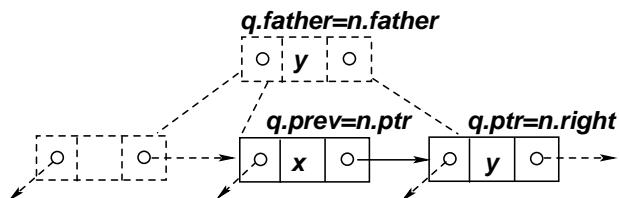
97     }
98     iterator_t find(elem_t elem, iterator_t p) {
99       if(p==end() || retrieve(p) == elem) return p;
100      iterator_t q,c = p.lchild();
101      while (c!=end()) {
102        q = find(elem,c);
103        if (q!=end()) return q;
104        else c = c.right();
105      }
106      return iterator_t();
107    }
108    void clear() { erase(begin()); }
109    iterator_t begin() {
110      return iterator_t(header->left_child,NULL,header);
111    }
112    iterator_t end() { return iterator_t(); }

```

Código 3.9: Implementación de la interfaz básica de árboles por punteros. [Archivo: treebas.h]

Una implementación de la interfaz básica código 3.7 por punteros puede observarse en el código 3.9.

- Tenemos primero las declaraciones “*hacia adelante*” de **tree** e **iterator_t**. Esto nos habilita a declarar **friend** a las clases **tree** e **iterator_t**.
- La clase **cell** sólo declara los campos para contener al puntero al hijo más izquierdo y al hermano derecho. El constructor inicializa los punteros a **NULL**.
- La clase **iterator_t** declara **friend** a **tree**, pero no es necesario hacerlo con **cell**. **iterator_t** declara los campos punteros a celdas y por comodidad declaramos un constructor privado **iterator_t(cell *p,cell *pv, cell *f)** que simplemente asigna a los punteros internos los valores de los argumentos.
- Por otra parte existe un constructor público **iterator_t(const iterator_t &q)**. Este es el “*constructor por copia*” y es utilizado cuando hacemos por ejemplo **iterator_t p(q);** con **q** un iterador previamente definido.
- El operador de asignación de iteradores (por ejemplo **p=q**) es *sintetizado* por el compilador, y simplemente se reduce a una copia *bit a bit* de los datos miembros de la clase (en este caso los tres punteros **ptr**, **prev** y **father**) lo cual es apropiado en este caso.
- Haber definido **iterator_t** como una clase (y no como un **typedef**) nos obliga a definir también los operadores **!=** y **==**. Esto nos permitirá comparar nodos por igualdad o desigualdad (**p==q** o **p!=q**). De la misma forma que con las posiciones en listas, los nodos *no* pueden compararse con los operadores de relación de orden (**<**, **<=**, **>** y **>=**). Notar que los operadores **==** y **!=** sólo comparan el campo **ptr** de forma que *todas las posiciones no dereferenciables (Λ) son “iguales” entre sí*. Esto permite comparar en los lazos cualquier posición **p** con **end()**, entonces **p==end()** retornará **true** incluso si **p** no es exactamente igual a **end()** (es decir tiene campos **father** y **prev** diferentes).
- El constructor por defecto **iterator_t()** devuelve un iterador con los tres punteros nulos. Este iterador no debería ser normalmente usado en ninguna operación, pero es invocado automáticamente por el compilador cuando declaramos iterators como en: **iterator p;**

Figura 3.23: La función `lchild()`.Figura 3.24: La función `right()`.

- Las operaciones `lchild()` y `right()` son las que nos permiten movernos dentro del árbol. Sólo pueden aplicarse a posiciones dereferenciables y pueden retornar posiciones dereferenciables o no. Para entender como funcionan consideremos la información contenida en un iterator. Si consideramos un iterator `n` (ver figura 3.23), entonces la posición de `m=n.lchild()` está definida por los siguientes punteros

- `m.ptr= n.ptr->left_child`
- `m.prev= NULL` (ya que `m` es un *hijo más izquierdo*)
- `m.father= n.ptr`

Por otra parte, si consideramos la función hermano derecho: `q=n.right()`, entonces `q` está definida por los siguientes punteros,

- `q.ptr= n.ptr->right`
- `q.prev= n.ptr`
- `q.father= n.father`

3.5.3. La clase tree

- La clase `tree` contiene un único dato que es un puntero a la celda de encabezamiento. Esta celda es alojada e inicializada en el constructor `tree()`.
- La función `retrieve(p)` simplemente retorna el dato contenido en la celda apuntada por `p.ptr`.
- La función `insert(p,x)` aloca una nueva celda `c` e inicializa sus campos datos. El dato `elem` se obtiene

del argumento a la llamada y el puntero al hermano derecho pasa a ser el puntero **ptr** de la posición donde se va a insertar, ya que (al igual que con las listas) el valor insertado queda *a la izquierda* de la posición donde se inserta. Como la nueva celda no va a tener hijos, el puntero **left_child** queda en **NULL** (esto se hace al crear la celda en el constructor de la clase **cell**). Después de insertar y enlazar la nueva celda hay que calcular la posición de la nueva celda insertada, que es el valor de retorno de **insert()**.

- **p.ptr=c**, la nueva celda insertada.
- **p.prev** no se altera ya que el hermano a la izquierda de la nueva celda es el mismo que el de la celda donde se insertó.
- **p.father** tampoco cambia, porque la nueva celda tiene el mismo padre que aquella posición donde se insertó.

Finalmente hay que actualizar los punteros en algunas celdas vecinas.

- Si **p.prev** no es nulo, entonces la celda *no es el hijo más izquierdo* y por lo tanto hay que actualizar el puntero **right** de la celda a la izquierda.
 - Caso contrario, la nueva celda *pasa a ser el hijo más izquierdo* de su padre y por lo tanto hay que actualizar el puntero **left_child** de éste.
 - En **erase(p)** la línea 70 eliminan todos los subárboles de **p** en forma recursiva. Notar que esta parte del código es genérica (independiente de la implementación particular). Finalmente las líneas 71–76 eliminan la celda correspondiente a **p** actualizando los punteros a las celdas vecinas si es necesario.
 - La función **splice(to, from)** primero elimina el subárbol de **from** (líneas 84–85) en forma muy similar a las líneas 73–74 de **erase** pero sin eliminar recursivamente el subárbol, como en **erase()** ya que debe ser insertado en la posición **to**. Esta inserción se hace en las líneas 87–90, notar la similitud de estas líneas con la función **insert()**.
 - La función **find(elem)** no fue descrita en la interfaz básica pero es introducida aquí. Retorna un iterator al nodo donde se encuentra el elemento **elem**. La implementación es completamente genérica se hace recursivamente definiendo la función auxiliar **find(elem,p)** que busca el elemento **elem** en el subárbol del nodo **p**.
 - **clear()** llama a **erase()** sobre la raíz.
 - **begin()** construye el iterator correspondiente a la raíz del árbol, es decir que los punteros correspondientes se obtienen a partir de la celda de encabezamiento como
 - **ptr=header->left_child**
 - **prev=NULL**
 - **father=header**
 - Se ha incluido un “constructor por copia” (línea 43) en la parte privada. Recordemos que el constructor por copia es usado cuando un usuario declara objetos en la siguiente forma
- ¹ **tree T2(T1);**
- es decir, al declarar un nuevo objeto **T2** a partir de uno preexistente **T1**.

Para todas las clases que contienen punteros a otros objetos (y que pertenecen a la clase, es decir que debe encargarse de alocarlos y desalocarlos, como son las celdas en el caso de listas y árboles) hay que tener cuidado. Si uno deja que el compilador sintetice un constructor por copia entonces la copia se hace “bit a bit” con lo cual para los punteros simplemente copia el valor del puntero, no creando duplicados de los objetos apuntados. A esto se le llama “shallow copy”. De esta forma el objeto original y el duplicado comparten objetos internos, y eso debe hacerse con cuidado, o puede traer problemas. A menos que uno, por alguna razón prefiera este comportamiento, en general hay que optar por una de las siguientes alternativas

- Implementar el constructor por copia correctamente, es decir copiando los componentes internos (“deep copy”). Este constructor funcionaría básicamente como la función `tree_copy()` descripta más arriba (ver §3.3.2.1). (La implementación de la interfaz avanzada y la de árbol binario están hechas así).
- Declarar al constructor por copia, implementándolo con un cuerpo vacío y poniéndolo en la parte privada. Esto hace que si el usuario intenta escribir un código como el de arriba, el compilador dará un mensaje de error. Esto evita que un usuario desprevenido que no sabe que el constructor por copia no hace el “deep copy”, lo use por accidente. Por ejemplo

```
1 tree T1;
2 // pone cosas en T1...
3 tree T2(T1);
4 // Obtiene un nodo en T2
5 iterator_t n = T2.find(x);
6 // vacia el arbol T1
7 T1.clear();
8 // Intenta cambiar el valor en 'n'
9 T2.retrieve(n) = y; // ERROR!!
```

En este ejemplo, el usuario obtiene una “shallow copy” `T2` del árbol `T1` y genera un iterator a una posición dereferenciable `n` en `T2`. Después de vaciar el árbol `T1` y querer acceder al elemento en `n` genera un error en tiempo de ejecución, ya que en realidad la estructura interna de `T2` era compartida con `T1`. Al vaciar `T1`, se vacía también `T2`.

Con la inclusión de la línea 43 del código 3.9., la instrucción `tree T2(T1);` genera un error en tiempo de compilación.

- Implementarlo como público, pero que de un error.

```
1 public:
2 tree(const tree &T) {
3     error("Constructor por copia no implementado!!");
4 }
```

De esta forma, si el usuario escribe `tree T2(T1);` entonces compilará pero en tiempo de ejecución, si pasa por esa línea va a dar un error.

3.6. Interfaz avanzada

```
1 #ifndef AED_TREE_H
2 #define AED_TREE_H
```

```
3 #include <cassert>
4 #include <iostream>
5 #include <cstddef>
6 #include <cstdlib>
7
8 namespace aed {
9
10 //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
11 template<class T>
12 class tree {
13 public:
14     class iterator;
15 private:
16     class cell {
17         friend class tree;
18         friend class iterator;
19         T t;
20         cell *right, *left_child;
21         cell() : right(NULL), left_child(NULL) {}
22     };
23     cell *header;
24
25     iterator tree_copy_aux(iterator nq,
26                           tree<T> &TT, iterator nt) {
27         nq = insert(nq,*nt);
28         iterator
29             ct = nt.lchild(),
30             cq = nq.lchild();
31         while (ct!=TT.end()) {
32             cq = tree_copy_aux(cq,TT,ct);
33             ct = ct.right();
34             cq = cq.right();
35         }
36         return nq;
37     }
38
39 public:
40     static int cell_count_m;
41     static int cell_count() { return cell_count_m; }
42     class iterator {
43     private:
44         friend class tree;
45         cell *ptr,*prev,*father;
46         iterator(cell *p,cell *prev_a,cell *f_a) : ptr(p),
47             prev(prev_a), father(f_a) { }
48     public:
49         iterator(const iterator &q) {
50             ptr = q.ptr;
51             prev = q.prev;
52             father = q.father;
53         }
54         T &operator*() { return ptr->t; }
55         T *operator->() { return &ptr->t; }
```

```
56     bool operator!=(iterator q) { return ptr!=q.ptr; }
57     bool operator==(iterator q) { return ptr==q.ptr; }
58     iterator() : ptr(NULL), prev(NULL), father(NULL) { }
59
60     iterator lchild() { return iterator(ptr->left_child,NULL,ptr); }
61     iterator right() { return iterator(ptr->right,ptr,father); }
62
63     // Prefix:
64     iterator operator++() {
65         *this = right();
66         return *this;
67     }
68     // Postfix:
69     iterator operator++(int) {
70         iterator q = *this;
71         *this = right();
72         return q;
73     }
74 };
75
76 tree() {
77     header = new cell;
78     cell_count_m++;
79     header->right = NULL;
80     header->left_child = NULL;
81 }
82 tree<T>(const tree<T> &TT) {
83     if (&TT != this) {
84         header = new cell;
85         cell_count_m++;
86         header->right = NULL;
87         header->left_child = NULL;
88         tree<T> &TTT = (tree<T> &) TT;
89         if (TTT.begin()!=TTT.end())
90             tree_copy_aux(begin(),TTT,TTT.begin());
91     }
92 }
93 tree &operator=(tree<T> &TT) {
94     if (this != &TT) {
95         clear();
96         tree_copy_aux(begin(),TT,TT.begin());
97     }
98     return *this;
99 }
100 ~tree() { clear(); delete header; cell_count_m--; }
101 iterator insert(iterator p,T t) {
102     assert(!(p.father==header && p.ptr));
103     cell *c = new cell;
104     cell_count_m++;
105     c->right = p.ptr;
106     c->t = t;
107     p.ptr = c;
108     if (p.prev) p.prev->right = c;
109     else p.father->left_child = c;
```

```
110     return p;
111 }
112 iterator erase(iterator p) {
113     if(p==end()) return p;
114     iterator c = p.lchild();
115     while (c!=end()) c = erase(c);
116     cell *q = p.ptr;
117     p.ptr = p.ptr->right;
118     if (p.prev) p.prev->right = p.ptr;
119     else p.father->left_child = p.ptr;
120     delete q;
121     cell_count_m--;
122     return p;
123 }
124
125 iterator splice(iterator to,iterator from) {
126     assert(!(to.father==header && to.ptr));
127     if (from.ptr->right == to.ptr) return from;
128     cell *c = from.ptr;
129
130     if (from.prev) from.prev->right = c->right;
131     else from.father->left_child = c->right;
132
133     c->right = to.ptr;
134     to.ptr = c;
135     if (to.prev) to.prev->right = c;
136     else to.father->left_child = c;
137
138     return to;
139 }
140 iterator find(T t) { return find(t,begin()); }
141 iterator find(T t,iterator p) {
142     if(p==end() || p.ptr->t == t) return p;
143     iterator q,c = p.lchild();
144     while (c!=end()) {
145         q = find(t,c);
146         if (q!=end()) return q;
147         else c++;
148     }
149     return iterator();
150 }
151 void clear() { erase(begin()); }
152 iterator begin() { return iterator(header->left_child,NULL,header); }
153 iterator end() { return iterator(); }
154 };
155
156 template<class T>
157 int tree<T>::cell_count_m = 0;
158
159 template<class T>
160 void swap(tree<T> &T1, tree<T> &T2) { T1.swap(T2); }
161
162 }
163 #endif
```

Código 3.10: Interfaz avanzada para árboles. [Archivo: tree.h]

Una interfaz similar a la descripta en la sección §3.4 pero incluyendo templates, clases anidadas y sobre-carga de operadores puede observarse en el código 3.10.

- La clase `tree` pasa a ser ahora un template, de manera que podremos declarar `tree<int>`, `tree<double>`.
- Las clases `cell` e `iterator` son ahora clases anidadas dentro de `tree`. Externamente se verán como `tree<int>::cell` y `tree<int>::iterator`. Sin embargo, sólo `iterator` es pública y es usada fuera de `tree`.
- La dereferenciación de posiciones (nodos) `x=retrieve(p)` se reemplaza por `x=*p`. Para eso debemos “sobrecargar” el operador `*`. Si el tipo elemento (es decir el tipo `T` del template) contiene campos, entonces vamos a querer extraer campos de un elemento almacenado en un nodo, por lo cual debemos hacer `(*p).campo`. Para poder hacer esto usando el operador `->` (es decir `p->campo`) debemos sobre-cargar el operador `->`. Ambos operadores devuelven referencias de manera que es posible usarlos en el miembro izquierdo, como en `*p=x` o `p->campo=z`.
- Igual que con la interfaz básica, para poder hacer comparaciones de iterators debemos sobrecargar también los operadores `==` y `!=`. También tiene definido el constructor por copia.
- El avance por hermano derecho `p = p.right();` ahora se puede hacer con `p++`, de todas formas mantenemos la función `right()` que a veces resulta ser más compacta. Por ejemplo `q = p.right()` se traduce en `q=p; q++;` en la versión con operadores.
- La función estática `cell_count()`, permite obtener el número total de celdas alojadas por todos las instancias de la clase, e incluye las celdas de encabezamiento. Esta función fue introducida para debugging, normalmente no debería ser usada por los usuarios de la clase. Como es estática puede invocarse como `tree<int>::cell_count()` o también sobre una instancia, `T.cell_count()`.
- Se ha incluido un constructor por copia, de manera que se puede copiar árboles usando directamente el operador `=`, por ejemplo

```
1  tree<int> T,Q;  
2 // carga elementos en T . . . .  
3 Q = T;
```

Así como también pasar árboles por copia y definir contenedores que contienen árboles como por ejemplo una lista de árboles de enteros. `list< tree<int> >`. Esta función necesita otra función recursiva auxiliar que hemos llamado `tree_copy_aux()` y que normalmente no debería ser usada directamente por los usuarios de la clase, de manera que la incluimos en la sección privada.

3.6.1. Ejemplo de uso de la interfaz avanzada

```
1 typedef tree<int> tree_t;  
2 typedef tree_t::iterator node_t;  
3  
4 int count_nodes(tree_t &T,node_t n) {  
5   if (n==T.end()) return 0;
```

```
6 int m=1;
7 node_t c = n.lchild();
8 while(c!=T.end()) m += count_nodes(T,c++);
9 return m;
10 }
11
12 int count_nodes(tree_t &T) {
13     return count_nodes(T,T.begin());
14 }
15
16 int height(tree_t &T, node_t n) {
17     if (n==T.end()) return -1;
18     node_t c = n.lchild();
19     if (c==T.end()) return 0;
20     int son_max_height = -1;
21     while (c!=T.end()) {
22         int h = height(T,c);
23         if (h>son_max_height) son_max_height = h;
24         c++;
25     }
26     return 1+son_max_height;
27 }
28
29 int height(tree_t &T) {
30     return height(T,T.begin());
31 }
32
33 void
34 node_level_stat(tree_t &T, node_t n,
35                 int level, vector<int> &nod_lev) {
36     if (n==T.end()) return;
37     assert(nod_lev.size()>=level);
38     if (nod_lev.size()==level) nod_lev.push_back(0);
39     nod_lev[level]++;
40     node_t c = n.lchild();
41     while (c!=T.end()) {
42         node_level_stat(T,c++,level+1,nod_lev);
43     }
44 }
45
46 void node_level_stat(tree_t &T,
47                      vector<int> &nod_lev) {
48     nod_lev.clear();
49     node_level_stat(T,T.begin(),0,nod_lev);
50     for (int j=0;j<nod_lev.size();j++) {
51         cout << "[level: " << j
52             << ", nodes: " << nod_lev[j] << "]";
53     }
54     cout << endl;
55 }
56
57 //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
58 int max_node(tree_t &T, node_t n) {
```

```
59 if (n==T.end()) return -1;
60 int w = *n;
61 node_t c = n.lchild();
62 while (c!=T.end()) {
63     int ww = max_node(T,c++);
64     if (ww > w) w = ww;
65 }
66 return w;
67 }

68
69 int max_node(tree_t &T) {
70     return max_node(T,T.begin());
71 }
72
73 //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
74 int max_leaf(tree_t &T,node_t n) {
75     if (n==T.end()) return -1;
76     int w = *n;
77     node_t c = n.lchild();
78     if (c==T.end()) return w;
79     w = 0;
80     while (c!=T.end()) {
81         int ww = max_leaf(T,c++);
82         if (ww > w) w = ww;
83     }
84     return w;
85 }
86
87 int max_leaf(tree_t &T) {
88     return max_leaf(T,T.begin());
89 }
90
91 //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
92 int leaf_count(tree_t &T,node_t n) {
93     if (n==T.end()) return 0;
94     node_t c = n.lchild();
95     if (c==T.end()) return 1;
96     int w = 0;
97     while (c!=T.end()) w += leaf_count(T,c++);
98     return w;
99 }
100
101 int leaf_count(tree_t &T) {
102     return leaf_count(T,T.begin());
103 }
```

Código 3.11: Algunos ejemplos de uso de la interfaz avanzada para árboles. [Archivo: treetools2.cpp]

En el código 3.11 vemos algunos ejemplos de uso de esta interfaz.

- Todo los ejemplos usan árboles de enteros, `tree<int>`. Los `typedef` de las líneas 1-2 permiten definir

tipos `tree_t` y `node_t` que abrevian el código.

- Las funciones implementadas son
 - `height(T)` su altura,
 - `count_nodes(T)` cuenta los nodos de un árbol,
 - `leaf_count(T)` el número de hojas,
 - `max_node(T)` el máximo valor del elemento contenido en los nodos,
 - `max_leaf(T)` el máximo valor del elemento contenido en las hojas,
- `node_level_stat(T, nod_lev)` calcula el número de nodos que hay en cada nivel del árbol, el cual se retorna en el `vector<int> nod_lev`, es decir, `nod_lev[1]` es el número de nodos en el nivel 1.

3.7. Tiempos de ejecución

Operación	$T(n)$
<code>begin()</code> , <code>end()</code> , <code>n.right()</code> , <code>n++</code> , <code>n.left_child()</code> , <code>*n</code> , <code>insert()</code> , <code>splice(to, from)</code>	$O(1)$
<code>erase()</code> , <code>find()</code> , <code>clear()</code> , <code>T1=T2</code>	$O(n)$

Tabla 3.1: Tiempos de ejecución para operaciones sobre árboles.

En la Tabla 3.1 vemos los tiempos de ejecución para las diferentes operaciones sobre árboles. Es fácil ver que todas las funciones básicas tienen costo $O(1)$. Es notable que una función como `splice()` también sea $O(1)$. Esto se debe a que la operación de mover todo el árbol de una posición a otra se realiza con una operación de punteros. Las operaciones que no son $O(1)$ son `erase(p)` que debe eliminar todos los nodos del subárbol del nodo `p`, `clear()` que equivale a `erase(begin())`, `find(x)` y el constructor por copia (`T1=T2`). En todos los casos `n` es o bien el número de nodos del subárbol (`erase(p)` y `find(x, p)`) o bien el número total de nodos del árbol (`clear()`, `find(x)` y el constructor por copia `T1=T2`).

3.8. Arboles binarios

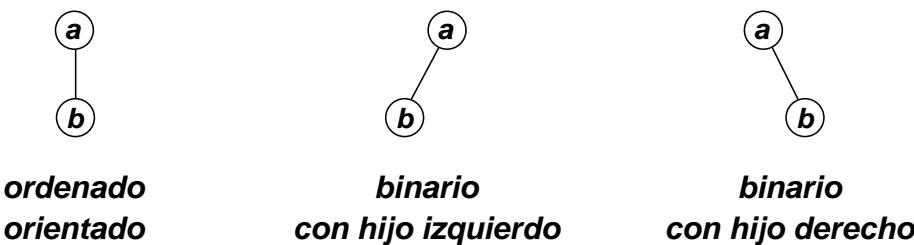


Figura 3.25: Diferentes casos de un árbol con dos nodos.

Los árboles que hemos estudiado hasta ahora son “árboles ordenados orientados” (AOO) ya que los hermanos están ordenados entre sí y hay una orientación de los caminos desde la raíz a las hojas. Otro tipo importante de árbol es el “árbol binario” (AB) en el cual cada nodo puede tener a lo sumo dos hijos. Además,

si un dado nodo n tiene un sólo hijo, entonces este puede ser el hijo derecho o el hijo izquierdo de n . Por ejemplo si consideramos las posibles estructuras de árboles con dos nodos (ver figura 3.25), tenemos que para el caso de un AOO la única posibilidad es un nodo raíz con un nodo hijo. Por otra parte, si el árbol es binario, entonces existen dos posibilidades, que el único hijo sea el hijo izquierdo o el derecho. Dicho de otra forma los AB del centro y la derecha son diferentes, mientras que si fueran AOO entonces serían ambos iguales al de la izquierda.

3.8.1. Listados en orden simétrico

Los listados en orden previo y posterior para AB coinciden con su versión correspondiente para AOO. El “listado en orden simétrico” se define recursivamente como

$$\begin{aligned} \text{osim}(\Lambda) &=<\text{lista vacía}> \\ \text{osim}(n) &= (\text{osim}(s_l), n, \text{osim}(s_r)) \end{aligned} \quad (3.14)$$

donde $s_{l,r}$ son los hijos izquierdo y derecho de n , respectivamente.

3.8.2. Notación Lisp

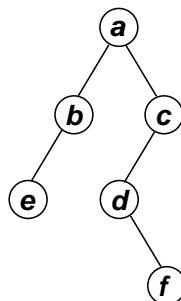


Figura 3.26: Ejemplo de árbol binario.

La notación Lisp para árboles debe ser modificada un poco con respecto a la de AOO, ya que debemos introducir algún tipo de notación para un hijo Λ . Básicamente, en el caso en que un nodo tiene un sólo hijo, reemplazamos con un punto la posición del hijo faltante. La definición recursiva es

$$\text{lisp}(n) = \begin{cases} n & ; \text{ si } s_l = \Lambda \text{ y } s_r = \Lambda \\ (n \text{ lisp}(s_l) \text{ lisp}(s_r)) & ; \text{ si } s_l \neq \Lambda \text{ y } s_r \neq \Lambda \\ (n \cdot \text{lisp}(s_r)) & ; \text{ si } s_l = \Lambda \text{ y } s_r \neq \Lambda \\ (n \text{ lisp}(s_l) \cdot) & ; \text{ si } s_l \neq \Lambda \text{ y } s_r = \Lambda \end{cases} \quad (3.15)$$

Por ejemplo, la notación Lisp del árbol de la figura 3.26 es

$$\text{lisp}(a) = (a (b e \cdot) (c (d \cdot f) \cdot)) \quad (3.16)$$

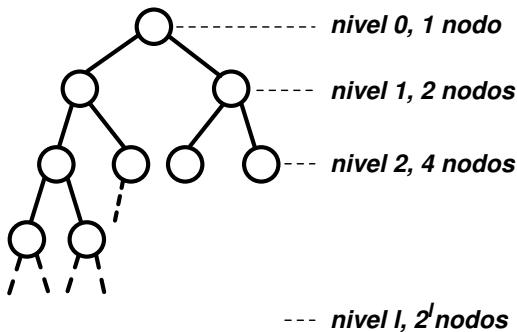


Figura 3.27: Cantidad máxima de nodos por nivel en un árbol binario lleno.

3.8.3. Árbol binario lleno

Un AOO no está limitado en cuanto a la cantidad de nodos que puede contener en un dado nivel. Por el contrario (ver figura 3.27) el árbol binario puede tener a lo sumo dos nodos en el nivel 1, 4 nodos en el nivel 2, y en general, 2^l nodos en el nivel l . En total, un árbol binario de l niveles puede tener a lo sumo

$$n \leq 1 + 2 + 4\dots + 2^l \quad (3.17)$$

nodos. Pero ésta es una serie geométrica de razón dos, por lo cual

$$n \leq \frac{2^{l+1} - 1}{2 - 1} = 2^{l+1} - 1. \quad (3.18)$$

O bien

$$n < 2^{l+1}. \quad (3.19)$$

Inversamente el número de niveles puede obtenerse a partir del número de nodos como

$$\begin{aligned} l + 1 &> \log_2 n \\ l + 1 &> \text{floor}(\log_2 n) \\ l &\geq \text{floor}(\log_2 n) \end{aligned} \tag{3.20}$$

3.8.4. Operaciones básicas sobre árboles binarios

Las operaciones sobre AB difieren de las de AOO (ver sección §3.3.4) en las funciones que permiten “moverse” en el árbol. Si usáramos las operaciones de AOO para acceder a los hijos de un nodo en un AB, entonces para acceder al hijo derecho deberíamos hacer una operación “*hijo-más-izquierdo*” para acceder al hijo izquierdo y después una operación “*hermano-derecho*”. Pero el hijo izquierdo no necesariamente debe existir, por lo cual la estrategia de movimientos en el árbol debe ser cambiada. La solución es que existan dos operaciones independientes “*hijo-izquierdo*” e “*hijo-derecho*”. También, en AB sólo se puede insertar en un nodo Λ ya que la única posibilidad de insertar en un nodo dereferenciable, manteniendo el criterio usado para AOO, sería insertar en un hijo izquierdo que no tiene hermano derecho. En ese caso (manteniendo el criterio usado para AOO) el hijo izquierdo debería pasar a ser el derecho y el nuevo elemento pasaría a ser el hijo izquierdo y de todas formas esta operación no agregaría ninguna funcionalidad.

Entonces, las operaciones para el AB son las siguientes.

- Dado un nodo, obtener su *hijo izquierdo*. (Puede retornar una posición Λ).
- Dado un nodo, obtener su *hijo derecho*. (Puede retornar una posición Λ).
- Dada una posición, determinar si es Λ o no.
- Obtener la posición de la raíz del árbol.
- Dado un nodo obtener una referencia al dato contenido en el nodo.
- Dada una posición *no dereferenciable* y un dato, insertar un nuevo nodo con ese dato en esa posición.
- Borrar un nodo y todo su subárbol correspondiente.

Notar que sólo cambian las dos primeras y la inserción con respecto a las de AOO.

3.8.5. Interfaces e implementaciones

3.8.5.1. Interfaz básica

```
1 class iterator_t {
2     /* ... */
3     public:
4     iterator_t left();
5     iterator_t right();
6 };
7
8 class btree {
9     /* ... */
10    public:
11    iterator_t begin();
12    iterator_t end();
13    elem_t & retrieve(iterator_t p);
14    iterator_t insert(iterator_t p, elem_t t);
15    iterator_t erase(iterator_t p);
16    void clear();
17    iterator_t splice(iterator_t to, iterator_t from);
18 };
```

Código 3.12: Interfaz básica para árboles binarios. [Archivo: btreetash.h]

En el código 3.12 vemos una interfaz posible (recordemos que las STL *no* tienen clases de árboles) para AB. Como siempre, la llamamos básica porque no tiene templates, clases anidadas ni sobrecarga de operadores. Es similar a la mostrada para AOO en código 3.7, la única diferencia es que en la clase **iterator** las funciones **left()** y **right()** retornan los *hijos izquierdo* y *derecho*, respectivamente, en lugar de las funciones **lchild()** (que en AOO retornaba el hijo más izquierdo) y **right()** (que en AOO retorna el *hermano* derecho).

3.8.5.2. Ejemplo de uso. Predicados de igualdad y espejo

```
1 bool equal_p (btree &T, iterator_t nt,
```

```

2     btree &Q, iterator_t nq) {
3         if (nt==T.end() xor nq==Q.end()) return false;
4         if (nt==T.end()) return true;
5         if (T.retrieve(nt) != Q.retrieve(nq)) return false;
6         return equal_p(T, nt.right(), Q, nq.right()) &&
7             equal_p(T, nt.left(), Q, nq.left());
8     }
9     bool equal_p(btree &T, btree &Q) {
10    return equal_p(T, T.begin(), Q, Q.begin());
11 }

```

Código 3.13: Predicado que determina si dos árboles son iguales. [Archivo: *equalp.cpp*]

Como ejemplo de uso de esta interfaz vemos en código 3.13 una función predicado (es decir una función que retorna un valor booleano) que determina si dos árboles binarios **T** y **Q** son iguales. Dos árboles son iguales si

- Ambos son vacíos
- Ambos no son vacíos, los valores de sus nodos son iguales y los hijos respectivos de su nodo raíz son iguales.

Como, descripto en §3.4.2 la función se basa en una función auxiliar recursiva que toma como argumento adicionales dos nodos **nt** y **nq** y determina si los subárboles de **nt** y **nq** son iguales entre sí.

La función recursiva primero determina si uno de los nodos es Λ y el otro no o viceversa. En ese caso la función debe retornar **false** inmediatamente. La expresión lógica buscada podría ser

```

1  if ((nt==T.end() && nq!=Q.end()) ||
2      (nt!=T.end() && nq==Q.end())) return false;

```

pero la expresión se puede escribir en la forma más compacta usada en la línea 3 usando el operador lógico **xor** (“o exclusivo”). Recordemos que **x xor y** retorna verdadero sólo si uno de los operandos es verdadero y el otro falso. Si el código llega a la línea 4 es porque o bien ambos nodos son Λ o bien los dos no lo son. Por lo tanto, si **nt** es Λ entonces ambos lo son y la función puede retornar **true** ya que dos árboles vacíos ciertamente son iguales. Ahora, si el código llega a la línea 5 es porque ambos nodos no son Λ . En ese caso, los valores contenidos deben ser iguales. Por lo tanto la línea 5 retorna **false** si los valores son distintos. Finalmente, si el código llega a la línea 6 sólo resta comparar los subárboles derechos de **nt** y **nq** y sus subárboles izquierdos, los cuales deben ser iguales entre sí. Por supuesto estas comparaciones se hacen en forma recursiva.

```

1 bool semejante_p (btree &T, iterator_t nt,
2                     btree &Q, iterator_t nq) {
3     if (nt==T.end() xor nq==Q.end()) return false;
4     if (nt==T.end()) return true;
5     return semejante_p(T, nt.right(), Q, nq.right()) &&
6         semejante_p(T, nt.left(), Q, nq.left());
7 }
8 bool semejante_p(btree &T, btree &Q) {
9     return semejante_p(T, T.begin(), Q, Q.begin());
10 }

```

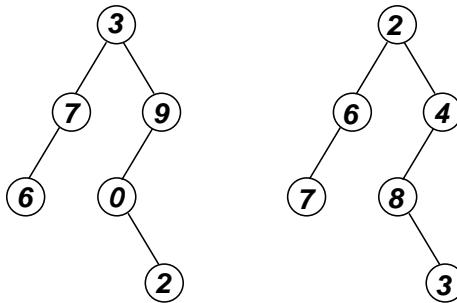


Figura 3.28: Dos árboles semejantes.

Código 3.14: Función predicado que determina si dos árboles son semejantes. [Archivo: *semejantep.cpp*]

Modificando ligeramente este algoritmo verifica si dos árboles son “semejantes” es decir, son iguales en cuanto a su estructura, sin tener en cuenta el valor de los nodos. Por ejemplo, los árboles de la figura 3.28 son semejantes entre sí. En forma recursiva la semejanza se puede definir en forma casi igual que la igualdad pero no hace falta que las raíces de los árboles sea igual. Dos árboles son semejantes si

- Ambos son vacíos
- Ambos no son vacíos, y los hijos respectivos de su nodo raíz son iguales.

Notar que la única diferencia es que no se comparan los valores de las raíces de los subárboles comparados. En el código 3.14 se muestra una función predicado que determina si dos árboles son semejantes. El código es igual al de *equal_p* sólo que se elimina la línea 5.

3.8.5.3. Ejemplo de uso. Hacer espejo “in place”

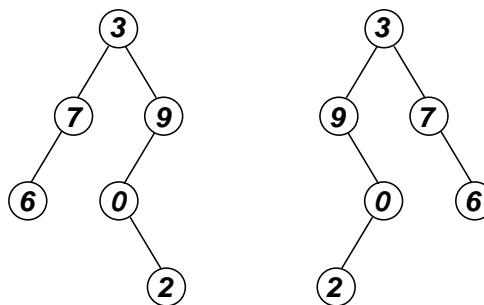


Figura 3.29: Copia espejo del árbol.

```

1 void mirror(btree &T, iterator_t n) {
2     if (n==T.end()) return;
  
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

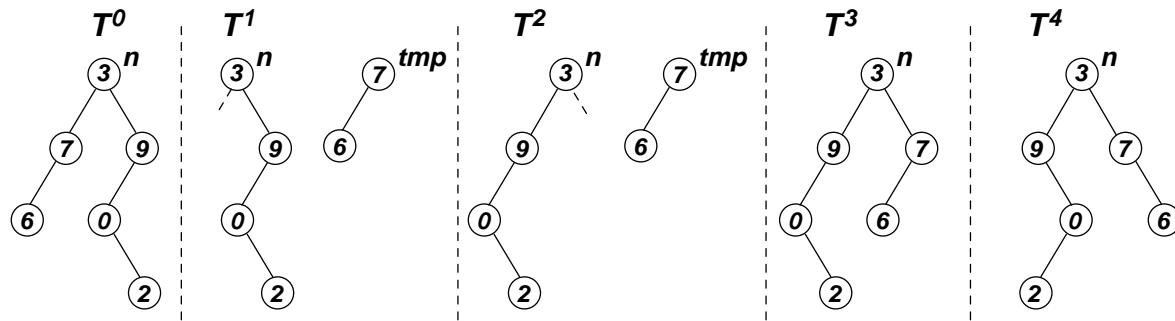


Figura 3.30: Descripción gráfica del procedimiento para copiar convertir “*in place*” un árbol en su espejo.

```

3   else {
4     btree tmp;
5     tmp.splice(tmp.begin(), n.left());
6     T.splice(n.left(), n.right());
7     T.splice(n.right(), tmp.begin());
8     mirror(T, n.right());
9     mirror(T, n.left());
10  }
11 }
12 void mirror(btree &T) { mirror(T, T.begin()); }
```

Código 3.15: Función para copiar convertir “*in place*” un árbol en su espejo. [Archivo: *bmmirror.cpp*]

Consideremos ahora una función **void mirror(tree &T)** que modifica el árbol **T**, dejándolo hecho igual a su espejo. Notar que esta operación es “*in place*”, es decir se hace en la estructura misma, sin crear una copia. El algoritmo es recursivo y se basa en intercambiar los subárboles de los hijos del nodo **n** y después aplicar recursivamente la función a los hijos (ver código 3.15). La operación del algoritmo sobre un nodo **n** del árbol se puede ver en la figura 3.30.

- La línea 5 extrae todo el subárbol del nodo izquierdo y lo inserta en un árbol vacío **tmp** con la operación **splice(to, from)**. Después de hacer esta operación el árbol se muestra como en el cuadro T^1 .
- La línea 6 mueve todo el subárbol del hijo derecho al hijo izquierdo, quedando como en T^2 .
- La línea 7 mueve todo el árbol guardado en **tmp** y que originariamente estaba en el hijo izquierdo al hijo derecho, quedando como en T^3 .
- Finalmente en las líneas 8–9 la función se aplica recursivamente a los hijos derecho e izquierdo, de manera que el árbol queda como en T^4 , es decir como el espejo del árbol original T^0 .

3.8.5.4. Implementación con celdas enlazadas por punteros

Así como la diferencia entre las interfaces de AOO y AB difieren en las funciones **lchild()** y **right()** que son reemplazadas por **left()** y **right()**. (Recordar que **right()** tiene significado diferente en AOO y AB.) Esto induce naturalmente a considerar que en la celda haya dos punteros que apunten al hijo izquierdo

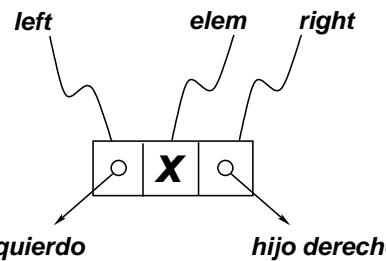


Figura 3.31: Celdas para representación de árboles binarios.

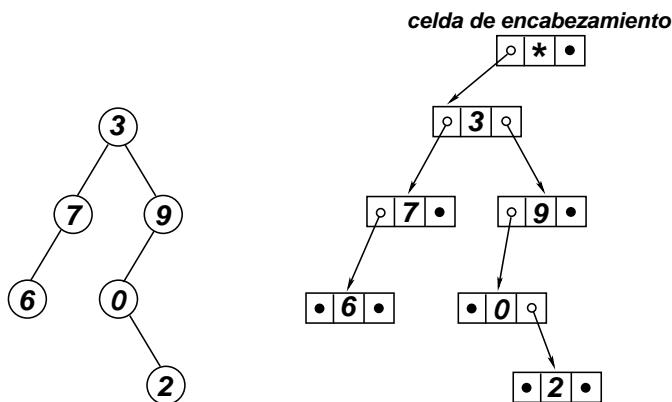


Figura 3.32: Representación de un árbol binario con celdas enlazadas.

y al hijo derecho, como se muestra en la figura 3.31. En la figura 3.32 se observa a la derecha el enlace de las celdas para representar el árbol binario de la izquierda.

```

1  typedef int elem_t;
2  class cell;
3  class iterator_t;
4
5  class cell {
6      friend class btree;
7      friend class iterator_t;
8      elem_t t;
9      cell *right,*left;
10     cell() : right(NULL), left(NULL) {}
11 };
12
13 class iterator_t {
14     private:
15     friend class btree;
16     cell *ptr,*father;
17     enum side_t {NONE,R,L};
18     side_t side;
19     iterator_t(cell *p,side_t side_a,cell *f_a)

```

```
20     : ptr(p), side(side_a), father(f_a) { }
21
22 public:
23     iterator_t(const iterator_t &q) {
24         ptr = q.ptr;
25         side = q.side;
26         father = q.father;
27     }
28     bool operator!=(iterator_t q) { return ptr!=q.ptr; }
29     bool operator==(iterator_t q) { return ptr==q.ptr; }
30     iterator_t() : ptr(NULL), side(NONE),
31                     father(NULL) { }
32
33     iterator_t left() {
34         return iterator_t(ptr->left,L,ptr);
35     }
36     iterator_t right() {
37         return iterator_t(ptr->right,R,ptr);
38     }
39 };
40
41 class btree {
42 private:
43     cell *header;
44     iterator_t tree_copy_aux(iterator_t nq,
45                               btree &TT, iterator_t nt) {
46         nq = insert(nq,TT.retrieve(nt));
47         iterator_t m = nt.left();
48         if (m != TT.end()) tree_copy_aux(nq.left(),TT,m);
49         m = nt.right();
50         if (m != TT.end()) tree_copy_aux(nq.right(),TT,m);
51         return nq;
52     }
53 public:
54     static int cell_count_m;
55     static int cell_count() { return cell_count_m; }
56     btree() {
57         header = new cell;
58         cell_count_m++;
59         header->right = NULL;
60         header->left = NULL;
61     }
62     btree(const btree &TT) {
63         if (&TT != this) {
64             header = new cell;
65             cell_count_m++;
66             header->right = NULL;
67             header->left = NULL;
68             btree &TTT = (btree &) TT;
69             if (TTT.begin()!=TTT.end())
70                 tree_copy_aux(begin(),TTT,TTT.begin());
71         }
72     }
```

```
73     }
74     ~btree() { clear(); delete header; cell_count_m--; }
75     elem_t & retrieve(iterator_t p) { return p.ptr->t; }
76     iterator_t insert(iterator_t p, elem_t t) {
77         cell *c = new cell;
78         cell_count_m++;
79         c->t = t;
80         if (p.side == iterator_t::R)
81             p.father->right = c;
82         else p.father->left = c;
83         p.ptr = c;
84         return p;
85     }
86     iterator_t erase(iterator_t p) {
87         if(p==end()) return p;
88         erase(p.right());
89         erase(p.left());
90         if (p.side == iterator_t::R)
91             p.father->right = NULL;
92         else p.father->left = NULL;
93         delete p.ptr;
94         cell_count_m--;
95         p.ptr = NULL;
96         return p;
97     }
98     iterator_t splice(iterator_t to, iterator_t from) {
99         cell *c = from.ptr;
100        from.ptr = NULL;
101        if (from.side == iterator_t::R)
102            from.father->right = NULL;
103        else
104            from.father->left = NULL;
105        if (to.side == iterator_t::R) to.father->right = c;
106        else to.father->left = c;
107        to.ptr = c;
108        return to;
109    }
110    iterator_t find(elem_t t) { return find(t,begin()); }
111    iterator_t find(elem_t t, iterator_t p) {
112        if(p==end() || p.ptr->t == t) return p;
113        iterator_t l = find(t,p.left());
114        if (l!=end()) return l;
115        iterator_t r = find(t,p.right());
116        if (r!=end()) return r;
117        return end();
118    }
119    void clear() { erase(begin()); }
120    iterator_t begin() {
121        return iterator_t(header->left,
122                          iterator_t::L,header);
123    }
124    iterator_t end() { return iterator_t(); }
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```

126 void lisp_print(iterator_t n) {
127     if (n==end()) { cout << ". "; return; }
128     iterator_t r = n.right(), l = n.left();
129     bool is_leaf = r==end() && l==end();
130     if (is_leaf) cout << retrieve(n);
131     else {
132         cout << "(" << retrieve(n) << " ";
133         lisp_print(l);
134         cout << " ";
135         lisp_print(r);
136         cout << ")";
137     }
138 }
139 void lisp_print() { lisp_print(begin()); }
140
141 };

```

Código 3.16: Implementación de árboles con celdas enlazadas por punteros. Declaraciones. [Archivo: btree-bas.h]

En el código 3.16 se muestra la implementación correspondiente.

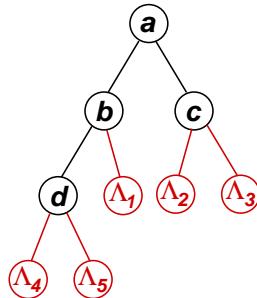


Figura 3.33: Iterators Λ en un árbol binario.

- **La clase iterator.** La clase `iterator_t` contiene un puntero a la celda, y otro al padre, como en el caso del AOO. Sin embargo el puntero `prev` que apuntaba al hermano a la izquierda, aquí ya no tiene sentido. Recordemos que el iterator nos debe permitir ubicar a las posiciones, incluso aquellas que son Λ . Para ello incluimos el iterator un miembro `side` de tipo tipo `enum side_t`, que puede tomar los valores `R` (right) y `L` (left). Por ejemplo consideremos el árbol de la figura 3.33. Además de los nodos $a - d$ existen 5 nodos Λ . Las posiciones de algunos nodos son representadas como sigue

- nodo b : `ptr=b, father=a, side=L`
- nodo c : `ptr=c, father=a, side=R`
- nodo d : `ptr=d, father=b, side=L`
- nodo Λ_1 : `ptr=NULL, father=b, side=R`
- nodo Λ_2 : `ptr=NULL, father=c, side=L`

- nodo Λ_3 : **ptr=NULL, father=c, side=R**
 - nodo Λ_4 : **ptr=NULL, father=d, side=L**
 - nodo Λ_5 : **ptr=NULL, father=d, side=R**
- La comparación de iterators (líneas 28–29) compara sólo los campos **ptr**, de manera que todos los iterators Λ resultan iguales entre sí (ya que tienen **ptr=NULL**). Como **end()** retorna un iterator Λ (ver más abajo), entonces esto habilita a usar los lazos típicos
- ```

1 while (c!=T.end()) {
2 // ...
3 c = c.right();
4 }
```
- La clase **btree** incluye un contador de celdas **cell\_count()** y constructor por copia **btree(const btree &)**, como para AOO.
  - La diferencia principal está en **insert(p,x)** y **erase(p)**. En **insert** se crea la celda (actualizando el contador de celdas) y se inserta el dato (líneas 77–79). Recordar que los campos punteros de la celda quedan en **NULL**, porque así se inicializan en el constructor de celdas. El único campo de celdas que se debe actualizar es, o bien el campo **left** o **right** de la celda padre. Cuál de ellos es el que debe apuntar a la nueva celda se deduce de **p.side** en el iterator. Finalmente se debe actualizar el iterator de forma que **ptr** apunte a la celda creada.
  - **erase(p)** elimina primero recursivamente todo el subárbol de los hijos izquierdo y derecho de **p**. Despues libera la celda actualizando el campo correspondiente del padre (dependiendo de **p.side**). También se actualiza el contador **cell\_count\_m** al liberar la celda. Notar la actualización del contador por la liberación de las celdas en los subárboles de los hijos se hace automáticamente dentro de la llamada recursiva, de manera que en **erase(p)** sólo hay que liberar explícitamente a la celda **p.ptr**.
  - El código de **splice(to, from)** es prácticamente un **erase** de **from** seguido de un **insert** en **to**.
  - La posición raíz del árbol se elige como el hijo izquierdo de la celda de encabezamiento. Esto es una convención, podríamos haber elegido también el hijo derecho.
  - El constructor por defecto de la clase **iterator** retorna un iterator no dereferenciable que no existe en el árbol. Todos sus punteros son nulos y **side** es un valor especial de **side\_t** llamado **NONE**. Insertar en este iterator es un error.
  - **end()** retorna un iterator no dereferenciable dado por el constructor por defecto de la clase **iterator\_t** (descripto previamente). Este iterator debería ser usado sólo para comparar. Insertar en este iterator es un error.

### 3.8.5.5. Interfaz avanzada

---

```

1 template<class T>
2 class btree {
3 /* ... */
4 public:
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```

5 class iterator {
6 /* ... */
7 public:
8 T &operator*();
9 T *operator>();
10 bool operator!=(iterator q);
11 bool operator==(iterator q);
12 iterator left();
13 iterator right();
14 };
15 iterator begin();
16 iterator end();
17 iterator insert(iterator p,T t);
18 iterator erase(iterator p);
19 iterator splice(iterator to,iterator from);
20 void clear();
21 };

```

**Código 3.17:** Interfaz avanzada para árboles binarios. [Archivo: btreet.h]

En el código 3.17 vemos una interfaz para árboles binarios incluyendo templates, clases anidadas y sobrecarga de operadores. Las diferencias principales son (ver también lo explicado en la sección §3.6)

- La clase es un template sobre el tipo contenido en el dato (**class T**) de manera que podremos declarar **btree<int>**, **btree<double>** ...
- La dereferenciación de nodo se hace sobrecargando los operadores **\*** y **->**, de manera que podemos hacer

```

1 x = *n;
2 *n = w;
3 y = n->member;
4 n->member = v;

```

donde **n** es de tipo iterator, **x,w** con de tipo **T** y **member** es algún campo de la clase **T** (si es una clase compuesta). También es válido hacer **n->f(...)** si **f** es un método de la clase **T**.

### 3.8.5.6. Ejemplo de uso. El algoritmo apply y principios de programación funcional.

A esta altura nos sería fácil escribir algoritmos que modifican los valores de un árbol, por ejemplo sumarle a todos los valores contenidos en un árbol un valor, o duplicarlos. Todos estos son casos particulares de un algoritmo más general **apply(Q,f)** que tiene como argumentos un árbol **Q** y una “función escalar” **T f(T)**. y le aplica a cada uno de los valores nodales la función en cuestión. Este es un ejemplo de “programación funcional”, es decir, programación en los cuales los datos de los algoritmos pueden ser también funciones.

C++ tiene un soporte básico para la programación funcional en la cual se pueden pasar “*punteros a funciones*”. Un soporte más avanzado se obtiene usando clases especiales que sobrecargan el operador **( )**, a tales funciones se les llama “*functors*”. Nosotros vamos a escribir ahora una herramienta simple llamada **apply(Q,f)** que aplica a los nodos de un árbol una función escalar **t f(T)** pasada por puntero.

```

1 template<class T>
2 void apply(btree<T> &Q,
3 typename btree<T>::iterator n,
4 T(*f)(T)) {
5 if (n==Q.end()) return;
6 *n = f(*n);
7 apply(Q,n.left(),f);
8 apply(Q,n.right(),f);
9 }
10 template<class T>
11 void apply(btree<T> &Q,T(*f)(T)) {
12 apply(Q,Q.begin(),f);

```

**Código 3.18:** Herramienta de programación funcional que aplica a los nodos de un árbol una función escalar.  
[Archivo: *apply.cpp*]

La función se muestra en el código 3.18. Recordemos que para pasar funciones como argumentos, en realidad se pasa *el puntero a la función*. Como las funciones a pasar son funciones que toman como un argumento un elemento de tipo **T** y retornan un elemento del mismo tipo, su “*signatura*” (la forma como se declara) es **T f(T)**. La declaración de punteros a tales funciones se hace reemplazando en la signatura el nombre de la función por **(\*f)**. De ahí la declaración en la línea 11, donde el segundo argumento de la función es de tipo **T(\*f)(T)**.

Por supuesto **apply** tiene una estructura recursiva y llama a su vez a una función auxiliar recursiva que toma un argumento adicional de tipo iterator. Dentro de esta función auxiliar el puntero a función **f** se aplica como una función normal, como se muestra en la línea 6.

Si **n** es  $\Lambda$  la función simplemente retorna. Si no lo está, entonces aplica la función al valor almacenado en **n** y después llama **apply** recursivamente a sus hijos izquierdo y derecho.

Otro ejemplo de programación funcional podría ser una función **reduce(Q,g)** que toma como argumentos un árbol **Q** y una función asociativa **T g(T,T)** (por ejemplo la suma, el producto, el máximo o el mínimo) y devuelve el resultado de aplicar la función asociativa a todos los valores nodales, hasta llegar a un único valor. Por ejemplo, si hacemos que **g(x,y)** retorne **x+y** retornará la suma de todas las etiquetas del árbol y si hacemos que retorne el máximo, entonces retornará el máximo de todas las etiquetas del árbol. Otra aplicación pueden ser “*filtros*”, como la función **prune\_odd** discutida en la sección §3.3.3. Podríamos escribir una función **remove\_if(Q,pred)** que tiene como argumentos un árbol **Q** y una función predicado **bool pred(T)**. La función **remove\_if** elimina todos los nodos **n** (y sus subárboles) para cuyos valores la función **pred(\*n)** retorna verdadero. La función **prune\_odd** se podría obtener entonces simplemente pasando a **remove\_if** una función predicado que retorna verdadero si el argumento es impar.

### 3.8.5.7. Implementación de la interfaz avanzada

```

1 #ifndef AED_BTREE_H
2 #define AED_BTREE_H
3
4 #include <iostream>
5 #include <cstddef>

```

```
6 #include <cstdlib>
7 #include <cassert>
8 #include <list>
9
10 using namespace std;
11
12 namespace aed {
13
14 //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
15 template<class T>
16 class btree {
17 public:
18 class iterator;
19 private:
20 class cell {
21 friend class btree;
22 friend class iterator;
23 T t;
24 cell *right,*left;
25 cell() : right(NULL), left(NULL) {}
26 };
27 cell *header;
28 enum side_t {NONE,R,L};
29 public:
30 static int cell_count_m;
31 static int cell_count() { return cell_count_m; }
32 class iterator {
33 private:
34 friend class btree;
35 cell *ptr,*father;
36 side_t side;
37 iterator(cell *p,side_t side_a,cell *f_a)
38 : ptr(p), side(side_a), father(f_a) {}
39 public:
40 iterator(const iterator &q) {
41 ptr = q.ptr;
42 side = q.side;
43 father = q.father;
44 }
45 T &operator*() { return ptr->t; }
46 T *operator->() { return &ptr->t; }
47 bool operator!=(iterator q) { return ptr!=q.ptr; }
48 bool operator==(iterator q) { return ptr==q.ptr; }
49 iterator() : ptr(NULL), side(NONE), father(NULL) {}
50
51 iterator left() { return iterator(ptr->left,L,ptr); }
52 iterator right() { return iterator(ptr->right,R,ptr); }
53 };
54 };
55
56 btree() {
57 header = new cell;
58 cell_count_m++;
```

```
59 header->right = NULL;
60 header->left = NULL;
61 }
62 btree<T>(const btree<T> &TT) {
63 if (&TT != this) {
64 header = new cell;
65 cell_count_m++;
66 header->right = NULL;
67 header->left = NULL;
68 btree<T> &TTT = (btree<T> &) TT;
69 if (TTT.begin() != TTT.end())
70 copy(begin(), TTT, TTT.begin());
71 }
72 }
73 btree &operator=(btree<T> &TT) {
74 if (this != &TT) {
75 clear();
76 copy(begin(), TT, TT.begin());
77 }
78 return *this;
79 }
80 ~btree() { clear(); delete header; cell_count_m--; }
81 iterator insert(iterator p, T t) {
82 assert(p==end());
83 cell *c = new cell;
84 cell_count_m++;
85 c->t = t;
86 if (p.side==R) p.father->right = c;
87 else p.father->left = c;
88 p.ptr = c;
89 return p;
90 }
91 iterator erase(iterator p) {
92 if(p==end()) return p;
93 erase(p.right());
94 erase(p.left());
95 if (p.side==R) p.father->right = NULL;
96 else p.father->left = NULL;
97 delete p.ptr;
98 cell_count_m--;
99 p.ptr = NULL;
100 return p;
101 }
102 iterator splice(iterator to, iterator from) {
103 if (from==end()) return to;
104 cell *c = from.ptr;
105 from.ptr = NULL;
106 if (from.side==R) from.father->right = NULL;
107 else from.father->left = NULL;
108
109 if (to.side==R) to.father->right = c;
110 else to.father->left = c;
111 to.ptr = c;
112 }
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```

113 return to;
114 }
115 iterator copy(iterator nq,btree<T> &TT,iterator nt) {
116 nq = insert(nq,*nt);
117 iterator m = nt.left();
118 if (m != TT.end()) copy(nq.left(),TT,m);
119 m = nt.right();
120 if (m != TT.end()) copy(nq.right(),TT,m);
121 return nq;
122 }
123 iterator find(T t) { return find(t,begin()); }
124 iterator find(T t,iterator p) {
125 if(p==end() || p.ptr->t == t) return p;
126 iterator l = find(t,p.left());
127 if (l!=end()) return l;
128 iterator r = find(t,p.right());
129 if (r!=end()) return r;
130 return end();
131 }
132 void clear() { erase(begin()); }
133 iterator begin() { return iterator(header->left,L,header); }
134
135 void lisp_print(iterator n) {
136 if (n==end()) { cout << "."; return; }
137 iterator r = n.right(), l = n.left();
138 bool is_leaf = r==end() && l==end();
139 if (is_leaf) cout << *n;
140 else {
141 cout << "(" << *n << " ";
142 lisp_print(l);
143 cout << " ";
144 lisp_print(r);
145 cout << ")";
146 }
147 }
148 void lisp_print() { lisp_print(begin()); }
149
150 iterator end() { return iterator(); }
151 };
152
153 template<class T>
154 int btree<T>::cell_count_m = 0;
155 }
156 #endif

```

**Código 3.19:** Implementación de la interfaz avanzada de árboles binarios por punteros. [Archivo: btree.h]

En el código 3.19 vemos una posible implementación de interfaz avanzada de AB con celdas enlazadas por punteros.

### 3.8.6. Arboles de Huffman

Los árboles de Huffman son un ejemplo interesante de utilización del TAD AB. El objetivo es comprimir archivos o mensajes de texto. Por simplicidad, supongamos que tenemos una cadena de  $N$  caracteres compuesta de un cierto conjunto reducido de caracteres  $C$ . Por ejemplo si consideramos las letras  $C = \{a, b, c, d\}$  entonces el mensaje podría ser  $abdcdaacabcbcdab$ . El objetivo es encontrar una representación del mensaje en bits (es decir una cadena de 0's y 1's) lo más corta posible. A esta cadena de 0's y 1's la llamaremos “el mensaje encodado”. El algoritmo debe permitir recuperar el mensaje original, ya que de esta forma, si la cadena de caracteres representa un archivo, entonces podemos guardar el mensaje encodado (que es más corto) con el consecuente ahorro de espacio.

Una primera posibilidad es el código provisto por la representación binaria ASCII de los caracteres. De esta forma cada carácter se encoda en un código de 8 bits. Si el mensaje tiene  $N$  caracteres, entonces el mensaje encodado tendrá una longitud de  $l = 8N$  bits, resultando en una longitud promedio de

$$\langle l \rangle = \frac{l}{N} = 8 \text{ bits/charácter} \quad (3.21)$$

Pero como sabemos que el mensaje sólo está compuesto de las cuatro letras  $a, b, c, d$  podemos crear un código de dos bits como el  $C1$  en la Tabla 3.2, de manera que un mensaje como  $abedcba$  se encoda en  $00011011100100$ . Desencodar un mensaje también es simple, vamos tomando dos caracteres del mensaje y consultando el código lo vamos convirtiendo al carácter correspondiente. En este caso la longitud del código pasa a ser de  $\langle l \rangle = 2$  bits/charácter, es decir la cuarta parte del código ASCII. Por supuesto esta gran compresión se produce por el reducido conjunto de caracteres utilizados. Si el conjunto de caracteres fuera de tamaño 8 en vez de 4 necesitaríamos al menos códigos de 3 bits, resultando en una tasa de 3 bits/charácter. En general si el número de caracteres es  $n_c$  la tasa será de

$$\langle l \rangle = \text{ceil}(\log_2 n_c) \quad (3.22)$$

Si consideramos texto común, el número de caracteres puede oscilar entre unos 90 y 128 caracteres, con lo cual la tasa será de 7 bits por carácter, lo cual representa una ganancia relativamente pequeña del 12.5 %.

| Letra | Código $C1$ | Código $C2$ | Código $C3$ |
|-------|-------------|-------------|-------------|
| $a$   | 00          | 0           | 0           |
| $b$   | 01          | 100         | 01          |
| $c$   | 10          | 101         | 10          |
| $d$   | 11          | 11          | 101         |

Tabla 3.2: Dos códigos posibles para un juego de 4 caracteres.

Una forma de mejorar la compresión es utilizar códigos de longitud variable, tratando de asignar códigos de longitud corta a los caracteres que tienen más probabilidad de aparecer y códigos más largos a los que tienen menos probabilidad de aparecer. Por ejemplo, si volvemos al conjunto de caracteres  $a, b, c, d$  y si suponemos que  $a$  tiene una probabilidad de aparecer del 70 % mientras que  $b, c, d$  sólo del 10 % cada uno, entonces podríamos considerar el código  $C2$  de la Tabla 3.2. Si bien la longitud en bits de  $a$  es menor en  $C2$  que en  $C1$ , la longitud de  $b$  y  $c$  es mayor, de manera que determinar cual de los dos códigos es mejor no es directo. Consideremos un mensaje típico de 100 caracteres. En promedio tendría 70 a's, 10 b's, 10c's y 10d's,

lo cual resultaría en un mensaje codificado de  $70 \times 1 + 10 \times 3 + 10 \times 3 + 10 \times 2 = 150$  bits, resultando en una longitud promedio de  $\langle l \rangle = 1.5$  bit/caracter. Notemos que en general

$$\begin{aligned}\langle l \rangle &= \frac{150 \text{ bits}}{100 \text{ caracteres}} \\ &= 0.70 \times 1 + 0.1 \times 3 + 0.1 \times 3 + 0.1 \times 2 \\ &= \sum_c P(c)l(c)\end{aligned}\tag{3.23}$$

Esta longitud media representa una compresión de 25% con respecto al  $C1$ . Por supuesto, la ventaja del código  $C2$  se debe a la gran diferencia en probabilidades entre  $a$  y los otros caracteres. Si la diferencia fuera menor, digamos  $P(a) = 0.4$ ,  $P(b) = P(c) = P(d) = 0.2$  ( $P(c)$  es la probabilidad de que en el mensaje aparezca un carácter  $c$ ), entonces la longitud de un mensaje típico de 100 caracteres sería  $40 \times 1 + 20 \times 3 + 20 \times 3 + 20 \times 2 = 200$  bits, o sea una tasa de 2 bits/caracter, igual a la de  $C1$ .

### **3.8.6.1. Condición de prefijos**

En el caso de que un código de longitud variable como el  $C_2$  sea más conveniente, debemos poder asegurarnos poder desencodar los mensajes, es decir que la relación entre mensaje y mensaje codificado sea única y que exista un algoritmo para encodar y desencodar mensajes en un tiempo razonable. Por empezar los códigos de los diferentes caracteres deben ser diferentes entre sí, pero esto no es suficiente. Por el ejemplo el código  $C_3$  tiene un código diferente para todos los caracteres, pero los mensajes  $dba$  y  $ccc$  se codifican ambos como 101010.

El error del código  $C_3$  es que el código de  $d$  empieza con el código de  $c$ . Decimos que el código de  $c$  es “prefijo” del de  $d$ . Así, al comenzar a desencodar el mensaje 101010. Cuando leemos los dos primeros bits 10, no sabemos si ya extraer una  $c$  o seguir con el tercer bit para formar una  $d$ . Notar que también el código de  $a$  es prefijo del de  $b$ . Por lo tanto, una condición para admitir un código es que cumpla con la “condición de prefijos”, a saber que *el código de un carácter no sea prefijo del código de ningún otro carácter*. Por ejemplo los códigos de longitud fija como el  $C_1$  trivialmente satisfacen la condición de prefijos. La única posibilidad de que violaran la condición sería si los códigos de dos caracteres son iguales, lo cual ya fue descartado.

### 3.8.6.2. Representación de códigos como árboles de Huffman

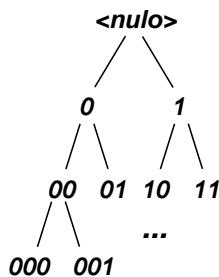


Figura 3.34: Representación de códigos binarios con árboles de Huffman.

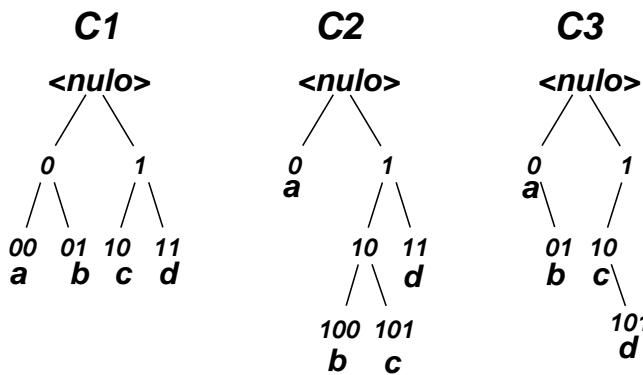


Figura 3.35: Representación de los códigos binarios  $C_1$ ,  $C_2$  y  $C_3$ .

Para códigos de longitud variable como el  $C_2$  la condición de prefijos se puede verificar carácter por carácter, pero es más simple y elegante si representamos el código como un “árbol de Huffman”. En esta representación, asociamos con cada nodo de un AB un código binario de la siguiente manera. Al nodo raíz lo asociamos con el código de longitud 0 para el resto de los nodos la definición es recursiva: si un nodo tiene código  $b_0b_1\dots b_{n-1}$  entonces el hijo izquierdo tiene código  $b_0b_1\dots b_{n-1}0$  y el hijo derecho  $b_0b_1\dots b_{n-1}1$ . Así se van generando todos los códigos binarios posibles. Notar que en el nivel  $l$  se encuentran todos los códigos de longitud  $l$ . Un dado código (como los  $C_1-C_3$ ) se puede representar como un AB marcando los nodos correspondientes a los códigos de cada uno de los caracteres y eliminando todos los nodos que no están contenidos dentro de los caminos que van desde esos nodos a la raíz. Es inmediato ver que la longitud del código de un carácter es la profundidad del nodo correspondiente. Una forma visual de construir el árbol es comenzar con un árbol vacío y comenzar a dibujar los caminos correspondientes al código de cada carácter. De esta manera los árboles correspondientes a los códigos  $C_1$  a  $C_3$  se pueden observar en la figura 3.35. Por construcción, a cada una de las hojas del árbol le corresponde un carácter. Notemos que en el caso del código  $C_3$  el camino del carácter  $a$  está contenido en el de  $b$  y el de  $c$  está contenido en el de  $d$ . De manera que la condición de prefijos se formula en la representación mediante árboles exigiendo que los caracteres estén sólo en las hojas, es decir *no en los nodos interiores*.

### 3.8.6.3. Códigos redundantes

| Letra | Código $C_2$ | Código $C_4$ |
|-------|--------------|--------------|
| $a$   | 0            | 0            |
| $b$   | 100          | 10100        |
| $c$   | 101          | 10101        |
| $d$   | 11           | 111          |

Tabla 3.3: Dos códigos posibles para un juego de 4 caracteres.

Consideremos ahora el código  $C_4$  mostrado en la Tabla 3.3 (ver figura 3.36). El código satisface la condición de prefijos, ya que los caracteres están asignados a 4 hojas distintas del árbol. Notemos que el árbol

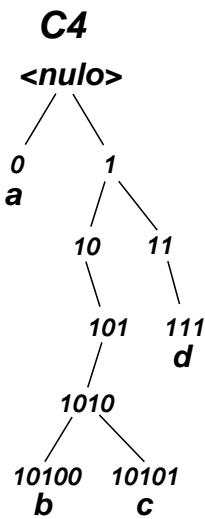


Figura 3.36: Código redundante.

tiene 3 nodos interiores 10, 101 y 11 que *tienen un sólo hijo*. Entonces podemos eliminar tales nodos interiores, “*subiendo*” todo el subárbol de 1010 a la posición del 10 y la del 111 a la posición del 11. El código resultante resulta ser igual al *C2* que ya hemos visto. Como todos los nodos suben y la profundidad de los nodos da la longitud del código, es obvio que la longitud del código medio será siempre menor para el código *C2* que para el *C4*, *independientemente de las probabilidades de los códigos de cada carácter*. Decimos entonces que un código como el *C4* que tiene nodos interiores con un sólo hijo es “*redundante*”, ya que puede ser trivialmente optimizado eliminando tales nodos y subiendo sus subárboles. Si un AB es tal que no contiene nodos interiores con un solo hijo se le llama “*árbol binario lleno*” (Full Binary Tree, FBT). Es decir, en un árbol binario lleno los nodos son o bien hojas, o bien nodos interiores con sus dos hijos.

#### 3.8.6.4. Tabla de códigos óptima. Algoritmo de búsqueda exhaustiva

El objetivo ahora es, *dado un conjunto de  $n_c$  caracteres, con probabilidades  $P_0, P_1, \dots, P_{n_c-1}$ , encontrar, de todos los posibles FBT con  $n_c$  hojas, aquel que minimiza la longitud promedio del código*. Como se discutió en primer capítulo, sección §1.1.4, una posibilidad es generar todos los posibles árboles llenos de  $n_c$  caracteres, calcular la longitud promedio de cada uno de ellos y quedarnos con el que tiene la longitud promedio mínima. Consideremos, por ejemplo el caso de  $n_c = 2$  caracteres. Es trivial ver que hay un solo posible FBT con dos hojas, el cual es mostrado en la figura 3.37 (árbol  $T_1$ ). Los posibles FBT de tres hojas se pueden obtener del de dos hojas convirtiendo a cada una de las hojas de  $T_1$  en un nodo interior, insertándole dos hijos. Así, el árbol  $T_2$  se obtiene insertándole dos nodos a la hoja izquierda de  $T_1$  y el  $T_3$  agregándole a la hoja derecha. En ambos casos los nodos agregados se marcan en verde. Así siguiendo, por cada FBT de 3 hojas se pueden obtener 3 FBT de 4 hojas, “*bifurcando*” cada una de sus hojas. En la figura  $T_4, T_5$  y  $T_6$  se obtienen bifurcando las hojas de  $T_2$  y  $T_7$ ,  $T_8$  y  $T_9$  las hojas de  $T_3$ . Como hay 2 FBT de 3 hojas, se obtienen en total 6 FBT de 4 hojas. Siguiendo con el razonamiento, el número de FBT de  $n_c$  hojas es  $(n_c - 1)!$ . Notar que algunas de las estructuras son redundantes, por ejemplo los árboles  $T_6$  y  $T_7$  son equivalentes.

Notemos que hasta ahora hemos considerado solamente “*la estructura del árbol*”. Los códigos se obtie-

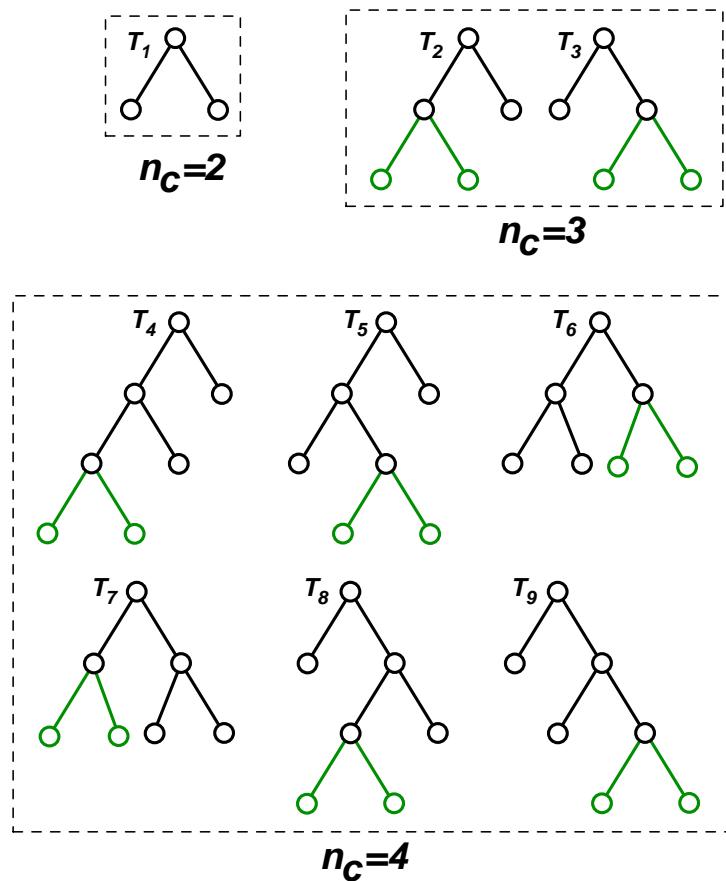


Figura 3.37: Posibles árboles binarios llenos con  $n_c$  hojas.

nen asignando cada uno de los  $n_c$  caracteres a una de las hojas del árbol. Entonces, por cada uno de los árboles de  $T_4$  a  $T_9$  se pueden obtener  $4! = 24$  posibles tablas de códigos permutando los caracteres entre sí. Por ejemplo, los tres árboles de la figura 3.38 son algunos de los árboles que se obtienen de la estructura de  $T_4$  en la figura 3.37 permutando las letras entre sí. Como el número de permutaciones de  $n_c$  caracteres es  $n_c!$ , tenemos que en total hay a lo sumo  $(n_c - 1)!n_c!$  posibles tablas de códigos. Usando las herramientas desarrolladas en la sección §1.3 (en particular §1.3.9), vemos que

$$T(n_c) = (n_c - 1)!n_c! < (n_c!)^2 = O(n_c^{2n_c}) \quad (3.24)$$

Repasando la experiencia que tuvimos con la velocidad de crecimiento de los algoritmos no polinomiales (ver §1.1.6), podemos descartar esta estrategia si pensamos aplicar el algoritmo a mensajes con  $n_c > 20$ .

**3.8.6.4.1. Generación de los árboles** Una forma de generar todos los árboles posibles es usando recursión. Dados una serie de árboles  $T_0, T_1, \dots, T_{n-1}$ , llamaremos  $\text{comb}(T_0, T_1, \dots, T_{n-1})$  a la lista de todos los posibles árboles formados combinando  $T_0, \dots, T_{n-1}$ . Para  $n = 2$  hay una sola combinación que consiste en poner a  $T_0$  como hijo izquierdo y  $T_1$  como hijo derecho de un nuevo árbol. Para aplicar el algoritmo de

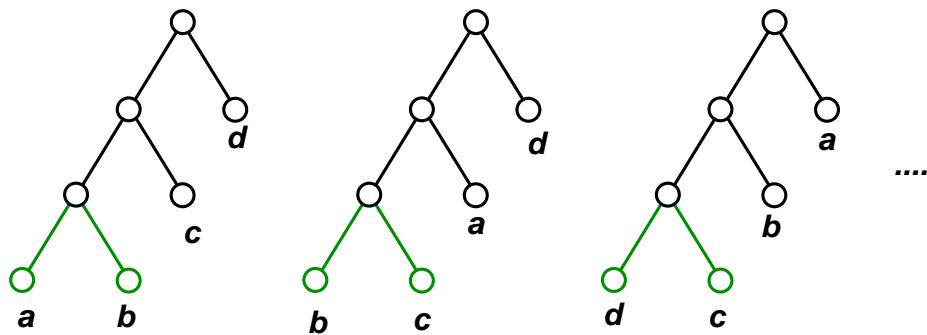


Figura 3.38: Posibles tablas de códigos que se obtienen permutando las letras en las hojas de una estructura dada.

búsqueda exhaustiva debemos poder generar la lista de árboles  $\text{comb}(T_0, T_1, \dots, T_{n_{\text{char}}-1})$  donde  $T_j$  es un árbol que contiene un sólo nodo con el carácter  $j$ -ésimo. Todas las combinaciones se pueden hallar tomando cada uno de los posibles pares de árboles  $(T_i, T_j)$  de la lista, combinándolos e insertando  $\text{comb}(T_i, T_j)$  en la lista, después de eliminar los árboles originales.

$$\begin{aligned} \text{comb}(T_0, T_1, T_2, T_3) = & (\text{comb}(\text{comb}(T_0, T_1), T_2, T_3), \\ & \text{comb}(\text{comb}(T_0, T_2), T_1, T_3), \\ & \text{comb}(\text{comb}(T_0, T_3), T_1, T_2), \\ & \text{comb}(\text{comb}(T_1, T_2), T_0, T_3), \\ & \text{comb}(\text{comb}(T_1, T_3), T_0, T_2), \\ & \text{comb}(\text{comb}(T_2, T_3), T_0, T_1)) \end{aligned} \quad (3.25)$$

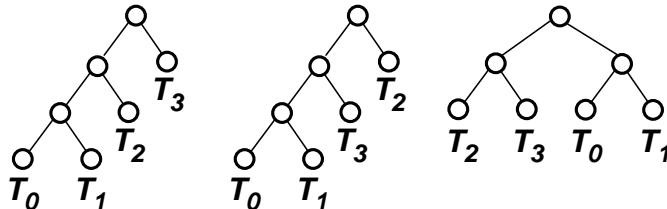


Figura 3.39: Generación de todos los posibles árboles binarios llenos de 4 hojas.

Ahora, recursivamente, cada uno de las sublistas se expande a su vez en 3 árboles, por ejemplo

$$\begin{aligned} \text{comb}(\text{comb}(T_0, T_1), T_2, T_3) = & (\text{comb}(\text{comb}(\text{comb}(T_0, T_1), T_2), T_3), \\ & \text{comb}(\text{comb}(\text{comb}(T_0, T_1), T_3), T_2), \\ & \text{comb}(\text{comb}(T_2, T_3), \text{comb}(T_0, T_1))) \end{aligned} \quad (3.26)$$

Estos tres posibles combinaciones pueden observarse en la figura 3.39. Para la figura hemos asumido que los árboles originales son nodos simples, pero podrían ser a su vez árboles.

Entonces, cada una de las sublistas en (3.25) genera 3 árboles. En general vamos a tener

$$N_{\text{abc}}(n) = (\text{número de posibles pares a escoger de } n \text{ árboles}) \times N_{\text{abc}}(n - 1) \quad (3.27)$$

donde  $N_{\text{abc}}(n)$  es el número de árboles binarios llenos de  $n$  hojas. El número de posibles pares a escoger de  $n$  árboles es  $n(n - 1)/2$ . La división por 2 proviene de que no importa el orden entre los elementos del par. De manera que

$$N_{\text{abc}}(n) = \frac{n(n - 1)}{2} N_{\text{abc}}(n - 1) \quad (3.28)$$

Aplicando recursivamente esta relación llegamos a

$$\begin{aligned} N_{\text{abc}}(n) &= \frac{n(n - 1)}{2} \frac{(n - 1)(n - 2)}{2} \cdots \frac{3 \cdot 2}{2} \\ &= \frac{n!(n - 1)!}{2^{n+1}} = \frac{(n!)^2}{n 2^{n+1}} \end{aligned} \quad (3.29)$$

Usando la aproximación de Stirling (1.34) llegamos a

$$N_{\text{abc}}(n) = O\left(\frac{n^{2n-1}}{2^n}\right) \quad (3.30)$$

Notar que esta estimación es menor que la obtenida en la sección anterior (3.24) ya que en aquella estimación se cuentan dos veces árboles que se obtienen intercambiando los hijos.

### 3.8.6.4.2. Agregando un condimento de programación funcional

```

1 typedef void (*traverse_tree_fun) (btree_t &T, void *data);
2
3 typedef list< btree<int> > list_t;
4 typedef list_t::iterator pos_t;
5
6 void comb(list_t &L, traverse_tree_fun f, void *data=NULL) {
7 if (L.size()==1) {
8 f(*L.begin(), data);
9 return;
10 }
11 int n=L.size();
12 for (int j=0; j<n-1; j++) {
13 for (int k=j+1; k<n; k++) {
14 btree_t T;
15 T.insert(T.begin(), -1);
16 node_t m = T.begin();
17
18 pos_t pk=L.begin();
19 for (int kk=0; kk<k; kk++) pk++;
20 T.splice(m.left(), pk->begin());
21 L.erase(pk);
22
23 pos_t pj=L.begin();
```

```

24 for (int jj=0; jj<j; jj++) pj++;
25 T.splice(m.right(),pj->begin());
26 L.erase(pj);
27
28 pos_t p = L.insert(L.begin(),btree_t());
29 p->splice(p->begin(),T.begin());
30
31 comb(L,f,data);
32
33 p = L.begin();
34 m = T.splice(T.begin(),p->begin());
35 L.erase(p);
36
37 pj=L.begin();
38 for (int jj=0; jj<j; jj++) pj++;
39 pj = L.insert(pj,btree_t());
40 pj->splice(pj->begin(),m.right());
41
42 pk=L.begin();
43 for (int kk=0; kk<k; kk++) pk++;
44 pk = L.insert(pk,btree_t());
45 pk->splice(pk->begin(),m.left());
46
47 }
48 }
49 }
```

**Código 3.20:** Implementación del algoritmo que genera todos los árboles llenos. [Archivo: allabc.cpp]

En el código 3.20 vemos una implementación del algoritmo descripto, es decir la función `comb(L, f)` genera todas las posibles combinaciones de árboles contenidos en la lista `L` y les aplica una función `f`. Este es otro ejemplo de “*programación funcional*”, donde uno de los argumentos de la función es, a su vez, otra función. Notemos que no cualquier función puede pasarse a `comb()` sino que tiene que responder a la “*signatura*” dada en la línea 1. Recordemos que la *signatura* de una función son los tipos de los argumentos de una función. En este caso el tipo `traverse_tree_fun` consiste en una función que retorna `void` y toma como argumentos una referencia a un árbol binario y puntero genérico `void *`. Cualquier función que tenga esa *signatura* podrá ser pasada a `comb()`.

En nuestro caso, debemos pasar a `comb()` una función que calcula la longitud promedio de código para un dado árbol y retenga el mínimo de todos aquellos árboles que fueron visitados hasta el momento. Como la `f` debe calcular el mínimo de todas las longitudes de código debemos guardar el estado actual del cálculo (la mínima longitud de código calculada hasta el momento) en una variable global. Para evitar el uso de variables globales, que son siempre una fuerte de error, utilizamos una variable auxiliar `void *data` que contiene el estado del cálculo y es pasado a `f` en las sucesivas llamadas.

De esta forma dividimos el problema en dos separados. Primero escribir un algoritmo `comb()` que recorre todos los árboles y le aplica una función `f`, pasándole el estado de cálculo actual `void *data` y, segundo, escribir la función `f` apropiada para este problema. Notar que de esta forma el algoritmo `comb()` es sumamente genérico y puede ser usado para otras aplicaciones que requieren recorrer todos los posibles árboles binarios llenos.

Una estrategia más simple sería escribir una función basada en este mismo algoritmo que genere todos los árboles posibles en una lista. Luego se recorre la lista calculando la longitud media para cada árbol y reteniendo la menor. La implementación propuesta es mucho más eficiente en uso de memoria ya que en todo momento sólo hay un árbol generado y también de tiempo de cálculo ya que en esta estrategia más simple los árboles se deben ir combinando por copia, mientras que en la propuesta el mismo árbol va pasando de una forma a la otra con simples operaciones de `splice()`.

**3.8.6.4.3. El algoritmo de combinación** El algoritmo verifica primero la condición de terminación de la recursión. Cuando la longitud de la lista es 1, entonces ya tenemos uno de los árboles generados y se le aplica la función `f`. Caso contrario el algoritmo prosigue generando  $n(n - 1)/2$  listas de árboles que se obtienen combinando alguno de los pares posibles  $T_i, T_j$  y reemplazando  $T_i, T_j$  por la combinación  $\text{comb}(T_i, T_j)$ . Una posibilidad es para cada par generar una copia de la lista de árboles, generar la combinación y reemplazar y llamar recursivamente a `comb()`. Pero esto resultaría en copiar toda la lista de árboles (incluyendo la copia de los árboles mismos), lo cual resultaría en un costo excesivo. En la implementación presentada esto se evita aplicando la transformación en las líneas 14–29 y deshaciendo la transformación en las líneas 33–45, después de haber llamado recursivamente a la función.

Detengámonos en la combinación de los árboles. Notemos que el lazo externo se hace sobre enteros, de manera que tanto para `j` como para `k` hay que realizar un lazo sobre la lista para llegar a la posición correspondiente. (Los lazos sobre `kk` y `jj`). En ambos casos obtenemos las posiciones correspondientes en la lista `pj` y `pk`. Para hacer la manipulación creamos un árbol temporal `T`. Insertamos un nodo interior con valor `-1` para diferenciar de las hojas, que tendrán valores no negativos (entre `0` y `nchar-1`). Cada uno de los árboles es movido (con `splice()`) a uno de los hijos de la raíz del árbol `T`. Notar que como `pj` y `pk` son iterators, los árboles correspondientes son `*pj` y `*pk` y para tomar la raíz debemos hacer `pj->begin()` y `pk->begin()`. Después de mover los árboles, la posición es eliminada de la lista con el `erase()` de listas. Notar que primero se elimina `k`, que es mayor que `j`. Si elimináramos primero `j` entonces el lazo sobre `kk` debería hacerse hasta `k-1` ya que todas las posiciones después de `j` se habrían corrido. Finalmente una nueva posición es insertada en el comienzo de la lista (cualquier posición hubiera estado bien, por ejemplo `end()`) y todo el árbol `T` es movido (con `splice()`) al árbol que está en esa posición.

Después de llamar a `comb()` recursivamente, debemos volver a “desarmar” el árbol que está en la primera posición de `L` y retornar los dos subárboles a las posiciones `j` y `k`. Procedemos en forma inversa a la combinación. Movemos todo el árbol en `L.begin()` a `T` y eliminamos la primera posición. Luego buscamos la posición `j`-ésima con un lazo e insertamos un árbol vacío, moviendo todo la rama derecha a esa posición. Lo mismo se hace después con la rama izquierda. La lista de árboles `L` debería quedar después de la línea 45 igual a la que comenzó el lazo en la línea 14, de manera que en realidad todo `comb()` no debe modificar a `L`. Notar que esto es a su vez necesario para poder llamar a `comb` en la línea 31, ya que si lo modificara no sería posible después desarmar la combinación.

---

```

1 double codelen(btreet_t &T, node_t n,
2 const vector<double> &prob, double &w) {
3 if (n.left() == T.end()) {
4 w = prob[*n];;
5 return 0. ;
6 } else {

```

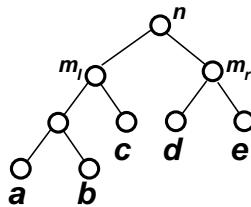


Figura 3.40: Cálculo de la longitud media del código.

```

7 double wl,wr,lr,ll;
8 ll = codelen(T,n.left(),prob,wl);
9 lr = codelen(T,n.right(),prob,wr);
10 w = wr+wl;
11 return wl+wr+ll+lr;
12 }
13 }
14
15 double codelen(btree_t &T,
16 const vector<double> &prob) {
17 double ww;
18 return codelen(T,T.begin(),prob,ww);
19 }
```

#### 3.8.6.4.4. Función auxiliar que calcula la longitud media del código. [Archivo: codelen.cpp]

#### Código 3.21: Cálculo de la longitud media

Consideremos el ejemplo del código de la figura 3.40. Tomando como base la expresión para la longitud media dada por (3.23) definimos la cantidad  $\text{cdln}(n)$  como

$$\text{cdln}(n) = P(a) 3 + P(b) 3 + P(c) 2 + P(d) 2 + P(e) 2 \quad (3.31)$$

es decir la suma de los productos de las probabilidades de las hojas del árbol por su distancia al nodo en cuestión. En el caso de ser  $n$  la raíz del árbol,  $\text{cdln}(n)$  es igual a la longitud media. Para nodos que no son la raíz el resultado no es la longitud media del subárbol correspondiente ya que la suma de las probabilidades en el subárbol es diferente de 1. Con un poco de álgebra podemos hallar una expresión recursiva

$$\begin{aligned}
 \text{cdln}(n) &= P(a) 3 + P(b) 3 + P(c) 2 + P(d) 2 + P(e) 2 \\
 &= [P(a) + P(b) + P(c)] + [P(a) 2 + P(b) 2 + P(c) 1] \\
 &\quad + [P(d) + P(e)] + [P(d) 1 + P(e) 1] \\
 &= P(m_l) + \text{cdln}(m_l) + P(m_r) + \text{cdln}(m_r)
 \end{aligned} \quad (3.32)$$

donde

$$\begin{aligned}
 P(m_l) &= P(a) + P(b) + P(c) \\
 P(m_r) &= P(d) + P(e)
 \end{aligned} \quad (3.33)$$

son la suma de probabilidades de la rama izquierda y derecha y

$$\begin{aligned} \text{cdln}(m_l) &= P(a) 2 + P(b) 2 + P(c) 1 \\ \text{cdln}(m_r) &= P(d) 1 + P(e) 1 \end{aligned} \quad (3.34)$$

son los valores de la función **cdln** en los hijos izquierdo y derecho.

En el código 3.21 se observa la función que calcula, dado el árbol binario y el vector de probabilidades **prob** la longitud media del código. Esta es una típica función recursiva como las ya estudiadas. La función **codelen(T,n,prob,w)** retorna la función **cdln** aplicada al nodo **n** aplicando la expresión recursiva, mientras que por el argumento **w** retorna la suma de las probabilidades de todas las hojas del subárbol. Para el nodo raíz del árbol **codelen()** coincide con la longitud media del código. La recursión se corta cuando **n** es una hoja, en cuyo caso **w** es la probabilidad de la hoja en cuestión (lo cual se obtiene de **prob**) y **cdln** es 0.

La función wrapper **codelen(T,prob)** se encarga de pasar los argumentos apropiados a la función recursiva auxiliar.

---

```

1 struct huf_exh_data {
2 btree_t best;
3 double best_code_len;
4 const vector<double> *prob;
5 };
6
7 void min_code_len(btree_t &T, void *data) {
8 huf_exh_data *hp = (huf_exh_data *)data;
9 double l = codelen(T,*(hp->prob));
10 if (l < hp->best_code_len) {
11 hp->best_code_len = l;
12 hp->best = T;
13 }
14 }
15
16 //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
17 void
18 huffman_exh(const vector<double> &prob, btree_t &T) {
19 int nchar = prob.size();
20 list_t L;
21 pos_t p;
22 huf_exh_data h;
23 h.prob = &prob;
24 h.best_code_len = DBL_MAX;
25 for (int j=0; j<nchar; j++) {
26 p = L.insert(L.end(),btree_t());
27 p->insert(p->begin(),j);
28 }
29 comb(L,&min_code_len,&h);
30 T.clear();
31 T.splice(T.begin(),h.best.begin());
32 }
```

---

### 3.8.6.4.5. Uso de comb y codeLEN Código 3.22: Función auxiliar para pasar a comb() [Archivo: hufexh.cpp]

Ahora debemos escribir una función auxiliar con signatura `void min_code_len(btreet_t &T, void *data);` para pasar a `comb()`. Esta debe encargarse de calcular la longitud media y mantener el mínimo de todas las longitudes medias encontradas y el árbol correspondiente. Llamamos a esto el “*estado del cálculo*” y para ello definimos una estructura auxiliar `huf_exh_data`. Además debemos pasar el vector de probabilidades `prob`. Para evitar de pasarlo por una variable global agregamos a `huf_exh_data` un puntero al vector.

Como `comb()` es genérica el estado global se representa por un puntero genérico `void *`. El usuario debe encargarse de convertir el puntero a un puntero a su estructura real, a través de una “conversión estática” (“static cast”). Esto se hace en la línea 8 donde se obtiene un puntero a la estructura real de tipo `huf_exh_data` que está detrás del puntero genérico. `T` es una referencia a uno de los árboles generados. Recordemos que `comb()` va a ir llamando a nuestra función `min_code_len()` con cada uno de los árboles que genere. En `min_code_len` simplemente calculamos la longitud promedio con `codeLEN()` y si es menor que el mínimo actual actualizamos los valores.

Finalmente, la función `huffman_exh(prob, T)` calcula en forma exhaustiva el árbol de menor longitud media usando las herramientas desarrolladas. Declara una lista de árboles `L` e inserta en ellos `nchar` árboles que constan de un sólo nodo (hoja) con el carácter correspondiente (en realidad un entero de 0 a `nchar-1`). También declara la estructura `h` que va a representar el estado del cálculo e inicializa sus valores, por ejemplo inicializa `best_code_len` a `DBL_MAX` (un doble muy grande). Luego llama a `comb()` el cual va a llamar a `min_code_len()` con cada uno de los árboles y un puntero al estado `h`. Una vez terminado `comb()` en `h.best` queda el árbol con el mejor código. Este lo movemos al árbol de retorno `T` con `splice()`.

### 3.8.6.5. El algoritmo de Huffman

Un algoritmo que permite obtener tablas de código óptimas en tiempos reducidos es el “*algoritmo de Huffman*”. Notemos que es deseable que los caracteres con mayor probabilidad (los más “pesados”) deberían estar cerca de la raíz, para tener un código lo más corto posible. Ahora bien, para que un carácter pesado pueda “subir”, es necesario que otros caracteres (los más livianos) “bajen”. De esta manera podemos pensar que hay una competencia entre los caracteres que tratan de estar lo más cerca posible de la raíz, pero tienden a “ganar” los más pesados. Caracteres con probabilidades similares deberían tender a estar en niveles parecidos. Esto sugiere ir apareando caracteres livianos con pesos parecidos en árboles que pasan a ser caracteres “comodines” que representan a un conjunto de caracteres.

**Ejemplo 3.3:** *Consigna:* Dados los caracteres  $a, b, c, d, e, f$  con pesos  $P(a) = 0.4, P(b) = 0.15, P(c) = 0.15, P(d) = 0.1, P(e) = 0.1, P(f) = 0.1$ , encontrar una tabla óptima de codificación, usando el algoritmo de Huffman.

El algoritmo procede de la siguiente manera,

1. Inicialmente se crean  $n_c$  árboles (tantos como caracteres hay). Los árboles tienen una sola hoja asociada con cada uno de los caracteres. Al árbol se le asocia un peso total, inicialmente cada árbol tiene el peso del carácter correspondiente.
2. En sucesivas iteraciones el algoritmo va combinando los dos árboles con menor peso en uno sólo. Si hay varias posibilidades se debe escoger aquella que da el menor peso del árbol combinado. Como en

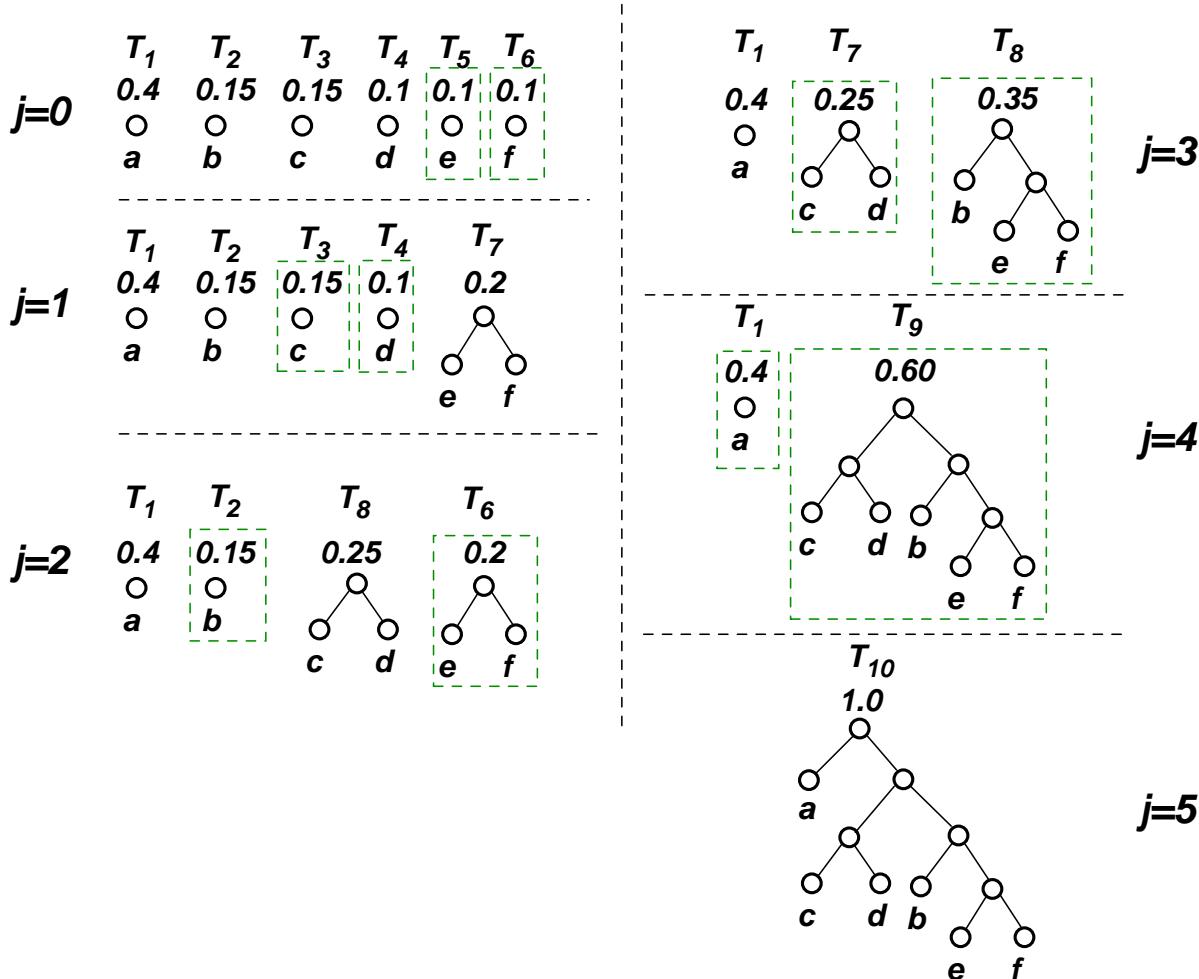


Figura 3.41: Algoritmo de Huffman.

cada combinación desaparecen dos árboles y aparece uno nuevo en  $n_c - 1$  iteraciones queda un sólo árbol con el código resultante.

Por ejemplo, en base a los datos del problema se construyen los 6 árboles de la figura 3.41 (marcados con  $j = 0$ ,  $j$  es el número de iteración). De los 6 árboles, los de menor peso son los  $T_4$ ,  $T_5$  y  $T_6$  correspondientes a los caracteres  $d$ ,  $e$  y  $f$ . Todos tienen la misma probabilidad 0.1. *Cuando varios árboles tienen la misma probabilidad se puede tomar cualquiera dos de ellos.* En el ejemplo elegimos  $T_5$  y  $T_6$ . Estos dos árboles se combinan como subárboles de un nuevo árbol  $T_7$ , con un peso asociado 0.2, igual a la suma de los pesos de los dos árboles combinados. Despues de esta operación quedan sólo los árboles  $T_1$ ,  $T_2$ ,  $T_3$ ,  $T_4$  y  $T_7$  con pesos 0.4, 0.15, 0.15, 0.1 y 0.2. Notar que en tanto antes como despues de la combinación, la suma de los pesos de los árboles debe dar 1 (como debe ocurrir siempre con las probabilidades).

Ahora los árboles de menor peso son  $T_2$ ,  $T_3$  y  $T_4$  con pesos 0.15, 0.15 y 0.1. Cualquiera de las combinaciones  $T_2$  y  $T_4$  o  $T_3$  y  $T_4$  son válidas, dando una probabilidad combinada de 0.25. Notar que la combinación

$T_2$  y  $T_3$  no es válida ya que daría una probabilidad combinada 0.3, mayor que las otras dos combinaciones. En este caso elegimos  $T_3$  y  $T_4$  resultando en el árbol  $T_8$  con probabilidad combinada 0.25. En las figuras se observa las siguientes etapas, hasta llegar al árbol final  $T_{10}$ .

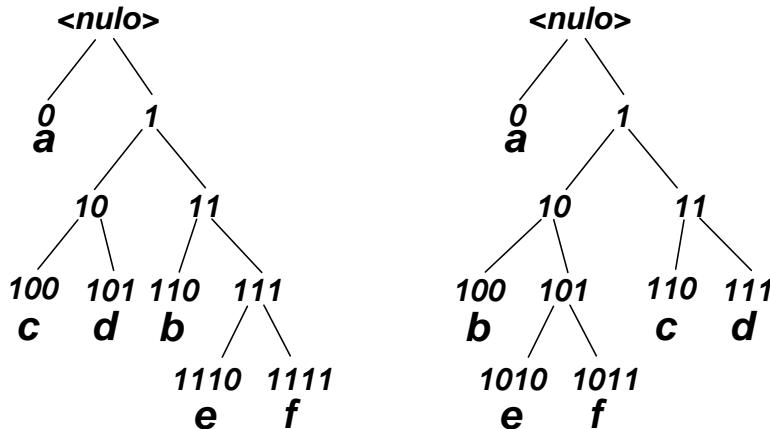


Figura 3.42: Arboles de códigos resultante del algoritmo de Huffman.

En la figura 3.42 se observa a la izquierda el árbol con los códigos correspondientes. La longitud media del código es, usando (3.23),

$$\langle l \rangle = 0.4 \times 1 + 0.15 \times 3 + 0.15 \times 3 + 0.1 \times 3 + 0.1 \times 5 + 0.1 \times 4 = 2.5 \text{ bits/caracter.} \quad (3.35)$$

resultando en una ganancia del 17 % con respecto a la longitud media de 3 que correspondería a la codificación con códigos de igual longitud dada por (3.22).

Notemos que al combinar dos árboles entre sí no estamos especificando cuál queda como hijo derecho y cuál como izquierdo. Por ejemplo, si al combinar  $T_7$  con  $T_8$  para formar  $T_9$  dejamos a  $T_7$  a la derecha, entonces el árbol resultante será el mostrado en la figura 3.42 a la derecha. Si bien el árbol resultante (y por lo tanto la tabla de códigos) es diferente, la longitud media del código será la misma, ya que la profundidad de cada carácter, que es su longitud de código, es la misma en los dos árboles. Por ejemplo,  $e$  y  $f$  tienen longitud 4 en los dos códigos.

A diferencia de los algoritmos heurísticos, la tabla de códigos así generada es “óptima” (la mejor de todas), y la longitud media por carácter coincide con la que se obtendría aplicando el algoritmo de búsqueda exhaustiva pero a un costo mucho menor.

### 3.8.6.6. Implementación del algoritmo

```

1 struct huffman_tree {
2 double p;
3 btree<int> T;
4 };
5
6 void
7 huffman(const vector<double> &prob, btree<int> &T) {

```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```
8 typedef list<huffman_tree> bosque_t;
9
10 // Contiene todos los arboles
11 bosque_t bosque;
12 // Numero de caracteres del codigo
13 int N = prob.size();
14 // Crear los arboles iniciales poniendolos en
15 // una lista Los elementos de la lista contienen
16 // la probabilidad de cada caracter y un arbol
17 // con un solo nodo. Los nodos interiores del
18 // arbol tienen un -1 (es solo para
19 // consistencia) y las hojas tienen el indice
20 // del caracter. (entre 0 y N-1)
21 for (int j=0; j<N; j++) {
22 // Agrega un nuevo elemento a la lista
23 bosque_t::iterator htree =
24 bosque.insert(bosque.begin(),huffman_tree());
25 htree->p = prob[j];
26 htree->T.insert(htree->T.begin(),j);
27 }
28
29 // Aqui empieza el algoritmo de Huffman.
30 // Tmp va a contener el arbol combinado
31 btree<int> Tmp;
32 for (int j=0; j<N-1; j++) {
33 // En la raiz de Tmp (que es un nodo interior)
34 // ponemos un -1 (esto es solo para chequear).
35 Tmp.insert(Tmp.begin(),-1);
36 // Tmp_p es la probabilidad del arbol combinado
37 // (la suma de las probabilidades de los dos subarboles)
38 double Tmp_p = 0.0;
39 // Para 'k=0' toma el menor y lo pone en el
40 // hijo izquierdo de la raiz de Tmp. Para 'k=1' en el
41 // hijo derecho.
42 for (int k=0; k<2; k++) {
43 // recorre el 'bosque' (la lista de arboles)
44 // busca el menor. 'qmin' es un iterator al menor
45 bosque_t::iterator q = bosque.begin(), qmin=q;
46 while (q != bosque.end()) {
47 if (q->p < qmin->p) qmin = q;
48 q++;
49 }
50 // Asigna a 'node' el hijo derecho o izquierdo
51 // de la raiz de 'Tmp' dependiendo de 'k'
52 btree<int>::iterator node = Tmp.begin();
53 node = (k==0 ? node.left() : node.right());
54 // Mueve todo el nodo que esta en 'qmin'
55 // al nodo correspondiente de 'Tmp'
56 Tmp.splice(node,qmin->T.begin());
57 // Acumula las probabilidades
58 Tmp_p += qmin->p;
59 // Elimina el arbol correspondiente del bosque.
60 bosque.erase(qmin);
```

```

61 }
62 // Inserta el arbol combinado en el bosque
63 bosque_t::iterator r =
64 bosque.insert(bosque.begin(),huffman_tree());
65 // Mueve todo el arbol de 'Tmp' al nodo
66 // recien insertado
67 r->T.splice(r->T.begin(),Tmp.begin());
68 // Pone la probabilidad en el elemento de la
69 // lista
70 r->p = Tmp_p;
71 }
72 // Debe haber quedado 1 solo elemento en la lista
73 assert(bosque.size()==1);
74 // Mueve todo el arbol que quedo a 'T'
75 T.clear();
76 T.splice(T.begin(),bosque.begin()->T.begin());
77 }
```

**Código 3.23:** Implementación del algoritmo de Huffman. [Archivo: huf.cpp]

En el código 3.23 vemos una implementación del algoritmo de Huffman usando la interfaz avanzada mostrada en el código 3.17.

- El código se basa en usar una lista de estructuras de tipo **huffman\_tree**, definida en las líneas 1-4 que contienen el árbol en cuestión y su probabilidad **p**.
- Los árboles son de tipo **btree<int>**. En los valores nodales almacenaremos para las hojas un índice que identifica al carácter correspondiente. Este índice va entre 0 y **N-1** donde **N** es el número de caracteres. **prob** es un vector de dobles de longitud **N**. **prob[j]** es la probabilidad del carácter **j**. En los valores nodales de los nodos interiores del árbol almacenaremos un valor **-1**. Este valor no es usado normalmente, sólo sirve como un chequeo adicional.
- El tipo **bosque\_t** definido en la línea 8 es un alias para una lista de tales estructuras.
- La función **huffman(prob,T)** toma un vector de dobles (las probabilidades) **prob** y calcula el árbol de Huffman correspondiente.
- En el lazo de las líneas 21–27 los elementos del bosque son inicializados, insertando el único nodo.
- El lazo de las líneas 31–77 es el lazo del algoritmo de Huffman propiamente dicho. En cada paso de este lazo se toman los dos árboles de **bosque** con probabilidad menor. Se combinan usando **splice** en un árbol auxiliar **Tmp**. Los dos árboles son eliminados de la lista y el combinado con la suma de las probabilidades es insertado en el bosque.
- Inicialmente **Tmp** está vacío, y dentro del lazo, la última operación es hacer un **splice** de todo **Tmp** a uno de los árboles del bosque, de manera que está garantizado que al empezar el lazo **Tmp** siempre está vacío.
- Primero insertamos en **Tmp** un nodo interior (con valor -1). Los dos árboles con pesos menores quedarán como hijos de este nodo raíz.
- La variable **Tmp\_p** es la probabilidad combinada. Al empezar el cuerpo del lazo es inicializada a 0 y luego le vamos acumulando las probabilidades de cada uno de los dos árboles a combinar.

- Para evitar la duplicación de código hacemos la búsqueda de los dos menores dentro del lazo sobre **k**, que se hace para **k=0** y **k=1**. Para **k=0** se encuentra el árbol del bosque con menor probabilidad y se inserta en el subárbol izquierdo de **Tmp**. Para **k=1** se inserta al segundo menor en el hijo derecho de de **Tmp**.
- La búsqueda del menor se hace en el lazo de las líneas 46–49. Este es un lazo sobre listas. **q** es un iterator a **list<huffman\_tree>** de manera que **\*q** es de tipo **huffman\_tree** y la probabilidad correspondiente está en **(\*q).p** o, lo que es lo mismo **q->p**. Vamos guardando la posición en la lista del mínimo actual en la posición **qmin**.
- En las líneas 52–53 hacemos que el iterator **node** apunte primero al nodo raíz de **Tmp** y después al hijo izquierdo o derecho, dependiendo de **k**.
- El árbol con menor probabilidad es pasado del elemento del bosque al subárbol correspondiente de **Tmp** en el splice de la línea 56. Recordar que, como el elemento de la lista es de tipo **huffman\_tree**, el árbol está en **qmin->T**. El elemento de la lista es borrado en la línea 60.
- Las probabilidades de los dos subárboles se van acumulando en la línea 58.
- Después de salir del lazo debe quedar en el bosque un solo árbol. **bosque.begin()** es un iterator al primer (y único) elemento del bosque. De manera que el árbol está en **bosque.begin()>T**. La probabilidad correspondiente debería ser 1. El **splice** de la línea 76 mueve todo este subárbol al valor de retorno, el árbol **T**.

### 3.8.6.7. Un programa de compresión de archivos

```

1 void
2 huffman_codes(btree<int> &T,btree<int>::iterator node,
3 const vector<double> &prob,
4 codigo_t &codigo, vector<codigo_t> &codigos) {
5 // 'codigo' es el codigo calculado hasta node.
6 // La funcion se va llamando recursivamente y a
7 // medida que va bajando en el arbol va
8 // agregando bits al codigo.
9 if (*node>=0) {
10 // Si es una hoja directamente inserta un
11 // codigo en 'codigos'
12 codigos[*node] = codigo;
13 return;
14 } else {
15 // Le va pasando 'codigo' a los hijos los
16 // cuales van agregando codigos a 'codigos'.
17 // 'codigo' se va pasando por referencia de
18 // manera que las llamadas recursivas lo deben
19 // dejar tal como estaba. Por eso, despues
20 // despues de agregar un 0 hay que sacarlo
21 // y lo mismo con el 1.
22 codigo.push_back(0);
23 huffman_codes(T,node.left(),prob,codigo,codigos);
24 codigo.pop_back();
25
26 codigo.push_back(1);
27 huffman_codes(T,node.right(),prob,codigo,codigos);

```

```
28 codigo.pop_back();
29 return;
30 }
31 }
32
33 void
34 huffman_codes(btree<int> &H,const vector<double> &prob,
35 vector<codigo_t> &codigos) {
36 // Este es el código de un carácter en particular. Es
37 // pasado por referencia, de manera que hay una sola instancia
38 // de código.
39 codigo_t código;
40 huffman_codes(H,H.begin(),prob,código,codigos);
41 }
42
43 const int NB = 8;
44 const int bufflen = 1024;
45
46 void qflush(queue<char> &Q, queue<char_t> &Qbytes,
47 int &nbits) {
48 // Convierte 'NB' bytes de 'Q' a un char.
49 // Si 'Q' queda vacía entonces rellena con 0's.
50 char_t c=0;
51 for (int j=0; j<NB; j++) {
52 int b = 0;
53 if (!Q.empty()) {
54 b = Q.front();
55 Q.pop();
56 nbits++;
57 }
58 c <= 1;
59 if (b) c |= 1;
60 else c &= ~1;
61 }
62 Qbytes.push(c);
63 }
64
65 void bflush(queue<char_t> &Qbytes,
66 vector<char_t> &buff,int &nbits,
67 FILE *zip) {
68 // Número de bits a ser escrito
69 int nb = nbits;
70 if (nb>bufflen*NB) nb = bufflen*NB;
71 nbits -= nb;
72 // Guarda en el archivo la longitud del siguiente bloque
73 fwrite(&nb,sizeof(int),1,zip);
74 // Pone en el buffer los 'nb' bits
75 int nbytes = 0;
76 while (nb>0) {
77 buff[nbytes++] = Qbytes.front();
78 Qbytes.pop();
79 nb -= NB;
80 }
```

```
81 fwrite(&buff[0], sizeof(char_t), nbytes, zip);
82 }
83
84 void hufzip(char *file, char *zipped) {
85 // Abre el archivo a compactar
86 FILE *fid;
87 if (file) {
88 fid = fopen(file, "r");
89 assert(fid);
90 } else fid = stdin;
91 // Numero total de caracteres posibles. Consideramos caracteres de 8
92 // bits, es decir que puede haber 256 caracteres
93 const int NUMCHAR=256;
94 // table[j] va a ser el numero de veces que aparece el caracter 'j'
95 // en el archivo. De manera que la probabilidad del caracter es
96 // prob[j] = table[j]/(sum_k=0^numchar table[k]) indx[j] es el
97 // indice asignado al caracter 'j'. Si el caracter 'j' no aparece en
98 // el archivo entonces hacemos indx[j]=-1 y si no le asignamos un
99 // numero correlativo de 0 a N-1. N es el numero total de
100 // caracteres distintos que aparecen en el archivo.
101 vector<int> table(NUMCHAR), indx(NUMCHAR);
102 // Ponemos los caracteres en una cola de 'char_t'
103 queue<char_t> fin;
104 // Contador de cuantos caracteres hay en el archivo
105 int n = 0;
106 while(1) {
107 int c = getc(fid);
108 if (c==EOF) break;
109 fin.push(c);
110 assert(c<NUMCHAR);
111 n++;
112 table[c]++;
113 }
114 fclose(fid);
115 // Detecta cuantos caracteres distintos hay fijandose en solo
116 // aquellos que tienen table[j]>0. Define prob[k] que es la
117 // probabilidad de aparecer del caracter con indice 'k'
118 int N=0;
119 // prob[k] es la probabilidad correspondiente al caracter de indice k
120 vector<double> prob;
121 // 'letters[k]' contiene el caracter (de 0 a NUMCHAR-1)
122 // correspondiente al indice 'k'
123 vector<char_t> letters;
124 for (int j=0; j<NUMCHAR; j++) {
125 if (table[j]) {
126 double p = double(table[j])/double(n);
127 indx[j] = N++;
128 letters.push_back((char_t)j);
129 prob.push_back(p);
130 } else indx[j] = -1;
131 }
132 // H va a contener al arbol de codigos
```

```
134 btree<int> H;
135 // Calcula el arbol usando el algoritmo de Huffman
136 huffman(prob,H);
137
138 // Construye la tabla de codigos. 'codigos[j]' va a ser
139 // un vector de enteros (bits)
140 vector<codigo_t> codigos(N);
141 // Calcula la tabla de codigos y la longitud media.
142 huffman_codes(H,prob,codigos);
143
144 // Abre el archivo zippeado
145 FILE *zip;
146 if (zipped) {
147 zip = fopen(zipped,"w");
148 assert(zip);
149 } else zip = stdout;
150
151 // Guarda encabezamiento en archivo zippeado conteniendo
152 // las probabilidades para despues poder reconstruir el arbol
153 for (int j=0; j<N; j++) {
154 fwrite(&prob[j],sizeof(double),1,zip);
155 fwrite(&letters[j],sizeof(char_t),1,zip);
156 }
157 // Terminador (probabilidad negativa)
158 double p = -1.0;
159 fwrite(&p,sizeof(double),1,zip);
160
161 vector<char_t> buff(bufflen);
162 // Cantidad de bits almacenados en buff
163 int nbits=0;
164
165 // Zippea. Va convirtiendo los caracteres de 'fin' en codigos y los
166 // inserta en la cola 'Q', o sea que 'Q' contiene todos elementos 0
167 // o 1. Por otra parte va sacan dode a 8 bits de Q y los convierte
168 // en un byte en 'Qbytes'. O sea que 'Qbytes' contiene caracteres que pueden
169 // tomar cualquier valor entre 0 y NUMCHAR-1.
170 queue<char> Q;
171 queue<char_t> Qbytes;
172 assert(fid);
173 while(!fin.empty()) {
174 // Va tomando de a un elemento de 'fin' y pone todo el codigo
175 // correspondiente en 'Q'
176 int c = fin.front();
177 fin.pop();
178 assert(c<NUMCHAR);
179 int k = indx[c];
180 assert(k>=0 && k<N);
181 codigo_t &cod = codigos[k];
182 for (int j=0; j<cod.size(); j++) Q.push(cod[j]);
183 // Convierte bits de 'Q' a caracteres
184 while (Q.size()>NB) qflush(Q,Qbytes,nbits);
185 // Escribe en el archivo zippeado.
```

```
187 while (Qbytes.size()>bufflen) bflush(Qbytes,buff,nbits,zip);
188 }
189
190 // Convierte el resto que puede quedar en Q
191 while (Q.size()>0) qflush(Q,Qbytes,nbits);
192 // Escribe el resto de lo que esta en Qbytes en 'zip'
193 while (Qbytes.size()>0) bflush(Qbytes,buff,nbits,zip);
194 // Terminador final con longitud de bloque=0
195 int nb=0;
196 // Escribe un terminador (un bloque de longitud 0)
197 fwrite(&nb,sizeof(int),1,zip);
198 fclose(zip);
199 }
200
201 int pop_char(queue<char> &Q,btree<int> &H,
202 btree<int>::iterator &m,int &k) {
203 // 'm' es un nodo en el arbol. Normalmente deberia estar en la raiz
204 // pero en principio puede estar en cualquier nodo. Se supone que ya
205 // se convirtieron una seride de bits. Si en la ultima llamada se
206 // llego a sacar un caracter entonces 'm' termina en la raiz, listo
207 // para extraer otro caracter. Entonces 'pop_char' extrae tantos
208 // caracteres como para llegar a una hoja y, por lo tanto, extraer
209 // un caracter. En ese caso pasa en 'k' el indice correspondiente,
210 // vuelve a 'm' a la raiz (listo para extraer otro caracter) y
211 // retorna 1. Si no, retorna 0 y deja a 'm' en el nodo al que llega.
212 while (!Q.empty()) {
213 int f = Q.front();
214 Q.pop();
215 // El valor binario 0 o 1 almacenado en 'Q' dice que hijo hay que tomar.
216 if (f) m = m.right();
217 else m = m.left();
218 // Verificar si llego a una hoja.
219 if (m.left()==H.end()) {
220 // Pudo sacar un caracter completo
221 k = *m;
222 assert(k != -1);
223 m = H.begin();
224 return 1;
225 }
226 }
227 // No pudo sacar un caracter completo.
228 return 0;
229 }
230
231 void hufunzip(char *zipped,char *unzipped) {
232 // Deszippe el archivo de nombre 'zipped' en 'unzipped'
233 // El vector de probabilidades (esta guardado en 'zipped').
234 vector<double> prob;
235 // Los caracteres correspondientes a cada indice
236 vector<char> letters;
237 // Numero de bits por caracter
238 const int NB=8;
239
240 // Abre el archivo 'zipped', si no es 'stdin'
```

```
241 FILE *zip;
242 if (zipped) {
243 zip = fopen(zipped, "r");
244 assert(zip);
245 } else zip = stdin;
246
247 // Lee la tabla de probabilidades y codigos, estan escritos
248 // en formato binario probabilidad,caracter,probabilidad,caracter, . .
249 // hasta terminar con una probabilidad <0
250 // Los va poniendo en 'prob[]' y las letras en 'letters[]'
251 int N=0;
252 int nread;
253 while (true) {
254 double p;
255 char c;
256 nread = fread(&p,sizeof(double),1,zip);
257 assert(nread==1);
258 if (p<0.0) break;
259 N++;
260 prob.push_back(p);
261 nread = fread(&c,sizeof(char),1,zip);
262 assert(nread==1);
263 letters.push_back(c);
264 }
265
266 // 'H' va a tener el arbol de codigos.
267 // 'huffman()' calcula el arbol.
268 btree<int> H;
269 huffman(prob,H);
270
271 // Los codigos se almacenan en un vector de
272 // codigos.
273 vector<codigo_t> codigos(N);
274 // 'huffman_codes()' calcula los codigos y tambien
275 // la longitud promedio del codigo.
276 huffman_codes(H,prob,codigos);
277
278 // El archivo donde descompacta. Si no se pasa
279 // el nombre entonces descompacta sobre 'stdout'.
280 FILE *unz;
281 if (unzipped) {
282 unz = fopen(unzipped, "w");
283 assert(unz);
284 } else unz = stdout;
285
286 // Los bloques de bytes del archivo compactado
287 // se van leyendo sobre una cola 'Q' y se va
288 // descompactando directamente sobre el archivo
289 // descompactado con 'putc' (el cual ya es
290 // buffereado)
291 queue<char> Q;
292 int read=0;
293 // Posicion en el arbol
```

```
294 btree<int>::iterator m = H.begin();
295 // indice de caracter que se extrajo
296 int k;
297 // Buffer para poner los bytes que se leen del
298 // archivo compactado.
299 vector<char_t> buff;
300 char_t c;
301 while (1) {
302 int nb;
303 // Lee longitud (en bits) del siguiente bloque.
304 nread = fread(&nb,sizeof(int),1,zip);
305 assert(nread==1);
306
307 // Detenerse si el siguiente bloque es nulo.
308 if (!nb) break;
309
310 // Redimensionar la longitud del
311 // buffer apropiadamente.
312 int nbytes = nb/NB + (nb % NB ? 1 : 0);
313 if (buff.size()<nbytes) buff.resize(nb);
314
315 // Lee el bloque
316 nread = fread(&buff[0],sizeof(char_t),nbytes,zip);
317 assert(nread==nbytes);
318
319 vector<char_t> v(NB);
320 int j = 0, read=0;
321 while (read<nb) {
322 c = buff[j++];
323 // Desempaquea el caracter tn bits
324 for (int l=0; l<NB; l++) {
325 int b = (c & 1 ? 1 : 0);
326 c >>= 1;
327 v[NB-l-1] = b;
328 }
329 for (int l=0; l<NB; l++) {
330 if (read++ < nb) Q.push(v[l]);
331 }
332 // Va convirtiendo bits de 'Q' en
333 // caracteres. Si 'pop_char()' no puede
334 // sacar un caracter, entonces va a devolver
335 // 0 y se termina el lazo. En ese caso 'm'
336 // queda en la posicion correspondiente en el
337 // arbol.
338 while(pop_char(Q,H,m,k)) putc(letters[k],unz);
339 }
340 }
341
342 assert(!.empty());
343 // Cerrar los archivos abiertos.
344 fclose(zip);
345 fclose(unz);
346 }
```

**Código 3.24:** Programa que comprime archivos basado en el algoritmo de Huffman. [Archivo: hufzipc.cpp]

Ahora vamos a presentar un programa que comprime archivos utilizando el algoritmo de Huffman.

- Primero vamos a describir una serie de funciones auxiliares. **huffman\_codes(T,prob,codigos)** calcula los codigos correspondientes a cada uno de los caracteres que aparecen en el archivo. La función es por supuesto recursiva y utiliza una función auxiliar **huffman\_codes(T,node,prob,level,codigo,codigos)**, donde se va pasando el “estado” del cálculo por las variables **node** que es la posición actual en el árbol y **codigo** que es un vector de enteros con 0’s y 1’s que representa el código hasta este nodo. La función agrega un código a la tabla si llega a una hoja (línea 12). Si no, agrega un 0 al código y llama recursivamente a la función sobre el hijo izquierdo y después lo mismo pero sobre el hijo derecho, pero agregando un 1. El código es siempre el mismo ya que es pasado por referencia. Por eso después de agregar un 0 o 1 y llamar recursivamente a la función hay que eliminar el bit, de manera que **codigo** debe quedar al salir en la línea 29 igual que al entrar antes de la línea 22. La función “wrapper” llama a la auxiliar pasándole como nodo la raíz y un código de longitud nula.
- Las funciones **hufzip(file,zipped)** y **hufunzip(zipped,unzipped)** comprimen y descomprimen archivos, respectivamente. Los argumentos son cadenas de caracteres que corresponden a los nombres de los archivos. El primer argumento es el archivo de entrada y el segundo el de salida. Si un string es nulo entonces se asume **stdin** o **stdout**.
- **hufzip()** va leyendo caracteres del archivo de entrada y cuenta cuantos instancias de cada carácter hay en el archivo. Dividiendo por el número total de caracteres obtenemos las probabilidades de ocurrencia de cada carácter. Consideramos caracteres de 8 bits de longitud, de manera que podemos comprimir cualquier tipo de archivos, formateados o no formateados.
- El árbol de Huffman se construye sólo para los **N** caracteres que existen actualmente en el archivo, es decir, aquellos que tienen probabilidad no nula. A cada carácter que existe en el archivo se le asigna un índice **indx** que va entre 0 y **N-1**. Se construyen dos tablas **indx[j]** que da el índice correspondiente al carácter **j** y **letters[k]** que da el carácter correspondiente al índice **k**. Otra posibilidad sería construir el árbol de Huffman para todos los caracteres incluyendo aquellos que tienen probabilidad nula.
- Como de todas formas estos caracteres tendrán posiciones en el árbol por debajo de aquellos caracteres que tienen probabilidad no nula, no afectará a la longitud promedio final.
- Notar que para poder comprimir hay que primero construir el árbol y para esto hay que recorrer el archivo para calcular las probabilidades. Esto obliga a leer los caracteres y mantenerlos en una cola **fin**, con lo cual todo el archivo a comprimir está en memoria principal. Otra posibilidad es hacer dos pasadas por el archivo, una primera pasada para calcular las probabilidades y una segunda para comprimir. Todavía otra posibilidad es comprimir con una tabla de probabilidades construida previamente, en base a estadística sobre archivos de texto o del tipo que se está comprimiendo. Sin embargo, en este caso las tasas de compresión van a ser menores.
- **hufzip()** va convirtiendo los caracteres que va leyendo a códigos de 0’s y 1’s que son almacenados temporalmente en una cola de bits **Q**. Mientras tanto se van tomando de a **NB=8** bits y se construye con operaciones de bits el carácter correspondiente el cual se va guardando en una cola de bytes **Qbytes**. A su vez, de **Qbytes** se van extrayendo bloques de caracteres de longitud **bufflen** para aumentar la eficiencia de la escritura a disco.

- Para descomprimir el archivo es necesario contar con el árbol que produjo la compresión. La solución utilizada aquí es guardar el vector de probabilidades utilizado **prob** y los caracteres correspondientes, ya que el árbol puede ser calculado únicamente usando la función **huffman()** a partir de las probabilidades. Para guardar el árbol se van escribiendo en el archivo la probabilidad y el carácter de a uno (líneas 154–157). Para indicar el fin de la tabla se escribe una probabilidad negativa (-1.0). La lectura de la tabla se hace en **hufunzip()** se hace en las líneas 253–264. Ambas funciones (**hufzip** y **hufunzip**) calculan el árbol usando **huffman()** en las líneas 136 y 269, respectivamente.
- El lazo que comprime son las líneas 174–188. Simplemente va tomando caracteres de **fin** y los convierte a bits, usando el código correspondiente, en **Q**. Simultáneamente va pasando tantas **NB**-tuplas de bits de **Q** a **Qbytes** con la función **qflush()** y tantos bytes de **Qbytes** al archivo zippeado con **bflush()**. La rutina **bflush()** va imprimiendo en el archivo de a **bufflen** bytes.
- En el archivo comprimido se van almacenando los bytes de a bloques de longitud **bufflen** o menor. Los bloques se almacenan grabando primero la longitud del bloque (un entero) (línea 73) y después el bloque de bytes correspondiente (línea 81).
- Después del lazo pueden quedar bits en **Q** y bytes en **Qbytes**. Los lazos de las líneas 191 y 193 terminan de procesar estos restos.
- El archivo se descomprime en las línea 301 de **hufunzip()**. Se leen bloques de bytes en la línea 316 y se van convirtiendo a bits en línea 330.
- La función **pop\_char(Q,H,m,k)** va sacando bits de **Q** y moviendo el nodo **m** en el árbol hasta que **m** llega a una hoja (en cuyo caso **pop\_char()** retorna un carácter por el argumento **k**, o mejor dicho el índice correspondiente) o hasta que **Q** queda vacía. En este caso **m** es vuelto a la raíz del árbol. Por ejemplo, refiriéndonos al código *C2* de la figura 3.35 si inicialmente **m** está en el nodo 1 y **Q={0110010110}**, entonces el primer bit 0 de la cola mueve **m** al nodo 10 y el segundo a 101. A esa altura **pop\_char()** devuelve el carácter **c** ya que ha llegado a una hoja y **m** vuelve a la raíz. La cola de bits queda en **Q={10010110}**. Si volvemos a llamar a **pop\_char()** repetidamente va a ir devolviendo los caracteres **b** (100) y **c** (101). A esa altura queda **Q={10}**. Si volvemos a llamar a **pop\_char()**, **m** va a descender hasta el nodo 10 y **Q** va a quedar vacía. En ese caso **pop\_char()** no retorna un carácter. La forma de retornar un carácter es la siguiente: Si **pop\_char()** pudo llegar a un carácter, entonces retorna 1 (éxito) y el índice del carácter se pasa a través del argumento **k**, si no retorna 0 (fallo).
- El lazo de la línea 338 extrae tantos caracteres como puede de **Q** y los escribe en el archivo **unzipped** con el macro **putc()** (de la librería estándar de C).
- Al salir del lazo de las líneas 301–340 no pueden quedar bits sin convertir en **Q**, ya que todos los caracteres del archivo comprimido han sido leídos y estos representan un cierto número de caracteres. Si después de salir del lazo quedan bits, esto quiere decir que hubo algún error en la lectura o escritura. El **assert()** de la línea 342 verifica esto.
- El programa así presentado difícilmente no puede recuperarse de un error en la lectura o escritura, salvo la detección de un error al fin del archivo, lo cual fue descripto en el párrafo anterior. Supongamos por ejemplo que encodamos el mensaje *accdededcedcd* con el código *C2*. Esto resulta en el string de bits **01011011101111011110111101111**. Ahora supongamos que al querer desencodar el mensaje se produce un error y el tercer bit pasa de 0 a 1, quedando el string **011110111101111011110111101111**. El proceso de decompresión resulta en el mensaje *addaddaddaddaddad*, quedando un remanente de un bit 1 en la cola. Detectamos que hay un error al ver que quedó un bit en la cola sin poder llegar a formar el carácter, pero lo peor es que todo el mensaje desencodado a partir del error es erróneo. Una posibilidad es encodar el código por bloques, introduciendo caracteres de control. La tasa de

compresión puede ser un poco menor, pero si un error se produce en alguna parte del archivo los bloques restantes se podrán descomprimir normalmente.

El programa de compresión aquí puede comprimir y descomprimir archivos de texto y no formateados. Para un archivo de texto típico la tasa de compresión es del orden del 35 %. Esto es poco comparado con los compresores usuales como **gzip**, **zip** o **bzip2** que presentan tasas de compresión en el orden del 85 %. Además, esos algoritmos comprimen los archivos “*al vuelo*” (“*on the fly*”) es decir que no es necesario tener todo el archivo a comprimir en memoria o realizar dos pasadas sobre el mismo.

# Capítulo 4

## Conjuntos

En este capítulo se introduce en mayor detalle el TAD “conjunto”, ya introducido en la sección §1.2 y los subtipos relacionados “diccionario” y “cola de prioridad”.

### 4.1. Introducción a los conjuntos

Un conjunto es una colección de “miembros” o “elementos” de un “conjunto universal”. Por contraposición con las listas y otros contenedores vistos previamente, todos los miembros de un conjunto deben ser diferentes, es decir no puede haber dos copias del mismo elemento. Si bien para definir el concepto de conjunto sólo es necesario el concepto de igualdad o desigualdad entre los elementos del conjunto universal, en general las representaciones de conjuntos asumen que entre ellos existe además una “relación de orden estricta”, que usualmente se denota como  $<$ . A veces un tal orden no existe en forma natural y es necesario saber definirlo, aunque sea sólo para implementar el tipo conjunto (ver sección §2.4.4).

#### 4.1.1. Notación de conjuntos

Normalmente escribimos un conjunto enumerando sus elementos entre llaves, por ejemplo  $\{1, 4\}$ . Debemos recordar que no es lo mismo que una lista, ya que, a pesar de que los enumeramos en forma lineal, *no existe un orden preestablecido entre los miembros de un conjunto*. A veces representamos conjuntos a través de una condición sobre los miembros del conjunto universal, por ejemplo

$$A = \{x \text{ entero} / x \text{ es par}\} \quad (4.1)$$

De esta forma se pueden definir conjuntos con un número infinito de miembros.

La principal relación en los conjuntos es la de “pertenencia”  $\in$ , esto es  $x \in A$  si  $x$  es un miembro de  $A$ . Existe un conjunto especial  $\emptyset$  llamado el “conjunto vacío”. Decimos que  $A$  está incluido en  $B$  ( $A \subseteq B$ , o  $B \supseteq A$ ) si todo miembro de  $A$  también es miembro de  $B$ . También decimos que  $A$  es un “subconjunto” de  $B$  y que  $B$  es un “supraconjunto” de  $A$ . Todo conjunto está incluido en sí mismo y el conjunto vacío está incluido en cualquier conjunto.  $A$  y  $B$  son “iguales” si  $A \subseteq B$  y  $B \subseteq A$ , por lo tanto dos conjuntos son distintos si al menos existe un elemento de  $A$  que no pertenece a  $B$  o viceversa. El conjunto  $A$  es un “subconjunto propio” (“supraconjunto propio”) de  $B$  si  $A \subseteq B$  ( $A \supsetneq B$ ) y  $A \neq B$ .

Las operaciones más básicas de los conjuntos son la “unión”, “intersección” y “diferencia”. La unión  $A \cup B$  es el conjunto de los elementos que pertenecen a  $A$  o a  $B$  mientras que la intersección  $A \cap B$  es el de los elementos que pertenecen a  $A$  y a  $B$ . Dos conjuntos son “disjuntos” si  $A \cap B = \emptyset$ . La diferencia  $A - B$  está formada por los elementos de  $A$  que no están en  $B$ . Es fácil demostrar la siguiente igualdad de conjuntos

$$A \cup B = (A \cap B) \cup (A - B) \cup (B - A) \quad (4.2)$$

siendo los tres conjuntos del miembro derecho disjuntos.

#### 4.1.2. Interfaz básica para conjuntos

```

1 typedef int elem_t;
2
3 class iterator_t {
4 private:
5 /* . . . */;
6 public:
7 bool operator!=(iterator_t q);
8 bool operator==(iterator_t q);
9 };
10
11 class set {
12 private:
13 /* . . . */;
14 public:
15 set();
16 set(const set &);
17 ~set();
18 elem_t retrieve(iterator_t p);
19 pair<iterator_t, bool> insert(elem_t t);
20 void erase(iterator_t p);
21 int erase(elem_t x);
22 void clear();
23 iterator_t next(iterator_t p);
24 iterator_t find(elem_t x);
25 iterator_t begin();
26 iterator_t end();
27 };
28 void set_union(set &A, set &B, set &C);
29 void set_intersection(set &A, set &B, set &C);
30 void set_difference(set &A, set &B, set &C);
```

**Código 4.1:** Interfaz básica para conjuntos [Archivo: setbash.h]

En el código 4.1 vemos una interfaz básica para conjuntos

- Como en los otros contenedores STL vistos, una clase **iterator** permite recorrer el contenedor. Los iterators soportan los operadores de comparación **==** y **!=**.

- Sin embargo, en el conjunto no se puede insertar un elemento en una posición determinada, por lo tanto la función **insert** no tiene un argumento posición como en listas o árboles. Sin embargo **insert** retorna un par, conteniendo un iterator al elemento insertado y un **bool** que indica si el elemento es un nuevo elemento o ya estaba en el conjunto.
- La función **erase(p)** elimina el elemento que está en la posición **p**. La posición **p** debe ser válida, es decir debe haber sido obtenida de un **insert(x)** o **find(x)**. **erase(p)** invalida **p** y todas las otras posiciones obtenidas previamente.
- **count=erase(x)** elimina el elemento **x** si estaba en el conjunto. Si no, el conjunto queda inalterado. Retorna el número de elementos *efectivamente eliminados* del conjunto. Es decir, si el elemento estaba previamente en el conjunto entonces retorna 1, de otra forma retorna 0. (Nota: Existe otro contenedor relacionado llamado “*multiset*” en el cual pueden existir varias copias de un mismo elemento. En *multiset* el valor de retorno puede ser mayor que 1).
- Como es usual **begin()**, **end()** y **next()** permiten iterar sobre el conjunto.
- Las operaciones binarias sobre conjuntos se realizan con las funciones **set\_union(A,B,C)**, **set\_intersection(A,B,C)** y **set\_difference(A,B,C)** que corresponden a las operaciones  $C = A \cup B$ ,  $C = A \cap B$  y  $C = A - B$ , respectivamente. Notar que estas *no son miembros de la clase*. Todas estas funciones binarias asumen que los conjuntos **A**, **B** y **C** son distintos.
- Una restricción muy importante en todas las funciones binarias es que ninguno de los conjuntos de entrada (ni **A** ni **B**) deben *superponerse* (overlap) con **C**. Esto se aplica también a la versión de las STL, en cuyo caso los *rangos* de entrada no se deben superponer con el de salida.
- **p=find(x)** devuelve un iterator a la posición ocupada por el elemento **x** en el conjunto. Si el conjunto no contiene a **x** entonces devuelve **end()**.

#### 4.1.3. Análisis de flujo de datos

Consideremos un programa simple como el mostrado en la figura 4.1 que calcula el máximo común divisor  $\gcd(p, q)$  de dos números enteros  $p, q$ , mediante el algoritmo de Euclides. Recordemos que el algoritmo de Euclides se basa en la relación recursiva

$$\gcd(p, q) = \begin{cases} q \text{ divide a } p : & q; \\ \text{si no:} & \gcd(q, \text{rem}(p, q)) \end{cases} \quad (4.3)$$

donde asumimos que  $p > q$  y  $\text{rem}(p, q)$  es el resto de dividir  $p$  por  $q$ . Por ejemplo, si  $p = 30$  y  $q = 12$  entonces

$$\gcd(30, 12) = \gcd(12, 6) = 6 \quad (4.4)$$

ya que  $\text{rem}(30, 12) = 6$  y 6 divide a 12.

Los bloques  $B_4$ ,  $B_5$  y  $B_7$  son la base recursiva del algoritmo. La lectura de los datos se produce en el bloque  $B_1$  y el condicional del bloque  $B_2$  se encarga de intercambiar  $p$  y  $q$  en el caso de que  $q > p$ .

Los bloques representan porciones de código en los cuales el código sucede secuencialmente línea a línea. Los condicionales en los bloques  $B_2$  y  $B_5$  y el lazo que vuelve del bloque  $B_7$  al  $B_4$  rompen la secuencialidad de este código.

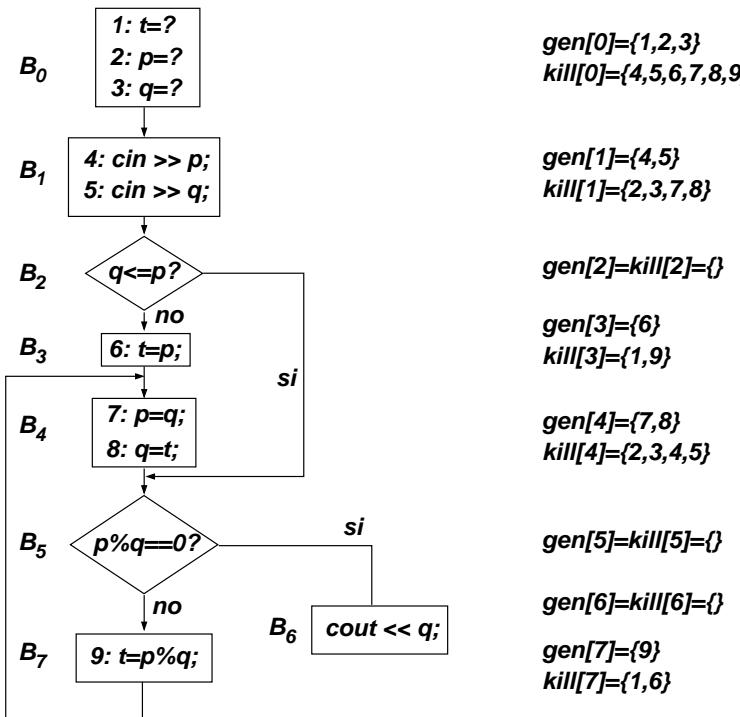


Figura 4.1:

En el diseño de compiladores es de importancia saber cuál es la última línea donde una variable puede haber tomado un valor al llegar a otra determinada línea. Por ejemplo, al llegar a la línea 7, **t** puede haber tomado su valor de una asignación en la línea 6 o en la línea 9. Este tipo de análisis es de utilidad para la optimización del código. Por ejemplo si al llegar a un cierto bloque sabemos que una variable **x** sólo puede haber tomado su valor de una asignación constante como **x=20;**, entonces en esa línea se puede reemplazar el valor de **x** por el valor 20. También puede servir para la detección de errores. Hemos introducido un bloque ficticio **B**<sub>0</sub> que asigna valores indefinidos (representados por el símbolo "?"). Si alguna de estas asignaciones están activas al llegar a una línea donde la variable es usada, entonces puede ser que se esté usando una variable indefinida. Este tipo de análisis es estándar en la mayoría de los compiladores.

Para cada bloque **B**<sub>j</sub> vamos a tener definidos 4 conjuntos a saber

- **gen[j]**: las asignaciones que son generadas en el bloque **B**<sub>j</sub>. Por ejemplo en el bloque **B**<sub>1</sub> se generan las asignaciones 4 y 5 para las variables **p** y **q**.
- **kill[j]**: las asignaciones que son eliminadas en el bloque. Por ejemplo, al asignar valores a las variables **p** y **q** en el bloque **B**<sub>1</sub> cualquier asignación a esas variables que llegue al bloque será eliminada, como por ejemplo, las asignaciones 2 y 3 del bloque ficticio **B**<sub>0</sub>. En este caso podemos detectar fácilmente cuáles son las asignaciones eliminadas pero en general esto puede ser más complejo, de manera que, conservativamente, introducimos en **kill[j]** todas las asignaciones a variables cuyo valor es reasignado en **B**<sub>j</sub>. En el caso del bloque **B**<sub>1</sub> las variables reasignadas son **p** y **q**, las cuales sólo tienen asignaciones en las líneas 2,3,7 y 8 (sin contar las propias asignaciones en el bloque). De manera que

**kill[1]={2, 3, 7, 8}**. Notar que, por construcción **gen[j]** y **kill[j]** son conjuntos disjuntos.

- **defin[j]** el conjunto total de definiciones que llegan al bloque  $B_j$ .
- **defout[j]** el conjunto total de definiciones que salen del bloque  $B_j$  ya sea porque son generadas en el bloque, o porque pasan a través del bloque sin sufrir reasignación.

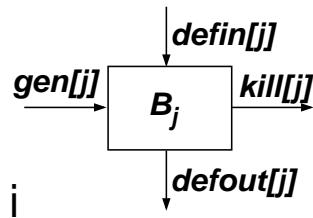


Figura 4.2: Ecuación de balance para las asignaciones que llegan y salen de un bloque.

En este análisis los conjuntos **gen[j]** y **kill[j]** pueden por simple observación del código, mientras que los **defin[j]** y **defout[j]** son el resultado buscado. Para obtenerlos debemos escribir una “*ecuación de balance de asignaciones*” en el bloque, a saber (ver figura 4.2)

$$\text{defout}[j] = (\text{defin}[j] \cup \text{gen}[j]) - \text{kill}[j] \quad (4.5)$$

la cual expresa que *las asignaciones que salen del bloque son aquellas que llegan, más las generadas en el bloque menos las que son eliminadas en el mismo*.

Ahora consideremos las asignaciones que llegan al  $B_4$ , es decir **defin[4]**. Estas pueden proceder o bien del bloque  $B_3$  o bien del  $B_7$ , es decir

$$\text{defin}[4] = \text{defout}[3] \cup \text{defout}[7] \quad (4.6)$$

En general tenemos que

$$\text{defin}[j] = \sum_{m \in \text{ent}[j]} \text{defout}[m] \quad (4.7)$$

donde **ent[j]** es el conjunto de bloques cuyas salidas confluyen a la entrada de  $B_j$ . En este caso tenemos

$$\begin{aligned}
 \text{ent}[0] &= \emptyset \\
 \text{ent}[1] &= \{0\} \\
 \text{ent}[2] &= \{1\} \\
 \text{ent}[3] &= \{2\} \\
 \text{ent}[4] &= \{3, 7\} \\
 \text{ent}[5] &= \{2, 4\} \\
 \text{ent}[6] &= \{5\} \\
 \text{ent}[7] &= \{5\}
 \end{aligned} \quad (4.8)$$

```

1 void dataflow(vector<set> &gen,
2 vector<set> &kill,
3 vector<set> &defin,
4 vector<set> &defout,
5 vector<set> &ent) {
6 int nblock = gen.size();
7 bool cambio=true;
8 while (cambio) {
9 cambio=false;
10 for (int j=0; j<nblock; j++) {
11 // Calcular la entrada al bloque 'defin[j]'
12 // sumando sobre los 'defout[m]' que
13 // confluyen al bloque j ...
14 }
15 int out_prev = defout[j].size();
16
17 cambio=false;
18 for (int j=0; j<nblock; j++) {
19 // Calcular el nuevo valor de 'defout[j]'
20 // usando la ec. de balance de asignaciones...
21 if (defout[j].size() != out_prev) cambio=true;
22 }
23 }
24 }
```

**Código 4.2:** Seudocódigo para el análisis de flujo de datos. [Archivo: dataflow1.cpp]

Un seudocódigo para este algoritmo puede observarse en el código 4.2. La función `dataflow()` toma como argumentos vectores de conjuntos de longitud `nblock` (el número de bloques, en este caso 8). `ent[]`, `gen[]` y `kill[]` son datos de entrada, mientras que `defin[]` y `defout[]` son datos de salida calculados por `dataflow()`. `tmp` es una variable auxiliar de tipo conjunto. El código entra en un lazo infinito en el cual va calculando para cada bloque las asignaciones a la entrada `defin[j]` aplicando (4.7) y luego calculando `defout[j]` mediante (4.5).

El proceso es iterativo, de manera que hay que inicializar las variables `defin[]` y `defout[]` y detectar cuando no hay mas cambios. Notemos primero que lo único que importa son las inicializaciones para `defin[]` ya que cualquier valor que tome `defout[]` al ingresar a `dataflow[]` será sobreescrito durante la primera iteración. Tomemos como inicialización  $\text{defin}[j]^0 = \emptyset$ . Después de la primera iteración los `defout[j]` tomarán ciertos valores  $\text{defout}[j]^0$ , posiblemente no nulos, de manera que en la iteración 1 los `defin[j]`<sup>1</sup> pueden eventualmente ser no nulos, pero vale que

$$\text{defin}[j]^0 \subseteq \text{defin}[j]^1 \quad (4.9)$$

Es fácil ver entonces que, después de aplicar (4.5) valdrá que

$$\text{defout}[j]^0 \subseteq \text{defout}[j]^1 \quad (4.10)$$

Siguiendo el razonamiento, puede verse que siempre seguirá valiendo que

$$\begin{aligned} \text{defin}[j]^k &\subseteq \text{defin}[j]^{k+1} \\ \text{defout}[j]^k &\subseteq \text{defout}[j]^{k+1} \end{aligned} \quad (4.11)$$

Nos preguntamos ahora cuantas veces hay que ejecutar el algoritmo. Notemos que, como tanto **defin[j]** como **defout[j]** deben ser subconjuntos del conjunto finito que representan todas las asignaciones en el programa, a partir de una cierta iteración los **defin[j]** y **defout[j]** no deben cambiar más. Decimos entonces que el algoritmo “*convergió*” y podemos detenerlo.

Notar que (4.11) garantiza que, para detectar la convergencia basta con verificar que el tamaño de ningún **defout[j]** cambie.

| <b>j</b>  | <b>iter=0</b> | <b>iter=1</b> | <b>iter=2</b> | <b>iter=3</b>   | <b>iter=4</b>   | <b>iter=5 y 6</b> |
|-----------|---------------|---------------|---------------|-----------------|-----------------|-------------------|
| defin[0]  | {}            | {}            | {}            | {}              | {}              | {}                |
| defout[0] | {1,2,3}       | {1,2,3}       | {1,2,3}       | {1,2,3}         | {1,2,3}         | {1,2,3}           |
| defin[1]  | {}            | {1,2,3}       | {1,2,3}       | {1,2,3}         | {1,2,3}         | {1,2,3}           |
| defout[1] | {4,5}         | {1,4,5}       | {1,4,5}       | {1,4,5}         | {1,4,5}         | {1,4,5}           |
| defin[2]  | {}            | {4,5}         | {1,4,5}       | {1,4,5}         | {1,4,5}         | {1,4,5}           |
| defout[2] | {}            | {4,5}         | {1,4,5}       | {1,4,5}         | {1,4,5}         | {1,4,5}           |
| defin[3]  | {}            | {}            | {4,5}         | {1,4,5}         | {1,4,5}         | {1,4,5}           |
| defout[3] | {6}           | {6}           | {4,5,6}       | {4,5,6}         | {4,5,6}         | {4,5,6}           |
| defin[4]  | {}            | {6,9}         | {6,9}         | {4,5,6,7,8,9}   | {4,5,6,7,8,9}   | {4,5,6,7,8,9}     |
| defout[4] | {7,8}         | {6,7,8,9}     | {6,7,8,9}     | {6,7,8,9}       | {6,7,8,9}       | {6,7,8,9}         |
| defin[5]  | {}            | {7,8}         | {4,5,6,7,8,9} | {1,4,5,6,7,8,9} | {1,4,5,6,7,8,9} | {1,4,5,6,7,8,9}   |
| defout[5] | {}            | {7,8}         | {4,5,6,7,8,9} | {1,4,5,6,7,8,9} | {1,4,5,6,7,8,9} | {1,4,5,6,7,8,9}   |
| defin[6]  | {}            | {}            | {7,8}         | {4,5,6,7,8,9}   | {1,4,5,6,7,8,9} | {1,4,5,6,7,8,9}   |
| defout[6] | {}            | {}            | {7,8}         | {4,5,6,7,8,9}   | {1,4,5,6,7,8,9} | {1,4,5,6,7,8,9}   |
| defin[7]  | {}            | {}            | {7,8}         | {4,5,6,7,8,9}   | {1,4,5,6,7,8,9} | {1,4,5,6,7,8,9}   |
| defout[7] | {9}           | {9}           | {7,8,9}       | {4,5,7,8,9}     | {4,5,7,8,9}     | {4,5,7,8,9}       |

Tabla 4.1: Iteraciones hasta llegar a convergencia para el algoritmo de análisis de flujo de datos.

La Tabla 4.1 muestra el avance de las iteraciones hasta llegar a convergencia. La iteración 6

```

1 void dataflow(vector<set> &gen,
2 vector<set> &kill,
3 vector<set> &defin,
4 vector<set> &defout,
5 vector<set> &ent) {
6 int nblock = gen.size();
7 set tmp;
8 bool cambio=true;
9 while (cambio) {
10 for (int j=0; j<nblock; j++) {
11 defin[j].clear();
12 iterator_t p = ent[j].begin();
13 while (p!=ent[j].end()) {
14 int k = ent[j].retrieve(p);
15 set_union(defin[j],defout[k],tmp);
16 defin[j] = tmp;
17 p = ent[j].next(p);
18 }
}

```

```

19 }
20 cambio=false;
21 for (int j=0; j<nblock; j++) {
22 int out_prev = defout[j].size();
23 set_union(defin[j],gen[j],tmp);
24 set_difference(tmp,kill[j],defout[j]);
25 if (defout[j].size()!=out_prev) cambio=true;
26 }
27 }
28 }
```

**Código 4.3:** Rutina para el análisis de flujo de datos. [Archivo: dtflow.cpp]

El código 4.3 muestra el código definitivo usando la interfaz básica para conjuntos (ver código 4.1). En las líneas 11–18 se calcula el `defin[j]` a partir de los `defout[k]` que confluyen a la entrada. El iterator `j` recorre los elementos del conjunto `ent[j]` que son los bloques cuya salida llega a la entrada del bloque `j`. Nótese que se debe usar un conjunto auxiliar `tmp` ya que el debido a que los argumentos de entrada a `set_union` deben ser diferentes entre sí. En las líneas 21–26 se calculan los conjuntos `defout[j]` a partir de las entradas `defin[j]` y los `gen[j]` y `kill[j]` usando la relación (4.5). Notar también el uso del conjunto auxiliar `tmp` para evitar la superposición de argumentos. Antes de actualizar `defout[j]` guardamos el número de elementos en una variable `out_prev` para poder después verificar si el conjunto creció, y así determinar si es necesario seguir iterando o no.

## 4.2. Implementación por vectores de bits

Tal vez la forma más simple de representar un conjunto es guardando un campo de tipo `bool` por cada elemento del conjunto universal. Si este campo es verdadero entonces el elemento está en el conjunto y viceversa. Por ejemplo, si el conjunto universal son los enteros de `0` a `N-1`

$$U = \{j \text{ entero, tal que } 0 \leq j < N\} \quad (4.12)$$

entonces podemos representar a los conjuntos por vectores de valores booleanos (puede ser `vector<bool>`) de longitud `N`. Si `v` es el vector, entonces `v[j]` indica si el entero `j` está o no en el conjunto. Por ejemplo, si el conjunto es `S={4,6,9}` y `N` es 10, entonces el vector de bits correspondiente sería `v={0,0,0,0, 1,0,1,0,0,1}`.

Para insertar o borrar elementos se prende o apaga el bit correspondiente. Todas estas son operaciones  $O(1)$ . Las operaciones binarias también son muy simples de implementar. Por ejemplo, si queremos hacer la unión  $C = A \cup B$ , entonces debemos hacer `C.v[j] = A.v[j] || B.v[j]`. La intersección se obtiene reemplazando `||` con `&&` y la diferencia  $C = A - B$  con `C.v[j] = A.v[j] && ! B.v[j]`.

Notar que el tiempo de ejecución de estas operaciones es  $O(N)$ , donde  $N$  es el número de elementos en el conjunto universal. La memoria requerida es `N` bits, es decir que también es  $O(N)$ . Es de destacar que también se podría usar `vector<T>` con cualquier tipo `T` convertible a un entero, por ejemplo `int`, `char`, `bool` y sus variantes. En cada caso la memoria requerida es `N*sizeof(T)`, de manera que siempre es  $O(N)$ . Esto representa 8 bits por elemento para `char` o 32 para `int`. En el caso de `bool`, el operador `sizeof(bool)`

reporta normalmente 1 byte por elemento, pero la representación interna de `vector<bool>` y en realidad requiere de un sólo bit por elemento.

#### 4.2.1. Conjuntos universales que no son rangos contiguos de enteros

Para representar conjuntos universales  $U$  que no son subconjuntos de los enteros, o que no son un subconjunto contiguo de los enteros  $[0, N)$  debemos definir funciones que establezcan la correspondencia entre los elementos del conjunto universal y el conjunto de los enteros entre  $[0, N)$ . Llamaremos a estas funciones

```

1 int indx(elem_t t);
2 elem_t element(int j);

1 const int N=50;
2 typedef int elem_t;
3 int indx(elem_t t) { return (t-100)/2; }
4 elem_t element(int j) { return 100+2*j; }
```

**Código 4.4:** Funciones auxiliares para definir conjuntos dentro de  $U = \{100, 102, \dots, 198\}$ . [Archivo: `element.cpp`]

```

1 const int N=52;
2 typedef char elem_t;
3 int indx(elem_t c) {
4 if (c>='a' && c<='z') return c-'a';
5 else if (c>='A' && c<='Z') return 26+c-'A';
6 else cout << "Elemento fuera de rango!!\n"; abort();
7 }
8 elem_t element(int j) {
9 assert(j<N);
10 return (j<26 ? 'a'+j : 'A'+j-26);
11 }
```

**Código 4.5:** Funciones auxiliares para definir conjuntos dentro de las letras  $a-z$  y  $A-Z$ . [Archivo: `setbasadefs.h`]

Por ejemplo, si queremos representar el conjunto de los enteros pares entre 100 y 198, entonces podemos definir estas funciones como en el código 4.4. Para el conjunto de las letras tanto minúsculas como mayúsculas (en total  $N=52$ ) podemos usar las funciones mostradas en el código código 4.5. Hacemos uso de que el código ASCII para las letras es correlativo. Para las minúsculas va desde 97 a 122 ( $a-z$ ) y para las mayúsculas va desde 65 hasta 90 ( $A-Z$ ). La función `indx()` correspondiente determina en qué rango (mayúsculas o minúsculas) está el carácter y restándole la base correspondiente lo convierte a un número entre 0 y 51. La función `element()`, en forma recíproca convierte un entero entre 0 y 51 a un carácter, fijándose primero si está en el rango 0-25, o 26-51.

#### 4.2.2. Descripción del código

```
1 typedef int iterator_t;
2
3 class set {
4 private:
5 vector<bool> v;
6 iterator_t next_aux(iterator_t p) {
7 while (p<N && !v[p]) p++;
8 return p;
9 }
10 typedef pair<iterator_t,bool> pair_t;
11 public:
12 set() : v(N,0) {}
13 set(const set &A) : v(A.v) {}
14 ~set() {}
15 iterator_t lower_bound(elem_t x) {
16 return next_aux(indx(x));
17 }
18 pair_t insert(elem_t x) {
19 iterator_t k = indx(x);
20 bool inserted = !v[k];
21 v[k] = true;
22 return pair_t(k,inserted);
23 }
24 elem_t retrieve(iterator_t p) { return element(p); }
25 void erase(iterator_t p) { v[p]=false; }
26 int erase(elem_t x) {
27 iterator_t p = indx(x);
28 int r = (v[p] ? 1 : 0);
29 v[p] = false;
30 return r;
31 }
32 void clear() { for(int j=0; j<N; j++) v[j]=false; }
33 iterator_t find(elem_t x) {
34 int k = indx(x);
35 return (v[k] ? k : N);
36 }
37 iterator_t begin() { return next_aux(0); }
38 iterator_t end() { return N; }
39 iterator_t next(iterator_t p) { next_aux(++p); }
40 int size() {
41 int count=0;
42 for (int j=0; j<N; j++) if (v[j]) count++;
43 return count;
44 }
45 friend void set_union(set &A,set &B,set &C);
46 friend void set_intersection(set &A,set &B,set &C);
47 friend void set_difference(set &A,set &B,set &C);
48 };
49
50 void set_union(set &A,set &B,set &C) {
```

```

51 for (int j=0; j<N; j++) C.v[j] = A.v[j] || B.v[j];
52 }
53 void set_intersection(set &A, set &B, set &C) {
54 for (int j=0; j<N; j++) C.v[j] = A.v[j] && B.v[j];
55 }
56 void set_difference(set &A, set &B, set &C) {
57 for (int j=0; j<N; j++) C.v[j] = A.v[j] && ! B.v[j];
58 }
```

**Código 4.6:** Implementación de conjuntos con vectores de bits. [Archivo: *setbasac.h*]

En el código 4.6 vemos una implementación posible con vectores de bits.

- El vector de bits está almacenado en un campo **vector<bool>**. El constructor por defecto dimensiona a **v** de tamaño **N** y con valores 0. El constructor por copia simplemente copia el campo **v**, con lo cual copia el conjunto.
- La clase iterator es simplemente un **typedef** de los enteros, con la particularidad de que el iterator corresponde al índice en el conjunto universal, es decir el valor que retorna la función **indx()**. Por ejemplo, en el caso de los enteros pares entre 100 y 198 tenemos 50 valores posibles del tipo iterator. El iterator correspondiente a 100 es 0, a 102 le corresponde 1, y así siguiendo hasta 198 que le corresponde 50.
- Se elige como iterator **end()** el índice **N**, ya que este no es nunca usado por un elemento del conjunto.
- Por otra parte, los iterators sólo deben avanzar sobre los valores definidos en el conjunto. Por ejemplo, si el conjunto es **S={120, 128, 180}** los iterators ocupados son 10, 14 y 40. Entonces **p=S.begin();** debe retornar 10 y aplicando sucesivamente **p=S.next(p);** debemos tener **p=14, 40 y 50** (que es **N=S.end()**).
- La función auxiliar **p=next\_aux(p)** (notar que esta declarada como privada) devuelve el primer índice siguiente a **p** ocupado por un elemento. Por ejemplo en el caso del ejemplo, **next\_aux(0)** debe retornar 10, ya que el primer índice ocupado en el conjunto siguiente a 0 es el 10.
- La función **next()** (que en la interfaz avanzada será sobrecargada sobre el operador **operator++**) incrementa **p** y luego le aplica **next\_aux()** para avanzar hasta el siguiente elemento del conjunto.
- La función **retrieve(p)** simplemente devuelve el elemento usando la función **element(p)**.
- **erase(x)** e **insert(x)** simplemente prenden o apagan la posición correspondiente **v[indx(x)]**. Notar que son  $O(1)$ .
- **find(x)** simplemente verifica si el elemento está en el conjunto, en ese caso retorna **indx(x)**, si no retorna **N** (que es **end()**).
- **size()** cuenta las posiciones en **v** que están prendidas. Por lo tanto es  $O(N)$ .
- Las funciones binarias **set\_union(A,B,C)**, **set\_intersection(A,B,C)** y **set\_difference(A,B,C)** hacen un lazo sobre todas las posiciones del vector y por lo tanto son  $O(N)$ .

| TAD conjunto                  | TAD Correspondencia                 |
|-------------------------------|-------------------------------------|
| <code>x = retrieve(p);</code> | <code>x = retrieve(p).first;</code> |
| <code>p=insert(x)</code>      | <code>p=insert(x,w)</code>          |
| <code>erase(p)</code>         | <code>erase(p)</code>               |
| <code>erase(x)</code>         | <code>erase(x)</code>               |
| <code>clear()</code>          | <code>clear()</code>                |
| <code>p = find(x)</code>      | <code>p = find(x)</code>            |
| <code>begin()</code>          | <code>begin()</code>                |
| <code>end()</code>            | <code>end()</code>                  |

Tabla 4.2: Tabla de equivalencia entre las operaciones del TAD conjunto y el TAD correspondencia.

## 4.3. Implementación con listas

### 4.3.0.1. Similaridad entre los TAD conjunto y correspondencia

Notemos que el TAD CONJUNTO es muy cercano al TAD CORRESPONDENCIA, de hecho podríamos representar conjuntos con correspondencias, simplemente usando las claves de la correspondencia como elementos del conjunto e ignorando el valor del contradominio. En la Tabla 4.2 vemos la equivalencia entre la mayoría de las operaciones de la clase. Sin embargo, las operaciones binarias **set\_union(A,B,C)**, **set\_intersection(A,B,C)** y **set\_difference(A,B,C)** no tiene equivalencia dentro de la correspondencia. De manera que si tenemos una implementación del TAD correspondencia, y podemos implementar las funciones binarias, entonces con pequeñas modificaciones adicionales podemos obtener una implementación de conjuntos.

En la sección §2.4, se discutieron las posibles implementaciones de correspondencias con contenedores lineales (listas y vectores) ordenados y no ordenados. Consideraremos por ejemplo la posibilidad de extender la implementación de correspondencia por listas ordenadas y no ordenadas a conjuntos. Como ya se discutió, en ambos casos las operaciones de inserción supresión terminan siendo  $O(n)$  ya que hay que recorrer el contenedor para encontrar el elemento.

Sin embargo, hay una gran diferencia para el caso de las operaciones binarias. Consideraremos por ejemplo **set\_union(A,B,C)**. En el caso de contenedores no ordenados, la única posibilidad es comparar cada uno de los elementos  $x_a$  de  $A$  con cada uno de los elementos de  $B$ . Si se encuentra el elemento  $x_a$  en  $B$ , entonces el elemento es insertado en  $C$ . Tal algoritmo es  $O(n_A n_B)$ , donde  $n_{A,B}$  es el número de elementos en  $A$  y  $B$ . Si el número de elementos en los dos contenedores es similar ( $n_A \sim n_B \sim n$ ), entonces vemos que es  $O(n^2)$ .

### 4.3.0.2. Algoritmo lineal para las operaciones binarias

Con listas ordenadas se puede implementar una versión de **set\_union(A,B,C)** que es  $O(n)$ . Mantenemos dos posiciones **pa** y **pb**, cuyos valores llamaremos  $x_a$  y  $x_b$ , tales que

- los elementos en  $A$  en posiciones anteriores a **pa** (y por lo tanto menores que  $x_a$ ) son menores que  $x_b$  y viceversa,
- todos los valores en  $B$  antes de **pb** son menores que  $x_a$ .

Estas condiciones son bastante fuertes, por ejemplo si el valor  $x_a < x_b$  entonces podemos asegurar que  $x_a \notin B$ . Efectivamente, en ese caso, todos los elementos anteriores a  $x_b$  son menores a  $x_a$  y por lo tanto distintos a  $x_a$ . Por otra parte los que están después de  $x_b$  son mayores que  $x_b$  y por lo tanto que  $x_a$ , con lo cual también son distintos. Como conclusión, no hay ningún elemento en  $B$  que sea igual a  $x_a$ , es decir  $x_a \notin B$ . Similarmente, si  $x_b < x_a$  entonces se ve que  $x_b \notin A$ .

Inicialmente podemos poner **pa=A.begin()** y **pb=B.begin()**. Como antes de **pa** y **pb** no hay elementos, la condición se cumple. Ahora debemos avanzar **pa** y **pb** de tal forma que se siga cumpliendo la condición. En cada paso puede ser que avancemos **pa**, **pb** o ambos. Por ejemplo, consideremos el caso

$$A = \{1, 3, 5, 7, 10\}, \quad B = \{3, 5, 7, 9, 10, 12\} \quad (4.13)$$

Inicialmente tenemos  $x_a = 1$  y  $x_b = 3$ . Si avanzamos **pb**, de manera que  $x_b$  pasa a ser 5, entonces la segunda condición no se satisfará, ya que el 3 en  $B$  no es menor que  $x_a$  que es 1. Por otra parte, si avanzamos **pa**, entonces sí se seguirán satisfaciendo ambas condiciones, y en general esto será siempre así, mientras *avancemos siempre el elemento menor*. Efectivamente, digamos que los elementos de  $A$  son  $x_a^0, x_a^1, \dots, x_a^{n-1}$ , y los de  $B$  son  $x_b^0, x_b^1, \dots, x_b^{m-1}$  y que en un dado momento **pa** está en  $x_a^j$  y **pb** en  $x_b^k$ . Esto quiere decir que, por las condiciones anteriores,

$$\begin{aligned} x_a^0, x_a^1, \dots, x_a^{j-1} &< x_b^k \\ x_b^0, x_b^1, \dots, x_b^{k-1} &< x_a^j \end{aligned} \quad (4.14)$$

Si

$$x_a^j < x_b^k \quad (4.15)$$

entonces en el siguiente paso **pa** avanza a  $x_a^{j+1}$ . Los elementos de  $B$  antes de **pb** siguen satisfaciendo

$$x_b^0, x_b^1, \dots, x_b^{k-1} < x_a^j < x_a^{j+1} \quad (4.16)$$

con lo cual la condición sobre **pa** se sigue cumpliendo. Por otra parte, ahora a los elementos antes de **pa** se agregó  $x_a^j$  con lo cual tenemos antes de **pa**

$$x_a^0, x_a^1, \dots, x_a^{j-1}, x_a^j \quad (4.17)$$

que cumplen la condición requerida por (4.14) y (4.15). Puede verse las condiciones también se siguen satisfaciendo si  $x_a^j > x_b^k$  y avanzamos **pb**, o si  $x_a^j = x_b^k$  y avanzamos ambas posiciones.

Para cualquiera de las operaciones binarias inicializamos los iterators con **pa=A.begin()** y **pb=B.begin()** y los vamos avanzando con el mecanismo explicado, es decir siempre el menor o los dos cuando son iguales. El proceso se detiene cuando alguno de los iterators llega al final de su conjunto. En cada paso alguno de los iterators avanza, de manera que es claro que en un número finito de pasos alguno de los iterators llegará al fin, de hecho en menos de  $n_a + n_b$  pasos. Las posiciones **pa** y **pb** recorren todos los elementos de alguno de los dos conjuntos, mientras que en el otro puede quedar un cierto “resto”, es decir una cierta cantidad de elementos al final de la lista.

Ahora consideremos la operación de **set\_union(A,B,C)**. Debemos asegurarnos de insertar todos los elementos de  $A$  y  $B$ , pero una sola vez y en forma ordenada. Esto se logra si en cada paso insertamos en el fin de  $C$  el elemento menor de  $x_a$  y  $x_b$  (si son iguales se inserta una sola vez). Efectivamente, si en un momento  $x_a < x_b$  entonces, por lo discutido previamente  $x_a$  seguramente no está en  $B$  y podemos insertarlo en  $C$ , ya que en el siguiente paso **pa** avanzará, dejándolo atrás, con lo cual seguramente no lo

insertaremos nuevamente. Además en pasos previos  $x_a$  no puede haber sido insertado ya que si era el menor **pa** habría avanzado dejándolo atrás y si era el mayor no habría sido insertado. El caso en que son iguales puede analizarse en forma similar. Puede verse que los elementos de  $C$  quedan ordenados, ya que de  $x_a$  y  $x_b$  siempre avanzamos el menor. Una vez que uno de los iteradores (digamos **pa**) llegó al final, si quedan elementos en  $B$  (el “resto”) entonces podemos insertarlos directamente al fin de  $C$  ya que está garantizado que estos elementos no pueden estar en  $A$ .

En el caso de los conjuntos en (4.13) un seguimiento de las operaciones arroja lo siguiente

- $x_a=1, x_b=3$ , inserta 1 en  $C$
- $x_a=3, x_b=3$ , inserta 3 en  $C$
- $x_a=5, x_b=5$ , inserta 5 en  $C$
- $x_a=7, x_b=7$ , inserta 7 en  $C$
- $x_a=10, x_b=9$ , inserta 9 en  $C$
- $x_a=10, x_b=10$ , inserta 10 en  $C$
- **pa** llega a **A.end()**
- Inserta todo el resto de  $B$  (el elemento 12) en  $C$ .

quedando  $C = \{1, 3, 5, 7, 9, 10, 12\}$  que es el resultado correcto.

En el caso de **set\_intersection(A,B,C)** sólo hay que insertar  $x_a$  cuando  $x_a = x_b$  y los restos no se insertan. El seguimiento arroja

- $x_a=1, x_b=3$ ,
- $x_a=3, x_b=3$ , inserta 3
- $x_a=5, x_b=5$ , inserta 5
- $x_a=7, x_b=7$ , inserta 7
- $x_a=10, x_b=9$ ,
- $x_a=10, x_b=10$ , inserta 10
- **pa** llega a **A.end()**

Para **set\_difference(A,B,C)** hay que insertar  $x_a$  si  $x_a < x_b$  y  $x_b$  no se inserta nunca. Al final sólo se inserta el resto de  $A$ .

- $x_a=1, x_b=3$ , inserta 1
- $x_a=3, x_b=3$ ,
- $x_a=5, x_b=5$ ,
- $x_a=7, x_b=7$ ,
- $x_a=10, x_b=9$ ,
- $x_a=10, x_b=10$ ,
- **pa** llega a **A.end()**

Con estos algoritmos, los tiempos de ejecución son  $O(n_A + n_B)$  ( $O(n)$  si los tamaños son similares).

#### 4.3.0.3. Descripción de la implementación

---

```
1 typedef int elem_t;
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```
2 typedef list<elem_t>::iterator iterator_t;
3
4 class set {
5 private:
6 list<elem_t> L;
7 public:
8 set() {}
9 set(const set &A) : L(A.L) {}
10 ~set() {}
11 elem_t retrieve(iterator_t p) { return *p; }
12 iterator_t lower_bound(elem_t t) {
13 list<elem_t>::iterator p = L.begin();
14 while (p!=L.end() && t>*p) p++;
15 return p;
16 }
17 iterator_t next(iterator_t p) { return ++p; }
18 pair<iterator_t,bool> insert(elem_t x) {
19 pair<iterator_t,bool> q;
20 iterator_t p;
21 p = lower_bound(x);
22 q.second = p==end() || *p!=x;
23 if(q.second) p = L.insert(p,x);
24 q.first = p;
25 return q;
26 }
27 void erase(iterator_t p) { L.erase(p); }
28 void erase(elem_t x) {
29 list<elem_t>::iterator
30 p = lower_bound(x);
31 if (p!=end() && *p==x) L.erase(p);
32 }
33 void clear() { L.clear(); }
34 iterator_t find(elem_t x) {
35 list<elem_t>::iterator
36 p = lower_bound(x);
37 if (p!=end() && *p==x) return p;
38 else return L.end();
39 }
40 iterator_t begin() { return L.begin(); }
41 iterator_t end() { return L.end(); }
42 int size() { return L.size(); }
43 friend void set_union(set &A,set &B,set &C);
44 friend void set_intersection(set &A,set &B,set &C);
45 friend void set_difference(set &A,set &B,set &C);
46 };
47
48 void set_union(set &A,set &B,set &C) {
49 C.clear();
50 list<elem_t>::iterator pa = A.L.begin(),
51 pb = B.L.begin(), pc = C.L.begin();
52 while (pa!=A.L.end() && pb!=B.L.end()) {
53 if (*pa<*pb) { pc = C.L.insert(pc,*pa); pa++; }
54 else { pc = C.L.insert(pc,*pb); pb++; }
```

```

55 else if (*pa>*pb) {pc = C.L.insert(pc,*pb); pb++; }
56 else {pc = C.L.insert(pc,*pa); pa++; pb++; }
57 pc++;
58 }
59 while (pa!=A.L.end()) {
60 pc = C.L.insert(pc,*pa);
61 pa++; pc++;
62 }
63 while (pb!=B.L.end()) {
64 pc = C.L.insert(pc,*pb);
65 pb++; pc++;
66 }
67 }
68 void set_intersection(set &A,set &B,set &C) {
69 C.clear();
70 list<elem_t>::iterator pa = A.L.begin(),
71 pb = B.L.begin(), pc = C.L.begin();
72 while (pa!=A.L.end() && pb!=B.L.end()) {
73 if (*pa<*pb) pa++;
74 else if (*pa>*pb) pb++;
75 else { pc=C.L.insert(pc,*pa); pa++; pb++; pc++; }
76 }
77 }
78 // C = A - B
79 void set_difference(set &A,set &B,set &C) {
80 C.clear();
81 list<elem_t>::iterator pa = A.L.begin(),
82 pb = B.L.begin(), pc = C.L.begin();
83 while (pa!=A.L.end() && pb!=B.L.end()) {
84 if (*pa<*pb) { pc=C.L.insert(pc,*pa); pa++; pc++; }
85 else if (*pa>*pb) pb++;
86 else { pa++; pb++; }
87 }
88 while (pa!=A.L.end()) {
89 pc = C.L.insert(pc,*pa);
90 pa++; pc++;
91 }
92 }
```

**Código 4.7:** Implementación de conjuntos con listas ordenadas. [Archivo: *setbas.h*]

En el código 4.7 vemos una posible implementación de conjuntos con listas ordenadas.

- Los métodos de la clase son muy similares a los de correspondencia y no serán explicados nuevamente (ver sección §2.4). Consideremos por ejemplo `p=insert(x)`. La única diferencia con el `insert()` de `map` es que en aquel caso hay que insertar un par que consiste en la clave y el valor de contradominio, mientras que en `set` sólo hay que insertar el elemento.
- Las funciones binarias no pertenecen a la clase y el algoritmo utilizado responde a lo descripto en la sección §4.3.0.2. Notar que al final de `set_union()` se insertan *los dos restos* de *A* y *B* a la cola de *C*. Esto se debe a que al llegar a este punto uno de los dos iterators está en `end()` de manera que sólo uno de los lazos se ejecutará efectivamente.

#### 4.3.0.4. Tiempos de ejecución

| Método                                                                                                                                | $T(N)$ |
|---------------------------------------------------------------------------------------------------------------------------------------|--------|
| retrieve(p), insert(x), erase(x), clear(), find(x), lower_bound(x), set_union(A,B,C), set_intersection(A,B,C), set_difference(A,B,C), | $O(n)$ |
| erase(p), begin(), end(),                                                                                                             | $O(1)$ |

Tabla 4.3: Tiempos de ejecución de los métodos del TAD conjunto implementado con listas ordenadas.

## 4.4. Interfaz avanzada para conjuntos

---

```

1 template<class T>
2 class set {
3 private:
4 /* . . . */
5 public:
6 class iterator {
7 friend class set;
8 T & operator*();
9 T *operator->();
10 bool operator!=(iterator q);
11 bool operator==(iterator q);
12 }
13 set() {}
14 set(const set &A) : L(A.L) {}
15 ~set() {}
16 set &operator=(set<T> &);
17 iterator lower_bound(T t);
18 pair<iterator, bool> insert(T x);
19 void erase(iterator p);
20 int erase(T x);
21 void clear();
22 iterator find(T x);
23 iterator begin();
24 iterator end();
25 int size();
26 };
27
28 template<class T>
29 void set_union(set<T> &A, set<T> &B, set<T> &C);
30
31 template<class T>
32 void set_intersection(set<T> &A, set<T> &B, set<T> &C);
33
34 template<class T>
35 void set_difference(set<T> &A, set<T> &B, set<T> &C);

```

---

**Código 4.8:** Interfaz avanzada para conjuntos [Archivo: seth.h]

En el código 4.8 se puede ver la interfaz avanzada para conjuntos, es decir utilizando templates, clases anidadas y sobrecarga de operadores. Las diferencias con la interfaz básica son similares a las de otros TAD (por ejemplo `tree<>`).

- La clase `set` pasa a ser ahora un template, de manera que podremos declarar `set<int>`, `set<double>`.
- La clase `iterator` es ahora una clase anidada dentro de `set`. Externamente se verá como `set<int>::iterator`
- La dereferenciación de posiciones (`x=retrieve(p)`) se reemplaza por `x=*p`. Sobrecargando el operador `*`. Si el tipo elemento (es decir el tipo `T` del template) contiene campos dato o métodos, podemos escribir `p->campo` o `p->f(...)`. Para esto sobrecargamos los operadores `operator*` y `operator->`.
- Igual que con la interfaz básica, para poder hacer comparaciones de iterators debemos sobrecargar también los operadores `==` y `!=` en la clase iterator.
- `erase(x)` retorna el número de elementos efectivamente eliminados.
- `insert(x)` retorna un `pair<iterator, bool>`. (ver sección §2.4.3.2). El primero es, como siempre, un iterator al elemento insertado. El segundo indica si el elemento realmente fue insertado o ya estaba en el conjunto.

---

```

1 template<class T>
2 class set {
3 private:
4 list<T> L;
5 public:
6 typedef typename list<T>::iterator iterator;
7 typedef pair<iterator,bool> pair_t;
8 set() {}
9 set(const set &A) : L(A.L) {}
10 ~set() {}
11 iterator lower_bound(T t) {
12 iterator p = L.begin();
13 while (p!=L.end() && t>*p) p++;
14 return p;
15 }
16 pair_t insert(T x) {
17 iterator p = lower_bound(x);
18 if(p==end() || *p!=x) {
19 p = L.insert(p,x);
20 return pair_t(p,true);
21 } else {
22 return pair_t(end(),false);
23 }
24 }
25 void erase(iterator p) { L.erase(p); }
26 int erase(T x) {
27 iterator p = lower_bound(x);
28 if (p!=end() && *p==x) {
29 L.erase(p); return 1;

```

```

30 } else return 0;
31 }
32 void clear() { L.clear(); }
33 iterator find(T x) {
34 iterator p = lower_bound(x);
35 if (p!=end() && *p==x) return p;
36 else return L.end();
37 }
38 iterator begin() { return L.begin(); }
39 iterator end() { return L.end(); }
40 int size() { return L.size(); }
41 bool empty() { return !L.size(); }
42 friend void set_union<>(set<T> &A, set<T> &B, set<T> &C);
43 friend void set_intersection<>(set<T> &A, set<T> &B, set<T> &C);
44 friend void set_difference<>(set<T> &A, set<T> &B, set<T> &C);
45 };
46
47 template<class T>
48 void set_union(set<T> &A, set<T> &B, set<T> &C) {
49 C.clear();
50 typename list<T>::iterator pa = A.L.begin(),
51 pb = B.L.begin(), pc = C.L.begin();
52 while (pa!=A.L.end() && pb!=B.L.end()) {
53 if (*pa<*pb) {pc = C.L.insert(pc,*pa); pa++; }
54 else if (*pa>*pb) {pc = C.L.insert(pc,*pb); pb++; }
55 else {pc = C.L.insert(pc,*pa); pa++; pb++; }
56 pc++;
57 }
58 while (pa!=A.L.end()) {
59 pc = C.L.insert(pc,*pa); pa++; pc++;
60 }
61 while (pb!=B.L.end()) {
62 pc = C.L.insert(pc,*pb); pb++; pc++;
63 }
64 }
65
66 template<class T>
67 void set_intersection(set<T> &A, set<T> &B, set<T> &C) {
68 C.clear();
69 typename list<T>::iterator pa = A.L.begin(),
70 pb = B.L.begin(), pc = C.L.begin();
71 while (pa!=A.L.end() && pb!=B.L.end()) {
72 if (*pa<*pb) pa++;
73 else if (*pa>*pb) pb++;
74 else {C.L.insert(pc,*pa); pa++; pb++; }
75 }
76 }
77
78 // C = A - B
79 template<class T>
80 void set_difference(set<T> &A, set<T> &B, set<T> &C) {
81 C.clear();
82 typename list<T>::iterator pa = A.L.begin(),

```

```

83 pb = B.L.begin(), pc = C.L.begin();
84 while (pa!=A.L.end() && pb!=B.L.end()) {
85 if (*pa<*pb) {C.L.insert(pc,*pa); pa++;}
86 else if (*pa>*pb) pb++;
87 else { pa++; pb++; }
88 }
89 while (pa!=A.L.end()) {
90 pc = C.L.insert(pc,*pa); pa++; pc++;
91 }
92 }
```

**Código 4.9:** Implementación de la interfaz avanzada para conjuntos con listas ordenadas. [Archivo: setl.h]

Una implementación de la interfaz avanzada basada en listas ordenadas puede observarse en el código 4.9.

## 4.5. El diccionario

En algunas aplicaciones puede ser que se necesite un TAD como el conjunto pero sin necesidad de las funciones binarias. A un tal TAD lo llamamos TAD DICCCIONARIO. Por supuesto cualquier implementación de conjuntos puede servir como diccionario, por ejemplo con vectores de bits, contenedores lineales ordenados o no. Sin embargo existe una implementación muy eficiente para la cual las inserciones y supresiones son  $O(1)$ , basada en la estructura “*tabla de dispersión*” (“hash tables”). Sin embargo, no es simple implementar en forma eficiente las operaciones binarias para esta implementación, por lo cual no es un buen candidato para conjuntos.

### 4.5.1. La estructura tabla de dispersión

La idea esencial es que dividimos el conjunto universal en  $B$  “cubetas” (“buckets” o “bins”), de tal manera que, a medida que nuevos elementos son insertados en el diccionario, estos son desviados a la cubeta correspondiente. Por ejemplo, si consideramos diccionarios de cadenas de caracteres en el rango **a-z**, es decir letras minúsculas, entonces podemos dividir al conjunto universal en  $B=26$  cubetas. La primera corresponde a todos los strings que comienzan con **a**, la segunda los que comienzan con **b** y así siguiendo hasta la **z**. En general tendremos una función de dispersión **int b = h(elem\_t t)** que nos da el número de cubeta **b** en el cual debe ser almacenado el elemento **t**. En el ejemplo descripto, la función podría implementarse como sigue

```

1 int h(string t) {
2 return t[0]-'a';
3 }
```

En este caso está garantizado que los números de cubetas devueltos por **h()** están en el rango  $[0, B)$ . En la práctica, el programador de la clase puede proveer funciones de hash para los tipos más usuales (como **int, double, string...**) dejando la posibilidad de que el usuario defina la función de hash para otros tipos, o también para los tipos básicos si considera que los que el provee son más eficientes (ya veremos cuáles son los requisitos para una buena función de hash). Asumiremos siempre que el tiempo de ejecución de la

función de dispersión es  $O(1)$ . Para mayor seguridad, asignamos al elemento  $\mathbf{t}$  la cubeta  $\mathbf{b}=\mathbf{h}(\mathbf{t})\%B$ , de esta forma está siempre garantizado que  $\mathbf{b}$  está en el rango  $[0, B)$ .

Básicamente, las cubetas son guardadas en un arreglo de cubetas (`vector<elem_t> v(B)`). Para insertar un elemento, simplemente calculamos la cubeta a usando la función de dispersión y guardamos el elemento en esa cubeta. Para hacer un `find(x)` o `erase(x)`, calculamos la cubeta y verificamos si el elemento está en la cubeta o no. De esta forma tanto las inserciones como las supresiones son  $O(1)$ . Este costo tan bajo es el interés principal de estas estructuras.

Pero normalmente el número de cubetas es mucho menor que el número de elementos del conjunto universal  $N$  (en muchos casos este último es infinito). En el ejemplo de los strings, todos los strings que empiezan con `a` van a la primera cubeta. Si un elemento es insertado y la cubeta correspondiente ya está ocupada decimos que hay una “*colisión*” y no podemos insertar el elemento en la tabla. Por ejemplo, si  $B = 10$  y usamos `h(x)=x`, entonces si se insertan los elementos  $\{1, 13, 4, 1, 24\}$  entonces los tres primeros elementos van a las cubetas 1, 3 y 4 respectivamente. El cuarto elemento también va a la cubeta 1, la cual ya está ocupada, pero no importa ya que es el *mismo* elemento que está en la cubeta. El problema es al insertar el elemento 24, ya que va a parar a la cubeta 4 la cual ya está ocupada, pero por *otro* elemento (el 4).

Si el número de elementos en el conjunto  $n$  es pequeño con respecto al número de cubetas ( $n \ll B$ ) entonces la probabilidad de que dos elementos vayan a la misma cubeta es pequeña, pero en este caso la memoria requerida (que es el tamaño del vector `v`, es decir  $O(B)$ ) sería mucho mayor que el tamaño del conjunto, con lo cual la utilidad práctica de esta implementación sería muy limitada. De manera que debe definirse una estrategia para “*resolver colisiones*”. Hay al menos dos formas bien conocidas:

- Usar “*tablas de dispersión abiertas*”.
- Usar redispersión en “*tablas de dispersión cerradas*”.

### 4.5.2. Tablas de dispersión abiertas

Esta es la forma más simple de resolver el problema de las colisiones. En esta implementación las cubetas no son elementos, sino que son listas (simplemente enlazadas) de elementos, es decir el vector `v` es de tipo `vector< list<elem_t> >`). De esta forma cada cubeta puede contener (teóricamente) infinitos elementos. Los elementos pueden insertarse en las cubetas en cualquier orden o ordenadas. La discusión de la eficiencia en este caso es similar a la de correspondencia con contenedores lineales (ver sección §2.4).

A continuación discutiremos la implementación del TAD diccionario implementado por tablas de dispersión abiertas con listas desordenadas. La inserción de un elemento `x` pasa por calcular el número de cubeta usando la función de dispersión y revisar la lista (cubeta) correspondiente. Si el elemento está en la lista, entonces no es necesario hacer nada. Si no está, podemos insertar el elemento en cualquier posición, puede ser en `end()`. El costo de la inserción es, en el peor caso, cuando elemento no esta en la lista, proporcional al número de elementos en la lista (cubeta). Si tenemos  $n$  elementos, el número de elementos por cubeta será, en promedio,  $n/B$ . Si el número de cubetas es  $B \approx n$ , entonces  $n/B \approx 1$  y el tiempo d ejecución es  $O(1 + n/B)$ . El 1 tiene en cuenta acá de que al menos hay que calcular la función de dispersión. Como el término  $n/B$  puede ser menor, e incluso mucho menor, que 1, entonces hay que mantener el término 1, de lo contrario estaríamos diciendo que en ese caso el costo de la función es mucho menor que 1. En el peor caso, todos los elementos pueden terminar en una sola cubeta, en cuyo caso la inserción sería  $O(n)$ . Algo similar pasa con `erase()`.

#### 4.5.2.1. Detalles de implementación

```
1 typedef int key_t;
2
3 class hash_set;
4 class iterator_t {
5 friend class hash_set;
6 private:
7 int bucket;
8 std::list<key_t>::iterator p;
9 iterator_t(int b, std::list<key_t>::iterator q)
10 : bucket(b), p(q) { }
11 public:
12 bool operator==(iterator_t q) {
13 return (bucket == q.bucket && p==q.p);
14 }
15 bool operator!=(iterator_t q) {
16 return !(*this==q);
17 }
18 iterator_t() { }
19 };
20 typedef int (*hash_fun)(key_t x);
21
22 class hash_set {
23 private:
24 typedef std::list<key_t> list_t;
25 typedef list_t::iterator listit_t;
26 typedef std::pair<iterator_t,bool> pair_t;
27 hash_set(const hash_set&) {}
28 hash_set& operator=(const hash_set&) {}
29 hash_fun h;
30 int B;
31 int count;
32 std::vector<list_t> v;
33 iterator_t next_aux(iterator_t p) {
34 while (p.p==v[p.bucket].end()
35 && p.bucket<B-1) {
36 p.bucket++;
37 p.p = v[p.bucket].begin();
38 }
39 return p;
40 }
41 public:
42 hash_set(int B_a, hash_fun h_a)
43 : B(B_a), v(B), h(h_a), count(0) { }
44 iterator_t begin() {
45 iterator_t p = iterator_t(0,v[0].begin());
46 return next_aux(p);
47 }
48 iterator_t end() {
49 return iterator_t(B-1,v[B-1].end());
```

```
50 }
51 iterator_t next(iterator_t p) {
52 p.p++; return next_aux(p);
53 }
54 key_t retrieve(iterator_t p) { return *p.p; }
55 pair_t insert(const key_t& x) {
56 int b = h(x) % B;
57 list_t &L = v[b];
58 listit_t p = L.begin();
59 while (p!=L.end() && *p!=x) p++;
60 if (p!=L.end())
61 return pair_t(iterator_t(b,p),false);
62 else {
63 count++;
64 p = L.insert(p,x);
65 return pair_t(iterator_t(b,p),true);
66 }
67 }
68 iterator_t find(key_t& x) {
69 int b = h(x) % B;
70 list_t &L = v[b];
71 listit_t p = L.begin();
72 while (p!=L.end() && *p!=x) p++;
73 if (p!=L.end())
74 return iterator_t(b,p);
75 else return end();
76 }
77 int erase(const key_t& x) {
78 list_t &L = v[h(x) % B];
79 listit_t p = L.begin();
80 while (p!=L.end() && *p!=x) p++;
81 if (p!=L.end()) {
82 L.erase(p);
83 count--;
84 return 1;
85 } else return 0;
86 }
87 void erase(iterator_t p) {
88 v[p.bucket].erase(p.p);
89 }
90 void clear() {
91 count=0;
92 for (int j=0; j<B; j++) v[j].clear();
93 }
94 int size() { return count; }
95 };
```

**Código 4.10:** Diccionario implementado por tablas de dispersión abiertas con listas desordenadas. [Archivo: *hashsetboso.h*]

En el código 4.10 vemos una posible implementación.

- Usamos `vector<>` y `list<>` de STL para implementar los vectores y listas, respectivamente.
- Los elementos a insertar en el diccionario son de tipo `key_t`.
- El `typedef` de la línea 20 define un tipo para las funciones admisibles como funciones de dispersión. Estas son funciones que deben tomar como argumento un elemento de tipo `key_t` y devuelve un entero.
- La clase contiene un puntero `h` a la función de dispersión. Este puntero se define en el constructor.
- El constructor toma como argumentos el número de cubetas y el puntero a la función de dispersión y los copia en los valores internos. Redimensiona el vector de cubetas `v` e inicializa el contador de elementos `count`.
- La clase iterator consiste de el número de cubeta (`bucket`) y un iterator en la lista (`p`). Las posiciones válidas son, como siempre, posiciones dereferenciables y `end()`.
- Los iterators dereferenciable consisten en un número de cubeta no vacía y una posición dereferenciable en la lista correspondiente.
- El iterator `end()` consiste en el par `bucket=B-1` y `p` el `end()` de esa lista (es decir, `v[bucket].end()`)
- Hay que tener cuidado de, al avanzar un iterator siempre llegar a otro iterator válido. La función privada `next_aux()` avanza cualquier combinación de cubeta y posición en la lista (puede no ser válida, por ejemplo el `end()` de una lista que no es la última) hasta la siguiente posición válida.
- El tiempo de ejecución de `next_aux()` es 1 si simplemente avanza una posición en la misma cubeta sin llegar a `end()`. Si esto último ocurre entonces entra en un lazo sobre las cubetas del cual sólo sale cuando llega a una cubeta no vacía. Si el número de elementos es mayor que  $B$  entonces en promedio todas las cubetas tienen al menos un elemento y `next_aux()` a lo sumo debe avanzar a la siguiente cubeta. Es decir, el cuerpo del lazo dentro de `next_aux()` se ejecuta una sola vez. Si  $n \ll B$  entonces el número de cubetas llenas es aproximadamente  $n$  y el de vacías  $\approx B$  (en realidad esto no es exactamente así ya que hay una cierta probabilidad de que dos elementos vayan a la misma cubeta). Entonces, por cada cubeta llena hay  $B/n$  cubetas vacías y el lazo en `next_aux()` debe ejecutarse  $B/n$  veces. Finalmente,  $B/n$  da infinito para  $n = 0$  y en realidad el número de veces que se ejecuta el lazo no es infinito sino  $B$ .
- `begin()` devuelve la primera posición válida en el diccionario. Para ello toma el iterator correspondiente a la posición `begin()` de la primera lista y le aplica `next_aux()`. Esto puede resultar en una posición dereferenciable o en `end()` (si el diccionario está vacío).
- `insert(x)` y `erase(x)` proceden de acuerdo a lo explicado en la sección previa.
- Notar la definición de referencias a listas en las líneas 57 y líneas 78. Al declarar `L` como referencias a listas (por el `&`) *no se crea una copia* de la lista.
- `next(p)` incrementa la posición en la lista. Pero con esto no basta ya que puede estar en el `end()` de una lista que no es la última cubeta. Por eso se aplica `next_aux()` que sí avanza a la siguiente posición válida.

#### 4.5.2.2. Tiempos de ejecución

En la Tabla 4.4 vemos los tiempos de ejecución correspondientes. El tiempo de ejecución de `next()` y `begin()` se debe a que llaman a que llaman a `next_aux()`.

| Método                       | $T(n, B)$ (promedio)                                                                                                | $T(n, B)$ (peor caso) |
|------------------------------|---------------------------------------------------------------------------------------------------------------------|-----------------------|
| retrieve(p), erase(p), end() | $O(1)$                                                                                                              | $O(1)$                |
| insert(x), find(x), erase(x) | $O(1 + n/B)$                                                                                                        | $O(n)$                |
| begin(), next(p)             | $\begin{cases} \text{si } n = 0, & O(B); \\ \text{si } n \leq B, & O(B/n); \\ \text{si } n > B & O(1); \end{cases}$ | $O(B)$                |
| clear()                      | $O(n + B)$                                                                                                          | $O(n + B)$            |

Tabla 4.4: Tiempos de ejecución de los métodos del TAD diccionario implementado con tablas de dispersión abiertas.

#### 4.5.3. Funciones de dispersión

De los costos que hemos analizado para las tablas de dispersión abierta se deduce que para que éstas sean efectivas los elementos deben ser distribuidos uniformemente sobre las cubetas. El diseño de una buena función de dispersión es precisamente ése. Pensemos por ejemplo en el caso de una tabla de dispersión para strings con  $B=256$  cubetas. Como función de dispersión podemos tomar

$$h_1(x) = (\text{Código ASCII del primer carácter de } x) \quad (4.18)$$

Si ésta función de dispersión es usada con strings que provienen por ejemplo de palabras encontradas en texto usual en algún lenguaje como español, entonces es probable que haya muchas más palabras que comiencen con la letra **a** y por lo tanto vayan a la cubeta 97 (el valor ASCII de **a**) que con la letra **x** (cubeta 120).

---

```

1 int h2(string s) {
2 int v = 0;
3 for (int j=0; j<s.size(); j++) {
4 v += s[j];
5 v = v % 256;
6 }
7 return v;
8 }
```

---

Código 4.11: Función de dispersión para strings [Archivo: hash.cpp]

Una mejor posibilidad es la función **h2()** mostrada en el código 4.11. Esta función calcula la suma de los códigos ASCII de todos los caracteres del string, módulo 256. Notamos primero que basta con que dos strings tengan un sólo carácter diferente para que sus valores de función de dispersión sean diferentes. Por ejemplo, los strings **argonauta** y **argonautas** irán a diferentes cubetas ya que difieren en un carácter. Sin embargo las palabras **vibora** y **bravio** irán a la misma ya que los caracteres son los mismos, pero en diferente orden (son anagramas la una de la otra). Sin embargo, parece intuitivo que en general  $h_2$  dispersará mucho mejor los strings que  $h_1$ . En última instancia el criterio para determinar si una función de dispersión es buena o no es estadístico. Se deben considerar “ensambles” de elementos (en este ejemplo strings) representativos y ver que tan bien se desempeñan las funciones potenciales para esos ensambles.

#### 4.5.4. Tablas de dispersión cerradas

Otra posibilidad para resolver el problema de las colisiones es usar otra cubeta cercana a la que indica la función de dispersión. A estas estrategias se les llama de “*redispersión*”. La tabla es un **vector<key\_t>** e inicialmente todos los elementos están inicializados a un valor especial que llamaremos **undef** (“*indefinido*”). Si el diccionario guarda valores enteros positivos podemos usar 0 como **undef**. Si los valores son reales (**double**'s o **float**'s) entonces podemos usar como **undef** el valor **DBL\_MAX** (definido en el header **float.h**). Este es el valor más grande representable en esa computadora (en el caso de un procesador de 32bits con el SO GNU/Linux y el compilador GCC es **1.7976931348623157e+308**). También puede ser **NAN**. Si estamos almacenando nombres de personas podemos usar la cadena **<NONE>** (esperando que nadie se llame de esa forma) y así siguiendo. Para insertar un elemento  $x$  calculamos la función de dispersión  $\text{init} = h(x)$  para ver cuál cubeta le corresponde *initialmente*. Si la cubeta está libre (es decir el valor almacenado es **undef**), entonces podemos insertar el elemento en esa posición. Si está ocupado, entonces podemos probar en la siguiente cubeta (en “*sentido circular*”, es decir si la cubeta es  $B - 1$  la siguiente es la 0) **(init+1)%B**. Si está libre lo insertamos allí, si está ocupada y el elemento almacenado no es  $x$ , seguimos con la siguiente, etc... hasta llegar a una libre. Si todas las cubetas están ocupadas entonces la tabla está llena y el programa debe señalar un error (o agrandar la tabla dinámicamente). *Nota:* En las STL la implementación de diccionario corresponde al header **unordered\_set** (esto recientemente fue cambiado, en versiones anteriores el nombre era **hash\_set** (pero no era parte del estándar)).

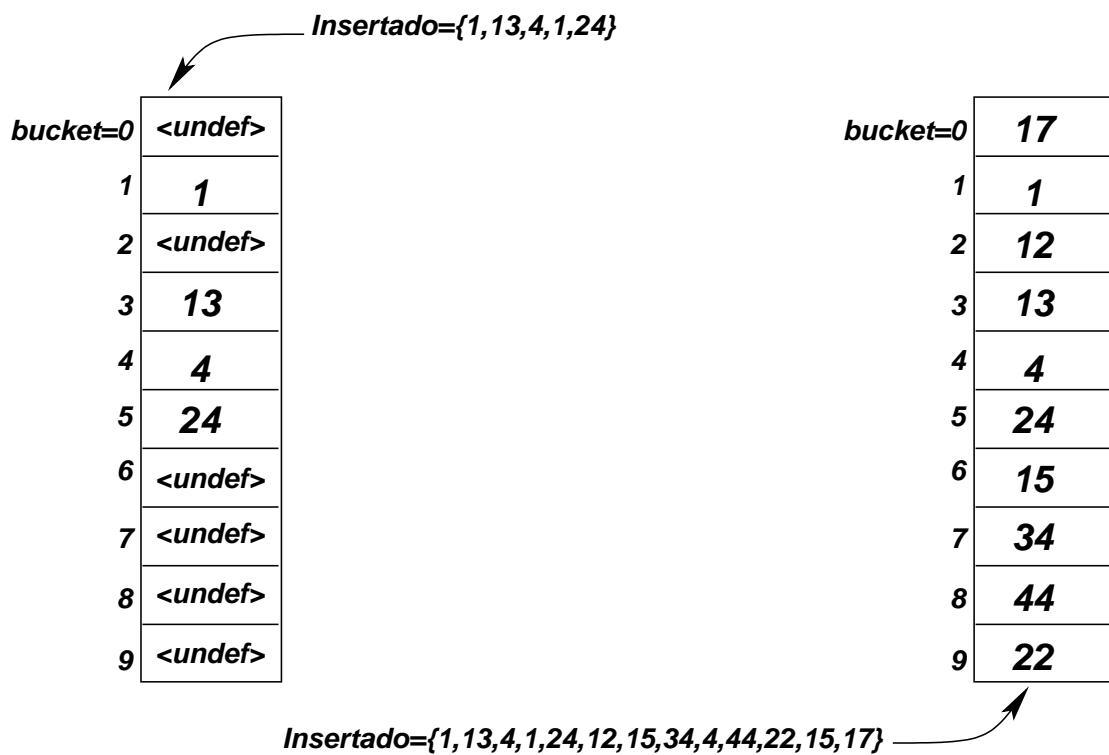


Figura 4.3: Ejemplo de inserción en tablas cerradas.

Consideremos por ejemplo una tabla de dispersión para enteros con  $B = 10$  cubetas y dispersión lineal

$(h(x) = x)$ , insertando los elementos 1, 13, 4, 1, 24, 12, 15, 34, 4, 44, 22, 15, 17, 19. Al insertar los elementos 1, 13 y 4 las cubetas respectivas (0,3,4) están vacías y los elementos son insertados. Al insertar el cuarto elemento (un 1) la cubeta respectiva (la 1) está ocupada, pero el elemento almacenado es el 1, de manera que no hace falta insertarlo nuevamente. El quinto elemento es un 24. Al querer insertarlo en la cubeta correspondiente (la 4) vemos que está ocupada y el elemento almacenado es un 4 (no un 24), de manera que probamos en la siguiente. Como está vacía el elemento es insertado. A esta altura la tabla está como se muestra en la figura 4.3 a la izquierda. Después de aplicar el procedimiento a todos los elementos a insertar, hasta el 17, la tabla queda como se muestra en la misma figura, a la derecha. Al querer insertar el 19 vemos que todas las cubetas están ocupadas, de manera que se debe señalar un error (o agrandar la tabla dinámicamente).

Ahora consideremos que ocurre al hacer **p=find(x)**. Por supuesto no basta con buscar en la cubeta  $h(x)$  ya que si al momento de insertar esa cubeta estaba llena, entonces el algoritmo de inserción lo insertó en la siguiente cubeta vacía, es decir que  $x$  puede estar en otra cubeta. Sin embargo, no hace falta revisar todas las cubetas, si revisamos las cubetas a partir de  $h(x)$ , es decir la  $h(x) + 1, h(x) + 2, \dots$  entonces cuando lleguemos a alguna cubeta que contiene a  $x$  podemos devolver el iterator correspondiente. Pero también si llegamos a un **undef**, sabemos que el elemento “*no está*” en el diccionario, ya que si estuviera debería haber sido insertado en alguna cubeta entre  $h(x)$  y el siguiente **undef**. De manera que al encontrar el primer **undef** podemos retornar **end()**.

Hasta ahora no hemos discutido como hacer las supresiones. Primero discutiremos en las secciones siguientes el costo de las operaciones de inserción y búsqueda, luego en la sección §4.5.4.4).

#### 4.5.4.1. Costo de la inserción exitosa

Definimos la “*tasa de ocupación*”  $\alpha$  como

$$\alpha = \frac{n}{B} \quad (4.19)$$

El costo de insertar un nuevo elemento (una “*inserción exitosa*”) es proporcional al número  $m$  de cubetas ocupadas que hay que recorrer hasta encontrar una cubeta libre. Cuando  $\alpha \ll 1$ , (muy pocas cubetas ocupadas) la probabilidad de encontrar una cubeta libre es grande y el costo de inserción es  $O(1)$ . A medida que la tabla se va llenando la probabilidad de encontrar una serie de cubetas ocupadas es alta y el costo de inserción ya no es  $O(1)$ .

Consideremos la cantidad de cubetas ocupadas que debemos recorrer hasta encontrar una cubeta libre en una tabla como la de la figura 4.3 a la izquierda. Consideremos que la probabilidad de que una dada cubeta sea la cubeta inicial es la misma para todas las cubetas. Si la cubeta inicial es una vacía (como las 0,2,6,7,8,9) entonces no hay que recorrer ninguna cubeta ocupada, es decir  $m = 0$ . Si queremos insertar en la cubeta 1, entonces ésta está ocupada, pero la siguiente está vacía, con lo cual debemos recorrer  $m = 1$  cubetas. El peor caso es al querer insertar en una cubeta como la 3, para la cual hay que recorrer  $m = 3$  cubetas antes de encontrar la siguiente vacía, que es la 6. Para las cubetas 4 y 5 tenemos  $m = 2$  y 1 respectivamente. El  $m$  promedio (que denotaremos por  $\langle m \rangle$ ) es la suma de los  $m$  para cada cubeta dividido el número de cubetas. En este caso es  $(1 + 3 + 2 + 1)/10 = 0.7$ . Notar que en este caso en particular es diferente de  $\alpha = n/B$  debido a que hay secuencias de cubetas ocupadas contiguas.

Si todas las cubetas tienen la misma probabilidad de estar ocupadas  $\alpha$ , entonces la probabilidad de que al insertar un elemento ésta este libre ( $m = 0$  intentos) es  $P(0) = 1 - \alpha$ . Para que tengamos que hacer un sólo intento debe ocurrir que la primera esté llena y la siguiente vacía. La probabilidad de que esto ocurra es

| Nro. de intentos infructuosos ( $m$ ) | Probabilidad de ocurrencia<br>$P(m) = \alpha^m(1 - \alpha)$ |
|---------------------------------------|-------------------------------------------------------------|
| 0                                     | 0.250000                                                    |
| 1                                     | 0.187500                                                    |
| 2                                     | 0.140625                                                    |
| 3                                     | 0.105469                                                    |
| 4                                     | 0.079102                                                    |
| 5                                     | 0.059326                                                    |
| 6                                     | 0.044495                                                    |
| 7                                     | 0.033371                                                    |
| 8                                     | 0.025028                                                    |
| 9                                     | 0.018771                                                    |
| 10                                    | 0.014078                                                    |

Tabla 4.5: Probabilidad de realizar  $m$  intentos en una tabla cerrada cuando la tasa de ocupación es  $\alpha = 0.75$

$P(1) = \alpha(1 - \alpha)$ . En realidad aquí hay una aproximación. Supongamos que  $B = 100$  de las cuales 75 están ocupadas. Si el primer intento da con una cubeta ocupada, entonces la probabilidad de que la segunda esté libre es un poco mayor que  $25/100 = (1 - \alpha)$  ya que sabemos que en realidad de las 99 cubetas restantes hay 25 libres, de manera que la probabilidad de encontrar una libre es en realidad  $25/99 \approx 0.253 > 0.25$ . Por simplicidad asumiremos que la aproximación es válida. Para dos intentos, la probabilidad es  $\alpha^2(1 - \alpha)$  y así siguiendo, la probabilidad de que haya que hacer  $m$  intentos es

$$P(m) = \alpha^m(1 - \alpha) \quad (4.20)$$

Por ejemplo, si la tasa de ocupación es  $\alpha = 0.75$ , entonces la probabilidad de tener que hacer  $m$  intentos es como se muestra en la Tabla 4.5. La cantidad de intentos en promedio será entonces

$$\begin{aligned} \langle m \rangle &= \sum_{m=0}^{B-1} m P(m) \\ &= \sum_{m=0}^{B-1} m \alpha^m (1 - \alpha) \end{aligned} \quad (4.21)$$

Suponiendo que  $B$  es relativamente grande y  $\alpha$  no está demasiado cerca de uno, podemos reemplazar la suma por una suma hasta  $m = \infty$ , ya que los términos decrecen muy fuertemente al crecer  $m$ . Para hacer la suma necesitamos hacer un truco matemático, para llevar la serie de la forma  $m\alpha^m$  a una geométrica simple, de la forma  $\alpha^m$

$$m\alpha^m = \alpha \frac{d}{d\alpha} \alpha^m \quad (4.22)$$

de manera que

$$\begin{aligned}
 \langle m \rangle &= \sum_{m=0}^{\infty} (1-\alpha) \alpha \frac{d}{d\alpha} \alpha^m \\
 &= \alpha (1-\alpha) \frac{d}{d\alpha} \left( \sum_{k=0}^{\infty} \alpha^m \right) \\
 &= \alpha (1-\alpha) \frac{d}{d\alpha} \left( \frac{1}{1-\alpha} \right) \\
 &= \frac{\alpha}{1-\alpha}
 \end{aligned} \tag{4.23}$$

Por ejemplo, en el caso de tener  $B = 100$  cubetas y  $\alpha = 0.9$  (90 % de cubetas ocupadas) el número de intentos medio es de  $\langle m \rangle = 0.9/(1-0.9) = 9$ .

#### 4.5.4.2. Costo de la inserción no exitosa

Llamaremos una “*inserción no exitosa*” cuando al insertar un elemento este ya está en el diccionario y por lo tanto en realidad no hay que insertarlo. El costo en este caso es menor ya que no necesariamente hay que llegar hasta una cubeta vacía, el elemento puede estar en cualquiera de las cubetas ocupadas. El número de intentos también depende de en qué momento fue insertado el elemento. Si fue insertado en los primeros momentos, cuando la tabla estaba vacía ( $\alpha$  pequeños), entonces es menos probable que el elemento esté en el segmento final de largas secuencias de cubetas llenas. Por ejemplo en el caso de la tabla en la figura 4.3 a la izquierda, el elemento 4 que fue insertado al principio necesita  $m = 0$  intentos infructuosos (es encontrado de inmediato), mientras que el 24, que fue insertado después, necesitará  $m = 1$ .

Si la tabla tiene una tasa de ocupación  $\alpha$ , entonces un elemento  $x$  puede haber sido insertado en cualquier momento previo en el cual la tasa era  $\alpha'$ , con  $0 \leq \alpha' \leq \alpha$ . Si fue insertado al principio ( $\alpha' \approx 0$ ) entonces habrá que hacer pocos intentos infructuosos para encontrarlo, si fue insertado recientemente ( $\alpha' \approx \alpha$ ) entonces habrá que hacer el mismo número promedio de intentos infructuosos que el que hay que hacer ahora para insertar un elemento nuevo, es decir  $\alpha/(1-\alpha)$ . Si asumimos que el elemento pudo haber sido insertado en cualquier momento cuando  $0 \leq \alpha' \leq \alpha$ , entonces el número de intentos infructuosos promedio será

$$\langle m_{n.e.} \rangle = \frac{1}{\alpha} \int_{\alpha'=0}^{\alpha} \langle m \rangle d\alpha' \tag{4.24}$$

Reemplazando  $\langle m \rangle$  de (4.23), tenemos que

$$\begin{aligned}
 \langle m_{n.e.} \rangle &= \frac{1}{\alpha} \int_{\alpha'=0}^{\alpha} \frac{\alpha'}{1-\alpha'} d\alpha' \\
 &= \frac{1}{\alpha} \int_{\alpha'=0}^{\alpha} \left( \frac{1}{1-\alpha'} - 1 \right) d\alpha' \\
 &= \frac{1}{\alpha} \left( -\log(1-\alpha') - \alpha' \right) \Big|_{\alpha'=0}^{\alpha} \\
 &= -\frac{1}{\alpha} \log(1-\alpha) - 1
 \end{aligned} \tag{4.25}$$

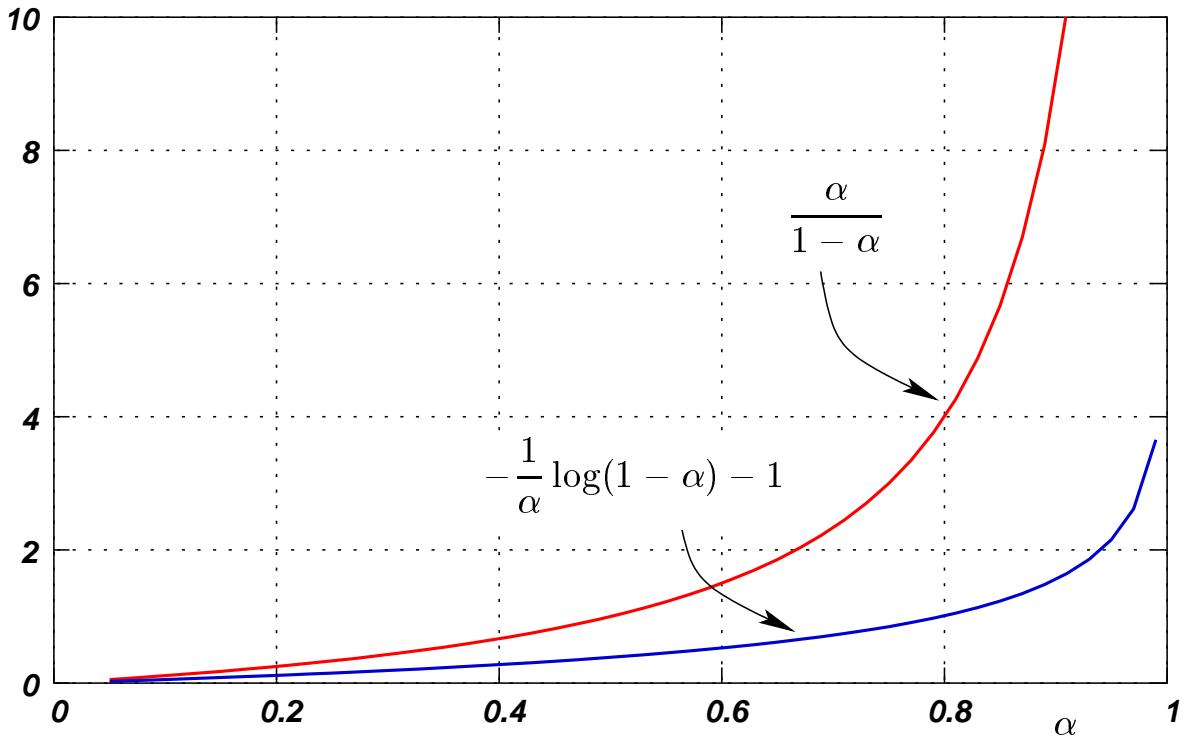


Figura 4.4: Eficiencia de las tablas de dispersión.

#### 4.5.4.3. Costo de la búsqueda

Ahora analicemos el costo de `p=find(k)`. El costo es  $O(1 + \langle m \rangle)$ , donde  $\langle m \rangle$  es el número de intentos infructuosos. Si el elemento no está en el diccionario (*búsqueda no exitosa*), entonces el número de intentos infructuosos es igual al de intentos infructuosos para inserción exitosa analizado en la sección §4.5.4.1 dado por (4.23). Por otra parte, si el elemento está en el diccionario (*búsqueda exitosa*), el número de intentos infructuosos es sensiblemente menor, ya que si el elemento fue insertado al comienzo, cuando la tabla estaba vacía es probable que esté en las primeras cubetas, bien cerca de  $h(x)$ . El análisis es similar al de la inserción no exitosa analizado en la sección §4.5.4.2, dado por (4.25).

#### 4.5.4.4. Supresión de elementos

Al eliminar un elemento uno estaría tentado de reemplazar el elemento por un `undef`. Sin embargo, esto dificultaría las posibles búsquedas futuras ya que ya no sería posible detenerse al encontrar un `undef` para determinar que el elemento no está en la tabla. La solución es introducir otro elemento `deleted` (“eliminado”) que marcará posiciones donde previamente hubo alguna vez un elemento que fue eliminado. Por ejemplo, para enteros positivos podríamos usar `undef=0` y `deleted=-1`. Ahora, al hacer `p=find(x)` debemos recorrer las cubetas siguientes a  $h(x)$  hasta encontrar  $x$  o un elemento `undef`. Los elementos `deleted` son tratados como una cubeta ocupada más. Sin embargo, al hacer un `insert(x)` de un nuevo elemento, podemos insertarlo en posiciones `deleted` además de `undef`.

#### 4.5.4.5. Costo de las funciones cuando hay supresión

El análisis de los tiempos de ejecución es similar al caso cuando no hay supresiones, pero ahora la tasa de ocupación debe incluir a los elementos **deleted**, es decir en base a la “*tasa de ocupación efectiva*”  $\alpha'$  dada por

$$\alpha' = \frac{n + n_{\text{del}}}{B} \quad (4.26)$$

donde ahora  $n_{\text{del}}$  es el número de elementos **deleted** en la tabla.

Esto puede tener un impacto muy negativo en la eficiencia, si se realizan un número de inserciones y supresiones elevado. Supongamos una tabla con  $B = 100$  cubetas en la cual se insertan 50 elementos distintos, y a partir de allí se ejecuta un lazo infinito en el cual se inserta un nuevo elemento al azar y se elimina otro del conjunto, también al azar. Después de cada ejecución del cuerpo del lazo el número de elementos se mantiene en  $n = 50$  ya que se inserta y elimina un elemento. La tabla nunca se llena, pero el número de suprimidos  $n_{\text{del}}$  crece hasta que eventualmente llega a ser igual  $B - n$ , es decir todas las cubetas están ocupadas o bien tienen **deleted**. En ese caso la eficiencia se degrada totalmente, cada operación (de hecho cada **locate()**) recorre toda la tabla, ya que en ningún momento encuentra un **undef**. De hecho, esto ocurre en forma relativamente rápida, en  $B - n$  supresiones/inserciones.

#### 4.5.4.6. Reinscripción de la tabla

Si el destino de la tabla es para insertar una cierta cantidad de elementos y luego realizar muchas consultas, pero pocas inserciones/supresiones, entonces el esquema presentado hasta aquí es razonable. Por el contrario, si el ciclo de inserción supresión sobre la tabla va a ser continuo, el incremento en los el número de elementos **deleted** causará tal deterioro en la eficiencia que no permitirá un uso práctico de la tabla. Una forma de corregir esto es “reinsertar” los elementos, es decir, se extraen todos los elementos guardándolos en un contenedor auxiliar (vector, lista, etc...) limpiando la tabla, y reinsertando todos los elementos del contenedor. Como inicialmente la nueva tabla tendrá todos los elementos **undef**, y las inserciones no generan elementos **deleted**, la tabla reinsertada estará libre de **deleted**’s. Esta tarea es  $O(B + n)$  y se ve compensada por el tiempo que se ahorrará en unas pocas operaciones.

Para determinar cuando se dispara la reinscripción se controla la tasa de suprimidos que existe actualmente

$$\beta = \frac{n_{\text{del}}}{B} \quad (4.27)$$

Cuando  $\beta \approx 1 - \alpha$  quiere decir que de la fracción de cubetas no ocupadas  $1 - \alpha$ , una gran cantidad de ellas dada por la fracción  $\beta$  está ocupada por suprimidos, degradando la eficiencia de la tabla. Por ejemplo, si  $\alpha = 0.5$  y  $\beta = 0.45$  entonces 50 % de las cubetas está ocupada, y del restante 50 % el 45 % está ocupado por **deleted**. En esta situación la eficiencia de la tabla es equivalente una con un 95 % ocupado.

En la reinscripción continua, cada vez que hacemos un borrado hacemos una serie de operaciones para dejar la tabla sin ningún **deleted**. Recordemos que al eliminar un elemento no podemos directamente insertar un **undef** en su lugar ya que de hacer el algoritmo de búsqueda no encontraría a los elementos que están después del elemento insertado y hasta el siguiente **undef**. Por ejemplo, si consideramos la tabla de la figura 4.5 a la izquierda, entonces si queremos eliminar el elemento 4, no podemos simplemente insertar insertar un **undef** ya que no encontraremos después el 24. Pero, si quisieramos eliminar un elemento como el 24 entonces sí podemos insertar allí un **undef**, ya que no queda ningún elemento entre la cubeta ocupada por el 24 y el siguiente **undef**. Ahora volvamos al caso en que queremos eliminar el 4. Podemos eliminar

| $S=\{\}$        | $S=\{24\}$           | $S=\{24\}$           | $S=\{24\}$           |
|-----------------|----------------------|----------------------|----------------------|
| <i>bucket=0</i> | <i>0</i>             | <i>0</i>             | <i>0</i>             |
| <b>1</b>        | <b>1</b>             | <b>1</b>             | <b>1</b>             |
| <b>2</b>        | <b>&lt;undef&gt;</b> | <b>&lt;undef&gt;</b> | <b>&lt;undef&gt;</b> |
| <b>3</b>        | <b>13</b>            | <b>13</b>            | <b>13</b>            |
| <b>4</b>        | <b>4</b>             | <b>4</b>             | <b>24</b>            |
| <b>5</b>        | <b>24</b>            | <b>&lt;undef&gt;</b> | <b>&lt;undef&gt;</b> |
| <b>6</b>        | <b>&lt;undef&gt;</b> | <b>&lt;undef&gt;</b> | <b>&lt;undef&gt;</b> |
| <b>7</b>        | <b>&lt;undef&gt;</b> | <b>&lt;undef&gt;</b> | <b>&lt;undef&gt;</b> |
| <b>8</b>        | <b>&lt;undef&gt;</b> | <b>&lt;undef&gt;</b> | <b>&lt;undef&gt;</b> |
| <b>9</b>        | <b>&lt;undef&gt;</b> | <b>&lt;undef&gt;</b> | <b>&lt;undef&gt;</b> |

Figura 4.5: Proceso de borrado con redispersión continua.

temporariamente el 24, guardándolo en una pila auxiliar  $S$  (puede ser cualquier otro contenedor como lista, vector o cola), y reemplazándolo por un **undef**, quedando en el estado mostrado en la segunda tabla desde la izquierda. De esta forma ahora el 4 está justo antes del **undef** y sí podemos reemplazarlo por un **undef**, quedando como la tercera tabla desde la izquierda. Ahora, finalmente, debemos reinsertar todos los elementos que están en  $S$  (en este caso sólo el 24), de nuevo en la tabla. En el caso más general, deberíamos guardar en el contenedor auxiliar todos los elementos que están entre la cubeta a suprimir y el siguiente (en sentido circular) **undef**. Es claro que de esta forma no tenemos en ningún momento elementos **deleted** en la tabla, ya que al eliminar el elemento nos aseguramos de que así sea. Las inserciones siguientes seguramente no generan elementos **deleted** (ninguna inserción los puede generar).

#### 4.5.4.7. Costo de las operaciones con supresión

Si se usa reinserción continua, entonces no hay en ningún momento elementos **deleted** y el número de intentos infructuosos es  $\langle m \rangle = \alpha / (1 - \alpha)$ . Las operaciones **find(x)** e **insert(x)** son ambas  $O(1 + \langle m \rangle)$ , pero **erase(x)** es  $O(1 + \langle m \rangle^2)$  ya que al reinsertar los elementos hay en promedio  $\langle m \rangle$  llamadas a **insert()** el cuál es a su vez  $O(1 + \langle m \rangle)$ .

En la reinserción discontinua, el costo de las tres operaciones es  $O(1 + \langle m \rangle)$ , pero  $\langle m \rangle = \alpha' / (1 - \alpha')$ , es decir que la tasa de ocupación efectiva crece con el número de elementos **deleted**.

#### 4.5.4.8. Estrategias de redispersión

Al introducir las tablas de dispersión cerrada (sección §4.5.4) hemos explicado como la redispersión permite resolver las colisiones, buscando en las cubetas  $h(x) + j$ , para  $j = 0, 1, \dots, B - 1$ , hasta encontrar una cubeta libre. A esta estrategia se le llama “*de redispersión lineal*”. Podemos pensar que si la función de dispersión no es del todo buena, entonces ciertas cubetas o ciertas secuencias de cubetas cercanas pueden tender a llenarse más que otras. Esto puede ocasionar que tiendan a formarse localmente secuencias de cubetas ocupadas, incluso cuando la tasa de ocupación no es tan elevada globalmente. En estos casos puede ayudar el tratar de no dispersar las cubetas en forma lineal sino de la forma  $h(x) + d_j$ , donde  $d_0$  es 0 y  $d_j, j = 1, \dots, B - 1$  es una permutación de los números de 1 a  $B - 1$ . Por ejemplo, si  $B = 8$ , podríamos tomar alguna permutación aleatoria como  $d_j = 7, 2, 5, 4, 1, 0, 3, 6$ . La forma de generar estas permutaciones es similar a la generación de números aleatorios, sin embargo está claro que no son números aleatorios, ya que la secuencia  $d_j$  debe ser la misma durante todo el uso de la tabla. Los elementos a tener en cuenta para elegir las posibles funciones de redispersión son las siguientes

- Debe ser determinística, es decir  $d_j$  debe ser sólo función de  $j$  durante todo el uso de la tabla.
- Debe ser  $d_0 = 0$  y los demás elementos  $d_1$  a  $d_{B-1}$  deben ser una permutación de los enteros 1 a  $B - 1$ , por ejemplo  $d_j = \text{rem}(mj, B)$  es válida si  $m$  y  $B$  son “coprimos” (no tienen factores en común). Por ejemplo si  $B = 8$  entonces  $m = 6$  no es válida ya que  $m$  y  $B$  tienen el factor 2 en común y la secuencia generada es  $d_j = \{0, 6, 4, 2, 0, 6, 4, 2\}$ , en la cual se generan sólo los números pares. Sin embargo  $m = 9$  está bien, ya que es coprimo con  $B$  y genera la secuencia  $d_j = \{0, 7, 6, 5, 4, 3, 2, 1\}$ .

Una posibilidad es generar una permutación aleatoria de los números de 0 a  $B - 1$  y guardarla en un vector, como un miembro estático de la clase. De esta forma habría que almacenar un sólo juego de  $d_j$  para todas las tablas. Esto serviría si tuviéramos muchas tablas relativamente pequeñas de dispersión pero no serviría si tuviéramos una sola tabla grande ya que en ese caso el tamaño de almacenamiento para los  $d_j$  sería comparable con el de la tabla misma.

Por otra parte, si los  $d_j$  se calculan en tiempo de ejecución cada vez que se va a aplicar una de las funciones **find()**, **insert()** o **erase()** entonces el costo de calcular cada uno de los  $d_j$  es importante ya que para aplicar una sola de esas funciones (es decir un sólo **insert()** o **erase()**) puede involucrar varias evaluaciones de  $d_j$  (tantos como intentos infructuosos).

Una posible estrategia para generar los  $d_j$  es la siguiente. Asumamos que  $B$  es una potencia de 2 y sea  $k$  un número entre 0 y  $B - 1$  a ser definido más adelante. Se comienza con un cierto valor para  $d_1$  y los  $d_{j+1}$  se calculan en términos del número anterior  $d_j$  con el algoritmo siguiente

$$d_{j+1} = \begin{cases} 2d_j < B & ; 2d_j \\ 2d_j \geq B & ; (2d_j - B) \oplus k \end{cases} \quad (4.28)$$

En esta expresión el operador  $\oplus$  indica el operador “*o exclusivo bit a bit*”. Consiste en escribir la expresión de cada uno de los argumentos y aplicarles el operador “*xor*” bit a bit. Por ejemplo, si queremos calcular  $25_{10} \oplus 13_{10}$  primero calculamos las expresiones binarias  $25_{10} = 11001_2$  y  $13_{10} = 1101_2$ . El resultado es entonces

$$\begin{aligned} 25_{10} &= 11001_2 \\ 13_{10} &= 01101_2 \\ 25_{10} \oplus 13_{10} &= 10100_2 \end{aligned} \quad (4.29)$$

---

```

1 int B=8, k=3, d=5;
2 for (int j=2; j<B; j++) {
3 int v = 2*d;
4 d = (v<B ? v : (v-B)^k);
5 }
```

**Código 4.12:** Generación de los índices de redispersión  $d_j$  [Archivo: *redisp.cpp*]

Esta operación puede ser implementada en forma eficiente en C++ usando el operador “*bitwise xor*” denotado por  $\wedge$ . El fragmento de código genera los  $d_j$  para el caso  $B = 8$ , tomando  $k = 3$  y  $d_1 = 5$ . En este caso los códigos generados son  $d_j = \{0, 5, 1, 2, 4, 3, 6, 7\}$ . Sin embargo, no está garantizado que cualquier combinación  $k$  y  $d_1$  genere una permutación válida, es decir, para algunas combinaciones puede ser que se generen elementos repetidos. En realidad el valor importante es  $k$ , si una dada combinación  $k, d_1$  es válida, entonces ese  $k$  será válido con cualquier otro valor de  $d_1$  en el rango  $[1, B)$ . No existe una forma sencilla de predecir para un dado  $B$  cuál es un  $k$  válido. Pero, asumiendo que el tamaño de las tablas será a lo sumo de  $2^{30} \approx 10^9$  elementos (si fueran enteros de 4bytes, una tal tabla ocuparía unos 4 GByte de memoria), podemos calcular previamente los  $k(p)$  apropiados para cada  $B = 2^p$ .

#### 4.5.4.9. Detalles de implementación

---

```

1 typedef int iterator_t;
2 typedef int (*hash_fun)(key_t x);
3 typedef int (*redisp_fun)(int j);
4
5 int linear_redisp_fun(int j) { return j; }
6
7 class hash_set {
8 private:
9 hash_set(const hash_set&){{}}
10 hash_set& operator=(const hash_set&){{}}
11 int undef, deleted;
12 hash_fun h;
13 redisp_fun rdf;
14 int B;
15 int count;
16 std::vector<key_t> v;
17 std::stack<key_t> S;
18 iterator_t locate(key_t x, iterator_t &fdel) {
19 int init = h(x);
20 int bucket;
21 bool not_found = true;
22 for (int i=0; i<B; i++) {
23 bucket = (init+rdf(i))% B;
24 key_t vb = v[bucket];
25 if (vb==x || vb==undef) break;
26 if (not_found && vb==deleted) {
```

```
27 fdel=bucket;
28 not_found = false;
29 }
30 }
31 if (not_found) fdel = end();
32 return bucket;
33 }
34 iterator_t next_aux(iterator_t bucket) {
35 int j=bucket;
36 while(j!=B && (v[j]==undef || v[j]==deleted)) {
37 j++;
38 }
39 return j;
40 }
41 public:
42 hash_set(int B_a,hash_fun h_a,
43 key_t undef_a,key_t deleted_a,
44 redisp_fun rdf_a=&linear_redisp_fun)
45 : B(B_a), undef(undef_a), v(B,undef_a), h(h_a),
46 deleted(deleted_a), rdf(rdf_a), count(0)
47 { }
48 std::pair<iterator_t, bool>
49 insert(key_t x) {
50 iterator_t fdel;
51 int bucket = locate(x,fdel);
52 if (v[bucket]==x)
53 return std::pair<iterator_t,bool>(bucket, false);
54 if (fdel!=end()) bucket = fdel;
55 if (v[bucket]==undef || v[bucket]==deleted) {
56 v[bucket]=x;
57 count++;
58 return std::pair<iterator_t,bool>(bucket, true);
59 } else {
60 std::cout << "Tabla de dispersion llena!!\n";
61 abort();
62 }
63 }
64 key_t retrieve(iterator_t p) { return v[p]; }
65 iterator_t find(key_t x) {
66 iterator_t fdel;
67 int bucket = locate(x,fdel);
68 if (v[bucket]==x) return bucket;
69 else return(end());
70 }
71 int erase(const key_t& x) {
72 iterator_t fdel;
73 int bucket = locate(x,fdel);
74 if (v[bucket]==x) {
75 v[bucket]=deleted;
76 count--;
77 // Trata de purgar elementos 'deleted'
78 // Busca el siguiente elemento 'undef'
```

```

79 int j;
80 for (j=1; j<B; j++) {
81 op_count++;
82 int b = (bucket+j) % B;
83 key_t vb = v[b];
84 if (vb==undef) break;
85 S.push(vb);
86 v[b]=undef;
87 count--;
88 }
89 v[bucket]=undef;
90 // Va haciendo erase/insert de los elementos
91 // de atras hacia adelante hasta que se llene
92 // 'bucket'
93 while (!S.empty()) {
94 op_count++;
95 insert(S.top());
96 S.pop();
97 }
98 return 1;
99 } else return 0;
100 }
101 iterator_t begin() {
102 return next_aux(0);
103 }
104 iterator_t end() { return B; }
105 iterator_t next(iterator_t p) {
106 return next_aux(p++);
107 }
108 void clear() {
109 count=0;
110 for (int j=0; j<B; j++) v[j]=undef;
111 }
112 int size() { return count; }
113 };

```

**Código 4.13:** Implementación de diccionario con tablas de dispersión cerrada y redispersión continua. [Archivo: *hashsetbash.h*]

En el código 4.13 se puede ver una posible implementación del TAD diccionario con tablas de dispersión cerrada y redispersión continua.

- El **typedef hash\_fun** define el tipo de funciones que pueden usarse como funciones de dispersión (toman como argumento un elemento de tipo **key\_t** y devuelven un entero). La función a ser usada es pasada en el constructor y guardada en un miembro **hash\_fun h**.
- El **typedef redisp\_fun** define el tipo de funciones que pueden usarse para la redispersión (los  $d_j$ ). Es una función que debe tomar como argumento un entero  $j$  y devolver otro entero (el  $d_j$ ). La función a ser usada es pasada por el usuario en el constructor y guardada en un miembro (**redisp\_fun rdf**). Por defecto se usa la función de redispersión lineal (**linear\_redisp\_fun()**).

- El miembro dato **int B** es el número de cubetas a ser usada. Es definido en el constructor.
- El miembro dato **int count** va a contener el número de elementos en la tabla (será retornado por **size()**).
- Las  $B$  cubetas son almacenadas en un **vector<key\_t> v**.
- Casi todos los métodos de la clase usan la función auxiliar **iterator\_t locate(key\_t x, iterator\_t &fde1)**. Esta función retorna un iterator de acuerdo a las siguiente reglas:
  - Si **x** está en la tabla **locate()** retorna la cubeta donde está almacenado el elemento **x**.
  - Si **x** no está en la tabla, retorna el primer **undef** después (en sentido circular) de la posición correspondiente a **x**.
  - Si no hay ningún **undef** (pero puede haber **deleted**) retorna alguna posición no especificada en la tabla.
  - Retorna por el argumento **fde1** la posición del primer **deleted** entre la cubeta correspondiente a **x** y el primer **undef**. Si no existe ningún **deleted** entonces retorna **end()**.
- La pila **S** es usada como contenedor auxiliar para la estrategia de reinserción. La reinserción se realiza en el bloque de las líneas 78–98
- La clase iterator consiste simplemente de un número de cubeta. **end()** es la cubeta ficticia **B**. La función **q=next(p)** que avanza iteradores, debe avanzar sólo sobre **cubetas ocupadas**. La función **next\_aux()** avanza un iterador (en principio inválido) hasta llegar a uno ocupado o a **end()**. La función **q=next(p)** simplemente incrementa **p** y luego aplica **next\_aux()**.
- La función **begin()** debe retornar el primer iterator válido (la primera cubeta ocupada) o bien **end()**. Para ello calcula el **next\_aux(0)** es decir la primera cubeta válida después de (o igual a) la cubeta 0.

## 4.6. Conjuntos con árboles binarios de búsqueda

Una forma muy eficiente de representar conjuntos son los árboles binarios de búsqueda (ABB). Un árbol binario es un ABB si es vacío ( $\Lambda$ ) o:

- Todos los elementos en los nodos del subárbol izquierdo son menores que el nodo raíz.
- Todos los elementos en los nodos del subárbol derecho son mayores que el nodo raíz.
- Los subárboles del hijo derecho e izquierdo son a su vez ABB.

### 4.6.1. Representación como lista ordenada de los valores

En la figura 4.6 vemos un posible árbol binario de búsqueda. La condición de ABB implica que si ordenamos todos los valores nodales de menor a mayor, quedan de la siguiente forma

$$\text{valores nodales ordenados} = \{(rama izquierda), r, (rama derecha)\} \quad (4.30)$$

donde  $r$  es el valor nodal de la raíz. Por ejemplo, si ordenamos los valores nodales del ejemplo obtenemos la lista  $\{5, 7, 10, 12, 14, 15, 18\}$  donde vemos que los valores de la rama izquierda 5 y 7 quedan a la izquierda de la raíz  $r = 10$  y los valores de la rama derecha 12, 14, 15 y 18 quedan a la izquierda. Este ordenamiento es válido también para subárboles. Si  $n$  es el subárbol del nodo  $n$ , entonces todos los elementos del subárbol aparecen contiguos en la lista ordenada global y los valores nodales del subárbol guardan entre sí la relación

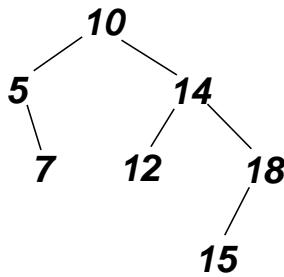


Figura 4.6: Ejemplos de árboles binarios de búsqueda

indicada en (4.30). En el ejemplo, si consideramos el subárbol de 14, entonces vemos que todos los valores del subárbol aparecen juntos (en este caso al final) y entre sí respetan (4.30), a saber primero 12 (la rama izquierda), 14 (la raíz del subárbol) y 15,18 (la rama derecha).

#### 4.6.2. Verificar la condición de ABB

```

1 bool abb_p(aed::btree<int> &T,
2 aed::btree<int>::iterator n, int &min, int &max) {
3 aed::btree<int>::iterator l,r;
4 int minr,maxr,minl,maxl;
5 min = +INT_MAX;
6 max = -INT_MAX;
7 if (n==T.end()) return true;
8
9 l = n.left();
10 r = n.right();
11
12 if (!abb_p(T,l,minl,maxl) || maxl>*n) return false;
13 if (!abb_p(T,r,minr,maxr) || minr<*n) return false;
14
15 min = (l==T.end() ? *n : minl);
16 max = (r==T.end() ? *n : maxr);
17 return true;
18 }
19
20 bool abb_p(aed::btree<int> &T) {
21 if (T.begin()==T.end()) return false;
22 int min,max;
23 return abb_p(T,T.begin(),min,max);
24 }
```

**Código 4.14:** Función predicado que determina si un dado árbol es ABB. [Archivo: abbp.cpp]

En el código 4.14 vemos una posible función predicado `bool abb_p(T)` que determina si un árbol binario es ABB o no. Para ello usa una función recursiva auxiliar `bool abb_p(T,n,min,max)` que determina si el

subárbol del nodo **n** es ABB y en caso de que si lo sea retorna a través de **min** y **max** los valores mínimos y máximos del árbol. Si no es ABB los valores de **min** y **max** están indeterminados. La función recursiva se aplica a cada uno de los hijos derechos e izquierdo, en caso de que alguno de estos retorne **false** la función retorna inmediatamente **false**. Lo mismo ocurre si el máximo del subárbol del hijo izquierdo **maxl** es mayor que el valor en el nodo **\*n**, o el mínimo del subárbol del hijo derecho es menor que **\*n**.

Por supuesto, si el nodo **n** está vacío (es **end()**) **abb\_p()** no falla y retorna un valor mínimo de **+INT\_MAX** y un valor máximo de **-INT\_MAX**. (**INT\_MAX** es el máximo entero representable y está definido en el header **float.h**). Estos valores garantizan que las condiciones de las líneas 12 y 13 no fallen cuando alguno de los hijos es  $\Lambda$ .

Si todas las condiciones se satisfacen entonces el mínimo de todo el subárbol de **n** es el mínimo del subárbol izquierdo, es decir **minl**, salvo si el nodo izquierdo es **end()** en cuyo caso el mínimo es el valor de **\*n**. Igualmente, el máximo del subárbol de **n** es **maxr** si el nodo izquierdo no es **end()** y **\*n** si lo es.

#### 4.6.3. Mínimo y máximo

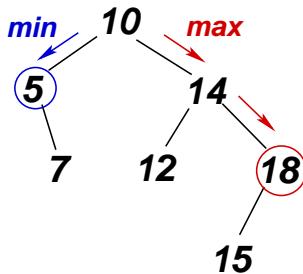


Figura 4.7:

Buscar los mínimos y máximos se puede hacer gráficamente en forma muy simple (ver figura 4.7).

- Para buscar el máximo se debe avanzar siempre por la derecha. Para encontrar el mínimo siempre por la izquierda.
- El listado en orden simétrico da la lista ordenada de los elementos del árbol.

#### 4.6.4. Buscar un elemento

```

1 node_t find(tree_t t, node_t n, T x) {
2 if (n==t.end()) return t.end();
3 if (x<*n) return find(t, n.left(), x)
4 elseif (x>*n) return find(t, n.right(), x)
5 else return n;
6 }

```

La función previa permite encontrar un elemento en el árbol. Retorna la posición (nodo) donde el elemento está o retornar **end()** si el elemento no está.

#### 4.6.5. Costo de mínimo y máximo

Si el árbol esta bien balanceado puede almacenar una gran cantidad de elementos en pocos niveles. Como las funciones descriptas previamente, (mínimo, máximo y buscar un elemento) siguen un camino en el árbol es de esperar que sean muy eficientes. Un “árbol binario completo” es un árbol que tiene todos sus niveles completamente ocupados, hasta un cierto nivel  $d$ . El árbol completo es el mejor caso en cuanto a balanceo de l árbol. Calculemos cuantos nodos tiene un árbol completo. En el primer nivel  $l = 0$  sólo está la raíz, es decir 1 nodo. En el nivel  $l = 1$  2 nodos y en general  $2^l$  nodos en el nivel  $l$ . Es decir que un árbol completo de altura  $d$  tiene

$$n = 1 + 2 + 2^2 + \dots + 2^d = 2^{d+1} - 1 \quad (4.31)$$

nodos, y por lo tanto el número de niveles, en función del número de nodos es (ver sección §3.8.3)

$$d = \log_2(n + 1) - 1 \quad (4.32)$$

Las funciones que iteran sólo sobre un camino en el árbol, como **insert(x)** o **find(x)**, tienen un costo  $O(l)$  donde  $l$  es el nivel o profundidad del nodo desde la raíz (ver sección 3.1). En el caso del árbol completo todos los caminos tienen una longitud  $l \leq d$ , de manera que los tiempos de ejecución son  $O(d) = O(\log(n))$ .

En realidad, los nodos interiores van a tener una profundidad  $l < d$  y no puede preguntarse si esto puede hacer bajar el costo promedio de las operaciones. En realidad, como el número de nodos crece muy rápidamente, de hecho exponencialmente con el número de nivel, el número de nivel promedio es muy cercano a la profundidad máxima para un árbol completo. Consideremos un árbol completo con profundidad 12, es decir con  $n = 2^{13} - 1 = 8191$  nodos. En el último nivel hay  $2^{12} = 4096$  nodos, es decir más de la mitad. En el anteúltimo ( $l = 11$ ) nivel hay 2048 nodos (1/4 del total) y en el nivel  $l = 10$  hay 1024 (1/8 del total). Vemos que en los últimos 3 niveles hay más del 87 % de los nodos. La profundidad media de los nodos es

$$\langle l \rangle = \frac{1}{n} \sum_{\text{nodo } m} l(m) \quad (4.33)$$

donde  $m$  recorre los nodos en el árbol y  $l(m)$  es la profundidad del nodo  $m$ . Como todos los nodos en el mismo nivel tienen la misma profundidad, tenemos

$$\langle l \rangle = \frac{1}{n} \sum_{l=0}^d (\text{Nro. de nodos en el nivel } l) \cdot l \quad (4.34)$$

Para el árbol completo tenemos entonces

$$\langle l \rangle = \frac{1}{n} \sum_{l=0}^d 2^l l \quad (4.35)$$

Esta suma se puede calcular en forma cerrada usando un poco de álgebra. Notemos primero que  $2^l = e^{l \log 2}$ . Introducimos una variable auxiliar  $\alpha$  tal que

$$2^l = e^{\alpha l} \Big|_{\alpha=\log 2} \quad (4.36)$$

Lo bueno de introducir esta variable  $\alpha$  es que la derivada con respecto a  $\alpha$  de la exponencial baja un factor  $l$ , es decir

$$\frac{d}{d\alpha} e^{\alpha l} = l e^{\alpha l}. \quad (4.37)$$

Entonces

$$\begin{aligned} \langle l \rangle &= \frac{1}{n} \sum_{l=0}^d 2^l l \\ &= \frac{1}{n} \sum_{l=0}^d \left[ \frac{d e^{\alpha l}}{d\alpha} \right]_{\alpha=\log 2}, \\ &= \frac{1}{n} \frac{d}{d\alpha} \left[ \sum_{l=0}^d e^{\alpha l} \right]_{\alpha=\log 2}. \end{aligned} \quad (4.38)$$

Ahora la sumatoria es una suma geométrica simple de razón  $e^\alpha$ , y se puede calcular en forma cerrada

$$\sum_{l=0}^d e^{\alpha l} = \frac{e^{\alpha(d+1)} - 1}{e^\alpha - 1}. \quad (4.39)$$

Su derivada con respecto a  $\alpha$  es

$$\frac{d}{d\alpha} \left[ \frac{e^{\alpha(d+1)} - 1}{e^\alpha - 1} \right] = \frac{(d+1) e^{\alpha(d+1)} (e^\alpha - 1) - (e^{\alpha(d+1)} - 1) e^\alpha}{(e^\alpha - 1)^2} \quad (4.40)$$

y reemplazando en (4.38) y usando  $e^\alpha = 2$  obtenemos

$$\begin{aligned} \langle l \rangle &= \frac{1}{n} \frac{(d+1) e^{\alpha(d+1)} (e^\alpha - 1) - (e^{\alpha(d+1)} - 1) e^\alpha}{(e^\alpha - 1)^2} \\ &= \frac{(d+1) 2^{d+1} - (2^{d+1} - 1) 2}{2^{d+1} - 1} \\ &= \frac{(d-1) 2^{d+1} + 2}{2^{d+1} - 1} \\ &\approx d - 1 \end{aligned} \quad (4.41)$$

La última aproximación proviene de notar que  $2^{d+1} \gg 1$  para, digamos,  $d > 10$ .

Esto dice que en un árbol completo la longitud promedio de los caminos es la profundidad del árbol menos uno.

Por el contrario, el peor caso es cuando el árbol está completamente desbalanceado, es decir en cada nivel del árbol hay un sólo nodo y por lo tanto hay  $n$  niveles. En este caso  $\langle l \rangle = n/2$  y los costos de **insert()** y **find()** son  $O(n)$ .

Notar que el balanceo del árbol depende del orden en que son insertados los elementos en el árbol. En la figura 4.8 se muestran los árboles obtenidos al insertar los enteros del 1 al 7, primero en forma ascendente, después descendente y finalmente en el orden  $\{4, 2, 6, 1, 3, 5, 7\}$ . Vemos que en los dos primeros casos el desbalanceo es total, el árbol degenera en dos listas por hijo derecho en el primer caso y por hijo izquierdo

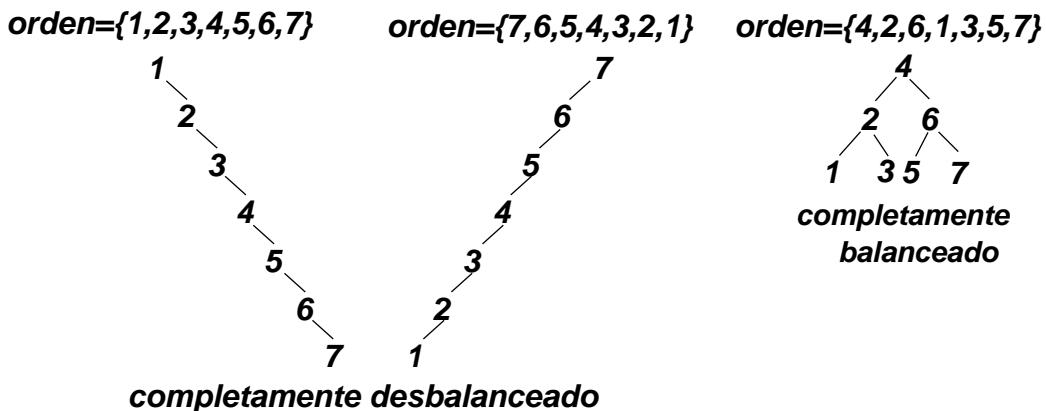


Figura 4.8: El balanceo del árbol depende del orden en que son ingresados los elementos.

en el segundo. En el tercer caso los elementos son ingresados en forma desordenada y el balanceo es el mejor posible. El árbol resultante es un arbol completo hasta el nivel 3.

Puede demostrarse que si los valores son insertados en forma aleatoria entonces

$$\langle m \rangle \approx 1.4 \log_2 n, \quad (4.42)$$

con lo cual el caso de inserción aleatoria está muy cercano al mejor caso.

#### 4.6.6. Operación de inserción

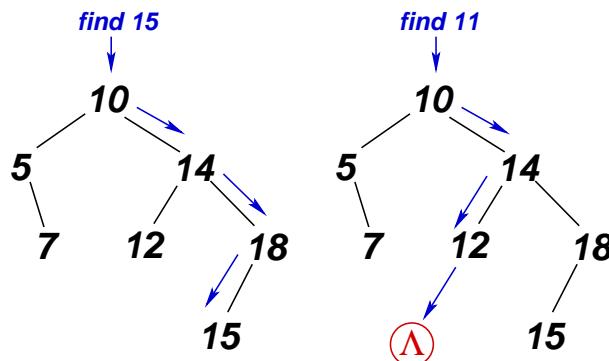


Figura 4.9:

Para insertar un elemento  $x$  hacemos un  $\text{find}(x)$ , si el nodo returned es  $\Lambda$  insertamos allí, si no el elemento ya está. Una vez ubicada la posición donde el elemento debe ser insertada, la inserción es  $O(1)$  de manera que toda la operación es básicamente la de  $\text{find}(x)$ :  $O(\langle m \rangle) = O(\log_2 n)$ .

#### 4.6.7. Operación de borrado

Para borrar hacemos de nuevo el `n=find(x)`. Si el elemento no está (“supresión no exitosa”), no hay que hacer nada. Si está hay varios casos, dependiendo del número de hijos del nodo que contiene el elemento a eliminar.

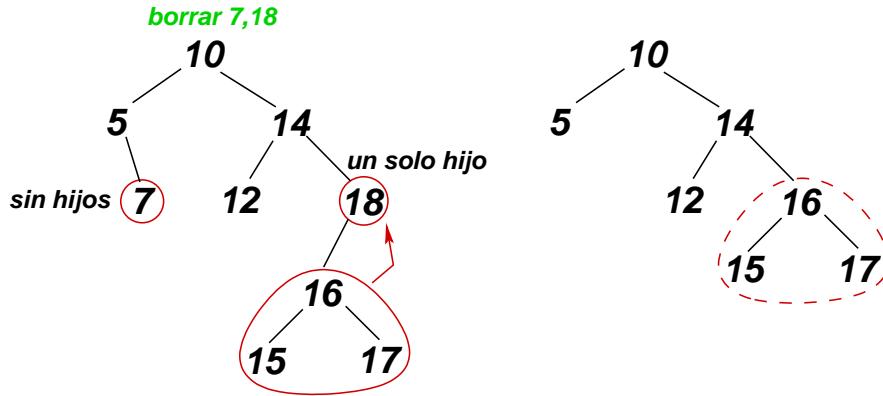


Figura 4.10: Supresión en ABB cuando el nodo no tiene o tiene un solo hijo.

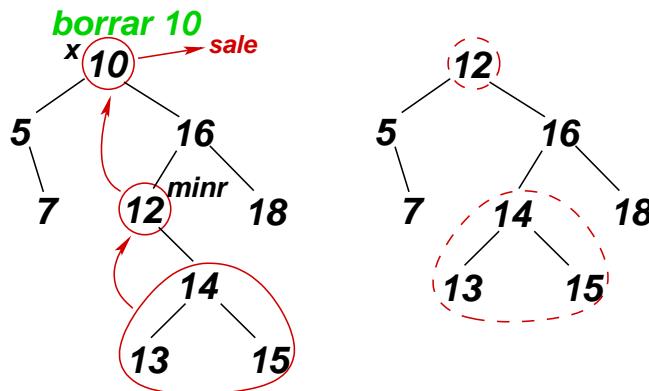


Figura 4.11: Supresión ABB cuando el nodo tiene dos hijos.

- Si no tiene hijos, como el elemento 7 en la figura 4.10 a la derecha, entonces basta con suprimir el nodo. Notar que si tiene hijos, no se puede aplicar directamente el `erase()` de árboles al nodo ya que eliminaría no sólo el elemento en cuestión sino todo su subárbol. Por eso consideramos a continuación los casos en que el nodo tiene uno y dos hijos.
- Si tiene un sólo hijo, como el elemento 18 en la figura 4.10 a la derecha, entonces basta con subir todo el subárbol del hijo existente (en este caso el izquierdo, ocupado por un 16) reemplazando al nodo eliminado.
- En el caso de tener dos hijos, como el elemento `x=10` en la figura 4.11, buscamos algún elemento de la rama derecha (también podría ser de la izquierda) para eliminar de la rama derecha y reemplazar

**x**. Para que el árbol resultante siga siendo un árbol binario de búsqueda, este elemento debe ser el mínimo de la rama derecha, (llamémoslo **minr**) en este caso ocupado por el 12. (También podría usarse el máximo de la rama izquierda.) Notar que, en general **minr** se buscaría como se explicó en la sección §4.6.3, pero a partir del hijo derecho (en este caso el 16). **minr** es eliminado de la rama correspondiente y reemplaza a **x**. Notar que como **minr** es un mínimo, necesariamente no tiene hijo izquierdo, por lo tanto al eliminar **minr** no genera una nueva cascada de eliminaciones. En el ejemplo, basta con subir el subárbol de 14, a la posición previamente ocupada por 12.

#### 4.6.8. Recorrido en el árbol

Hemos discutido hasta aquí las operaciones propias del TAD conjunto, **find(x)**, **insert(x)** y **erase(x)**. Debemos ahora implementar las operaciones para recorrer el árbol en forma ordenada, como es propio del tipo **set<>** en las STL, en particular **begin()** debe retornar un iterator al menor elemento del conjunto y el operador de incremento **n++** debe incrementar el iterator **n** al siguiente elemento en forma ordenada. Estas operaciones son algo complejas de implementar en el ABB.

Para **begin()** basta con seguir el algoritmo explicado en la sección §4.6.3 partiendo de la raíz del árbol.

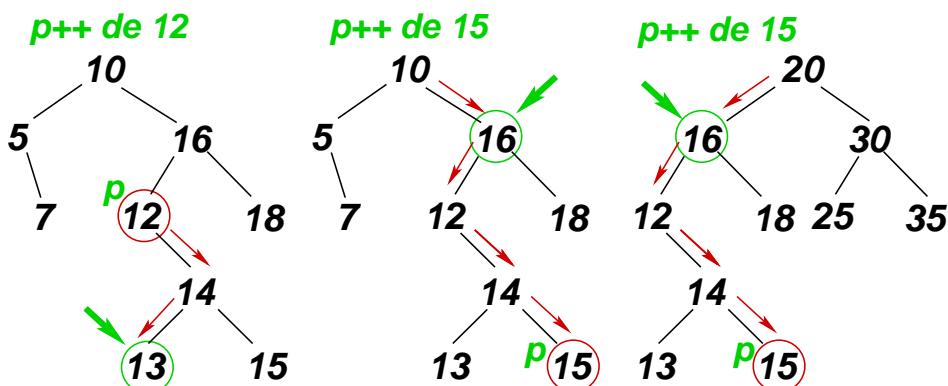


Figura 4.12: Operación **p++** en **set<>** implementado por ABB.

Dado un nodo dereferenciable **n** la operación de incremento **n++** procede de la siguiente forma

- Si **n** tiene hijo derecho (como el 12 en la figura 4.12 a la izquierda) entonces el siguiente es el mínimo de la rama derecha. Por lo tanto basta con bajar por la derecha hasta el hijo derecho (en este caso el 14) y después seguir siempre por la izquierda hasta la última posición dereferenciable (en este caso el 13).
- Si el nodo no tiene hijo derecho, entonces hay que buscar en el camino que llega al nodo el último “padre derecho”. En el ejemplo de la figura 4.12 centro, 15 no tiene hijo derecho, por lo cual recorremos el camino desde la raíz que en este caso corresponde a los elementos  $\{10, 16, 12, 14, 15\}$ . Hay un sólo parente que es derecho en el camino (es decir tal que el nodo siguiente en el camino es hijo izquierdo) y es el 16. Notar que al ser 16 el último parente derecho, esto garantiza que 15 es el máximo de todo el subárbol izquierdo de 16, por lo tanto en si representamos todos los elementos del subárbol de 16 en forma ordenada, 15 está inmediatamente antes que el 16. Como todos los elementos del subárbol

de 16 aparecen juntos en la lista ordenada global (ver sección §4.6.1) en la lista 16 será también el siguiente a 15 en la lista ordenada global. Esto ocurre también si el último padre derecho (16 en este caso) es hijo izquierdo como en el ejemplo de la figura 4.12 a la derecha.

#### 4.6.9. Operaciones binarias

Las funciones binarias requieren un párrafo aparte. Notemos primero que podemos implementar estas funciones en forma genérica. Por ejemplo podemos implementar `set_union(A,B,C)` insertando en **C** primero todos los elementos de **A** y después los de **B**. El mismo `insert(x)` se encargará de no insertar elementos duplicados. Si pudiéramos garantizar un tiempo  $O(\log n)$  para cada inserción, entonces el costo global sería a lo sumo de  $O(n \log n)$  lo cual sería aceptable. Sin embargo, en este caso si iteramos sobre los conjuntos con `begin()`, `operator++()` y `end()`, estaríamos insertando los elementos en forma ordenada en **C** con lo cual se produciría el peor caso discutido en la sección §4.6.5, en el cual el árbol degenera en una lista y las inserciones cuestan  $O(n)$ , dando un costo global para `set_union()` de  $O(n^2)$ , lo cual es inaceptable.

Para evitar esto usamos una estrategia en la cual insertamos **A** y **B** en forma recursiva, insertando primero la raíz y después los árboles izquierdo y derecho. De esta forma puede verse que si los árboles para **A** y **B** están bien balanceados el árbol resultante para **C** también lo estará.

Para `set_intersection()` adoptamos una estrategia similar. Para cada nodo en **A** insertamos primero el valor nodal si está también contenido en **B**, y después aplicamos recursivamente a las ramas izquierda y derecha. La estrategia para `set_difference()` es similar sólo que verificamos que el valor nodal *no esté* en **B**.

#### 4.6.10. Detalles de implementación

```

1 // Forward declarations
2 template<class T>
3 class set;
4 template<class T> void
5 set_union(set<T> &A, set<T> &B, set<T> &C);
6 template<class T> void
7 set_intersection(set<T> &A, set<T> &B, set<T> &C);
8 template<class T> void
9 set_difference(set<T> &A, set<T> &B, set<T> &C);
10
11 template<class T>
12 class set {
13 private:
14 typedef btree<T> tree_t;
15 typedef typename tree_t::iterator node_t;
16 tree_t bstree;
17 node_t min(node_t m) {
18 if (m == bstree.end()) return bstree.end();
19 while (true) {
20 node_t n = m.left();
21 if (n==bstree.end()) return m;
22 m = n;
23 }
24 }
25 }
```

```

24 }
25
26 void set_union_aux(tree_t &t, node_t n) {
27 if (n==t.end()) return;
28 else {
29 insert(*n);
30 set_union_aux(t,n.left());
31 set_union_aux(t,n.right());
32 }
33 }
34 void set_intersection_aux(tree_t &t,
35 node_t n, set &B) {
36 if (n==t.end()) return;
37 else {
38 if (B.find(*n)!=B.end()) insert(*n);
39 set_intersection_aux(t,n.left(),B);
40 set_intersection_aux(t,n.right(),B);
41 }
42 }
43 void set_difference_aux(tree_t &t,
44 node_t n, set &B) {
45 if (n==t.end()) return;
46 else {
47 if (B.find(*n)==B.end()) insert(*n);
48 set_difference_aux(t,n.left(),B);
49 set_difference_aux(t,n.right(),B);
50 }
51 }
52 int size_aux(tree_t t, node_t n) {
53 if (n==t.end()) return 0;
54 else return 1+size_aux(t,n.left())
55 +size_aux(t,n.right());
56 }
57 public:
58 class iterator {
59 private:
60 friend class set;
61 node_t node;
62 tree_t *bstree;
63 iterator(node_t m,tree_t &t)
64 : node(m), bstree(&t) {}
65 node_t next(node_t n) {
66 node_t m = n.right();
67 if (m!=bstree->end()) {
68 while (true) {
69 node_t q = m.left();
70 if (q==bstree->end()) return m;
71 m = q;
72 }
73 } else {
74 // busca el padre
75 m = bstree->begin();
76 if (n==m) return bstree->end();

```

```

77 node_t r = bstree->end();
78 while (true) {
79 node_t q;
80 if (*n<*m) { q = m.left(); r=m; }
81 else q = m.right();
82 if (q==n) break;
83 m = q;
84 }
85 return r;
86 }
87 }
88 public:
89 iterator() : bstree(NULL) { }
90 iterator(const iterator &n)
91 : node(n.node), bstree(n.bstree) {}
92 iterator& operator=(const iterator& n) {
93 bstree=n.bstree;
94 node = n.node;
95 }
96 const T &operator*() { return *node; }
97 const T *operator>() { return &*node; }
98 bool operator!=(iterator q) {
99 return node!=q.node; }
100 bool operator==(iterator q) {
101 return node==q.node; }

102 // Prefix:
103 iterator operator++() {
104 node = next(node);
105 return *this;
106 }
107 // Postfix:
108 iterator operator++(int) {
109 node_t q = node;
110 node = next(node);
111 return iterator(q,*bstree);
112 }
113 };
114 }
115 private:
116 typedef pair<iterator,bool> pair_t;
117 public:
118 set() {}
119 set(const set &A) : bstree(A.bstree) {}
120 ~set() {}
121 pair_t insert(T x) {
122 node_t q = find(x).node;
123 if (q == bstree.end()) {
124 q = bstree.insert(q,x);
125 return pair_t(iterator(q,bstree),true);
126 } else return pair_t(iterator(q,bstree),false);
127 }
128 void erase(iterator m) {
129 node_t p = m.node;

```

```

130 node_t qr = p.right(),
131 ql = p.left();
132 if (qr==bstree.end() && ql==bstree.end())
133 p = bstree.erase(p);
134 else if (qr == bstree.end()) {
135 btree<T> tmp;
136 tmp.splice(tmp.begin(),ql);
137 p = bstree.erase(p);
138 bstree.splice(p,tmp.begin());
139 } else if (ql == bstree.end()) {
140 btree<T> tmp;
141 tmp.splice(tmp.begin(),p.right());
142 p = bstree.erase(p);
143 bstree.splice(p,tmp.begin());
144 } else {
145 node_t r = min(qr);
146 T minr = *r;
147 erase(iterator(r,bstree));
148 *p = minr;
149 }
150 }
151 int erase(T x) {
152 iterator q = find(x);
153 int ret;
154 if (q==end()) ret = 0;
155 else {
156 erase(q);
157 ret = 1;
158 }
159 return ret;
160 }
161 void clear() { bstree.clear(); }
162 iterator find(T x) {
163 node_t m = bstree.begin();
164 while (true) {
165 if (m == bstree.end())
166 return iterator(m,bstree);
167 if (x<*m) m = m.left();
168 else if (x>*m) m = m.right();
169 else return iterator(m,bstree);
170 }
171 }
172 iterator begin() {
173 return iterator(min(bstree.begin()),bstree);
174 }
175 iterator end() {
176 return iterator(bstree.end(),bstree);
177 }
178 int size() {
179 return size_aux(bstree,bstree.begin()); }
180 friend void
181 set_union<T>(set<T> &A, set<T> &B, set<T> &C);
182 friend void

```

```

183 set_intersection<>(set<T> &A, set<T> &B, set<T> &C);
184 friend void
185 set_difference<>(set<T> &A, set<T> &B, set<T> &C);
186 friend void f();
187 };
188
189 template<class T> void
190 set_union(set<T> &A, set<T> &B, set<T> &C) {
191 C.clear();
192 C.set_union_aux(A.bstree, A.bstree.begin());
193 C.set_union_aux(B.bstree, B.bstree.begin());
194 }
195
196 template<class T> void
197 set_intersection(set<T> &A, set<T> &B, set<T> &C) {
198 C.clear();
199 C.set_intersection_aux(A.bstree,
200 A.bstree.begin(), B);
201 }
202
203 // C = A - B
204 template<class T> void
205 set_difference(set<T> &A, set<T> &B, set<T> &C) {
206 C.clear();
207 C.set_difference_aux(A.bstree,
208 A.bstree.begin(), B);
209 }
```

Código 4.15: Implementación de `set<T>` son ABB. [Archivo: `setbst.h`]

En el código 4.15 vemos una posible implementación de conjuntos por ABB usando la clase `btree<T>` discutida en el capítulo §3. En este caso pasamos directamente a la interfaz avanzada, es decir compatible con las STL.

- El tipo `set<T>` es un template que contiene un árbol binario `btree<T>` `bstree`.
- Los `typedef tree_t` y `node_t` son abreviaciones (privadas) para acceder convenientemente al árbol subyacente.
- La función `min(m)` retorna un iterator al nodo con el menor elemento del subárbol del nodo `m`. El nodo se encuentra bajando siempre por el hijo izquierdo (como se explicó en la sección §4.6.3).
- La clase iterator contiene no sólo el nodo en el árbol sino también un puntero al árbol mismo. Esto es necesario ya que algunas operaciones de árboles (por ejemplo la comparación con `end()`) necesitan tener acceso al árbol donde está el nodo.
- `begin()` utiliza `min(bstree.begin())` para encontrar el nodo con el menor elemento en el árbol.
- La implementación incluye un constructor por defecto (el conjunto esta vacío), un constructor por copia que invoca al constructor por copia de árboles. El operador de asignación es sintetizado automáticamente por el compilador y funciona correctamente ya que utiliza el operador de asignación para `btree<T>` el cual fue correctamente implementado (hace una “*deep copy*” del árbol).

- En este caso no mantenemos un contador de nodos, por lo cual la función `size()` calcula el número de nodo usando una función recursiva auxiliar `size_aux()`.
- Las operadores de incremento para iterators (tanto prefijo como postfijo) utilizan una función auxiliar `next()` que calcula la posición correspondiente al siguiente nodo en el árbol (en el sentido ordenado, como se describió en la sección §4.6.8). Esta función se complica un poco ya que nuestra clase árbol no cuenta con una función “padre”. Entonces, cuando el nodo no tiene hijo derecho y es necesario buscar el último “padre derecho”, el camino no se puede seguir desde abajo hacia arriba sino desde arriba hacia abajo, comenzando desde la raíz. De todas formas, el costo es  $O(d)$  donde  $d$  es la altura del árbol, gracias a que la estructura de ABB nos permite ubicar el nodo siguiendo un camino.
- La función `erase()` es implementada eficientemente en términos de `splice()` cuando el nodo tiene sus dos hijos.
- Las funciones binarias utilizan funciones recursivas miembros privados de la clase `set_union_aux(T,n)`, `set_intersection_aux(T,n,B)` y `set_difference_aux(T,n,B)` que aplican la estrategia explicada en la sección §4.6.9.

#### 4.6.11. Tiempos de ejecución

| Método                                                                                | $T(n)$ (promedio) | $T(n)$ (peor caso) |
|---------------------------------------------------------------------------------------|-------------------|--------------------|
| <code>*p, end()</code>                                                                | $O(1)$            | $O(1)$             |
| <code>insert(x), find(x), erase(p),<br/>erase(x), begin(), p++, ++p</code>            | $O(\log n)$       | $O(n)$             |
| <code>set_union(A,B,C),<br/>set_intersection(A,B,C),<br/>set_difference(A,B,C)</code> | $O(n \log n)$     | $O(n^2)$           |

Tabla 4.6: Tiempos de ejecución de los métodos del TAD set implementado con ABB.

#### 4.6.12. Balanceo del árbol

Es claro de los tiempos de ejecución que la eficiencia de esta implementación reside en mantener el árbol lo más balanceado posible. Existen al menos dos estrategias para mantener el árbol balanceado. Los “árboles AVL” mantienen el árbol balanceado haciendo rotaciones de los elementos en forma apropiada ante cada inserción o supresión. Por otra parte los “treaps” mantiene el árbol balanceado introduciendo en cada nodo un campo elemento adicional llamado “prioridad”. La prioridad es generada en forma aleatoria en el momento de insertar el elemento y lo acompaña hasta el momento en que el elemento sea eliminado. Si bien el árbol sigue siendo un ABB con respecto al campo elemento, además se exige que el árbol sea “parcialmente ordenado” con respecto a la prioridad. Un árbol binario parcialmente ordenado es aquel tal que la prioridad de cada nodo es menor o igual que la de sus dos hijos. De esta forma las rotaciones se hacen en forma automática al insertar o eliminar elementos con sólo mirar las prioridades de un nodo y sus dos hijos, mientras que en el caso de los árboles AVL se debe considerar la altura de las ramas derecha e izquierda.

# Capítulo 5

## Ordenamiento

El proceso de ordenar elementos (“*sorting*”) en base a alguna relación de orden (ver sección §2.4.4) es un problema tan frecuente y con tantos usos que merece un capítulo aparte. Nosotros nos concentraremos en el ordenamiento interno, es decir cuando todo el contenedor reside en la memoria principal de la computadora. El tema del ordenamiento externo, donde el volumen de datos a ordenar es de tal magnitud que requiere el uso de memoria auxiliar, tiene algunas características propias y por simplicidad no será considerado aquí.

### 5.1. Introducción

Asumiremos normalmente que los elementos a ordenar pertenecen a algún tipo **key\_t** con una relación de orden  $\lessdot$  y que están almacenados en un contenedor lineal (vector o lista). El problema de ordenar un tal contenedor es realizar una serie de intercambios en el contenedor de manera de que los elementos queden ordenados, es decir  $k_0 \leq k_1 \leq \dots \leq k_{n-1}$ , donde  $k_j$  es la clave del  $j$ -ésimo elemento.

Si el ordenamiento se hace sin requerir ningún tipo de memoria adicional (que crezca con  $n$ ), decimos que es “*in-place*”. Si no, se debe tener en cuenta también como crece la cantidad de memoria requerida.

#### 5.1.1. Relaciones de orden débiles

Ya hemos discutido como se define una relación de orden en la sección §2.4.4. Igual que para definir conjuntos o correspondencias, la relación de ordenamiento puede a veces ser elegida por conveniencia. Así, podemos querer ordenar un conjunto de números por su valor absoluto. En ese caso al ordenar los números  $(1, 3, -2, -4)$  el contenedor ordenado resultaría en  $(1, -2, 3, -4)$ . En este caso la relación de orden la podríamos denotar por  $\triangleleft$  y la definiríamos como

$$a \triangleleft u, \quad \text{si } |a| < |u|. \quad (5.1)$$

La relación binaria definida así no resulta ser una relación de orden en el sentido fuerte, como definida en la sección §2.4.4, ya que, por ejemplo, para el par  $-2, 2$  no son ciertos  $2 \triangleleft -2$ , ni  $-2 \triangleleft 2$  ni  $-2 = 2$ .

La segunda condición sobre las relaciones de orden puede relajarse un poco y obtener la definición de relaciones de orden débiles: **Definición:** “ $\triangleleft$ ” es una relación de orden débil en el conjunto  $C$  si,

1.  $\triangleleft$  es transitiva, es decir, si  $a \triangleleft b$  y  $b \triangleleft c$ , entonces  $a \triangleleft c$ .

2. Dados dos elementos cualquiera  $a, b$  de  $C$

$$(a \triangleleft b) \&& (b \triangleleft a) = \text{false} \quad (5.2)$$

Esta última condición se llama *de antisimetría*. Usamos aquí la notación de C++ para los operadores lógicos, por ejemplo `&&` indica “and” y los valores booleanos `true` y `false`. Es decir  $a \triangleleft b$  y  $b \triangleleft a$  no pueden ser verdaderas al mismo tiempo (son exclusivas).

Los siguientes, son ejemplos de relaciones de orden débiles pero no cumplen con la condición de antisimetría fuerte.

- Menor en valor absoluto para enteros, es decir

$$a \triangleleft b \text{ si } |a| < |b| \quad (5.3)$$

La antisimetría fuerte no es verdadera para el par  $2, -2$ , es decir, ni es cierto que  $-2 \triangleleft 2$  ni  $2 \triangleleft -2$  ni  $2 = -2$ .

- Pares de enteros por la primera componente, es decir  $(a, b) \triangleleft (c, d)$  si  $a < c$ . En este caso la antisimetría fuerte se viola para  $(2, 3)$  y  $(2, 4)$ , por ejemplo.
- Legajos por el nombre del empleado. Dos legajos para diferentes empleados que tienen el mismo nombre (`nombre=Perez, Juan,DNI=14231235`) y (`nombre=Perez, Juan,DNI=12765987`) violan la condición.

Si dos elementos satisfacen que  $!(a \triangleleft b) \&& !(b \triangleleft a)$  entonces decimos que son *equivalentes* y lo denotamos por  $a \equiv b$ .

También se pueden definir otras relaciones derivadas como

|                      |                                                                |
|----------------------|----------------------------------------------------------------|
| mayor:               | $(a \triangleright b) = (b \triangleleft a)$                   |
| equivalencia:        | $(a \equiv b) = !(a \triangleleft b) \&& !(b \triangleleft a)$ |
| menor o equivalente: | $(a \trianglelefteq b) = !(b \triangleleft a)$                 |
| mayor o equivalente: | $(a \trianglerighteq b) = !(a \triangleleft b)$                |

(5.4)

También en algunos lenguajes (e.g. Perl) es usual definir una función `int cmp(T x,T y)` asociada a una dada relación de orden  $\triangleright$  que retorna 1, 0 o -1 dependiendo si  $x \triangleright y$ ,  $x \equiv y$  o  $x \triangleleft y$ . En ese caso, el valor de `cmp(x,y)` se puede obtener de

$$\text{cmp}(x, y) = (y \triangleleft x) - (x \triangleleft y) \quad (5.5)$$

donde en el miembro derecho se asume que las expresiones lógicas retornan 0 o 1 (como es usual en C). En la Tabla 5.1 se puede observar que todos los operadores de comparación se pueden poner en términos de cualquiera de los siguientes  $\triangleleft, \trianglelefteq, \triangleright, \trianglerighteq$ ,  $\text{cmp}(\cdot, \cdot)$ . Notar que los valores retornados por  $\text{cmp}(\cdot, \cdot)$  son comparados directamente con la relación de orden para enteros  $<, >$  etc... y no con la relación  $\triangleleft, \triangleright, \dots$  y amigos.

### 5.1.2. Signatura de las relaciones de orden. Predicados binarios.

En las STL las relaciones de orden se definen mediante “*predicados binarios*”, es decir, su signatura es de la forma

| La expresión:          | Usando: $\triangleleft$                            | Usando: $\trianglelefteq$                                  |
|------------------------|----------------------------------------------------|------------------------------------------------------------|
| $a \triangleleft b$    | $a \triangleleft b$                                | $! (b \trianglelefteq a)$                                  |
| $a \trianglelefteq b$  | $! (b \triangleleft a)$                            | $a \trianglelefteq b$                                      |
| $a \triangleright b$   | $b \triangleleft a$                                | $! a \trianglelefteq b$                                    |
| $a \trianglerighteq b$ | $! a \triangleleft b$                              | $b \trianglerighteq a$                                     |
| $\text{cmp}(a, b)$     | $(b \triangleleft a) - (a \triangleleft b)$        | $(b \trianglelefteq a) - (a \trianglelefteq b)$            |
| $a = b$                | $! (a \triangleleft b \mid\mid b \triangleleft a)$ | $a \trianglelefteq b \&\& b \trianglelefteq a$             |
| $a \neq b$             | $a \triangleleft b \mid\mid b \triangleleft a$     | $(! a \trianglelefteq b) \mid\mid (! b \trianglelefteq a)$ |

| La expresión:          | Usando: $\triangleright$                             | Usando: $\trianglerighteq$                                   |
|------------------------|------------------------------------------------------|--------------------------------------------------------------|
| $a \triangleleft b$    | $b \triangleright a$                                 | $! a \trianglerighteq b$                                     |
| $a \trianglelefteq b$  | $! (b \triangleright a)$                             | $b \trianglerighteq a$                                       |
| $a \triangleright b$   | $a \triangleright b$                                 | $! b \trianglerighteq a$                                     |
| $a \trianglerighteq b$ | $! b \triangleright a$                               | $a \trianglerighteq b$                                       |
| $\text{cmp}(a, b)$     | $(a \triangleright b) - (b \triangleright a)$        | $(a \trianglerighteq b) - (b \trianglerighteq a)$            |
| $a = b$                | $! (a \triangleright b \mid\mid b \triangleright a)$ | $a \trianglerighteq b \&\& b \trianglerighteq a$             |
| $a \neq b$             | $a \triangleright b \mid\mid b \triangleright a$     | $(! a \trianglerighteq b) \mid\mid (! b \trianglerighteq a)$ |

| La expresión:          | Usando: $\text{cmp}(\cdot, \cdot)$ |
|------------------------|------------------------------------|
| $a \triangleleft b$    | $\text{cmp}(a, b) = -1$            |
| $a \trianglelefteq b$  | $\text{cmp}(a, b) \leq 0$          |
| $a \triangleright b$   | $\text{cmp}(a, b) = 1$             |
| $a \trianglerighteq b$ | $\text{cmp}(a, b) \geq 0$          |
| $\text{cmp}(a, b)$     | $\text{cmp}(a, b)$                 |
| $a = b$                | $! \text{cmp}(a, b)$               |
| $a \neq b$             | $\text{cmp}(a, b)$                 |

Tabla 5.1: Equivalencia entre los diferentes operadores de comparación.

```
bool (*binary_pred)(T x, T Y);
```

**Ejemplo 5.1:** *Consigna:* Escribir una relación de orden para comparación lexicográfica de cadenas de caracteres. Hacerlo en forma dependiente e independiente de mayúsculas y minúsculas (*case-sensitive* y *case-insensitive*).

La clase **string** de las STL permite encapsular cadenas de caracteres con ciertas características mejoradas con respecto al manejo básico de C. Entre otras cosas, tiene sobrecargado el operador “ $<$ ” con el orden lexicográfico (es decir, el orden alfabético). El orden lexicográfico consiste en comparar los primeros caracteres de ambas cadenas, si son diferentes entonces es menor aquel cuyo valor ASCII es menor, si son iguales se continua con los segundos caracteres y así siguiendo hasta que eventualmente uno de las dos cadenas se termina. En ese caso si una de ellas continúa es la mayor, si sus longitudes son iguales los strings son iguales. Esta relación de orden es fuerte.

---

```
1 bool string_less_cs(const string &a,const string &b) {
2 int na = a.size();
```

```
3 int nb = b.size();
4 int n = (na>nb ? nb : na);
5 for (int j=0; j<n; j++) {
6 if (a[j] < b[j]) return true;
7 else if (b[j] < a[j]) return false;
8 }
9 return na<nb;
10 }
```

**Código 5.1:** Función de comparación para cadenas de caracteres con orden lexicográfico. [Archivo: *stringlcs.cpp*]

En el código 5.1 vemos la implementación del predicado binario correspondiente. Notar que este predicado binario termina finalmente comparando caracteres en las líneas 6–7. A esa altura se está usando el operador de comparación sobre el tipo **char** que es un subtipo de los enteros.

Si ordenamos las cadenas **pepe juana PEPE Juana JUANA Pepe**, con esta función de comparación obtendremos

JUANA Juana PEPE Pepe juana pepe (5.6)

Recordemos que en la serie ASCII las mayúsculas están antes que las minúsculas, las minúsculas **a-z** están en el rango 97-122 mientras que las mayúsculas **A-Z** están en el rango 65-90.

```
1 bool string_less_cs3(const string &a,const string &b) {
2 return a<b;
3 }
```

**Código 5.2:** Función de comparación para cadenas de caracteres con orden lexicográfico usando el operador “**<**” de C++. [Archivo: *stringlcs3.cpp*]

```
1 template<class T>
2 bool less(T &x,T &y) {
3 return x<y;
4 }
```

**Código 5.3:** Template de las STL que provee un predicado binario al operador intrínseco “**<**” del tipo **T**. [Archivo: *lesst.h*]

Como ya mencionamos el orden lexicográfico está incluido en las STL, en forma del operador “**<**” sobrecargado, de manera que también podría usarse como predicado binario la función mostrada en el código 5.2. Finalmente, como para muchos tipos básicos (**int**, **double**, **string**) es muy común ordenar por el operador “**<**” del tipo, las STL proveen un template **less<T>** que devuelve el predicado binario correspondiente al operador intrínseco “**<**” del tipo **T**.

```

1 char tolower(char c) {
2 if (c>='A' && c<='Z') c += 'a'-'A';
3 return c;
4 }
5
6 bool string_less_ci(const string &a,
7 const string &b) {
8 int na = a.size();
9 int nb = b.size();
10 int n = (na>nb ? nb : na);
11 for (int j=0; j<n; j++) {
12 char
13 aa = tolower(a[j]),
14 bb = tolower(b[j]);
15 if (aa < bb) return true;
16 else if (bb < aa) return false;
17 }
18 return na<nb;
19 }
```

**Código 5.4:** Función de comparación para cadenas de caracteres con orden lexicográfico independiente de mayúsculas y minúsculas. [Archivo: *stringci.cpp*]

Si queremos ordenar en forma independiente de mayúsculas/minúsculas, entonces podemos definir una relación binaria que básicamente es

$$(a \triangleleft b) = (\text{tolower}(a) < \text{tolower}(b)) \quad (5.7)$$

donde la función **tolower()** convierte su argumento a minúsculas. La función **tolower()** está definida en la librería estándar de C (libc) para caracteres. Podemos entonces definir la función de comparación para orden lexicográfico independiente de mayúsculas/minúsculas como se muestra en el código 5.4. La función **string\_less\_ci()** es básicamente igual a la **string\_less\_cs()** sólo que antes de comparar caracteres los convierte con **tolower()** a minúsculas. De esta forma **pepe**, **Pepe** y **PEPE** resultan equivalentes.

### 5.1.3. Relaciones de orden inducidas por composición

La relación de orden para cadenas de caracteres (5.7) es un caso particular de una forma bastante general de generar relaciones de orden que se obtienen componiendo otra relación de orden con una función escalar. Dada una relación de orden “<” y una función escalar  $y = f(x)$  podemos definir una relación de orden “ $\triangleleft$ ” mediante

$$(a \triangleleft b) = (f(a) < f(b)) \quad (5.8)$$

Por ejemplo, la relación de orden *menor en valor absoluto* definida en (5.3) para los enteros, puede interpretarse como la composición de la relación de orden usual “<” con la función valor absoluto. Si la relación de orden “<” y  $f()$  es biunívoca, entonces “ $\triangleleft$ ” resulta ser también fuerte. Si estas condiciones no se cumplen, la relación resultante puede ser débil. Volviendo al caso de la relación *menor en valor absoluto*, si bien la relación de partida “<” es fuerte, la composición con una función no biunívoca como el valor absoluto resulta en una relación de orden débil.

Notar que la función de mapeo puede ser de un conjunto universal  $U$  a otro  $U' \neq U$ . Por ejemplo, si queremos ordenar un vector de listas por su longitud, entonces el conjunto universal  $U$  es el conjunto de las listas, mientras que  $U'$  es el conjunto de los enteros.

#### 5.1.4. Estabilidad

Ya vimos que cuando la relación de orden es débil puede haber elementos que son equivalentes entre sí sin ser iguales. Al ordenar el contenedor por una relación de orden débil los elementos equivalentes entre sí deben quedar contiguos en el contenedor ordenado pero el orden entre ellos no está definido en principio.

Un algoritmo de ordenamiento es “estable” si aquellos elementos equivalentes entre sí quedan en el orden en el que estaban originalmente. Por ejemplo, si ordenamos  $(-3, 2, -4, 5, 3, -2, 4)$  por valor absoluto, entonces podemos tener

$$\begin{aligned} (2, -2, -3, 3, -4, 4, 5), & \text{ estable} \\ (-2, 2, -3, 3, 4, -1, 5), & \text{ no estable} \end{aligned} \tag{5.9}$$

#### 5.1.5. Primeras estimaciones de eficiencia

Uno de los aspectos más importantes de los algoritmos de ordenamiento es su tiempo de ejecución como función del número  $n$  de elementos a ordenar. Ya hemos visto en la sección §1.4.2 el algoritmo de ordenamiento por el método de la burbuja (“bubble-sort”), cuyo tiempo de ejecución resultó ser  $O(n^2)$ . Existen otros algoritmos de ordenamiento simples que también llevan a un número de operaciones  $O(n^2)$ . Por otra parte, cualquier algoritmo de ordenamiento debe por lo menos recorrer los elementos a ordenar, de manera que al menos debe tener un tiempo de ejecución  $O(n)$ , de manera que en general todos los algoritmos existentes están en el rango  $O(n^\alpha)$  con  $1 \leq \alpha \leq 2$  (los algoritmos  $O(n \log n)$  pueden considerarse como  $O(n^{1+\epsilon})$  con  $\epsilon \rightarrow 0$ ).

Si los objetos que se están ordenando son grandes (largas cadenas de caracteres, listas, árboles...) entonces puede ser más importante considerar el número de intercambios que requiere el algoritmo, en vez del número de operaciones. Puede verse también que los algoritmos más simples hacen  $O(n^2)$  intercambios (por ej. el método de la burbuja).

#### 5.1.6. Algoritmos de ordenamiento en las STL

La firma del algoritmo genérico de ordenamiento en las STL es

```
1 void sort(iterator first, iterator last);
2 void sort(iterator first, iterator last, binary_pred f);
```

el cual está definido en el header **algorithm**. Notemos que en realidad **sort()** es una función “sobrecargada”, existen en realidad dos funciones **sort()**. La primera toma un rango **[first, last)** de iteradores en el contenedor y ordena ese rango del contenedor, dejando los elementos antes de **first** y después de **last** inalterados. Si queremos ordenar todo el contenedor, entonces basta con pasar el rango **[begin(), end())**. Esta versión de **sort()** no se puede aplicar a cualquier contenedor sino que tiene que ser un “contenedor de acceso aleatorio”, es decir un contenedor en el cual los iteradores soportan operaciones de aritmética entera, es decir si tenemos un iterador **p**, podemos hacer **p+j** (avanzar el iterador **j** posiciones, en tiempo  $O(1)$ ). Los operadores de acceso aleatorio en las STL son **vector<>** y **deque<>**.

El ordenamiento se realiza mediante la relación de orden **operator<** del tipo **T** del cual están compuestos los elementos del contenedor. Si el tipo **T** es una clase definida por el usuario, este debe sobrecargar el **operator<**.

La segunda versión toma un argumento adicional que es la función de comparación. Esto puede ser útil cuando se quiere ordenar un contenedor por un orden diferente a **operator<** o bien **T** no tiene definido **operator<**. Las STL contiene en el header **functional** unos templates **less<T>**, **greater<T>** que devuelven funciones de comparación basados en **operator<** y **operator>** respectivamente. Por ejemplo, si queremos ordenar de mayor a menor un vector de enteros, basta con hacer

```
1 vector<int> v;
2 // Inserta elementos en v...
3 sort(v.begin(), v.end(), greater<int>);
```

Usar **less<int>** es totalmente equivalente a usar la versión **sort(first, last)** es decir sin función de comparación.

Si queremos ordenar un vector de strings por orden lexicográfico independientemente de minúsculas/mayúsculas debemos hacer

```
1 vector<string> v;
2 // Inserta elementos en v...
3 sort(v.begin(), v.end(), string_less_ci);
```

## 5.2. Métodos de ordenamiento lentos

Llamamos “*rápidos*” a los métodos de ordenamiento con tiempos de ejecución menores o iguales a  $O(n \log n)$ . Al resto lo llamaremos “*lentos*”. En esta sección estudiaremos tres algoritmos lentos, a saber burbuja, selección e inserción.

### 5.2.1. El método de la burbuja

---

```
1 template<class T> void
2 bubble_sort(typename std::vector<T>::iterator first,
3 typename std::vector<T>::iterator last,
4 bool (*comp)(T&,T&)) {
5 int size = last-first;
6 for (int j=0; j<size-1; j++) {
7 for (int k=size-1; k>j; k--) {
8 if (comp(*(first+k),*(first+k-1))) {
9 T tmp = *(first+k-1);
10 *(first+k-1) = *(first+k);
11 *(first+k) = tmp;
12 }
13 }
14 }
15 }
16
17 template<class T> void
18 bubble_sort(typename std::vector<T>::iterator first,
19 typename std::vector<T>::iterator last) {
```

```
20 bubble_sort(first, last, less<T>);
21 }
```

Código 5.5: Algoritmo de ordenamiento de la burbuja. [Archivo: bubsort.h]

El método de la burbuja fue introducido en la sección §1.4.2. Nos limitaremos aquí a discutir la conversión al formato compatible con la STL. El código correspondiente puede observarse en el código 5.5.

- Para cada algoritmo de ordenamiento presentamos las dos funciones correspondientes, con y sin función de comparación.
- Ambas son templates sobre el tipo **T** de los elementos a ordenar.
- La que no tiene operador de comparación suele ser un “wrapper” que llama a la primera pasándole como función de comparación **less<T>**.
- Notar que como las funciones no reciben un contenedor, sino un rango de iteradores, no se puede referenciar directamente a los elementos en la forma **v[j]** sino a través del operador de dereferenciación **\*p**. Así, donde normalmente pondríamos **v[first+j]** debemos usar **\*(first+j)**.
- Recordar que las operaciones aritméticas con iteradores son válidas ya que los contenedores son de acceso aleatorio. En particular, las funciones presentadas son sólo válidas para **vector<>**, aunque se podrían modificar para incluir a **deque<>** en forma relativamente fácil.
- Notar la comparación en la línea 8 usando el predicado binario **comp()** en vez de **operator<**. Si queremos que nuestros algoritmos de ordenamiento puedan ordenar con predicados binarios arbitrarios, la comparación para elementos de tipo **T** se debería hacer siempre usando **comp()**.
- Para la discusión de los diferentes algoritmos haremos abstracción de la posición de comienzo **first**, como si fuera 0. Es decir consideraremos que se está ordenando el rango **[0, n)** donde **n=last-first**.

### 5.2.2. El método de inserción

```
1 template<class T> void
2 insertion_sort(typename
3 std::vector<T>::iterator first,
4 typename
5 std::vector<T>::iterator last,
6 bool (*comp)(T&,T&)) {
7 int size = last-first;
8 for (int j=1; j<size; j++) {
9 T tmp = *(first+j);
10 int k=j-1;
11 while (comp(tmp,*(first+k))) {
12 *(first+k+1) = *(first+k);
13 if (--k < 0) break;
14 }
15 *(first+k+1) = tmp;
16 }
17 }
18
19
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```
20 template<class T> void
21 insertion_sort(typename
22 std::vector<T>::iterator first,
23 typename
24 std::vector<T>::iterator last) {
25 insertion_sort(first, last, less<T>);
26 }
```

Código 5.6: Algoritmo de ordenamiento por inserción. [Archivo: *inssorta.h*]

En este método (ver código 5.6) también hay un doble lazo. En el lazo sobre  $j$  el rango  $[0, j)$  está ordenado e insertamos el elemento  $j$  en el rango  $[0, j)$ , haciendo los desplazamientos necesarios. El lazo sobre  $k$  va recorriendo las posiciones desde  $j$  hasta 0 para ver donde debe insertarse el elemento que en ese momento está en la posición  $j$ .

### 5.2.3. El método de selección

```
1 template<class T> void
2 selection_sort(typename
3 std::vector<T>::iterator first,
4 typename
5 std::vector<T>::iterator last,
6 bool (*comp)(T&, T&)) {
7 int size = last - first;
8 for (int j=0; j < size-1; j++) {
9 typename std::vector<T>::iterator
10 min = first+j,
11 q = min+1;
12 while (q < last) {
13 if (comp(*q, *min)) min = q;
14 q++;
15 }
16 T tmp = *(first+j);
17 *(first+j) = *min;
18 *min = tmp;
19 }
20 }
```

Código 5.7: Algoritmo de ordenamiento por selección. [Archivo: *selsort.h*]

En este método (ver código 5.7) también hay un doble lazo (esta es una característica de todos los algoritmos lentos). En el lazo  $j$  se elige el menor del rango  $[j, N)$  y se intercambia con el elemento en la posición  $j$ .

#### 5.2.4. Comparación de los métodos lentos

- En los tres métodos el rango  $[0, j]$  está siempre ordenado. Este rango va creciendo hasta que en la iteración  $j=n$  ocupa todo el vector y por lo tanto queda ordenado.
- Además en los métodos de burbuja y selección el rango  $[0, j]$  está en su posición definitiva, es decir los elementos en ese rango son los  $j$  menores elementos del vector, de manera que en las iteraciones sucesivas no cambiarán de posición. Por el contrario, en el método de inserción el elemento  $j$ -ésimo viaja hasta la posición que le corresponde de manera que hasta en la última ejecución del lazo sobre  $j$  todas las posiciones del vector pueden cambiar.
- Tanto burbuja como selección son exactamente  $O(n^2)$ . Basta ver que en ambos casos el lazo interno se ejecuta incondicionalmente  $j$  veces.
- En el caso de inserción el lazo interno puede ejecutarse entre 0 y  $j$  veces, dependiendo de donde se encuentre el punto de inserción para el elemento  $j$ . El mejor caso es cuando el vector está ordenado, y el punto de inserción en ese caso es directamente la posición  $k=j-1$  para todos los  $j$ . En este caso el lazo interno se ejecuta una sola vez para cada valor de  $j$  con lo cual el costo total es  $O(n)$ .
- El peor caso para inserción es cuando el vector está ordenado en forma inversa (de mayor a menor). En ese caso, para cada  $j$  el elemento  $j$  debe viajar hasta la primera posición y por lo tanto el lazo interno se ejecuta  $j$  veces. El costo total es en este caso  $\sim n^2/2$ .
- En el caso promedio (es decir cuando los valores del vector están aleatoriamente distribuidos) la posición de inserción  $k$  está en el medio del rango  $[0, j]$  de manera que el lazo interno se ejecuta  $\sim j/2$  veces y el costo total es  $\sim n^2/4$ .
- En cuanto al número de intercambios, ambos burbuja e inserción hacen en el peor caso (cuando el vector está ordenado en forma inversa)  $j$  intercambios en el lazo interior y  $\sim n^2/2$  en total.
- En el mejor caso (cuando el vector está ordenado) ambos hacen 0 intercambios. En el caso promedio ambos hacen  $\sim n^2/4$  intercambios (Hay un 50 % de probabilidad de que el intercambio se haga o no).
- Selección hace siempre sólo  $n$  intercambios. Notar que el intercambio se hace fuera del lazo interno.

Debido a estas consideraciones resulta que *inserción* puede ser de interés cuando el vector está parcialmente ordenado, es decir hay relativamente pocos elementos fuera de posición.

Por otra parte *selección* puede ser una opción interesante cuando se debe minimizar el número de intercambios. Sin embargo, veremos en la siguiente sección que, ordenando “*indirectamente*” los elementos, cualquier se puede lograr que cualquier método de ordenación haga sólo  $n$  intercambios.

#### 5.2.5. Estabilidad

Una forma de verificar si un algoritmo de ordenamiento es estable o no es controlar que la estabilidad no se viole en ningún intercambio. Por ejemplo en el caso del método de la burbuja la estabilidad se mantiene ya que el intercambio se realiza sólo en las líneas 9–11. Pero como el intercambio sólo se realiza si  $*(\text{first}+k)$  es *estrictamente menor* que  $*(\text{first}+k-1)$  y los dos elementos están en posiciones consecutivas *el intercambio nunca viola la estabilidad*.

En el caso del método de inserción pasa algo similar. En las líneas 11–14 el elemento  $*(\text{first}+j)$  es intercambiado con todo el rango  $[\text{first}+k, \text{first}+j]$  que son elementos *estrictamente mayores* que  $*(\text{first}+j)$ .

En cambio, en el caso del método de selección, después de buscar la posición del mínimo en el lazo de las líneas 12–15, el intercambio se realiza en las líneas 16–18. Pero al realizar este intercambio, el ele-

mento `*(first+j)`, que va a ir a la posición `min`, puede estar cambiando de posición relativa con elementos equivalentes en el rango `(first+j, min)`, violando la estabilidad.

### 5.3. Ordenamiento indirecto

Si el intercambio de elementos es muy costoso (pensemos en largas listas, por ejemplo) podemos reducir notablemente el número de intercambios usando “ordenamiento indirecto”, el cual se basa en ordenar un vector de cursores o punteros a los objetos reales. De esta forma el costo del intercambio es en realidad el de intercambio de los cursores o punteros, lo cual es mucho más bajo que el intercambio de los objetos mismos.

```
1 template<class T>
2 void apply_perm(typename std::vector<T>::iterator first,
3 typename std::vector<T>::iterator last,
4 std::vector<int> &idx) {
5 int size = last-first;
6 assert(idx.size()==size);
7 int sorted = 0;
8 T tmp;
9 while (sorted<size) {
10 if(idx[sorted]!=sorted) {
11 int k = sorted;
12 tmp = *(first+k);
13 while (idx[k]!=sorted) {
14 int kk = idx[k];
15 *(first+k)=*(first+kk);
16 idx[k] = k;
17 k = kk;
18 }
19 *(first+k) = tmp;
20 idx[k] = k;
21 }
22 sorted++;
23 }
24 }
25
26 template<class T>
27 void ibubble_sort(typename std::vector<T>::iterator first,
28 typename std::vector<T>::iterator last,
29 bool (*comp)(T&,T&)) {
30 int size = last-first;
31 std::vector<int> idx(size);
32 for (int j=0; j<size; j++) idx[j] = j;
33
34 for (int j=0; j<size-1; j++) {
35 for (int k=size-1; k>j; k--) {
36 if (comp(*(first+idx[k]),*(first+idx[k-1]))) {
37 int tmp = idx[k-1];
38 idx[k-1] = idx[k];
39 idx[k] = tmp;
```

```

40
41 }
42 }
43 apply_perm<T>(first, last, indx);
44 }
45
46 template<class T>
47 void ibubble_sort(typename std::vector<T>::iterator first,
48 typename std::vector<T>::iterator last) {
49 ibubble_sort(first, last, less<T>);
50 }
```

**Código 5.8:** Método de la burbuja con ordenamiento indirecto. [Archivo: *ibub.h*]

En el código 5.8 vemos el método de la burbuja combinado con ordenamiento indirecto. Igual que para el ordenamiento directo existen versiones con y sin función de comparación (*ibubble\_sort(first, last, comp)* y *ibubble\_sort(first, last)*). Se utiliza un vector auxiliar de enteros *indx*. Inicialmente este vector contiene los enteros de 0 a *size-1*. El método de la burbuja procede en las líneas 34–42, pero en vez de intercambiar los elementos en el contenedor real, se mueven los cursosres en *indx*. Por ejemplo, después de terminar la ejecución del lazo para *j=0*, en vez de quedar el menor en *\*first*, en realidad queda la posición correspondiente al menor en *indx[0]*. En todo momento *indx[]* es una permutación de los enteros en *[0, size)*. Cuando termina el algoritmo de la burbuja, al llegar a la línea 43, *indx[]* contiene la permutación que hace ordena *[first, last)*. Por ejemplo el menor de todos los elementos está en la posición *first+indx[0]*, el segundo menor en *first+indx[1]*, etc... La última operación de *ibubble\_sort()* es llamar a la función *apply\_perm(first, last, indx)* que aplica la permutación obtenida a los elementos en *[first, last)*. Puede verse que esta función realiza *n* intercambios o menos. Esta función es genérica y puede ser combinada con cualquier otro algoritmo de ordenamiento indirecto. En la figura 5.1 vemos como queda el vector *v[]* con los elementos desordenados, y la permutación *indx[]*.

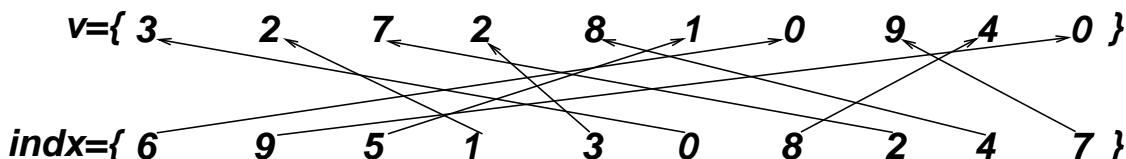


Figura 5.1: Ordenamiento indirecto.

El código en las líneas 34–42, es básicamente igual al de la versión directa *bubble\_sort()* sólo que donde en ésta se referencian elementos como *\*(first+j)* en la versión indirecta se referencian como *\*(first+indx[j])*. Por otra parte, donde la versión indirecta intercambia los elementos *\*(first+k-1)* y *\*(first+k)*, la versión indirecta intercambia los índices en *indx[k-1]* y *indx[k]*.

En el caso de usar ordenamiento indirecto debe tenerse en cuenta que se requiere memoria adicional para almacenar *indx[]*.

### 5.3.1. Minimizar la llamada a funciones

Si la función de comparación se obtiene por composición, y la función de mapeo es muy costosa. Entonces tal vez convenga generar primero un vector auxiliar con los valores de la función de mapeo, ordenarlo y después aplicar la permutación resultante a los elementos reales. De esta forma el número de llamadas a la función de mapeo pasa de ser  $O(n^2)$  a  $O(n)$ . Por ejemplo, supongamos que tenemos un conjunto de árboles y queremos ordenarlos por la suma de sus valores nodales. Podemos escribir una función `int sum_node_val(tree<int> &A);` y escribir una función de comparación

```
1 bool comp_tree(tree<int> &A, tree<int> &B) {
2 return sum_node_val(A) < sum_node_val(B);
3 }
```

Si aplicamos por ejemplo **bubble\_sort**, entonces el número de llamadas a la función será  $O(n^2)$ . Si los árboles tienen muchos nodos, entonces puede ser preferible generar un vector de enteros con los valores de las sumas y ordenarlo. Luego se aplicaría la permutación correspondiente al vector de árboles.

En este caso debe tenerse en cuenta que se requiere memoria adicional para almacenar el vector de valores de la función de mapeo, además del vector de índices para la permutación.

## 5.4. El método de ordenamiento rápido, quick-sort

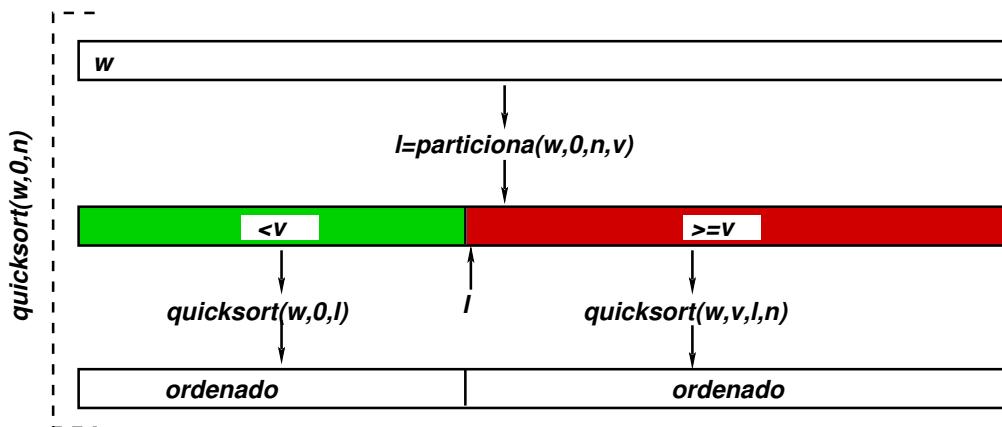


Figura 5.2: Esquema “dividir para vencer” para el algoritmo de ordenamiento rápido, quick-sort.

Este es probablemente uno de los algoritmos de ordenamiento más usados y se basa en la estrategia de “dividir para vencer”. Se escoge un elemento del vector  $v$  llamado “pivot” y se “particiona” el vector de manera de dejar los elementos  $\geq v$  a la derecha (rango  $[l, n]$ ), donde  $l$  es la posición del primer elemento de la partición derecha) y los  $< v$  a la izquierda (rango  $[0, l)$ ). Está claro que a partir de entonces, los elementos en cada una de las particiones quedarán en sus respectivos rangos, ya que todos los elementos en la partición derecha son estrictamente mayores que los de la izquierda. Podemos entonces aplicar **quick-sort** recursivamente a cada una de las particiones.

```
1 void quicksort(w, j1, j2) {
2 // Ordena el rango [j1, j2] de 'w'
```

```

3 if (n==1) return;
4 // elegir pivote v ...
5 l = partition(w,j1,j2,v);
6 quicksort(w,j1,l);
7 quicksort(w,l,j2);
8 }

```

Código 5.9: Seudocódigo para el algoritmo de ordenamiento rápido. [Archivo: qsortsc.cpp]

Si garantizamos que cada una de las particiones tiene al menos un elemento, entonces en cada nivel de recursividad los rangos a los cuales se le aplica quick-sort son estrictamente menores. La recursión termina cuando el vector a ordenar tiene un sólo elemento.

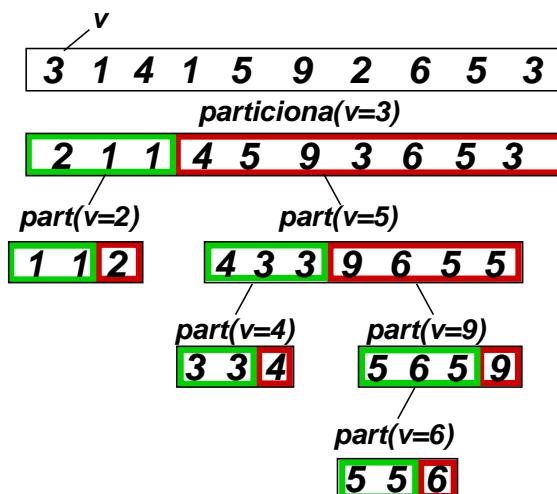


Figura 5.3: Ejemplo de aplicación de quick-sort

En la figura 5.3 vemos un ejemplo de aplicación de quick-sort a un vector de enteros. Para el ejemplo hemos elegido como estrategia para elegir el pivote tomar el mayor de los dos primeros elementos distintos. Si la secuencia a ordenar no tiene elementos distintos, entonces la recursión termina. En el primer nivel los dos primeros elementos distintos (de izquierda a derecha) son el 3 y el 1, por lo que el pivote será  $v = 3$ . Esto induce la partición que se observa en la línea siguiente, donde tenemos los elementos menores que 3 en el rango  $[0, 3)$  y los mayores o iguales que 3 en el rango  $[3, 10)$ . Todavía no hemos explicado cómo se hace el algoritmo de partición, pero todavía esto no es necesario para entender como funciona quick-sort. Ahora aplicamos quick-sort a cada uno de los dos rangos. En el primer caso, los dos primeros elementos distintos son 2 y 1 por lo que el pivote es 2. Al particionar con este pivote quedan dos rangos en los cuales no hay elementos distintos, por lo que la recursión termina allí. Para la partición de la derecha, en cambio, todavía debe aplicarse quick-sort dos niveles más. Notar que la estrategia propuesta de elección del pivote garantiza que cada una de las particiones tendrá al menos un elemento ya que si los dos primeros elementos distintos de la secuencia son  $a$  y  $b$ , y  $a < b$ , por lo cual  $v = b$ , entonces al menos hay un elemento en la partición derecha (el elemento  $b$ ) y otro en la partición izquierda (el elemento  $a$ ).

### 5.4.1. Tiempo de ejecución. Casos extremos

El tiempo de ejecución del algoritmo aplicado a un rango  $[j_1, j_2)$  de longitud  $n = j_2 - j_1$  es

$$T(n) = T_{\text{part-piv}}(n) + T(n_1) + T(n_2) \quad (5.10)$$

donde  $n_1 = l - j_1$  y  $n_2 = j_2 - l$  son las longitudes de cada una de las particiones.  $T_{\text{part-piv}}(n)$  es el costo de particionar la secuencia y elegir el pivote.

El mejor caso es cuando podemos elegir el pivote de tal manera que las particiones resultan ser bien balanceadas, es decir  $n_1 \approx n_2 \approx n/2$ . Si además asumimos que el algoritmo de particionamiento y elección del pivote son  $O(n)$ , es decir

$$T_{\text{part-piv}} = cn \quad (5.11)$$

(más adelante se explicará un algoritmo de particionamiento que satisface esto) Entonces

$$\begin{aligned} T(n) &= T_{\text{part-piv}}(n) + T(n_1) + T(n_2) \\ &= cn + T(n/2) + T(n/2) \end{aligned} \quad (5.12)$$

Llamando  $T(1) = d$  y aplicando sucesivamente

$$\begin{aligned} T(2) &= c + 2T(1) = c + 2d \\ T(4) &= 4c + 2T(2) = 3 \cdot 4c + 4d \\ T(8) &= 8c + 2T(4) = 4 \cdot 8c + 8d \\ T(16) &= 16c + 2T(8) = 5 \cdot 16c + 16d \\ \vdots &= \vdots \\ T(2^p) &= (p+1)n(c+d) \end{aligned} \quad (5.13)$$

pero como  $n = 2^p$  entonces  $p = \log_2 n$  y por lo tanto

$$T(n) = O(n \log n). \quad (5.14)$$

Por otro lado el peor caso es cuando la partición es muy desbalanceada, es decir  $n_1 = 1$  y  $n_2 = n - 1$  o viceversa. En este caso tenemos

$$\begin{aligned} T(n) &= cn + T(1) + T(n-1) \\ &= cn + d + T(n-1) \end{aligned} \quad (5.15)$$

y aplicando sucesivamente,

$$\begin{aligned} T(2) &= 2c + 2d \\ T(3) &= 3c + d + (2c + 2d) = 5c + 3d \\ T(4) &= 4c + d + (5c + 3d) = 9c + 4d \\ T(5) &= 5c + d + (9c + 4d) = 14c + 5d \\ \vdots &= \vdots \\ T(n) &= \left( \frac{n(n+1)}{2} - 2 \right) c + nd = O(n^2) \end{aligned} \quad (5.16)$$

El peor caso ocurre, por ejemplo, si el vector está inicialmente ordenado y usando como estrategia para el pivote el mayor de los dos primeros distintos. Si tenemos en  $v$ , por ejemplo los enteros 1 a 100 ordenados, que podemos denotar como un rango  $[1, 100]$ , entonces el pivote sería inicialmente 2. La partición izquierda tendría sólo al 1 y la derecha sería el rango  $[2, 99]$ . Al particionar  $[2, 99]$  tendríamos el pivote 3 y las particiones serían 2 y  $[3, 99]$  (ver figura 5.4). Puede verificarse que lo mismo ocurriría si el vector está ordenado al revés.

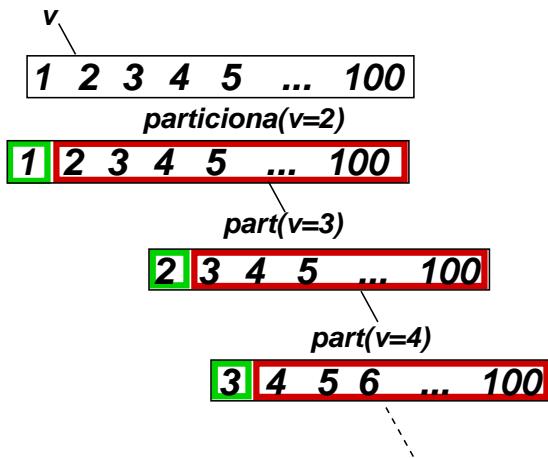


Figura 5.4: Particionamiento desbalanceado cuando el vector ya está ordenado.

#### 5.4.2. Elección del pivote

En el caso promedio, el balance de la partición dependerá de la estrategia de elección del pivote y de la distribución estadística de los elementos del vector. Compararemos a continuación las estrategias que corresponden a tomar la “*mediana*” de los  $k$  primeros distintos. Recordemos que para  $k$  impar la mediana de  $k$  elementos es el elemento que queda en la posición del medio del vector (la posición  $(k - 1)/2$  en base 0) después de haberlos ordenado. Para  $k$  par tomamos el elemento en la posición  $k/2$  (base 0) de los primeros  $k$  distintos, después de ordenarlos.

No confundir la mediana con el “*promedio*” o “*media*” del vector que consiste en sumar los valores y dividirlos por el número de elementos. Si bien es muy sencillo calcular el promedio en  $O(n)$  operaciones, la mediana requiere en principio ordenar el vector, por lo que claramente no se puede usar la mediana como estrategia para elegir el pivote. En lo que resta, cuando hablamos de elegir la mediana de  $k$  valores del vector, asumimos que  $k$  es un valor constante y pequeño, más concretamente que no crece con  $n$ .

En cuanto al promedio, tampoco es una buena opción para el pivote. Consideremos un vector con 101 elementos 1,2,...,99,100,100000. La mediana de este vector es 51 y, por supuesto da una partición perfecta, mientras que el promedio es 1040.1, lo cual daría una pésima partición con 100 elementos en la partición izquierda y 1 elemento en la derecha. Notemos que esta mala partición es causada por la no uniformidad en la distribución de los elementos, para distribuciones más uniformes de los elementos tal vez, es posible que el promedio sea un elección razonable. De todas formas el promedio es un concepto que es sólo aplicable a tipos para los cuales las operaciones algebraicas tienen sentido. No es claro como podríamos calcular el promedio de una serie de cadenas de caracteres.

Volviendo En el caso  $k = 2$  tomamos de los dos primeros elementos distintos el que está en la posición 1, es decir el mayor de los dos, de manera que  $k = 2$  equivale a la estrategia propuesta en las secciones anteriores. El caso del balance perfecto (5.12) se obtiene tomando como pivote la mediana de todo el vector, es decir  $k = n$ .

Para una dada estrategia de elección del pivote podemos preguntarnos, cual es la probabilidad  $P(n, n_1)$  de que el pivote genere subparticiones de longitud  $n_1$  y  $n - n_1$ , con  $1 \leq n_1 < n$ . Asumiremos que los elementos del vector están distribuidos aleatoriamente. Si, por ejemplo, elegimos como pivote el primer elemento del vector (o sea la mediana de los primeros  $k = 1$  distintos), entonces al ordenar el vector este elemento puede terminar en cualquier posición del vector, ordenado de manera que  $P(n, n_1) = 1/(n - 1)$  para cualquier  $n_1 = 1, \dots, n - 1$ . Por supuesto, recordemos que esta no es una elección aceptable para el pivote en la práctica ya que no garantizaría que ambas particiones sean no nulas. Si el primer elemento resultara ser el menor de todos, entonces la partición izquierda resultaría ser nula. En la figura 5.6 vemos esta distribución de probabilidad. Para que la curva sea independiente de  $n$  hemos graficado  $nP(n, n_1)$  en función de  $n_1/n$ .

|     |     |     |     |
|-----|-----|-----|-----|
| $a$ | $b$ | $x$ | $x$ |
| $a$ | $x$ | $b$ | $x$ |
| $a$ | $x$ | $x$ | $b$ |
| $b$ | $a$ | $x$ | $x$ |
| $x$ | $a$ | $b$ | $x$ |
| $x$ | $a$ | $x$ | $b$ |
| $b$ | $x$ | $a$ | $x$ |
| $x$ | $b$ | $a$ | $x$ |
| $x$ | $x$ | $a$ | $b$ |
| $b$ | $x$ | $x$ | $a$ |
| $x$ | $b$ | $x$ | $a$ |
| $x$ | $x$ | $b$ | $a$ |

Figura 5.5: Posibilidades al ordenar un vector 5 elementos.  $a$  y  $b$  son los primeros dos elementos antes de ordenar.

Ahora consideremos la estrategia propuesta en las secciones anteriores, es decir el mayor de los dos primeros elementos distintos. Asumamos por simplicidad que todos los elementos son distintos. Sean  $a$  y  $b$  los dos primeros elementos del vector, entonces después de ordenar los elementos estos elementos pueden terminar en cualquiera de las posiciones  $j, k$  del vector con la misma probabilidad. Si la longitud del vector es  $n = 4$ , entonces hay  $n(n - 1) = 12$  posibilidades esencialmente distintas como se muestra en la figura 5.5.

Notemos que las primeras tres corresponden a que  $a$  termine en la posición 0 (base 0) y  $b$  en cada una de las tres posiciones restantes. Las siguientes 3 corresponden a  $a$  en la posición 1 (base 0), y así siguiendo. En el primero de los doce casos el pivote sería  $b$  y terminaría en la posición 1. Revisando todos los posibles casos tenemos que en 2 casos (líneas 1 y 4) el pivote termina en la posición 1, en 4 casos (líneas 2, 5, 7 y 8) termina en la posición 2 y en 6 casos (líneas 3, 6, 9, 10, 11 y 12) termina en la posición 3. Notemos que en este caso es más probable que el pivote termine en las posiciones más a la derecha que en las que están más a la izquierda. Por supuesto esto se debe a que estamos tomando el mayor de los dos. Una forma de contar estas posibilidades es considerar que para que el mayor esté en la posición  $j$  debe ocurrir que  $a$  esté en la posición  $j$  y  $b$  en las posiciones 0 a  $j - 1$ , o que  $b$  quede en la posición  $j$  y  $a$  en las posiciones 0 a  $j - 1$ , o sea un total de  $2j$  posibilidades.

Ahora veamos que ocurre si tomamos como estrategia para la elección del pivote la mediana de los primeros  $k = 3$  distintos. En este caso, si denotamos los tres primeros distintos como  $a, b$  y  $c$ , entonces

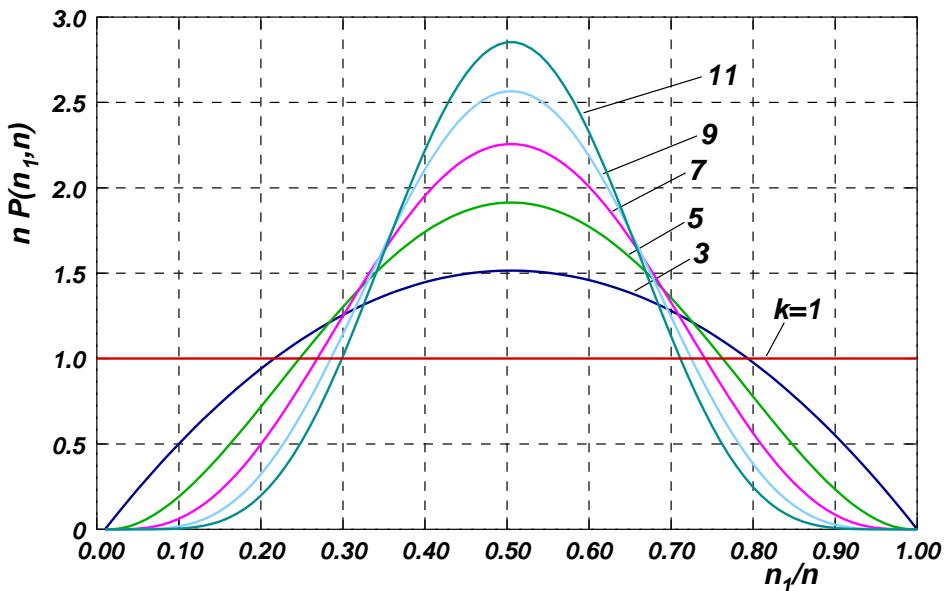


Figura 5.6: Probabilidad de que el pivote aparezca en una dada posición  $n_1$ , para diferentes valores de  $k$ .

existen  $n(n - 1)(n - 2)$  casos distintos:  $a$  en cualquiera de las  $n$  posiciones,  $b$  en cualquiera de las  $n - 1$  restantes y  $c$  en cualquiera de las  $n - 2$  restantes. Para que el pivote quede en la posición  $j$  debería ocurrir que, por ejemplo,  $a$  quede en la posición  $j$ ,  $b$  en una posición  $[0, j)$  y  $c$  en una posición  $(j, n)$ , o sea  $j(n-j-1)$  posibilidades. Las restantes posibilidades se obtienen por permutación de los elementos  $a$ ,  $b$  y  $c$ , en total

- $a$  en la posición  $j$ ,  $b$  en  $[0, j)$   $c$  en  $(j, n)$
- $a$  en la posición  $j$ ,  $c$  en  $[0, j)$   $b$  en  $(j, n)$
- $b$  en la posición  $j$ ,  $a$  en  $[0, j)$   $c$  en  $(j, n)$
- $b$  en la posición  $j$ ,  $c$  en  $[0, j)$   $a$  en  $(j, n)$
- $c$  en la posición  $j$ ,  $a$  en  $[0, j)$   $b$  en  $(j, n)$
- $c$  en la posición  $j$ ,  $b$  en  $[0, j)$   $a$  en  $(j, n)$

O sea que en total hay  $6j(n-j-1)$  posibilidades. Esto está graficado en la figura 5.6 como  $k = 3$ . Notemos que al tomar  $k = 3$  hay mayor probabilidad de que el pivote particione en forma más balanceada. En la figura se muestra la distribución de probabilidades para los  $k$  impares y se observa que a medida que crece  $k$  ha más certeza de que el pivote va a quedar en una posición cercana al centro de la partición.

#### 5.4.3. Tiempo de ejecución. Caso promedio.

Hemos obtenido el tiempo de ejecución en el mejor (5.12) y el peor (5.15) caso como casos particulares de (5.10). Conociendo la distribución de probabilidades para las diferentes particiones podemos obtener una expresión general para el caso promedio sumando sobre todas las posibles posiciones finales del pivote,

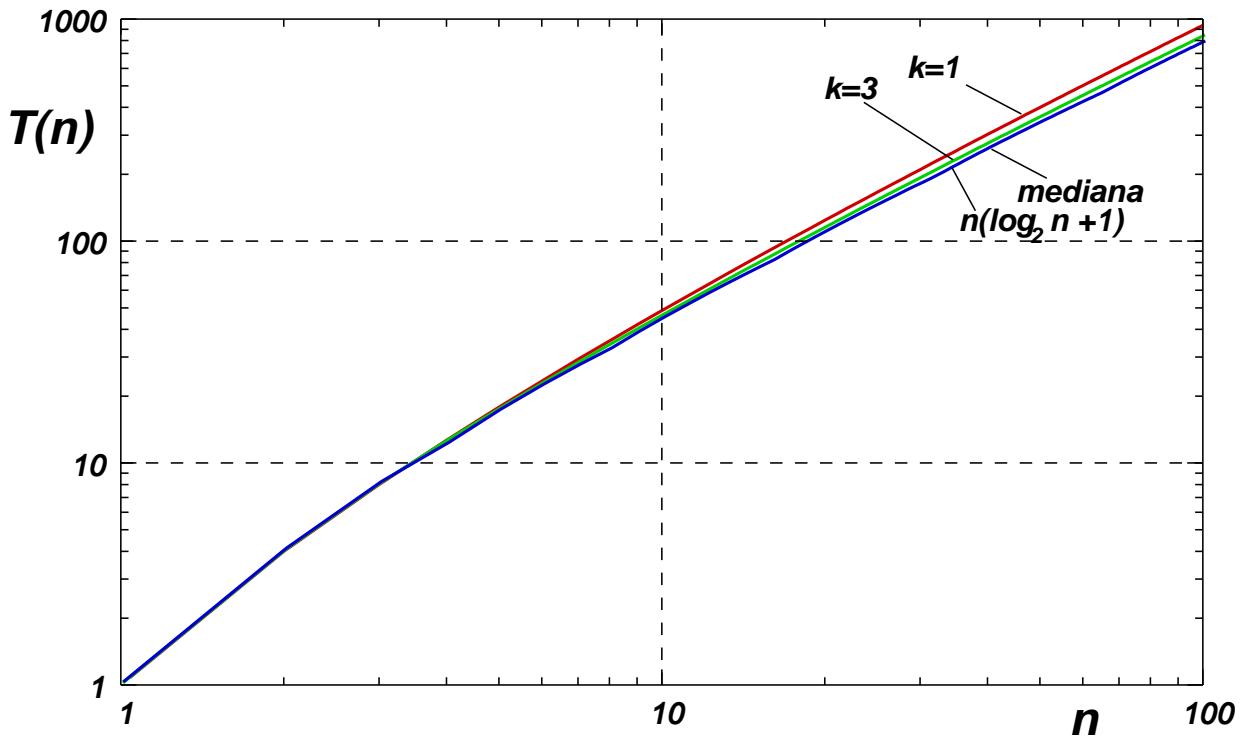


Figura 5.7:

desde  $j = 0$  hasta  $n - 1$  multiplicado por su correspondiente probabilidad

$$T(n) = cn + \sum_{n_1=1}^{n-1} P(n, n_1)(T(n_1) + T(n - n_1)). \quad (5.17)$$

Notar que en la suma para un  $n$  dado sólo aparecen los  $T(n_1)$  para  $n_1 < n$ , con lo cual puede fácilmente implementarse en un pequeño programa que va calculando los valores de  $T(n)$  para  $n = 1, 2, 3, \dots$ . En la figura 5.7 se observan los tiempos de ejecución así calculados para los casos  $k = 1$  y  $k = 3$  y también para el caso de elegir como pivote la mediana (equivale a  $k = n$ ). Para los cálculos, se ha tomado  $c = 1$  (en realidad puede verse que la velocidad de crecimiento no depende del valor de  $c$ , mientras sea  $c > 0$ ). También se graficó la función  $n(\log_2 n + 1)$  que corresponde al mejor caso (5.12), al menos para  $n = 2^p$ . Observamos que el tiempo de ejecución no presenta una gran dependencia con  $k$  y en todos los casos está muy cerca del mejor caso,  $O(n \log_2 n)$ . Esto demuestra (al menos “experimentalmente”) que en el caso promedio, e incluso para estrategias muy simples del pivote, el tiempo de ejecución en el caso promedio es

$$T_{\text{prom}}(n) = O(n \log_2 n) \quad (5.18)$$

Una demostración rigurosa de esto puede encontrarse en Aho et al. [1987].

#### 5.4.4. Dispersión de los tiempos de ejecución

Sin embargo, la estrategia en la elección del pivote (en este caso el valor de  $k$ ) sí tiene una incidencia notable en la “dispersión” de los valores de tiempos de ejecución, al menos para valores de  $k$  pequeños. Es decir, si bien para todos los valores de  $k$  el tiempo de ejecución promedio es  $O(n \log n)$  para valores de  $k$  altos, es más probable que la partición sea siempre balanceada, debido a que las campanas de probabilidad (ver figura 5.6) son cada vez más concentradas cerca de  $n_1 = n/2$ . De hecho, cuando tomamos como pivote la mediana ( $k = n$ ) el balanceo es siempre perfecto, a medida que reducimos el valor de  $k$  es más probable que para ciertas distribuciones de los elementos del vector los tiempos de ejecución sean más grande que promedio.

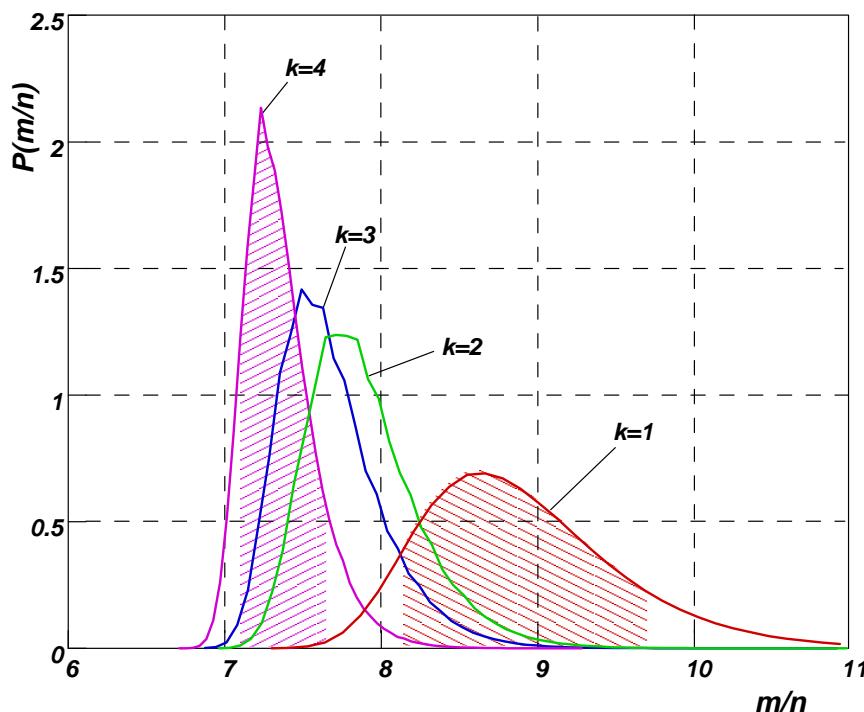


Figura 5.8: Dispersión de valores en el número de operaciones  $m$  al ordenar con quick-sort un vector generado aleatoriamente.

Para verificar esto realizamos un experimento en el que generamos un cierto número muy grande  $N$  (en este caso usamos concretamente  $N = 10^6$ ) de vectores de longitud  $n$  y contamos el número de operaciones  $m$  al aplicar quick-sort a ese vector. En realidad lo que contamos es la suma  $m$  del largo de las particiones que se van generando. En el caso de la figura 5.3, esta suma daría  $m = 30$  (no se cuenta la longitud del vector inicial, es decir sólo se suman las longitudes de las particiones rojas y verdes en la figura). En la figura 5.8 graficamos la probabilidad  $P(\xi)$  de ocurrencia de un dado valor de  $\xi = m/n$ . En el caso de la figura 5.3 tendríamos  $m = 30$  y  $n = 10$ , por lo tanto  $\xi = 30/10 = 3$ . La probabilidad  $P(\xi)$  se obtiene dividiendo el eje de las abscisas  $\xi$  en una serie de intervalos  $[\xi_i, \xi_{i+1}]$ , y contando para cada intervalo el número de vectores  $N_i$  (de los  $N$  simulados) cuyo valor de  $\xi$  cae en el intervalo. La probabilidad correspondiente al intervalo es

entonces

$$P(\xi) \approx \frac{N_i}{N}. \quad (5.19)$$

Haciendo tender el número de simulaciones en el experimento  $N$  a infinito el miembro derecho tiende a la probabilidad  $P(x)$ .

Basta con observar la gráfica para ver que a medida que  $k$  se incrementa la distribución de los valores es más concentrada, resultando en una campana más delgada y puntiaguda. Esto se puede cuantificar buscando cuáles son los valores de  $\xi$  que delimitan el 80 % de los valores centrales. Por ejemplo, se observa que para el valor más bajo  $k = 1$  el 80 % de los valores está entre  $\xi = 8.1$  y  $9.8$  (ancho de la campana 1.7), mientras que para  $k = 4$  el 80 % de los valores está entre  $\xi = 7.1$  y  $7.65$  (ancho de la campana 0.55). En la figura se muestran sombreadas las áreas que representan el 80 % central de los valores para  $k = 1$  y  $k = 4$ .

#### 5.4.5. Elección aleatoria del pivote

Vimos que tomando la mediana de los primeros  $k$  valores podemos hacer que cada vez sea menos probable obtener tiempos de ejecución demasiado altos, o mejor dicho, mucho más altos que el promedio *si el vector está inicialmente desordenado*. Pero si el vector está inicialmente ordenado, entonces en todos los niveles la partición va a ser desbalanceada independientemente del valor de  $k$ . El problema es que la situación en que el vector tiene cierto grado de ordenamiento previo es relativamente común, pensemos por ejemplo en un vector que está inicialmente ordenado y se le hacen un pequeño número de operaciones, como insertar o eliminar elementos, o permutar algunos de ellos. Cuando queremos volver a ordenar el vector estaríamos en un caso bastante cercano al peor.

Para decirlo en forma más rigurosa, cuando calculamos el tiempo promedio asumimos que todas las posibles permutaciones de los elementos del vector son igualmente probables, lo cual es cierto si, por ejemplo, generamos el vector tomando los elementos con un generador aleatorio. En la práctica es común que secuencias donde los elementos estén parcialmente ordenados sean más frecuentes y es malo que éste sea justo el peor caso. Una solución puede ser “desordenar” inicialmente el vector, aplicando un algoritmo como el `random_shuffle()` de STL [SGI, 1999]. La implementación de `random_shuffle()` es  $O(n)$ , con lo cual no cambia la tasa de crecimiento. Sin embargo podría poner a quick-sort en desventaja con otros algoritmos que también son  $O(n \log n)$  como `heap_sort()`. Otro inconveniente es que el `random_shuffle()` inicial haría prácticamente imposible una implementación estable del ordenamiento.

#### 5.4.6. El algoritmo de partición

Una implementación eficiente del algoritmo de partición es clave para la eficiencia de quick-sort. Mantenemos dos cursores `l` y `r` de tal manera que todos los elementos a la izquierda de `l` son estrictamente menores que el pivote `v` y los que están a la derecha de `r` son mayores o iguales que `v`. Los elementos a la izquierda de `l` son la partición izquierda que va a ir creciendo durante el algoritmo de partición, y lo mismo para `r`, *mutatis mutandis*. Inicialmente podemos poner `l=first` y `r=last-1`, ya que corresponde a que ambas particiones sean inicialmente nulas. A partir de allí vamos aplicando un proceso iterativo que se muestra en el seudocódigo 5.10

---

```

1 int partition(w,first,last,v) {
2 // Particiona el rango [j1,j2] de 'w'
```

```

3 // con respecto al pivote 'v'
4 if (n==1) return (w[first]<v ? first : last);
5 int middle = (first+last)/2;
6 l1 = partition(w,first,middle,v);
7 l2 = partition(w,middle,last,v);
8 // Intercambia [l1,middle] con [middle,l2]
9 swap(l1,middle,l2);
10 }

```

**Código 5.10:** Seudocódigo para el algoritmo de particionamiento. [Archivo: partsc.cpp]

Avanzar **l** lo más a la derecha posible significa avanzar **l** hasta encontrar un elemento mayor o igual que **v**. Notar que intercambiar los elementos, garantiza que en la siguiente ejecución del lazo cada uno de los cursores **l** y **r** avanzarán al menos una posición ya que, después de intercambiar, el elemento en **l** será menor que **v** y el que está en **r** será mayor o igual que **v**. El algoritmo termina cuando **l** y **r** se “cruzan”, es decir cuando **l>r**. En ese caso **l** representa el primer elemento de la partición derecha, el cual debe ser retornado por **partition()** para ser usado en quick-sort.

```

1 template<class T>
2 typename std::vector<T>::iterator
3 partition(typename std::vector<T>::iterator first,
4 typename std::vector<T>::iterator last,
5 bool (*comp)(T&, T&), T &pivot) {
6 typename std::vector<T>::iterator
7 l = first,
8 r = last;
9 r--;
10 while (true) {
11 T tmp = *l;
12 *l = *r;
13 *r = tmp;
14 while (comp(*l,pivot)) l++;
15 while (!comp(*r,pivot)) r--;
16 if (l>r) break;
17 }
18 return l;
19 }

```

**Código 5.11:** Algoritmo de partición para quick-sort. [Archivo: qspart.h]

#### 5.4.7. Tiempo de ejecución del algoritmo de particionamiento

En el código 5.11 vemos una implementación de **partition()**. Recordemos que para que las estimaciones del tiempo de ejecución de quick-sort sean válidas, en particular (5.14) y (5.18), debe valer (5.11), es decir que el tiempo de particionamiento sea lineal. El tiempo de particionamiento es el del lazo de las líneas 10–17, es decir la suma de los lazos de avances de **l** y retrocesos de **r** más los intercambios. Ahora

bien, en cada avance de **l** y retroceso de **r** la longitud del rango **[l,r]** que es la que todavía falta particionar se reduce en uno, de manera que a lo sumo puede haber  $n = \text{last} - \text{first}$  avances y retrocesos. Por otra parte, por cada intercambio debe haber al menos un avance de **l** y un retroceso de **r** de manera que hay a lo sumo  $n/2$  intercambios, con lo cual se demuestra que todo el tiempo de **partition()** es a lo sumo  $O(n)$ .

#### 5.4.8. Búsqueda del pivote por la mediana

```

1 template<class T>
2 int median(typename std::vector<T>::iterator first,
3 typename std::vector<T>::iterator last,
4 std::vector<T> &dif, int k,
5 bool (*comp)(T&, T&)) {
6 typename std::vector<T>::iterator
7 q = first;
8 int ndif=1;
9 dif[0] = *q++;
10 while (q<last) {
11 T val = *q++;
12 int j;
13 for (j=0; j<ndif; j++)
14 // Aca debe compararse por 'equivalente'
15 // es decir usando comp
16 if (!comp(dif[j],val)
17 && !comp(val,dif[j])) break;
18 if (j==ndif) {
19 dif[j] = val;
20 ndif++;
21 if (ndif==k) break;
22 }
23 }
24 typename std::vector<T>::iterator
25 s = dif.begin();
26 bubble_sort(s,s+ndif,comp);
27 return ndif;
28 }
```

Código 5.12: Función para calcular el pivote en quick-sort. [Archivo: qsmedian.h]

En el código 5.12 podemos ver una implementación de la función que calcula el pivote usando la estrategia de la mediana de los  $k$  primeros. La función **ndif=median(first,last,dif,k,comp)** busca los  $k$  primeros elementos distintos del rango **[first,last]** en el vector **dif** y retorna el número exacto de elementos distintos encontrados (que puede ser menor que **k**). Recordemos que el número de elementos distintos es usado en quick-sort para cortar la recursión. Si no hay al menos dos elementos distintos entonces no es necesario particionar el rango.

El algoritmo es  $O(n)$  mientras que  $k$  sea fijo, esto es, que no crezca con  $n$ . Se recorre el rango y se van introduciendo los nuevos elementos distintos en el vector **dif**. Para ver si un elemento es distinto se compara con todos los elementos previamente insertados en **dif**. Es muy importante que la comparación debe

realizarse por *equivalencia* (ver línea 17), y no por *igualdad*. Es decir dos elementos  $a$  y  $b$  son equivalentes si `comp(a,b) && comp(b,a)` es verdadero. Para relaciones de orden débiles esto es muy importante ya que si todos los elementos en el rango son equivalentes pero no iguales (pensemos en  $(-1, 1, -1)$  con la relación de orden (5.3), menor en valor absoluto), entonces si `partition()` comparara por igualdad reportaría dos elementos distintos, pero después al particionar una de las particiones resultaría vacía y entraría en un lazo infinito.

#### 5.4.9. Implementación de quick-sort

```

1 template<class T> void
2 quick_sort(typename std::vector<T>::iterator first,
3 typename std::vector<T>::iterator last,
4 bool (*comp)(T&,T&)) {
5 int size = last-first;
6 int max_bub_size = 9;
7 if (size<max_bub_size) {
8 bubble_sort(first,last,comp);
9 return;
10 }
11 if (size<=1) return;
12 int k=3;
13 std::vector<T> dif(k);
14 int ndif = median(first, last, dif, k, comp);
15 if (ndif==1) return;
16 T pivot = dif[ndif/2];
17 typename std::vector<T>::iterator l;
18 l = partition(first, last, comp, pivot);
19 quick_sort(first,l,comp);
20 quick_sort(l,last,comp);
21 }
22
23 template<class T> void
24 quick_sort(typename std::vector<T>::iterator first,
25 typename std::vector<T>::iterator last) {
26 quick_sort(first,last,less<T>);
27 }
```

Código 5.13: Algoritmo de ordenamiento rápido (quick-sort) [Archivo: qsort.h]

En el código 5.13 vemos la implementación de la rutina principal de quick-sort.

- Una mejora para quick-sort consiste en usar para las particiones más pequeñas otro algoritmo, por ejemplo el método de la burbuja. Recordemos que si bien la tasa de crecimiento  $O(n^2)$  nos indica que para valores grandes de  $n$  burbuja será siempre más ineficiente que quick-sort, para valores pequeños puede ser más eficiente y de hecho lo es. Por eso, para longitudes de rangos menores que `max_bub_size` la función llama a `bubble_sort()` en vez de llamar recursivamente a `quick_sort()`. La constante `max_bub_size` óptima se halla por prueba y error y en nuestro caso se ha elegido `max_bub_size=9`.

- Notemos que en este caso `quick_sort()` corta la recursión de varias formas posibles, a saber:
  - Cuando cambia a `bubble_sort()` como se mencionó en el punto anterior.
  - Cuando la longitud del rango es 1. Esto sólo puede ocurrir si se elige `max_bub_size=0` para evitar cambiar a llamar a `bubble_sort()` (por la razón que fuere).
  - Cuando no se detectan dos o más elementos distintos (mejor dicho, no equivalentes).
- Las últimas tres líneas de `quick_sort()` simplemente reflejan el seudocódigo 5.9.

#### 5.4.10. Estabilidad

Quick-sort es estable si el algoritmo de partición lo es, y tal cual como está implementado aquí, el algoritmo de *partición no es estable*, ya que al hacer el intercambio en `partition()` un elemento puede ser intercambiado con un elemento equivalente.

```

1 template<class T>
2 typename std::vector<T>::iterator
3 stable_partition(typename std::vector<T>::iterator first,
4 typename std::vector<T>::iterator last,
5 bool (*comp)(T&, T&), T &pivot) {
6 int size = (last-first);
7 if (size==1) return (comp(*first,pivot)? last : first);
8 typename std::vector<T>::iterator
9 middle = first + size/2,
10 l1, l2;
11 l1 = stable_partition(first,middle,comp,pivot);
12 l2 = stable_partition(middle,last,comp,pivot);
13 range_swap<T>(l1,middle,l2);
14 return l1+(l2-middle);
15 }
```

Código 5.14: Seudocódigo para el algoritmo de partición estable. [Archivo: stabpart.h]

Es sencillo implementar una variante estable de `partition()` como se observa en el seudo código 5.14. El algoritmo de partición es recursivo. Primero dividimos el rango `[first, last)` por el punto medio `middle`. Aplicamos recursivamente `partition()` a cada una de los rangos izquierdo (`[first, middle)`) y derecho (`[middle, last)`) retornando los puntos de separación de cada una de ambas particiones `l1` y `l2`. Una vez que ambos rangos están particionados sólo hace falta intercambiar (“swap”) los rangos `[l1, middle)` y `[middle, l2)`. Si el particionamiento de ambos subrangos fue estable y al hacer el swap mantenemos el orden relativo de los elementos en cada uno de los rangos, entonces la partición de `[first, last)` será estable, ya que los elementos de `[middle, l2)` son estrictamente mayores que los de `[l1, middle)`.

#### 5.4.11. El algoritmo de intercambio (swap)

Llamaremos a esta operación `swap(l1, middle, l2)`. Notemos que a cada posición `l1+k1` en el rango `[l1, l2)` le corresponde, después del intercambio, otra posición `l1+k2` en el rango `[l1, l2)`. La relación

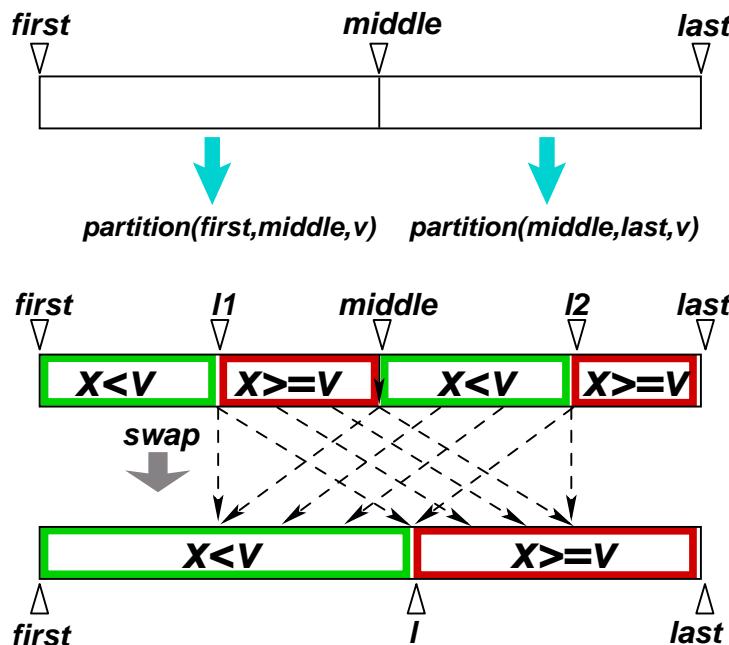


Figura 5.9: Algoritmo de partición estable.

entre  $k_1$  y  $k_2$  es biunívoca, es decir, la operación de intercambio es una “permutación” de los elementos del rango. Si llamamos  $n_1$  y  $n_2$  las longitudes de cada uno de los rangos a intercambiar, entonces tenemos

$$k_2 = \begin{cases} k_1 + n_2; & \text{si } k_1 < n_1 \\ k_1 - n_1; & \text{si } k_1 \geq n_1 \end{cases} \quad (5.20)$$

o recíprocamente,

$$k_1 = \begin{cases} k_2 + n_1; & \text{si } k_2 < n_2 \\ k_2 - n_2; & \text{si } k_2 \geq n_2 \end{cases} \quad (5.21)$$

Para describir el algoritmo es más simple pensar que tenemos los elementos del rango  $[11, 12)$  en un vector  $w$  de longitud  $n_1+n_2$ . Consideremos por ejemplo el caso  $n_1 = 4$ ,  $n_2 = 6$  (ver figura 5.11). De acuerdo con (5.21) el elemento que debe ir a la primera posición es el que esta en la posición 4. Podemos guardar el primer elemento (posición 0) en una variable temporal  $\text{tmp}$  y traer el 4 a la posición 0. A su vez podemos poner en 4 el que corresponde allí, que está inicialmente en la posición 8 y así siguiendo se desencadenan una serie de intercambios hasta que el que corresponde poner en la posición a rellenar es el que tenemos guardado en la variable temporal (por ahora el 0).

```

1 T tmp = w[0];
2 int k2 = 0;
3 while (true) {
4 int k1 = (k2<n2 ? k2+n1 : k2-n2);
5 if (k1==0) break;
6 w[k2] = w[k1];

```

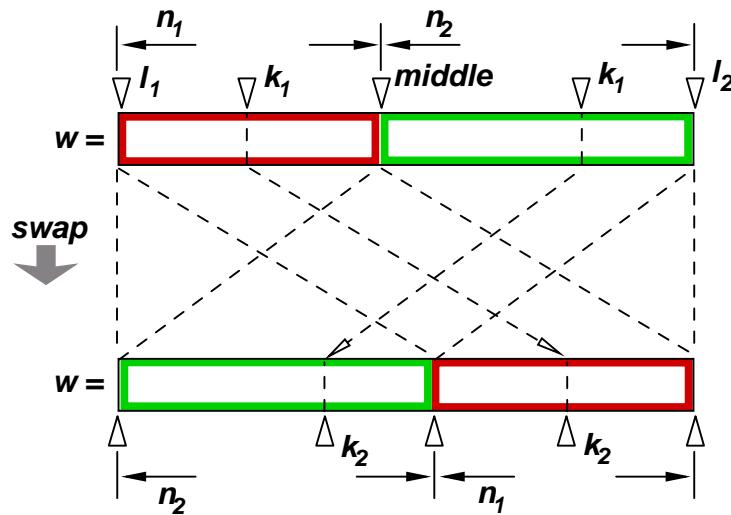


Figura 5.10:

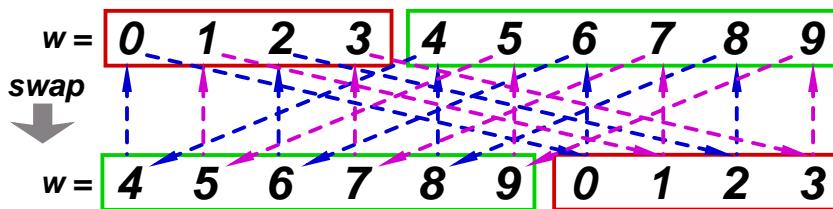


Figura 5.11: Algoritmo de intercambio de rangos.

```

7 k2 = k1;
8 }
9 w[k2] = tmp;
```

**Código 5.15:** Algoritmo de rotación para el swap de rangos. [Archivo: swapsc.cpp]

En el código 5.15 vemos como sería el algoritmo de rotación. En el ejemplo, los intercambios producidos serían

$$\text{tmp} \leftarrow w[0] \leftarrow w[4] \leftarrow w[8] \leftarrow w[2] \leftarrow w[6] \leftarrow \text{tmp} \quad (5.22)$$

Esta rotación de los elementos se muestra en la figura con flechas azules. Podemos ver que esto rota 5 elementos, los otros 5 rotan de la misma forma si comenzamos guardando en `tmp` el elemento de la posición 1, trayendo al 1 el de la posición 5, y así siguiendo

$$\text{tmp} \leftarrow w[1] \leftarrow w[5] \leftarrow w[9] \leftarrow w[3] \leftarrow w[7] \leftarrow \text{tmp} \quad (5.23)$$

Notar que los elementos que se rotan son exactamente los que fueron rotados previamente, incrementados en uno.

Si  $n_1 = n_2$  (por ejemplo  $n_1 = n_2 = 5$ ), entonces hay cinco rotaciones de dos elementos,

$$\begin{aligned} \text{tmp} &\leftarrow w[0] \leftarrow w[5] \leftarrow \text{tmp} \\ \text{tmp} &\leftarrow w[1] \leftarrow w[6] \leftarrow \text{tmp} \\ \text{tmp} &\leftarrow w[2] \leftarrow w[7] \leftarrow \text{tmp} \\ \text{tmp} &\leftarrow w[3] \leftarrow w[8] \leftarrow \text{tmp} \\ \text{tmp} &\leftarrow w[4] \leftarrow w[9] \leftarrow \text{tmp} \end{aligned} \tag{5.24}$$

Si  $n_1$  divide a  $n_2$  (por ejemplo  $n_1 = 2$  y  $n_2 = 8$ ), entonces se generan 2 rotaciones de 5 elementos, a saber

$$\begin{aligned} \text{tmp} &\leftarrow w[0] \leftarrow w[2] \leftarrow w[4] \leftarrow w[6] \leftarrow w[8] \leftarrow \text{tmp} \\ \text{tmp} &\leftarrow w[1] \leftarrow w[3] \leftarrow w[5] \leftarrow w[7] \leftarrow w[9] \leftarrow \text{tmp} \end{aligned} \tag{5.25}$$

Observando con detenimiento podemos encontrar una regla general, a saber que el número de rotaciones es  $m = \gcd(n_1, n_2)$ , donde  $\gcd(n_1, n_2)$  es el máximo común divisor de  $n_1$  y  $n_2$ . Además, como el número de rotaciones por el número de elementos rotados en cada rotación debe ser igual al número total de elementos  $n_1 + n_2$  debemos tener que el número de elementos rotados en cada rotación es  $(n_1 + n_2)/m$ . Las rotaciones empiezan en los elementos 0 a  $m - 1$ .

---

```

1 int gcd(int m, int n) {
2 int M, N;
3 if (m>n) {
4 M=m; N=n;
5 } else {
6 N=m; M=n;
7 }
8 while (true) {
9 int rest = M% N;
10 if (!rest) return N;
11 M = N; N= rest;
12 }
13 }
14
15 template<class T>
16 void range_swap(typename std::vector<T>::iterator first,
17 typename std::vector<T>::iterator middle,
18 typename std::vector<T>::iterator last) {
19
20 int n1 = middle-first,
21 n2 = last-middle;
22 if (!n1 || !n2) return;
23 int m = gcd(n1,n2);
24 for (int j=0; j<m; j++) {
25 T tmp = *(first+j);
26 int k2 = j;
27 while (true) {
28 int k1 = (k2<n2 ? k2+n1 : k2-n2);
29 if (k1==j) break;
30 *(first+k2) = *(first+k1);
31 k2 = k1;
32 }
33 }
```

```

32 }
33 *(first+k2) = tmp;
34 }
35 }
```

**Código 5.16:** Algoritmo de intercambio de dos rangos consecutivos. [Archivo: swap.h]

El algoritmo de particionamiento estable se observa en el código 5.16. La función `int gcd(int, int)` calcula el máximo común divisor usando el algoritmo de Euclides (ver sección §4.1.3). El tiempo de ejecución de `swap()` es  $O(n)$ , donde  $n$ , ya que en cada ejecución del lazo un elemento va a su posición final.

Si reemplazamos en quick-sort (código 5.13) la función `partition()` (código 5.11) por `stable_partition()` (código 5.14) el algoritmo se hace estable.

#### 5.4.12. Tiempo de ejecución del quick-sort estable

Sin embargo el algoritmo de particionamiento estable propuesto ya no es  $O(n)$ . De hecho el tiempo de ejecución de `stable_partition()` se puede analizar de la misma forma que el de quick-sort mismo en el mejor caso. Es decir, su tiempo de ejecución satisface una relación recursiva como (5.12) donde el tiempo  $cn$  ahora es el tiempo de `swap()`. Por lo tanto el tiempo de ejecución de `stable_partition()` es  $O(n \log n)$  en el peor caso.

Ahora para obtener el tiempo de ejecución de la versión estable de quick-sort en el mejor caso volvemos a (5.12) teniendo en cuenta la nueva estimación para el algoritmo de partición

$$T(n) = n \log n + 2T(n/2) \quad (5.26)$$

Ambas (5.26) y (5.12) son casos especiales de una relación de recurrencia general que surge naturalmente al considerar algoritmos de tipo “dividir para vencer”

$$T(n) = f(n) + 2T(n/2), \quad (5.27)$$

donde  $f(n) = n \log n$  en el caso de (5.26) y  $f(n) = cn$  para (5.12). Aplicando recursivamente obtenemos

$$\begin{aligned} T(2) &= f(2) + 2T(1) = f(2) + 2d \\ T(4) &= f(4) + 2T(2) = f(4) + 2f(2) + 4d \\ T(8) &= f(8) + 2T(4) = f(8) + 2f(4) + 4f(2) + 8d \\ &\vdots = \vdots \end{aligned} \quad (5.28)$$

Si  $f(n)$  es una función “cóncava hacia arriba” (ver figura 5.12, izquierda) entonces es válido que

$$2f(n/2) \leq f(n), \quad (5.29)$$

mientras que si es cóncava hacia abajo (ver figura 5.12, derecha) entonces

$$2f(n/2) \geq f(n). \quad (5.30)$$

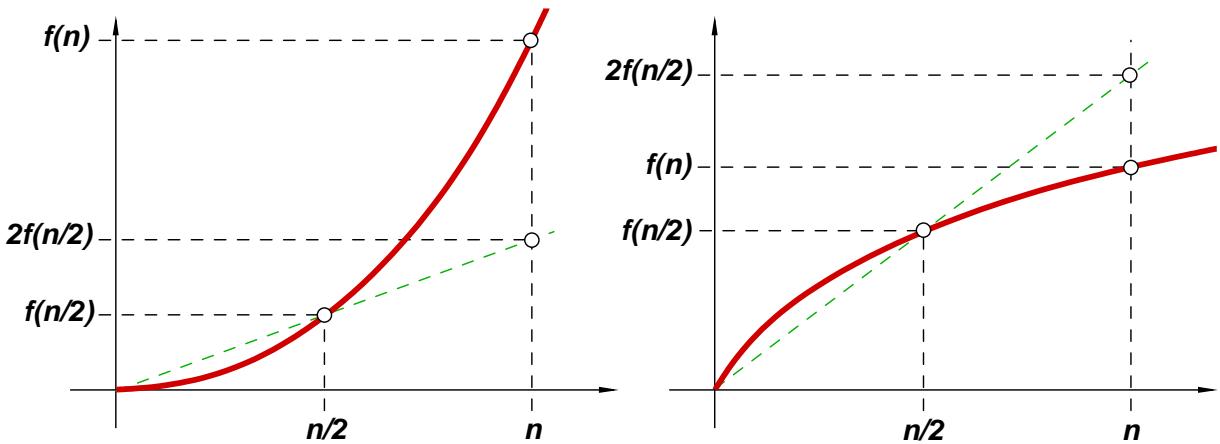


Figura 5.12: Ejemplo de crecimiento “más que lineal” (izquierda) y menos que lineal (derecha)

También decimos que la función tiene crecimiento “más que lineal” o “menos que lineal”, respectivamente. Si la función crece más que linealmente, como en el caso de  $n \log n$ , entonces podemos acotar

$$\begin{aligned} 2f(2) &\leq f(4) \\ 2f(4) &\leq f(8) \\ 4f(2) &\leq 2f(4) \leq f(8) \\ &\vdots \quad \vdots \end{aligned} \tag{5.31}$$

de manera que

$$\begin{aligned} T(8) &\leq 3f(8) + 8d \\ T(16) &= f(16) + 2T(8) \leq f(16) + 6f(8) + 16d \leq 4f(16) + 16d \\ &\vdots \quad \vdots \end{aligned} \tag{5.32}$$

y, en general

$$T(n) \leq (\log n) f(n) + nd. \tag{5.33}$$

Si lo aplicamos a quick-sort estable con  $f(n) = n \log n$  llegamos a

$$T(n) = O(n (\log n)^2) \tag{5.34}$$

## 5.5. Ordenamiento por montículos

---

```

1 // Fase inicial
2 // Pone todos los elementos en S
3 while (!L.empty()) {
4 x = *L.begin();
5 S.insert(x);
6 L.erase(L.begin());

```

```

7 }
8 // Fase final
9 // Saca los elementos de S usando 'min'
10 while (!S.empty()) {
11 x = *S.begin();
12 S.erase(S.begin());
13 L.push(L.end(), x);

```

**Código 5.17:** Algoritmo de ordenamiento usando un conjunto auxiliar. [Archivo: heapsortsc.cpp]

Podemos ordenar elementos usando un **set** de STL ya que al recorrer los elementos con iteradores estos están guardados en forma ordenada. Si logramos una implementación de **set** tal que la inserción y supresión de elementos sea  $O(\log n)$ , entonces basta con insertar los elementos a ordenar y después extraerlos recorriendo el conjunto con **begin()** y **operator++()**. Si, por ejemplo, asumimos que los elementos están en una lista, el algoritmo sería como se muestra en el seudocódigo 5.17.

- La representación por *árboles binarios de búsqueda* (ver sección §4.6.5) sería en principio válida, aunque si los elementos no son insertados en forma apropiada el costo de las inserciones o supresiones puede llegar a  $O(n)$ . Hasta ahora no hemos discutido ninguna implementación de **set** en el cual inserciones y supresiones sean  $O(\log n)$  *siempre*.
- Una desventaja es que el conjunto no acepta elementos diferentes, de manera que esto serviría sólo para ordenar contenedores con elementos diferentes. Esto se podría remediar usando un **multiset** [SGI, 1999].
- Esta forma de ordenar no es *in-place* y de hecho las representaciones de **set** requieren una considerable cantidad de memoria adicional por elemento almacenado.

### 5.5.1. El montículo

El “*montículo*” (“*heap*”) es una estructura de datos que permite representar en forma muy conveniente un TAD similar al conjunto llamado “*cola de prioridad*”. La cola de prioridad difiere del conjunto en que no tiene las operaciones binarias ni tampoco operaciones para recorrer el contenedor como **end()** y **operator++()**. Si tiene una función **min()** que devuelve una posición al menor elemento del conjunto, y por lo tanto es equivalente a **begin()**.

El montículo representa la cola de prioridad almacenando los elementos en un árbol binario con las siguientes características

- Es “*parcialmente ordenado*” (PO), es decir el padre es siempre menor o igual que sus dos hijos.
- Es “*parcialmente completo*”: Todos los niveles están ocupados, menos el último nivel, en el cual están ocupados todos los lugares más a la izquierda.

En la figura 5.13 vemos tres árboles binarios de los cuales sólo el de más a la izquierda cumple con todas las condiciones de montículo. El del centro no cumple con la condición de PO ya que los elementos 22 y 13 (marcados en rojo) son menores que sus padres. Por otra parte, el de la derecha satisface la condición de PO pero no es parcialmente completo debido a los nodos marcados como  $\Lambda$ .

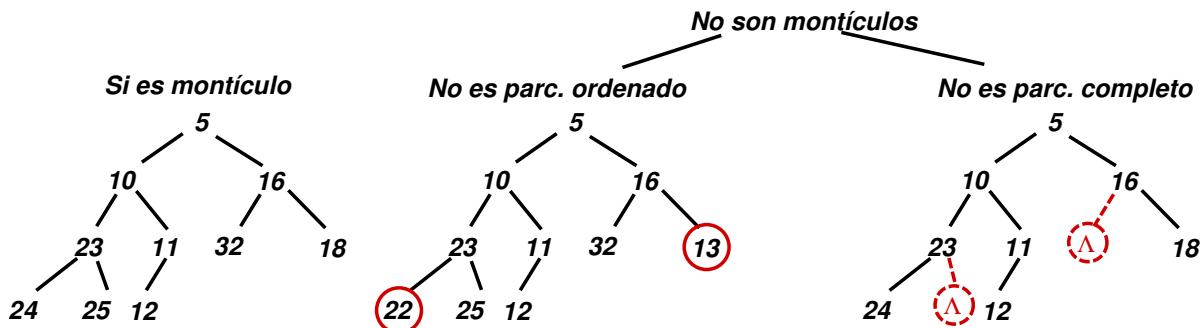


Figura 5.13: Ejemplo de árboles que cumplen y no cumplen la condición de montículo.

Notemos que la propiedad de PO es recursiva. Podemos decir que un árbol binario es PO si la raíz es menor que sus hijos (es decir, es *localmente* PO) y los subárboles de sus hijos son PO. Por otra parte la propiedad de parcialmente completo *no es recursiva*.

### 5.5.2. Propiedades

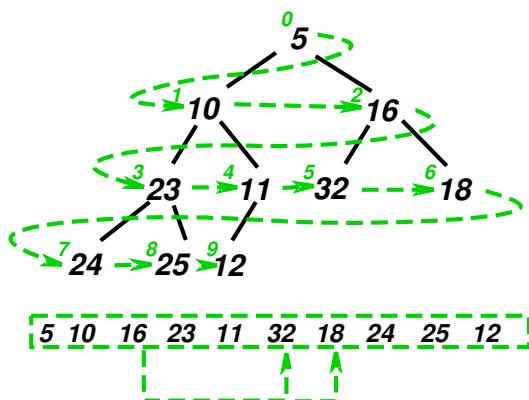


Figura 5.14:

La condición de que sea PO implica que el mínimo está siempre en la raíz. La condición de parcialmente completo permite implementarlo eficientemente en un vector. Los elementos son almacenados en el vector *por orden de nivel*, es decir primero la raíz, en la posición 0 del vector, después los dos hijos de la raíz de izquierda a derecha en las posiciones 1 y 2, después los 4 elementos del nivel 2 y así siguiendo. Esto se representa en la figura 5.14 donde los números en verde representan la posición en el vector. Notemos que las posiciones de los hijos se pueden calcular en forma algebraica a partir de la posición del padre

$$\begin{aligned} \text{hijo izquierdo de } j &= 2j + 1, \\ \text{hijo derecho de } j &= 2j + 2. \end{aligned} \tag{5.35}$$

Notemos también que cada subárbol del montículo es a su vez un montículo.

### 5.5.3. Inserción

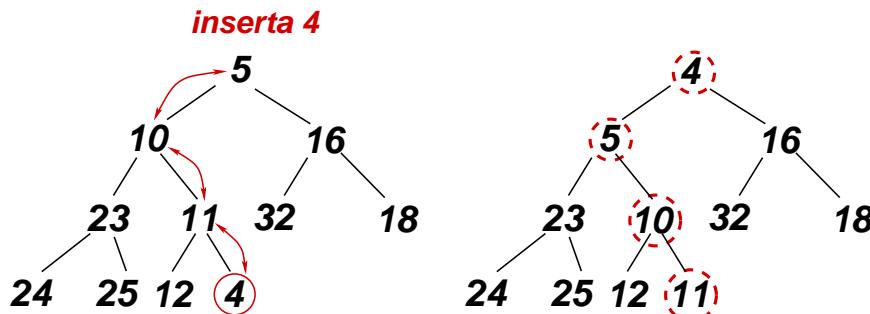


Figura 5.15: Inserción en un montículo.

Para poder usar el montículo para poder ordenar debemos poder insertar nuevos elementos, manteniendo la propiedad de montículo. El procedimiento consiste en insertar inicialmente el elemento en la *primera posición libre*, es decir la posición libre lo más a la izquierda posible del último nivel semicompleto o, si no hay ningún nivel semicompleto, la primera posición a la izquierda del primer nivel vacío. En la figura 5.15 vemos un ejemplo en el cual insertamos el elemento 4 en un montículo que hasta ese momento contiene 10 elementos. Una vez así insertado el elemento, se cumple la condición de parcialmente completo pero probablemente no la de PO. Esta última se restituye haciendo una serie de intercambios. Notar que los intercambios de elementos no pueden quebrar la propiedad de parcialmente completo.

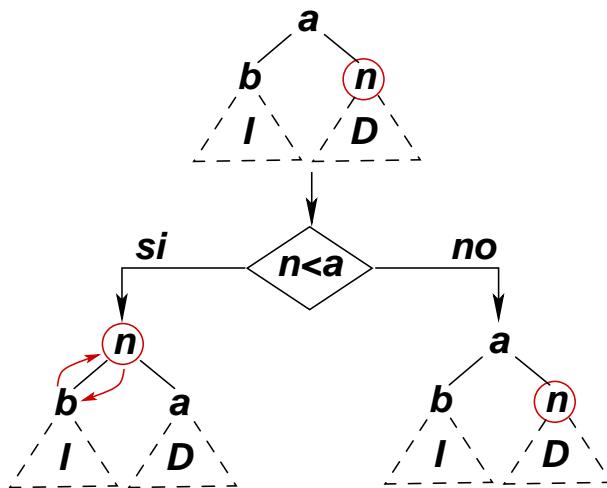


Figura 5.16: Esquema del restablecimiento de la propiedad de montículo al subir el nuevo elemento.

Para restituir la propiedad de PO vamos intercambiando el elemento insertado con su padre, si éste es estrictamente mayor. El proceso se detiene al encontrar un padre que no es mayor o equivalente al elemento. En cierta forma, este procedimiento es similar al que usa el método de la burbuja para ir “flotando” cada uno de los elementos en el lazo interno, con la salvedad que aquí el proceso se realiza sobre el camino que va desde el nuevo elemento a la raíz. Por supuesto, en el montículo pueden existir elementos iguales (en

realidad equivalentes), es decir *no es un conjunto*. Cada vez que se intercambia el elemento con su padre el subárbol de la nueva posición donde va el elemento recupera la propiedad de montículo. Consideremos por ejemplo el caso de la figura 5.16 donde el nuevo elemento  $n$  ha subido hasta ser raíz del subárbol  $D$ . Queremos ver como se restituye la propiedad de montículo al intercambiar (o no)  $n$  con su padre  $a$ . El caso en que el nuevo elemento es raíz del subárbol izquierdo es exactamente igual, *mutatis mutandis*. Notemos que el subárbol  $I$  del hijo izquierdo de  $a$  también debe ser un montículo ya que no fue tocado durante el proceso de subir  $n$  hasta su posición actual (todo el camino por donde subió  $n$  debe estar contenido dentro de  $D$ ). Ahora supongamos que  $a \leq n$ , entonces no intercambiaremos  $a$  con  $n$  como se muestra en la parte inferior derecha de la figura. Es claro que se verifica localmente la condición de PO y como a su vez cada uno de los subárboles  $I$  y  $D$  son PO, todo el subárbol de  $a$  es PO. Si por otra parte  $a > n$  (en la parte inferior izquierda de la figura) entonces al intercambiar  $a$  con  $n$  es claro que  $D$  quedará como un montículo, ya que hemos reemplazado la raíz por un elemento todavía menor. Por otra parte  $I$  ya era un montículo y lo seguirá siendo porque no fue modificado. Finalmente la condición de PO se satisface localmente entre  $n$ ,  $b$  y  $a$  ya que  $n < a$  y como el la condición se satisfacía localmente antes de que subiera  $n$ , debía ser  $a \leq b$ , por lo tanto  $n < a \leq b$ .

#### 5.5.4. Costo de la inserción

Notemos que cada intercambio es  $O(1)$  por lo tanto todo el costo de la inserción es básicamente orden de la longitud  $l$  del camino que debe subir el nuevo elemento desde la nueva posición inicial hasta algún punto, eventualmente en el peor de los casos hasta la raíz. Pero como se trata de un árbol parcialmente completo la longitud de los caminos contenidos en el árbol está acotada por (ver sección §3.8.3)

$$T(n) = O(l) = O(\log_2 n). \quad (5.36)$$

Notemos que esta estimación es válida *en el peor caso*. A diferencia del ABB (ver sección §4.6), el montículo no sufre de problemas de “balanceo” ya que siempre es mantenido en un árbol parcialmente completo.

Este algoritmo de inserción permite implementar la línea 5 en el seudocódigo 5.17 en  $O(n \log_2 n)$ .

#### 5.5.5. Eliminar el mínimo. Re-heap.

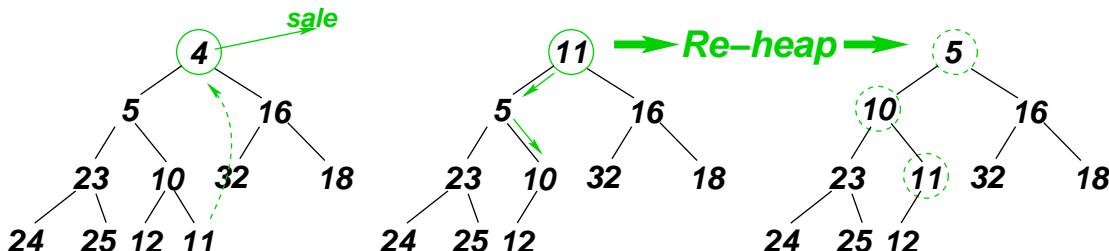


Figura 5.17: Procedimiento de eliminar el elemento mínimo en montículos.

Ahora vamos a implementar las líneas 11–12, esto es una operación combinada en la cual se recupera el valor del mínimo y se elimina este valor del contenedor. En el montículo el elemento menor se encuentra en la raíz, por construcción, es decir en la primera posición del vector. Para eliminar el elemento debemos

rellenar el “hueco” con un elemento. Para mantener la propiedad de parcialmente completo, subimos a la raíz el último elemento (el más a la derecha) del último nivel semicompleto. Por ejemplo, considerando el montículo inicial de la figura 5.17 a la izquierda, entonces al eliminar el 4 de la raíz, inmediatamente subimos el 11 para “rellenar” el hueco creado, quedando el árbol como se muestra en la figura, en el centro. Este árbol satisface todas las condiciones de montículo menos localmente la condición PO en la raíz. Al proceso de realizar una serie de intercambios para restaurar la condición de montículo en un árbol en el cual la única condición que se viola es la de PO (localmente) en la raíz se le llama “rehacer el montículo” (“re-heap”). Este proceso consiste en ir bajando el elemento de la raíz, intercambiándolo con el menor de sus hijos hasta encontrar una hoja, o una posición en la cual los dos hijos son mayores o equivalentes que el elemento que baja. En el caso de la figura, el 11 (que subió a la raíz para reemplazar al 4 que se fue), baja por el 5, ya que 5 es el menor de los dos hijos. Luego por el 10 y se detiene allí ya que en ese momento su único hijo es el 12 que es mayor que él. La situación final es como se muestra a la derecha.

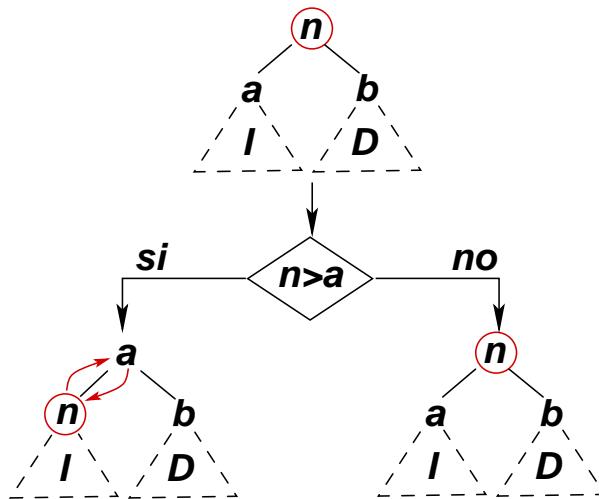


Figura 5.18: Intercambio básico en el re-heap. Asumimos  $a \leq b$ .

Ahora para entender mejor porque funciona este algoritmo consideremos el caso general mostrado en la figura 5.18. Tenemos un árbol que satisface las propiedades de montículo en todos sus puntos salvo, eventualmente, la condición de PO localmente en la raíz. Llámese  $n$  al nodo de la raíz y  $a, b$  sus hijos. Para fijar ideas asumimos que  $a \leq b$ , ya que el caso  $b > a$  es exactamente igual, *mutatis mutandis*. Ahora si  $n > a$  entonces intercambiamos  $n$  con  $a$ , quedando como en la parte inferior izquierda de la figura, caso contrario el árbol queda igual. Queremos ver que cada vez que el nodo  $n$  va bajando una posición la condición de PO se viola (eventualmente) sólo en el nodo que está bajando y en ese caso se viola localmente. Consideremos primero el caso  $n > a$ , podemos ver que se ha restablecido la condición PO en la raíz, ya que  $a < n$  y habíamos asumido que  $a \leq b$ . Entonces a esta altura la condición de PO puede violarse solamente en la raíz del subárbol izquierdo  $I$ . Por otra parte si  $n \leq a$  entonces todo el árbol es un montículo ya que  $n \leq a$  y  $n \leq b$  (ya que  $a \leq b$ ).

### 5.5.6. Costo de re-heap

De nuevo, el costo de re-heap está dado por el número de intercambios hasta que el elemento no baja más o llega a una hoja. En ambos casos el razonamiento es igual al de la inserción, como el procedimiento de bajar el elemento en el re-heap se da por un camino, el costo es a los sumo  $O(l)$  donde  $l$  es la longitud del máximo camino contenido en el montículo y, por lo tanto  $O(\log_2 n)$ . El costo de toda la segunda fase del algoritmo (seudo código 5.17) es  $O(n \log_2 n)$ .

### 5.5.7. Implementación in-place

Con el procedimiento de inserción y re-heap podemos ya implementar un algoritmo de ordenamiento  $O(n \log_2 n)$ . Alocamos un nuevo vector de longitud  $n$  que contendrá el montículo. Vamos insertando todos los elementos del vector original en el montículo, el cual va creciendo hasta llenar todo el vector auxiliar. Una vez insertados todos los elementos, se van eliminando del montículo, el cual va disminuyendo en longitud, y se van re-insertando en el vector original de adelante hacia atrás. Después de haber re-insertado todos los elementos, el vector original queda ordenado y se desaloca el vector auxiliar.

Sin embargo, la implementación descrita no es in-place, ya que necesita un vector auxiliar. Sin embargo, es fácil modificar el algoritmo para que sea in-place. Notemos que a medida que vamos sacando elementos del vector para insertarlos en el montículo en la primera fase y en el otro sentido en la segunda fase, en todo momento la suma de las longitudes del vector y del montículo es exactamente  $n$  de manera que podemos usar el mismo vector original para almacenar el vector ordenado y el montículo en la fase inicial y el vector ordenado y el montículo en la fase final.

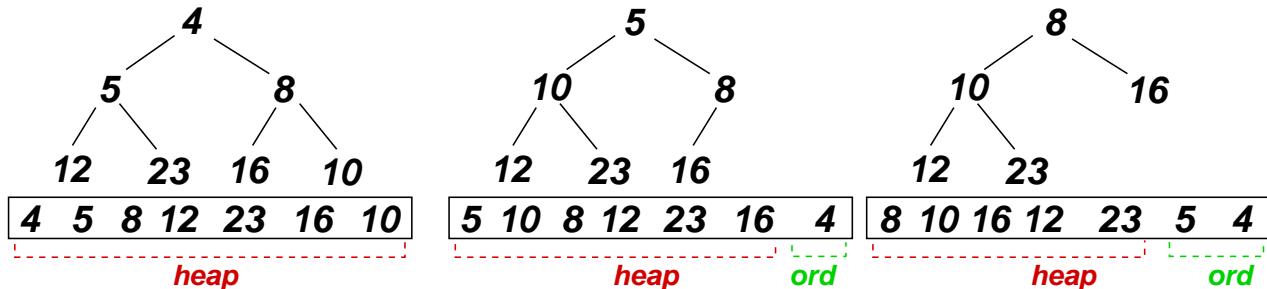


Figura 5.19: Implementación in-place de la fase final de heap-sort.

Para ilustrar mostramos en la figura 5.19 los primeros pasos de la fase final de heap-sort. En la figura de la izquierda se ve el vector que tiene 7 elementos, ocupado totalmente por el montículo. Ahora sacamos el elemento menor que es un 4. Como el tamaño del montículo se reduce en uno, queda al final del vector exactamente un lugar para guardar el elemento eliminado. En el paso siguiente se extrae el nuevo mínimo que es 5, el cual va la segunda posición desde el final. A esta altura el vector contiene al montículo en sus 5 primeros elementos y el vector ordenado ocupa los dos últimos elementos. A medida que se van sacando elementos del montículo, este se reduce en longitud y la parte ordenada del final va creciendo hasta que finalmente todo el vector queda ocupado por el vector ordenado.

Sin embargo, tal cual como se describió aquí, el vector queda ordenado de mayor a menor, lo cual no es exactamente lo planeado en un principio. Pero invertirlo es un proceso muy simple y  $O(n)$  de manera que

es absorbido en el costo global  $O(n \log_2 n)$ . Sin embargo existe una posibilidad mejor aún, en vez de usar un montículo minimal, como se describió hasta aquí, podemos usar uno maximal (que es exactamente igual, pero donde el padre es mayor o igual que los hijos). De esta forma en la fase final vamos siempre extrayendo el máximo, de manera que el vector queda ordenado de menor a mayor.

### 5.5.8. El procedimiento make-heap

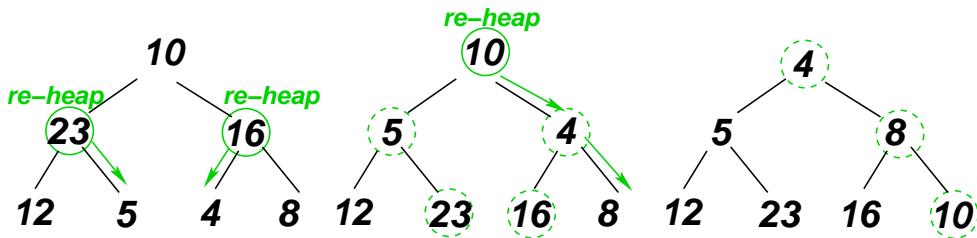


Figura 5.20: Procedimiento make-heap

La fase inicial tiene como objetivo convertir al vector, inicialmente desordenado, en un montículo. Existe una forma más eficiente de realizar esta tarea mediante un procedimiento llamado *make-heap*. En este procedimiento consiste en aplicar *re-heap* a cada uno de los nodos interiores (aquellos que no son hojas) *desde abajo hacia arriba*. Consideremos por ejemplo el árbol de la figura 5.20 a la izquierda. Aplicamos inicialmente *re-heap* a los nodos que están en el nivel 1 (el segundo desde abajo hacia arriba). Recordemos que el *re-heap* asume que la condición de PO se viola únicamente en la raíz. Como los árboles de los nodos en el nivel 1 tienen profundidad 1, esta condición ciertamente se cumple. Una vez aplicado el *re-heap* a cada uno de los nodos del nivel 1 el árbol queda como en la figura del centro. Ahora están dadas las condiciones para aplicar el *re-heap* el árbol en su totalidad ya que la condición de PO sólo se viola eventualmente en la raíz. En general, si ya hemos aplicado el *re-heap* a todos los nodos del nivel  $l$  entonces ciertamente podemos aplicárselo a cualquier nodo del nivel  $l - 1$  ya que sus hijos derecho e izquierdo pertenecen al nivel  $l$  y por lo tanto sus subárboles ya son montículos.

Puede verse fácilmente que este algoritmo es  $O(n \log_2 n)$  ya que cada *re-heap* es, a lo sumo,  $O(\log_2 n)$  y se hacen  $O(n/2)$  *re-heaps*. Sin embargo puede demostrarse que en realidad el *make-heap* es  $O(n)$ , es decir que tiene un costo aún menor. Notemos que esto no afecta la velocidad de crecimiento global de *heap-sort* ya que sigue dominando el costo de la etapa final que es  $O(n \log_2 n)$ , pero de todas formas el reemplazar la etapa inicial implementada mediante inserciones por el *make-heap*, baja prácticamente el costo global de *heap-sort* a la mitad.

Para ver que el costo de *make-heap* es  $O(n)$  contemos los intercambios que se hacen en los *re-heap*. Notemos que al aplicarle los *re-heap* a los nodos del nivel  $j$ , de los cuales hay  $2^j$ , el número de intercambios máximos que hay que hacer es  $l - j$  de manera que para cada nivel son a lo sumo  $2^j(l - j)$  intercambios. La cantidad total de intercambios es entonces

$$T(n) = \sum_{j=0}^l 2^j(l - j) \quad (5.37)$$

Notemos que podemos poner (5.37) como

$$T(n) = l \sum_{j=0}^l 2^j - \sum_{j=0}^l j 2^j \quad (5.38)$$

La primera sumatoria es una serie geométrica

$$\sum_{j=0}^l 2^j = \frac{2^{l+1} - 1}{2 - 1} = 2^{l+1} - 1. \quad (5.39)$$

La segunda sumatoria se puede calcular en forma cerrada en forma similar a la usada en la sección §4.5.4.1. Notemos primero que podemos reescribirla como

$$\sum_{j=0}^l j 2^j = \sum_{j=0}^l j e^{\alpha j} \Big|_{\alpha=\log 2}, \quad (5.40)$$

pero

$$j e^{\alpha j} = \frac{d}{d\alpha} e^{\alpha j} \quad (5.41)$$

de manera que

$$\sum_{j=0}^l j 2^j = \sum_{j=0}^l \left( \frac{d}{d\alpha} e^{\alpha j} \right) \Big|_{\alpha=\log_2} = \frac{d}{d\alpha} \left( \sum_{j=0}^l e^{\alpha j} \right) \Big|_{\alpha=\log_2} \quad (5.42)$$

pero ahora la sumatoria es una suma geométrica de razón  $e^\alpha$ , de manera que

$$\sum_{j=0}^l e^{\alpha j} = \frac{e^{\alpha(l+1)} - 1}{e^\alpha - 1} \quad (5.43)$$

y

$$\frac{d}{d\alpha} \left( \frac{e^{\alpha(l+1)} - 1}{e^\alpha - 1} \right) = \frac{(l+1)e^{\alpha(l+1)}(e^\alpha - 1) - (e^{\alpha(l+1)} - 1)e^\alpha}{(e^\alpha - 1)^2} \quad (5.44)$$

de manera que,

$$\begin{aligned} \sum_{j=0}^l j 2^j &= \frac{(l+1)e^{\alpha(l+1)}(e^\alpha - 1) - (e^{\alpha(l+1)} - 1)e^\alpha}{(e^\alpha - 1)^2} \Big|_{\alpha=\log_2} \\ &= (l+1)2^{l+1} - 2(2^{l+1} - 1) \\ &= (l-1)2^{l+1} + 2. \end{aligned} \quad (5.45)$$

Reemplazando en (5.39) y (5.45) en (5.37) tenemos que

$$\begin{aligned} T(n) &= \sum_{j=0}^l 2^j(l-j) = (2^{l+1} - 1)l - (l-1)2^{l+1} - 2 \\ &= 2^{l+1} - l - 2 = O(2^{l+1}) = O(n) \end{aligned} \quad (5.46)$$

### 5.5.9. Implementación

```
1 template<class T> void
2 re_heap(typename std::vector<T>::iterator first,
3 typename std::vector<T>::iterator last,
4 bool (*comp)(T&,T&),int j=0) {
5 int size = (last-first);
6 T tmp;
7 while (true) {
8 typename std::vector<T>::iterator
9 higher,
10 father = first + j,
11 l = first + 2*j+1,
12 r = l + 1;
13 if (l>=last) break;
14 if (r<last)
15 higher = (comp(*l,*r) ? r : l);
16 else higher = l;
17 if (comp(*father,*higher)) {
18 tmp = *higher;
19 *higher = *father;
20 *father = tmp;
21 }
22 j = higher - first;
23 }
24 }
25
26 template<class T> void
27 make_heap(typename std::vector<T>::iterator first,
28 typename std::vector<T>::iterator last,
29 bool (*comp)(T&,T&)) {
30 int size = (last-first);
31 for (int j=size/2-1; j>=0; j--)
32 re_heap(first,last,comp,j);
33 }
34
35 template<class T> void
36 heap_sort(typename std::vector<T>::iterator first,
37 typename std::vector<T>::iterator last,
38 bool (*comp)(T&,T&)) {
39 make_heap(first,last,comp);
40 typename std::vector<T>::iterator
41 heap_last = last;
42 T tmp;
43 while (heap_last>first) {
44 heap_last--;
45 tmp = *first;
46 *first = *heap_last;
47 *heap_last = tmp;
48 re_heap(first,heap_last,comp);
49 }
50 }
```

```
51 template<class T> void
52 heap_sort(typename std::vector<T>::iterator first,
53 typename std::vector<T>::iterator last) {
54 heap_sort(first, last, less<T>);
```

Código 5.18: Algoritmo de ordenamiento por montículos. [Archivo: heapsort.h]

En el código 5.18 vemos una posible implementación del algoritmo de ordenamiento por montículos. El montículo utilizado es maximal.

- La función `re_heap(first, last, comp, j)` realiza el procedimiento descripto re-heap sobre el subárbol de los nodos `j` (relativo a `first`).
- Durante la segunda fase `re_heap()` será llamado siempre sobre la raíz del montículo (`j=0`) pero en la fase inicial de `make_heap()` será llamado sobre los nodos interiores.
- Los iteradores `father`, `l` y `r` apuntan al nodo padre y sus dos hijos donde el padre es el nodo que inicialmente está en la raíz y va bajando por el mayor (recordemos que el montículo es maximal).
- El algoritmo termina cuando el `father` es una hoja, lo cual se detecta en la línea 13.
- El iterator `higher` es igual a `verb+r+` y `l`, dependiendo de cuál de ellos sea el mayor. Si el elemento en `higher` es mayor que el de `father` entonces los elementos son intercambiados.
- La línea 16 contempla el caso especial de que `father` tenga un sólo hijo. (Como el 12 en la figura 5.14.)
- `make_heap()` simplemente aplica `re_heap()` a los nodos interiores que van de `size/2-1` hasta `j=0`. (De abajo hacia arriba).
- `heap_sort(first, last, comp)` aplica `make_heap()` para construir el montículo en el rango `[first, last]`.
- En la segunda fase, el iterator `heap_last` marca la separación entre el montículo `[first, heap_last]` y el vector ordenado `[heap_last, last]`. Inicialmente `heap_last=last` y se va reduciendo en uno con cada re-heap hasta que finalmente `heap_last=first`.
- Las líneas 45–47 extraen el mínimo del montículo y suben el último elemento del mismo, insertando el mínimo en el frente del vector ordenado. Finalmente la línea 48 restituye la propiedad de montículo.

### 5.5.10. Propiedades del ordenamiento por montículo

Lo notable de heap-sort es que el tiempo de ejecución es  $O(n \log n)$  en el peor caso y además es in-place. Sin embargo, en el caso promedio quick-sort resulta ser más rápido (por un factor constante) por lo que muchas veces es elegido.

Heap-sort no es estable ya que en el momento de extraer el mínimo del montículo e intercambiarlo y subir el último elemento del mismo, no hay forma de garantizar que no se pase por encima de elementos equivalentes.

## 5.6. Ordenamiento por fusión

---

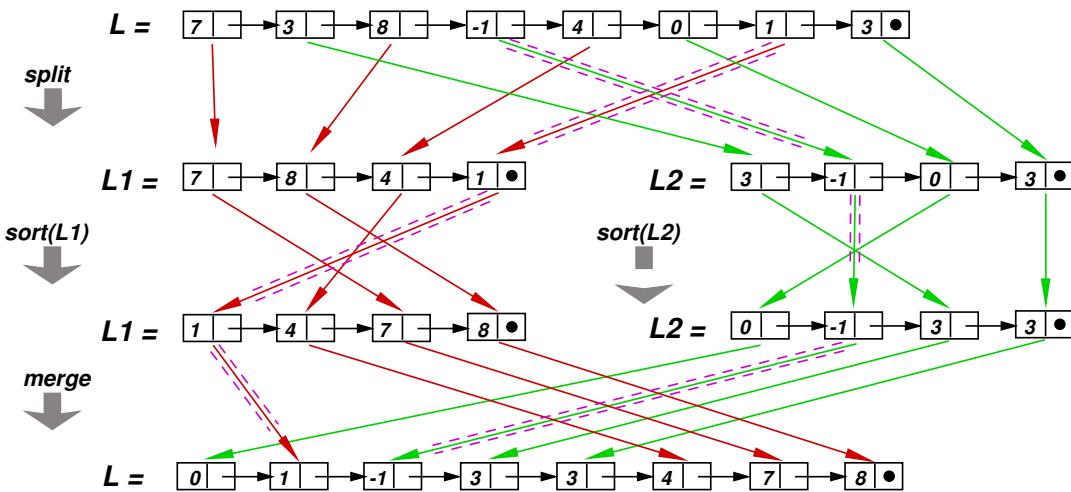


Figura 5.21: Ordenamiento de listas por fusión con splitting par/impar

```

1 void merge_sort(list<T> &L, bool (comp*)(T&, T&)) {
2 list<T>::iterator p = L.begin();
3 if (p==L.end() || ++p==L.end()) return;
4 list<T> L1,L2;
5 // Separacion: separar L en dos sublistas de
6 // tamano similar 'L1' y 'L2' ...
7 merge_sort(L1,comp);
8 merge_sort(L2,comp);
9 // Fusion: concatenar las listas 'L1' y 'L2' en 'L' ...
10 }

```

Código 5.19: Seudocódigo para el algoritmo de ordenamiento por fusión. [Archivo: mergesortsc.cpp]

Conceptualmente, uno de los algoritmos rápidos más simples de comprender es el algoritmo de “ordenamiento por fusión” o “intercalamiento” (“merge-sort”). Si pensamos en el ordenamiento de listas, entonces el esquema sería como se muestra en el seudocódigo 5.17. Como quick-sort, la estrategia es también típica de “dividir para vencer” e intrínsecamente recursiva. Inicialmente (ver figura 5.21) la lista se divide (“split”) en dos sublistas del mismo tamaño y se aplica recursivamente `merge_sort()` a cada una de las listas. Luego estas se concatenan (también “fusionan” o “merge”) en  $L$  manteniendo el orden, como en `set_union()` (ver sección §4.3.0.2) para conjuntos por listas ordenadas. Si la división se hace en forma balanceada y las operaciones de división y concatenación se pueden lograr en tiempo  $O(n)$  entonces el análisis de costo es similar al de quick-sort en el mejor caso y finalmente se obtiene un tiempo  $O(n \log n)$ . Ya hemos visto que el algoritmo de concatenación para listas ordenadas es  $O(n)$  y para dividir la lista en dos de igual tamaño simplemente podemos ir tomando un elemento de  $L$  e ir poniéndolo alternadamente en  $L_1$  y  $L_2$ , lo cual es  $O(n)$ . A esta forma de separar la lista en dos de igual tamaño lo llamamos “splitting par/impar”. De manera que ciertamente es posible implementar merge-sort para listas en tiempo  $O(n \log n)$ .

El concepto de si el ordenamiento es *in-place* o no cambia un poco para listas. Si bien usamos contenedores auxiliares (las listas  $L_1$  y  $L_2$ ), la cantidad total de celdas en juego es siempre  $n$ , si tomamos la

precaución de ir eliminando las celdas de **L** a medida que insertamos los elementos en las listas auxiliares, y viceversa al hacer la fusión. De manera que podemos decir que merge-sort es in-place.

Merge-sort es el algoritmo de elección para listas. Es simple,  $O(n \log n)$  en el peor caso, y es *in-place*, mientras que cualquiera de los otros algoritmos rápidos como quick-sort y heap-sort se vuelven cuadráticos (o peor aún) al querer adaptarlos a listas, debido a la falta de iteradores de acceso aleatorio. También merge-sort es la base de los algoritmos para *ordenamiento externo*.

### 5.6.1. Implementación

```
1 template<class T> void
2 merge_sort(std::list<T> &L, bool (*comp)(T&, T&)) {
3 std::list<T> L1, L2;
4 list<T>::iterator p = L.begin();
5 if (p==L.end() || ++p==L.end()) return;
6 bool flag = true;
7 while (!L.empty()) {
8 std::list<T> &LL = (flag ? L1 : L2);
9 LL.insert(LL.end(), *L.begin());
10 L.erase(L.begin());
11 flag = !flag;
12 }
13
14 merge_sort(L1, comp);
15 merge_sort(L2, comp);
16
17 typename std::list<T>::iterator
18 p1 = L1.begin(),
19 p2 = L2.begin();
20 while (!L1.empty() && !L2.empty()) {
21 std::list<T> &LL =
22 (comp(*L2.begin(), *L1.begin()) ? L2 : L1);
23 L.insert(L.end(), *LL.begin());
24 LL.erase(LL.begin());
25 }
26 while (!L1.empty()) {
27 L.insert(L.end(), *L1.begin());
28 L1.erase(L1.begin());
29 }
30 while (!L2.empty()) {
31 L.insert(L.end(), *L2.begin());
32 L2.erase(L2.begin());
33 }
34 }
35
36 template<class T>
37 void merge_sort(std::list<T> &L) {
38 merge_sort(L, less<T>);
```

---

Código 5.20: Algoritmo de ordenamiento por fusión. [Archivo: mergesort.h]

- Para merge-sort hemos elegido una firma diferente a la que hemos utilizado hasta ahora y que es usada normalmente en las STL para vectores. `merge_sort()` actúa directamente sobre la lista y no sobre un rango de iteradores. Sin embargo, sería relativamente fácil adaptarla para un rango usando la función `splice()` para extraer el rango en una lista auxiliar, ordenarla y volverla al rango original. Estas operaciones adicionales de `splice()` serían  $O(1)$  de manera que no afectarían el costo global del algoritmo.
- El bucle de las líneas 7–12 realiza la separación en las listas auxiliares. Mantenemos una bandera lógica `flag` que va tomando los valores `true/false` alternadamente. Cuando `flag` es `true` extraemos un elemento de `L` y lo insertamos en `L1`. Si es `false` lo insertamos en `L2`.
- Notar el uso de la *referencia a lista* `LL` para evitar la duplicación de código. Dependiendo de `flag`, la referencia `LL` “apunta” a `L1` o `L2` y después la operación de pasaje del elemento de `L` a `L1` o `L2` se hace vía `LL`.
- A continuación `merge_sort()` se aplica recursivamente para ordenar cada una de las listas auxiliares.
- El código de las líneas 17–33 realiza la fusión de las listas. El código es muy similar a `set_union()` para conjuntos implementados por listas ordenadas (ver sección §4.3.0.2). Una diferencia es que aquí si hay elementos duplicados no se eliminan, como se hace con conjuntos.
- En la fusión se vuelve a utilizar una referencia a lista para la duplicación de código.

### 5.6.2. Estabilidad

Es fácil implementar la etapa de fusión (“*merge*”) en forma estable, basta con tener cuidado al elegir el primer elemento de `L1` o `L2` en la línea 22 cuando ambos elementos son equivalentes. Notar que, de la forma como está implementado allí, cuando ambos elementos son equivalentes se elige el de la lista `L1`, lo cual es estable. Si reemplazáramos por

```
std::list<T> &LL =
(comp(*L1.begin(), *L2.begin()) ? L1 : L2);
```

entonces en caso de ser equivalentes estaríamos tomando el elemento de la lista `L2`. Por supuesto, ambas implementaciones serían equivalentes si no consideramos la estabilidad.

Entonces, si implementamos la fusión en forma estable y asumimos que las etapas de ordenamiento serán (recursivamente) estables, sólo falta analizar la etapa de split. Puede verse que la etapa de split, tal cual como está implementada aquí (split par/impar) es inestable. Por ejemplo, si nos concentramos en la figura 5.21, de los dos 1's que hay en la lista, queda antes el segundo (que estaba originalmente en la posición 6. Esto se debe a la etapa de split, donde el primer uno va a la lista `L2`, mientras que el segundo va a la lista `L1`.

### 5.6.3. Versión estable de split

```
1 int size = L.size();
2 if (size==1) return;
3 std::list<T> L1, L2;
4 int n1 = size/2;
5 int n2 = size-n1;
6 for (int j=0; j<n1; j++) {
```

```

7 L1.insert(L1.end(),*L.begin());
8 L1.erase(L1.begin());
9 }
10 for (int j=0; j<n2; j++) {
11 L2.insert(L2.end(),*L.begin());
12 L2.erase(L2.begin());
13 }
```

Código 5.21: Versión estable de split. [Archivo: stabsplit.h]

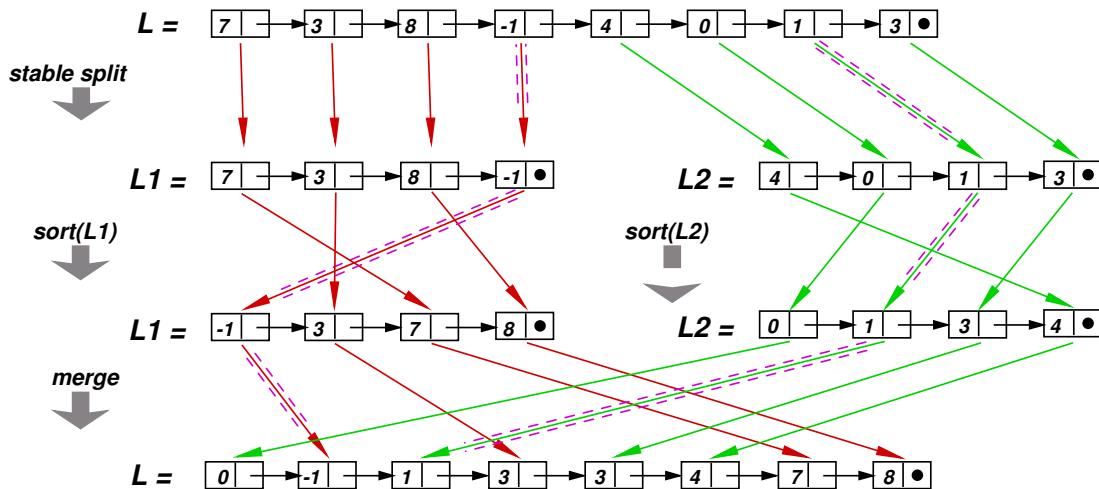


Figura 5.22: Versión estable de merge\_sort()

Se puede modificar fácilmente la etapa de split de manera que sea estable, basta con poner los primeros  $\text{floor}(n/2)$  elementos en **L1** y los restantes  $\text{ceil}(n/2)$  en **L2**, como se muestra en el código 5.21, el esquema gráfico de ordenamiento de un vector aleatorio de ocho elementos puede verse en la figura 5.22. Notar que ahora en ningún momento el camino de los 1's se cruzan entre sí.

#### 5.6.4. Merge-sort para vectores

```

1 template<class T> void
2 merge_sort(typename std::vector<T>::iterator first,
3 typename std::vector<T>::iterator last,
4 typename std::vector<T> &tmp,
5 bool (*comp)(T&, T)) {
6 int
7 n = last-first;
8 if (n==1) return;
9 int n1 = n/2, n2 = n-n1;
10 typename std::vector<T>::iterator
11 middle = first+n1,
```

((version aed-3.1-12-gc28b6c4c) (date Thu Aug 17 16:54:16 2017 -0300) (proc-date Thu Aug 17 16:55:19 2017 -0300))

```

12 q = tmp.begin(),
13 q1 = first,
14 q2 = first+n1;
15
16 merge_sort(first, middle, tmp, comp);
17 merge_sort(first+n1, last, tmp, comp);
18
19 while (q1!=middle && q2!=last) {
20 if (comp(*q2,*q1)) *q++ = *q2++;
21 else *q++ = *q1++;
22 }
23 while (q1!=middle) *q++ = *q1++;
24 while (q2!=last) *q++ = *q2++;
25
26 q1=first;
27 q = tmp.begin();
28 for (int j=0; j<n; j++) *q1++ = *q++;
29 }
30
31 template<class T> void
32 merge_sort(typename std::vector<T>::iterator first,
33 typename std::vector<T>::iterator last,
34 bool (*comp)(T&, T&)) {
35 std::vector<T> tmp(last-first);
36 merge_sort(first, last, tmp, comp);
37 }
38
39 template<class T> void
40 merge_sort(typename std::vector<T>::iterator first,
41 typename std::vector<T>::iterator last) {
42 merge_sort(first, last, less<T>);
43 }
```

**Código 5.22:** Implementación de merge-sort para vectores con un vector auxiliar. [Archivo: mergevec.h]

Es muy simple implementar una versión de merge-sort para vectores, si se usa un vector auxiliar, es decir no *in-place*. El proceso es igual que para listas, pero al momento de hacer la fusión, esta se hace sobre el vector auxiliar. Este vector debe ser en principio tan largo como el vector original y por una cuestión de eficiencia es creado en una función “wrapper” auxiliar y pasada siempre por referencia.

- Para vectores no es necesario hacer explícitamente el *split* ya que basta con pasar los extremos de los intervalos. Es decir, la operación de split es aquí un simple cálculo del iterador **middle**, que es  $O(1)$ .
- El vector auxiliar **tmp** se crea en la línea 35. Este vector auxiliar es pasado a una función recursiva **merge\_sort(first, last, tmp, comp)**.
- El intercalamiento o fusión se realiza en las líneas 19–24. El algoritmo es igual que para listas, pero los elementos se van copiando a elementos del vector **tmp** (el iterador **q**).
- El vector ordenado es recopiado en **[first, last)** en las líneas 26.

Esta implementación es estable,  $O(n \log n)$  en el peor caso, pero no es *in-place*. Si relajamos la condición de estabilidad, entonces podemos usar el algoritmo de intercalación discutido en la sección §2.3.1.

Recordemos que ese algoritmo es  $O(n)$  y no es in-place, pero requiere de menos memoria adicional,  $O(\sqrt{n})$  en el caso promedio,  $O(n)$  en el peor caso, en comparación con el algoritmo descripto aquí que que requiere  $O(n)$  siempre,

### 5.6.5. Ordenamiento externo

```
1 void merge_sort(list<block> &L, bool (comp*)(T&, T&)) {
2 int n = L.size();
3 if (n==1) {
4 // ordenar los elementos en el unico bloque de
5 // 'L'
6 } else {
7 list<T> L1,L2;
8 // Separacion: separar L en dos sublistas de
9 // tamano similar 'L1' y 'L2' ...
10 int
11 n1 = n/2,
12 n2 = n-n1;
13 list<block>::iterator
14 p = L.begin(),
15 q = L1.begin();
16 for (int j=0; j<n1; j++)
17 q = L1.insert(q,*p++);
18
19 q = L2.begin();
20 for (int j=0; j<n2; j++)
21 q = L2.insert(q,*p++);
22
23 // Sort individual:
24 merge_sort(L1,comp);
25 merge_sort(L2,comp);
26
27 // Fusion: concatenar las listas
28 // 'L1' y 'L2' en 'L' ...
29 }
30 }
```

Código 5.23: Algoritmo para ordenar bloques de dato. Ordenamiento externo. [Archivo: extsortsc.cpp]

De todos los algoritmos de ordenamiento vistos, merge-sort es el más apropiado para ordenamiento externo, es decir, para grandes volúmenes de datos que no entran en memoria principal. Si tenemos  $n$  objetos, entonces podemos dividir a los  $n$  objetos en  $m$  bloques de  $b = n/m$  objetos cada uno. Cada bloque se almacenará en un archivo independiente. El algoritmo procede entonces como se muestra en el código 5.23.

- A diferencia del merge-sort de listas, cuando la longitud de la lista se reduce a uno, esto quiere decir que la lista tiene un sólo bloque, no un solo elemento, de manera que hay que ordenar los elementos

del bloque entre sí. Esto se puede hacer cargando todos los elementos del bloque en un vector y ordenándolos con algún algoritmo de ordenamiento interno.

- Además, cuando se hace la fusión de las listas de bloques en la línea 28 se debe hacer la fusión elemento a elemento (no por bloques).
- Por supuesto las operaciones sobre los bloques deben ser implementadas en forma “*indirecta*”, es decir sin involucrar una copia explícita de los datos. Por ejemplo, si los bloques son representados por archivos entonces podríamos tener en las listas los nombres de los archivos.

Merge-sort externo es estable si el algoritmo de ordenamiento para los bloques es estable y si la fusión se hace en forma estable.

En el código **extsort.cpp**, que acompaña este libro se ha implementado el algoritmo descripto. El algoritmo genera un cierto número **Nb** de bloques de **M** enteros. Para ordenar cada bloque (archivo) se ha usado el **sort()** de STL para vectores. El **sort()** de STL no es estable, de manera que esta implementación tampoco lo es, pero de todas formas estamos ordenando enteros por  $<$ , que es una relación de orden fuerte, de manera que la estabilidad no es relevante.

## 5.7. Comparación de algunas implementaciones de algoritmos de ordenamiento

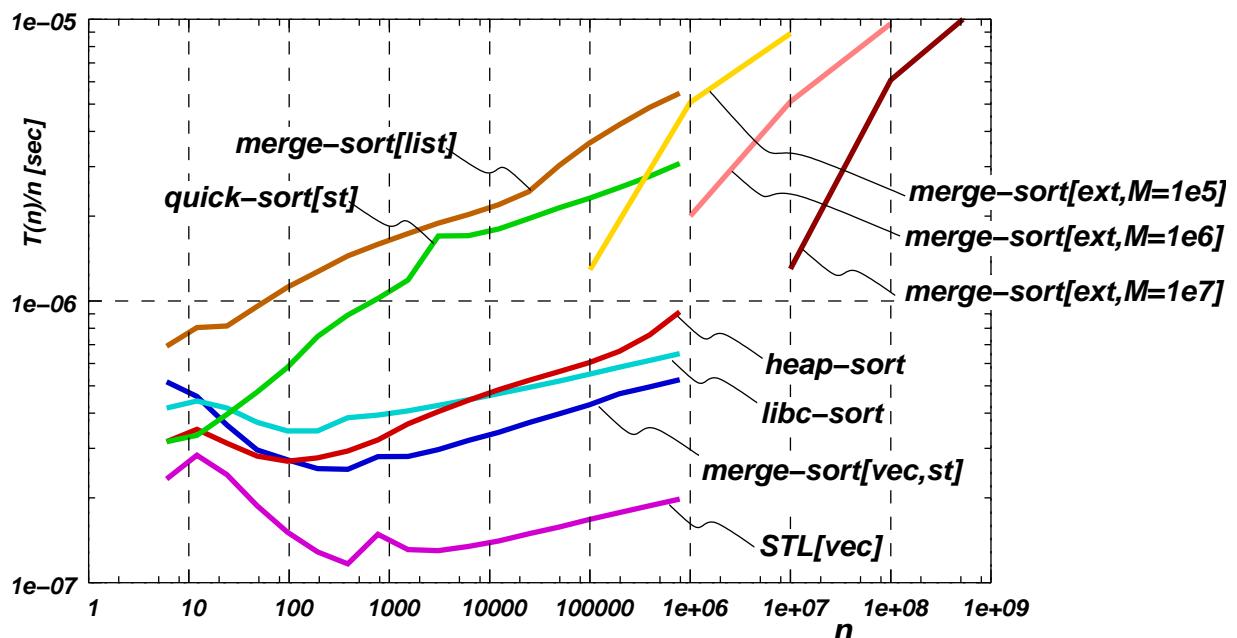


Figura 5.23: Tiempos de ejecución para varios algoritmos de ordenamiento.

En la figura 5.23 vemos una comparación de varias implementaciones de algoritmos de ordenamiento.

- **STL[vec]** es la versión de sort que viene en el header `<algorithm>` de C++ estándar con la versión de `g++` usada en este libro.

- **merge-sort[vec, st]** es la versión de merge-sort estable para vectores (no in-place) descripta en la sección §5.6.4.
- **libc-sort** es la rutina de ordenamiento que viene con el compilador **gcc** y que forma parte de la librería estándar de C (llamada **libc**).
- **heap-sort** es el algoritmo de ordenamiento por montículos implementado en este libro.
- **quick-sort[st]** es el algoritmo de ordenamiento rápido, en su versión estable descripta en la sección §5.4.10.
- **merge-sort[list]** Es la implementación de merge-sort para listas descripta en la sección §5.6.1.
- **merge-sort[ext,M=]** es la versión de merge-sort externa descripta en la sección §5.6.5. Se han hecho experimentos con varios valores del tamaño del bloque **M**.

Podemos hacer las siguientes observaciones

- Para resaltar las diferencias entre los diferentes algoritmos se ha graficado en las ordenadas  $T(n)/n$ . Para un algoritmo estrictamente lineal (es decir  $O(n)$ ) este valor debería ser constante. Como todos los algoritmos genéricos son, en el mejor de los casos,  $O(n \log n)$  se observa un cierto crecimiento. De todas formas este cociente crece a lo sumo un factor 10 o menos en 5 órdenes de magnitud de variación de  $n$ .
- Para algunos algoritmos se observa un decrecimiento para valores bajos de  $n$  (entre 10 y 100). Esto se debe a inefficiencias de las implementaciones para pequeños valores de  $n$ .
- El algoritmo de ordenamiento interno más eficiente de los comparados resulta ser la versión que viene con las STL. Esto puede deberse a que la implementación con template puede implementar “*inline*” las funciones de comparación.
- Merge-sort para vectores resulta ser muy eficiente (recordemos que además es estable, pero no es in-place). Sin embargo, para listas resulta ser notablemente más lento. Esto se debe sobre todo a que en general la manipulación de listas es más lenta que el acceso a vectores.
- La versión estable de quick-sort es relativamente inefficiente, sin embargo es el único algoritmo rápido, estable e *in-place* de los discutidos (recordemos que es  $O(n(\log n)^2)$ ).
- Los algoritmos de ordenamiento interno se ha usado hasta  $n = 10^6$ . El *merge-sort* se ha usado hasta cerca de  $n = 10^9$ . Notar que a esa altura el tamaño de los datos ordenados es del orden de 4 GBytes.
- El algoritmo de ordenamiento externo parece tener una velocidad de crecimiento mayor que los algoritmos de ordenamiento interno. Sin embargo esto se debe a la influencia del tamaño del bloque usado. Para valores de  $n$  mucho mayores que  $M$  el costo tiende a desacelerarse y debería ser  $O(n \log n)$ .

## Capítulo 6

# GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

### 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

### 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if

the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

#### 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. In any section entitled “Acknowledgements” or “Dedications”, preserve the section’s title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document. If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

## 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

### How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

*Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled “GNU Free Documentation License”.*

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being LIST”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

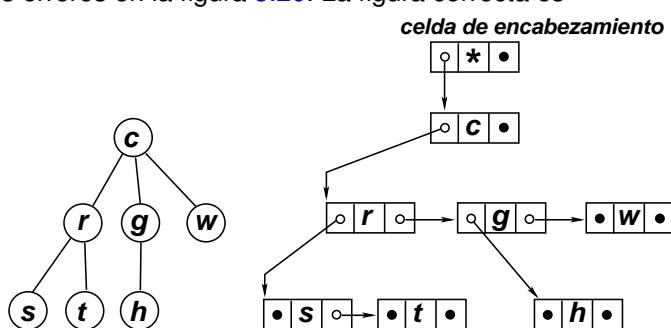
# Errata

- [2004-03-11] En la sección “Eficiencia de la implementación por arreglos”, las ecuaciones contienen errores. Deben ser:

$$T_{\text{prom}}(n) = \sum_{j=0}^{n-1} P_j T(j)$$

$$\begin{aligned} T_{\text{prom}}(n) &= \frac{1}{n} \sum_{j=0}^{n-1} n - j - 1 \\ &= \frac{1}{n} ((n-1) + (n-2) + \cdots + 1 + 0) \\ &= \frac{1}{n} \frac{(n-1)n}{2} = \frac{n-1}{2} \approx \frac{n}{2} = O(n) \end{aligned}$$

- [2004-03-11] Por un error en la generación del PDF, en los códigos el operador != salía como ! (faltaba el =).
- [2004-04-06] En la sección “Tipos de datos abstractos fundamentales. Tiempos de ejecución para listas ordenadas”, el tiempo de ejecución para `erase()` en correspondencias por listas en la tabla es  $O(1)/O(n)/O(n)$ .
- [2004-04-13] Varias entradas en la figura 1.1 estaban mal. Faltaba el par  $(a, d)$  en la listas de pares de  $G$ .
- [2004-04-13] En el último párrafo de la sección §3.2.4 “Reconstrucción del árbol a partir de sus órdenes” donde dice “...para hallar la estructura de los descendientes de  $s$ ” debe decir “...para hallar la estructura de los descendientes de  $c$ ”
- [2004-04-19] Varios errores en la figura 3.20. La figura correcta es



- [2004-05-09] En la sección *Operaciones básicas sobre árboles binarios* donde dice:
    - ▷ “Entonces, las operaciones abstractas son las siguientes: ... Dada una posición (dereferenciable o no) y un dato, insertar un nuevo nodo con ese dato en esa posición.”
- debe decir:
- ▷ “Entonces, las operaciones abstractas son las siguientes: ... Dada una posición no dereferenciable y un dato, insertar un nuevo nodo con ese dato en esa posición.”
- [2004-05-21] En los ejemplos **preorder()** y **postorder()**, en la sección *Interfaz básica para árboles*, la comparación en las funciones “wrapper” debe ser exactamente al revés. En ambas funciones donde dice **if (T.begin() != T.end()) return;** debe decir **if (T.begin() == T.end()) return;**.
  - [2004-06-10] En “*Conjuntos. Algoritmo lineal para las operaciones binarias*” debe decir

▷ “Estas condiciones son bastante fuertes, por ejemplo si el valor  $x_a < x_b$  entonces podemos asegurar que  $x_a \notin B$ . Efectivamente, en ese caso, todos los elementos anteriores a  $x_b$  son menores a  $x_a$  y por lo tanto distintos a  $x_a$ . Por otra parte los que están después de  $x_b$  son mayores que  $x_b$  y por lo tanto que  $x_a$ , con lo cual también son distintos. Como conclusión, no hay ningún elemento en  $B$  que sea igual a  $x_a$ , es decir  $x_a \notin B$ . Similarmente, si  $x_b < x_a$  entonces se ve que  $x_b \notin A$ .”

- [2004-06-13] En “*Conjuntos. Costo de la inserción no exitosa*”, en varias ecuaciones para el número medio de intentos infructuosos  $\langle m \rangle$ , el índice de sumación está mal (donde dice  $k$  debe ser  $m$ ) y los límites para sumación están mal.

Donde dice

$$\begin{aligned}\langle m \rangle &= \sum_{k=0}^B m P(m) \\ &= \sum_{k=0}^B m \alpha^m (1 - \alpha)\end{aligned}$$

debe decir

$$\begin{aligned}\langle m \rangle &= \sum_{m=0}^{B-1} m P(m) \\ &= \sum_{m=0}^{B-1} m \alpha^m (1 - \alpha)\end{aligned}$$

y donde dice

$$\langle m \rangle = \sum_{k=0}^{\infty} (1 - \alpha) \alpha \frac{d}{d\alpha} \alpha^m$$

debe decir

$$\langle m \rangle = \sum_{m=0}^{\infty} (1 - \alpha) \alpha \frac{d}{d\alpha} \alpha^m$$

- [2004-07-17] En “Arboles. Reconstrucción del árbol a partir de sus órdenes”

Donde dice

$$\text{opost}(n) = (\text{descendientes}(n_1), n_1, \text{descendientes}(n_2), n_2, \dots, \text{descendientes}(n_m), n_m, n_1).$$

debe decir:

$$\text{opost}(n) = (\text{descendientes}(n_1), n_1, \text{descendientes}(n_2), n_2, \dots, \text{descendientes}(n_m), n_m, n).$$

# Indice alfabético

- ~btree, 157, 163
- ~list, 80
- ~set, 200, 205, 207, 208, 237
- ~tree, 137, 143
  
- abb.p, 228
- abiertas, tablas de dispersión a., 211
- adaptador, 83
- adyacente, 11
- agente viajero, problema del a.v., 9
- aleatorio, contenedor de acceso a., 246
- aleatorio, contenedores de acceso a., 112
- alfabético, véase lexicográfico
- algoritmo, 9
  - ávido, 19
    - heurístico, 9, 19
  - altura de un nodo, 116
  - antecesor, 116
  - antisimetría, cond. de a. para rel. de orden, 242
  - apply, 161
  - apply\_perm, 251
  - árbol, 114
  - árbol binario, 148
  - árbol binario completo, 230
  - árbol binario lleno, 167
  - árboles ordenados orientados, 148
  - árboles binarios de búsqueda, 227
  - aristas de un grafo, 11
  - arreglos, implementación de listas por a., 58
  - asintótica, véase notación a.
  - ávido, véase algoritmo
  - AVL, árboles, 240
  
  - bases de datos, analogía con correspondencias, 97
  - begin, 80, 104, 108, 111, 138, 144, 157, 164, 200, 205, 209, 212, 226, 238
  
- bflush, 182
- bin, 210
- binary search, 111
- bsearch, 46
- bsearch2, 46
- btree, 156, 162
- bubble-sort, véase burbuja
- bubble\_sort, 42, 247
- bucket, 210
- buffer, 90
- burbuja, método de ordenamiento, 42
- burbuja, ordenamiento, 247
- búsqueda binaria, 46, 111
- búsqueda exhaustiva, 9, 12
  
- calculadora, 83
- camino, en un árbol, 115
- cell, 64, 79, 136, 142, 155, 162
- cell::cell, 73
- cell\_count, 142, 156, 162
- cerradas, tablas de dispersión c., 216
- check1, 85
- check2, 85
- check\_sum, 57, 78
- circular, sentido c., 216
- clave, 97
- clear, 80, 90, 96, 104, 108, 111, 138, 144, 157, 164, 200, 205, 209, 213, 226, 238
- codelen, 173, 174
- cola, 90
- cola de prioridad, 191, 271
- colisión, 211
- colorear grafos, 11
- comb, 171
- comp\_tree, 253
- complejidad algorítmica, 32

conjunto, 49, 191  
    TAD, 27  
conjunto vacío, 191  
constructor por copia, 138, 140  
contenedores, 28  
contradominio, 97  
convergencia de mét. iterativos, 197  
copy, 164  
correspondencia, 97  
count\_nodes, 145, 146  
crecimiento, tasa o velocidad de c., 32  
cubetas, 210  
  
dataflow, 195, 197  
deleted, véase eliminado  
dereferenciable, 117  
dereferenciables, posiciones d. en listas, 49  
descendiente, 116  
diccionario, 191, 210  
diferencia de conjuntos, 191  
disjuntos, conjuntos, 10  
dblemente enlazadas, véase listas doblemente enlazadas  
dominio, 97  
  
edge, 24  
efecto colateral, 78  
eficiencia, 30  
element, 199  
eliminado, elemento deleted, 220  
empty, 90, 96, 104, 107, 111, 209  
encabezamiento, celda de e., 66  
end, 80, 104, 108, 111, 138, 144, 157, 164, 200, 205, 209, 212, 226, 238  
ensamble, 31  
envoltorio, 133  
equal\_p, 151, 152  
equivalencia, en rel. de orden, 242  
erase, 80, 104, 107, 111, 137, 144, 157, 163, 200, 205, 208, 213, 225, 237, 238  
estabilidad  
    métodos lentos, 246  
    quick-sort, 265  
estable, algoritmo de ordenamiento e., 246  
  
etiqueta, 114  
Euclides, algoritmo de E. para gcd(), 193, 269  
exitosa, inserción e. en tablas de dispersión, 217  
experimental, determinación e. de la tasa de crecimiento, 37  
externo, ordenamiento, 241, 282  
  
factorial, 36  
ficticia, posición, 49  
FIFO, 90  
find, 104, 107, 110, 137, 138, 144, 157, 164, 200, 205, 209, 213, 225, 229, 238  
floor( $x$ ), función de la librería estándar de C. , 112  
for, 224, 284  
free store, 64  
front, 96  
relación de o. fuerte, 241  
 fusión, ordenamiento por f., 281  
  
gcd, 268  
genérico, función, 62  
grafo, 11  
    denso, 40  
    no orientado, 11  
    ralo, 40  
graph, 24  
greedy, 25  
greedy, 22–24  
  
h, 210  
h2, 215  
hash tables, 210  
hash\_set, 212, 224, 225  
heap, 64, 271  
heap\_sort, 279, 280  
height, 146  
hermanos, 116  
heurístico, véase algoritmo  
hijos, 114  
hoja, 116  
huffman, 178  
Huffman, árboles de H., 165  
Huffman, algoritmo de H., 176  
huffman\_codes, 181, 182  
huffman\_exh, 175

hufunzip, 185  
hufzip, 183  
  
ibubble\_sort, 251, 252  
if, 107, 111, 143, 152, 163, 283  
if, tiempos de ejec. de bloques if, 40  
implementación de un TAD, 27  
in-place, ordenamiento i.p., 241  
indefinido, 216  
indirecto, ordenamiento, 251  
indx, 199  
inicialización, lista de i. de una clase, 62  
inserción, método de i., 248  
insert, 80, 104, 137, 143, 157, 163, 200, 205, 208, 213, 225, 237  
insertar, en listas, 49  
insérer, 29  
insertion\_sort, 248, 249  
inssort, 91  
intercalamiento, alg. de ordenamiento por i., 38  
interfaz, 27  
interno, ordenamiento, 241  
intersección de conjuntos, 191  
inválidas, posiciones, 51  
iteradores, 22, 28  
iterativo, método, 196  
iterator, 50, 79, 142, 143, 162, 236, 237  
iterator\_t, 136, 155, 156, 212  
iterators, 22, 28  
  
key, 97, 105  
  
lazos, tiempo de ejec. de l., 41  
lchild, 136, 143  
leaf\_count, 147  
left, 156, 162  
less, 244  
lexicográfico, orden, 103, 243  
LIFO, 82  
lineales, contenedores, 102  
linear\_redisp\_fun, 224  
Lisp, 48  
Lisp, notación L. para árboles, 121  
Lisp, notación L. para árboles binarios, 149  
lisp\_print, 125, 131, 158, 164  
  
list, 80  
list::~list, 60, 67, 74  
list::begin, 60, 67, 75  
list::cell\_space\_init, 74  
list::clear, 62, 67, 75  
list::delete\_cell, 74  
list::end, 60, 67, 75  
list::erase, 61, 67, 75  
list::insert, 61, 67, 75  
list::list, 60, 66, 73  
list::new\_cell, 74  
list::next, 60, 67, 74  
list::prev, 61, 67, 75  
list::print, 62, 68, 75  
list::printd, 68, 76  
list::retrieve, 60, 67, 74  
list::size, 68  
listas, 48  
    dblemente enlazadas, 82  
locate, 224  
longitud, 49  
longitud, l. de un camino en un árbol, 115  
lower\_bound, algoritmo, 105  
lower\_bound, 104, 107, 110, 200, 205, 208  
  
máquina de Turing, 39  
main, 53–55, 85  
make\_heap, 279  
map, 104, 107, 110  
mapas, coloración de m., 12  
max\_leaf, 147  
max\_node, 146, 147  
median, 263  
mediana, 256  
memoria adicional, en ordenamiento, 241  
memoria asociativa, 97  
memory leaks, 64  
merge, 94, 96  
merge\_sort, 281, 282, 284–286  
miembro de un conjunto, 191  
miembro izquierdo, 100  
min, 53, 54, 235  
min\_code\_len, 175  
mirror, 153, 154

mirror\_copy, 127, 132  
montículo, 271  
  
next, 108, 200, 205, 213, 226, 236  
next\_aux, 200, 212, 225  
nivel, 116  
node\_level\_stat, 146  
nodos, 114  
notación asintótica, 32  
NP, problemas, 39  
  
operaciones abstractas, 27  
operator[], sobrecarga de, 108  
orden, relación de, 113, 191, 241  
orden posterior, listado en, 119  
orden previo, listado en, 119  
ordenados, árboles, 116  
ordenados, contenederos, 103  
ordenamiento, 241  
  
P, problemas, 39  
padre, 114  
pair, 107, 110  
pair\_t, clase, 108  
parcialmente completo, condición de, 271  
parcialmente ordenado, 240  
parcialmente ordenado, condición de, 271  
particionamiento, algoritmo de p., 253  
partition, 261, 262  
permutación, 265  
pertenencia, en conjuntos, 191  
pila, 73, 82  
pivot, 253  
PO, véase parcialmente ordenado  
polaca invertida, notación , 83, 121  
polish notation, reverse, véase polaca invertida, notación  
pop, 90, 96  
pop\_char, 185  
posición, en listas, 49  
posiciones, operaciones con p. en listas, 52  
postfijo, operador de incremento p., 78  
postorder, 119, 125, 131  
[p, q), véase rango  
predicado, 152  
  
prefijo, condición de p., 166  
prefijo, operador de incremento p., 78  
preorder, 119, 124, 130, 133  
preorder\_aux, 133  
print, 53, 54, 80  
printf, 80  
prioridad, cola de, véase cola de prioridad  
profundidad, 116  
programación funcional, 172  
promedio, 31  
propios, descendientes y antecesores, 116  
prune\_odd, 128, 132  
pulmón, 90  
puntero, 64  
puntero, implementación de lista por p., 64  
purge, 55  
purge, algoritmos genérico, 55  
push, 90, 96  
  
qflush, 182  
queue, 96  
quick-sort, 253  
quick\_sort, 264  
quicksort, 253  
  
rápido, ordenamiento, 253  
raíz, 114  
range\_swap, 268  
rango, 76, 111  
re\_heap, 279  
redispersión, 216  
redispersión lineal, 223  
referencia, 100  
referencia, funciones que retornan r., 51  
refinamiento, 21  
relación de orden, véase orden  
retorno, valor de r. del operador de incremento, 78  
retrieve, 104, 137, 141, 157, 200, 205, 213, 225  
reverse polish notation, véase polaca invertida, notación  
right, 136, 143, 156, 162  
RPN, véase polaca invertida, notación  
  
Scheme, 48  
search, 31

selección, método de s., 249  
selection\_sort, 249  
semejante\_p, 152  
serializar, 122  
set, 200, 205, 207, 208, 237  
set\_difference, 201, 206, 209, 239  
set\_difference\_aux, 236  
set\_intersection, 201, 206, 209, 239  
set\_intersection\_aux, 236  
set\_union, 200, 205, 209, 239  
set\_union\_aux, 236  
seudo-código, 21  
side effect, 78  
signatura, 161  
simétrico, listado en orden s., 149  
sincronización en cálculo distribuido, 10  
size, 81, 90, 96, 104, 111, 200, 205, 209, 213, 226, 238  
size\_aux, 236  
sobrecarga de operadores, 145  
sobrecarga, del nombre de funciones, 133  
sort, algoritmo, 63  
splice, 137, 144, 157, 163  
stable\_partition, 265  
stack, 90  
stack::clear, 89  
stack::empty, 89  
stack::pop, 89  
stack::push, 89  
stack::size, 89  
stack::stack, 89  
stack::top, 89  
Standard Template Library, STL, 21  
Stirling, aproximación de, 36  
STL, 21  
string\_less\_ci, 245  
string\_less\_cs, 243  
string\_less\_cs3, 244  
subconjunto, 191  
subconjunto propio, 191  
supraconjunto, 191  
supraconjunto propio, 191  
suprimir, en listas, 49  
swap, 144  
tabla de dispersión, 210  
TAD, véase tipo abstracto de datos  
TAD conjunto, véase conjunto  
tasa de crecimiento, 32  
tasa de ocupación en tablas de dispersión, 217  
tiempo de ejecución, 15, 26, 30  
tipo abstracto de datos, 27  
tolower, 245  
top, 90  
tope de pila, 82  
treaps, 240  
tree, 136, 137, 141, 143  
tree\_copy, 126, 131  
tree\_copy\_aux, 142, 156  
TSP, véase agente viajero  
undef, valor indefinido, 216  
unión de conjuntos, 191  
universal, conjunto u., 191  
value, 105  
velocidad de crecimiento, 32  
while, 99, 101, 108, 159, 266, 270, 271  
wrapper, 133

# Bibliografía

- A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison Wesley, 1987.
- Free Software Foundation. *GNU Compiler Collection GCC manual*, a. GCC version 3.2, <http://www.gnu.org>.
- Free Software Foundation. *The GNU C Library Reference Manual*, b. Version 2.3.x of the GNU C Library, edition 0.10, <http://www.gnu.org/software/libc/libc.html>.
- R. Hernández, J.C. Lázaro, R. Dormido, and S. Ros. *Estructuras de Datos y Algoritmos*. Prentice Hall, 2001.
- D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1981.
- SGI. *Standard Template Library Programmer's Guide*, 1999. <http://www.sgi.com/tech/stl>.