Capítulo 1

Problemas, programas, estructuras y algoritmos

El proceso de resolución de problemas, en programación, está compuesto por una serie de pasos que se aplican de forma metódica. Empieza con el análisis de las características del problema, continúa con el diseño de una solución, la implementación del diseño y termina con la verificación y evaluación del programa resultante. Los dos ejes básicos en los que se articula el desarrollo de un programa son las estructuras de datos y los algoritmos. Las estructuras de datos determinan la forma de almacenar la información necesaria para resolver el problema. Los algoritmos manipulan esa información, para producir unos datos de salida a partir de unos datos de entrada. El objetivo último de la algorítmica es encontrar la mejor forma de resolver los problemas, en términos de eficiencia y de calidad del software.

Objetivos del capítulo:

- Entender el desarrollo de programas como un proceso metódico e ingenieril, formado por una serie de etapas con distintos niveles de abstracción, frente a la idea de la programación como arte.
- Tomar conciencia de la importancia de realizar siempre un análisis y diseño previos del problema, como pasos anteriores a la implementación en un lenguaje de programación.
- Motivar el estudio de los algoritmos y las estructuras de datos, como una disciplina fundamental de la informática.
- Repasar algunos de los principales conceptos de programación, que el alumno debe conocer de cursos previos como: tipos de datos, estructuras de datos, tipos abstractos, algoritmos, complejidad algorítmica y eficiencia.
- Entender la importancia del análisis de algoritmos, el objetivo del análisis, los factores que influyen y el tipo de notaciones utilizadas para su estudio.
- Distinguir entre algoritmos y esquemas algorítmicos, presentando brevemente los principales esquemas algorítmicos que serán estudiados.

Contenido del capítulo:

1.1. Resolución de problemas 3
1.1.1. Análisis de requisitos del problema 3
1.1.2. Modelado del problema y algoritmos abstractos 4
1.1.3. Diseño de la solución 5
1.1.4. Implementación del diseño 6
1.1.5. Verificación y evaluación de la solución 7
1.2. Tipos de datos 7
1.2.1. Definición de tipo de datos, tipo abstracto y estructura 8
1.2.2. Tipos de tipos 10
1.2.3. Repaso de tipos y pseudolenguaje de definición 12
1.3. Algoritmos y algorítmica 14
1.3.1. Definición y propiedades de algoritmo 15
1.3.2. Análisis de algoritmos 17
1.3.3. Diseño de algoritmos 21
1.3.4. Descripción del pseudocódigo utilizado 25
1.4. Consejos para una buena programación 26
1.4.1. Importancia del análisis y diseño previos 26
1.4.2. Modularidad: encapsulación y ocultamiento 28
1.4.3. Otros consejos 29
Ejercicios propuestos 31
Cuestiones de autoevaluación 31
Referencias bibliográficas 32

1.1. Resolución de problemas

La habilidad de programar ordenadores constituye una de las bases necesarias de todo informático. Pero, por encima del nivel de programación, la característica fundamental de un informático es su capacidad para **resolver problemas**. La resolución de problemas informáticos en la vida real implica otras muchas más habilidades que únicamente saber programar –habilidades que distinguen a un ingeniero de un simple programador. Implica comprender y analizar las necesidades de los problemas, saber modelarlos de forma abstracta diferenciando lo importante de lo irrelevante, disponer de una variada batería de herramientas conceptuales que se puedan aplicar en su resolución, trasladar un diseño abstracto a un lenguaje y entorno concretos y, finalmente, ser capaz de evaluar la corrección, prestaciones y posibles inconvenientes de la solución planteada.

Las herramientas que surgen en el desarrollo de programas son esencialmente de dos clases: **estructuras de datos** y **algoritmos**. Las estructuras de datos representan la parte estática de la solución al problema, el componente almacenado, donde se encuentran los datos de entrada, de salida y los necesarios para posibles cálculos intermedios. Los algoritmos representan la parte dinámica del sistema, el componente que manipula los datos para obtener la solución. Algoritmos y estructuras de datos se relacionan de forma muy estrecha: las estructuras de datos son manipuladas mediante algoritmos que añaden o modifican valores en las mismas; y cualquier algoritmo necesita manejar datos que estarán almacenados en cierta estructura. La idea se puede resumir en la fórmula magistral de Niklaus Wirth: Algoritmos + Estructuras de datos = Programas.

En ciertos ámbitos de aplicación predominará la componente algorítmica –como, por ejemplo, en problemas de optimización y cálculo numérico– y en otros la componente de estructuras –como en entornos de bases de datos y sistemas de información–, pero cualquier aplicación requerirá siempre de ambos. Esta dualidad aparece en el nivel abstracto: un tipo abstracto está formado por un dominio de valores (estructura abstracta) y un conjunto de operaciones (algoritmos abstractos). Y la dualidad se refleja también en el nivel de implementación: un módulo (en lenguajes estructurados) o una clase (en lenguajes orientados a objetos) están compuestos por una estructura de atributos o variables, y una serie de procedimientos de manipulación de los anteriores.

Antes de entrar de lleno en el estudio de los algoritmos y las estructuras de datos, vamos a analizar los pasos que constituyen el proceso de resolución de problemas.

1.1.1. Análisis de requisitos del problema

El proceso de resolución de problemas parte siempre de un **problema**, de un enunciado más o menos claro que alguien plantea porque le vendría bien que estuviera resuelto[1]. El primer paso es la comprensión del problema, entender las características y peculiaridades de lo que se necesita. Este **análisis de los requisitos** del problema puede ser, de por sí, una de las grandes dificultades; sobre todo en grandes aplicaciones, donde los documentos de requisitos son ambiguos, incompletos y contradictorios.

[1]Esta afirmación puede parecer irrelevante por lo obvia. Pero no debemos perder de vista el objetivo último de la utilidad práctica, en el estudio de los algoritmos y las estructuras de datos.

El análisis debe producir como resultado un **modelo abstracto** del problema. El modelo abstracto es un modelo conceptual, una abstracción del problema, que reside exclusivamente en la mente del individuo y donde se desechan todas las cuestiones irrelevantes para la resolución del problema. Supongamos, por ejemplo, los dos siguientes problemas, con los cuales trabajaremos en el resto de los puntos.

Ejemplo 1.1 El problema de las cifras. Dado un conjunto de seis números enteros, que pueden ser del 1 al 10, ó 25, 50, 75 ó 100, encontrar la forma de conseguir otro entero dado entre 100 y 999, combinando los números de partida con las operaciones de suma, resta, producto y división entera. Cada uno de los seis números iniciales sólo se puede usar una vez.

Ejemplo 1.2 El problema de detección de caras humanas. Dada una imagen en color en un formato cualquiera (por ejemplo, bmp, jpeg o gif) encontrar el número de caras humanas presentes en la misma y la posición de cada una de ellas.

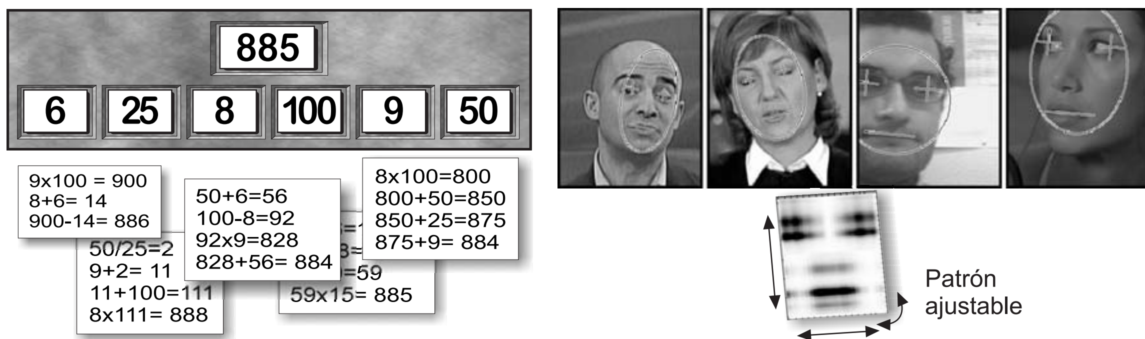


Figura 1.1: Ejemplos del problema de las cifras (izquierda) y detección de caras (derecha).

Ambos enunciados representan categorías de problemas muy distintas. El problema 1.1 tiene un enunciado más o menos claro, aunque cabrían algunas dudas. Es de tipo matemático, se puede modelar formalmente y es previsible que exista un algoritmo adecuado. El problema 1.2 tiene un enunciado más corto pero mucho más ambiguo, no está claro exactamente lo que se pide. Es más, incluso teniéndolo bien claro, el problema parece más adecuado para ser resuelto de cabeza por un humano, pero difícil de implementar en un ordenador. Aun así, ambos problemas tienen algo en común: son de interés, así que necesitamos resolverlos de la mejor forma que podamos.

1.1.2. Modelado del problema y algoritmos abstractos

El primer paso sería crear un **modelo abstracto**, en el que nos quedamos con lo esencial del problema. Normalmente, este modelo se crea a través de una **analogía** con algo conocido previamente. Por ejemplo, para enseñar a un alumno inexperto lo que es un algoritmo (concepto para él desconocido) se utiliza la analogía con una receta de cocina

(concepto conocido, esperemos), y las estructuras de datos se asimilan con la disposición de armarios, cajones y recipientes donde se guardan los ingredientes y el bizcocho resultante.

Ejemplo 1.1 (cont.) Modelo abstracto. Para el problema de las cifras, el modelo abstracto puede prescindir del hecho de que los números sean seis, de que estén en ciertos rangos y que se trabaje sólo con enteros. Una solución sería una expresión matemática –usando los números de partida– que produce el número buscado. Entonces, el problema se podría resolver generando todas las posibles expresiones que contengan los n números iniciales, o menos, y quedándonos la que obtenga un resultado más próximo al buscado.

Pero el modelo de un problema no tiene por qué ser único. Otra persona podría interpretarlo como un conjunto de cajas, que se combinan entre sí. Ahora tendríamos un modelo "temporal": tenemos que buscar dos cajas que combinándolas nos den el objetivo o un valor próximo. Si no llegamos al resultado, buscar otra caja y así sucesivamente. La idea resultante sería similar a lo que aparece en la figura 1.1, abajo a la izquierda.

Ejemplo 1.2 (cont.) Modelo abstracto. En el problema de detección de caras, el modelo abstracto elimina el hecho de que las imágenes tengan distinto formato o incluso que los objetos buscados sean caras humanas. Un posible modelo sería un problema de patrones –patrones como los de un sastre– que admiten cierta variabilidad: se pueden mover, estirar, cambiar el tamaño, rotar, etc. El problema consistiría en buscar esos patrones en las imágenes, en todos los sitios que se pueda. Grosso modo, un algoritmo podría consistir en generar todas las posibles colocaciones, rotaciones y tamaños de los patrones y quedarnos con las que "encajen" con lo que hay en la imagen de partida.

1.1.3. Diseño de la solución

Una vez que tengamos un modelo completo y adecuado, significará que hemos entendido bien el problema. Como hemos visto en los ejemplos, el modelo conlleva también un algoritmo informal, que no es más que una vaga idea de cómo resolver el problema. El siguiente paso de refinamiento consistiría en diseñar una solución. Usando otra analogía, el **diseño de un programa** es equivalente a los planos de un arquitecto: el diseño describe el aspecto que tendrá el programa, los materiales que se necesitan y dónde y cómo colocarlos. El diseño es siempre un paso previo a la implementación, pero que se olvida muy a menudo por los programadores primerizos. Un ingeniero informático debería ser tan consciente del error de programar sin partir de un buen diseño previo, como lo es un arquitecto del fallo de hacer un puente sin tener antes los planos.

El diseño de un programa consta de dos clases de elementos: tipos de datos y algoritmos. Los tipos de datos usados a nivel de diseño son **tipos abstractos**, en los cuales lo importante son las operaciones que ofrecen y no la representación en memoria. Los algoritmos son una versión refinada de los algoritmos abstractos del paso anterior. No obstante, son aún **algoritmos en pseudocódigo**, donde se dan cosas por supuestas.

Definir los tipos de datos para almacenar la solución suele ser mucho más difícil que definirlos para los datos de entrada. Además, modificaciones en los algoritmos pueden requerir cambios en los tipos de datos y viceversa.

Ejemplo 1.1 (cont.) Diseño de la solución. En el problema de las cifras, los tipos de datos para la entrada podrían ser una tabla de 6 enteros y otro entero para el

número buscado. Pero ¿cómo representar la solución? En este caso, la solución es una serie de operaciones sobre los números del array. Cada operación podría ser una tupla $(indice_1, indice_2, operacion)$, que significa: "operar el número de $indice_1$ en el array con el de $indice_2$ usando *operacion*". La representación de la solución sería una lista de estas tuplas. Con esta representación, el algoritmo debería tener en cuenta que $indice_1$ e $indice_2$ ya están usados. Además aparece otro nuevo número, que puede usarse con posterioridad. Entonces, puede ser necesario definir otros tipos o modificar los existentes, por ejemplo haciendo que la tupla signifique: "operar el número ..., almacenando el resultado en $indice_1$ y colocando un 0 en $indice_2$".

Ejemplo 1.2 (cont.) Diseño de la solución. En el problema de detección de caras, la entrada sería una imagen, que se puede representar con matrices de enteros, de cierto ancho y alto. Además, hay otro dato *oculto*, el modelo de lo que se considera una cara. ¿Cómo se puede modelar la forma de todas las posibles caras? ¿Qué información se debería almacenar? La representación no es nada trivial. Por ejemplo, una solución sencilla sería tener un conjunto variado de imágenes de caras de ejemplo. La representación de la solución debe indicar el número de caras y la posición de cada una; por ejemplo centro, (x_i, y_i), escala, s_i, y rotación, α_i, de cada cara. Podríamos usar una lista de tuplas de tipo $(x_i, y_i, s_i, \alpha_i)$. Con todo esto, habría que diseñar un algoritmo en pseudolenguaje que pruebe todos los valores de los parámetros anteriores, y diga dónde se encuentra una cara.

1.1.4. Implementación del diseño

El siguiente paso en la resolución del problema es la implementación. La implementación parte del diseño previo, que indica qué cosas se deben programar, cómo se estructura la solución del problema, dónde colocar cada funcionalidad y qué se espera en concreto de cada parte en la que se ha descompuesto la solución. Esta descripción de lo que debe hacer cada parte es lo que se conoce como la **especificación**. Una implementación será correcta si cumple su especificación.

La dualidad tipos/algoritmos del diseño se traslada en la implementación a estructuras/algoritmos. Para cada uno de los tipos de datos del diseño, se debe elegir una **estructura de datos** adecuada. Por ejemplo, los dos ejemplos de problemas analizados necesitan listas. Una lista se puede representar mediante una variedad de estructuras: listas enlazadas con punteros o cursores, mediante celdas adyacentes en un array, enlazadas simple o doblemente, con nodos cabecera o no, etc. La elección debe seguir los criterios de eficiencia –en cuanto a tiempo y a uso de memoria– que se hayan especificado para el problema. En cada caso, la estructura más adecuada podrá ser una u otra.

Por otro lado, la **implementación de los algoritmos** también se puede ver como una serie de tomas de decisiones, para trasladar un algoritmo en pseudocódigo a un lenguaje concreto. Por el ejemplo, si se necesita repetir un cálculo varias veces, se puede usar un bucle `for`, un `while`, se puede usar recursividad e incluso se puede poner varias veces el mismo código si el número de ejecuciones es fijo[2].

Realmente, la implementación de estructuras de datos y algoritmos no va por separado. Ambas son simultáneas. De hecho, así ocurre con los mecanismos de los lenguajes

[2] Algo nada aconsejable, de todos modos.

de programación –módulos o clases– en los que se asocia cada estructura con los algoritmos que la manejan. La organización temporal de la implementación es, más bien, por funcionalidades: cuestiones de interface con el usuario, accesos a disco, funcionalidad matemática, etc.

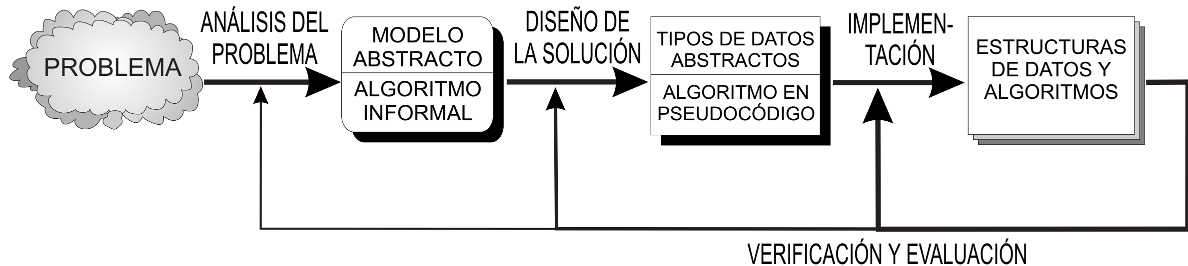


Figura 1.2: Esquema del proceso cíclico de desarrollo de software.

1.1.5. Verificación y evaluación de la solución

La resolución de un problema no acaba con la implementación de un programa[3]. Los programas deberían ser comprobados de forma exhaustiva, verificando que se ajustan a los objetivos deseados, obtienen los resultados correctos sin provocar fallos de ejecución. En la práctica, hablamos de un **proceso cíclico de desarrollo**: implementar, verificar, corregir la implementación, volver a verificar, volver a corregir, etcétera. En cierto momento, puede que tengamos que cambiar el diseño, e incluso puede que lo que esté fallando sea el modelo abstracto. Cuanto más atrás tengamos que volver, más tiempo habremos perdido haciendo cosas que luego resultan inútiles.

La evaluación de los programas incluye no sólo la corrección del resultado sino también la medición de las prestaciones, lo que normalmente se denomina el **análisis de eficiencia**. La eficiencia es una proporción entre los resultados obtenidos y los recursos consumidos. Fundamentalmente, un programa consume recursos de tiempo y de memoria. Aunque usualmente hablamos de "análisis de algoritmos", tanto las estructuras de datos como los algoritmos influyen en el consumo de recursos de un programa.

Incluso antes de llegar a la implementación de un programa, es posible y adecuado hacer el análisis del algoritmo en pseudocódigo. Este análisis será una previsión de la eficiencia que obtendría la solución diseñada si la implementamos. Se trata, por lo tanto, de un estudio teórico. Cuando el estudio se realiza sobre una implementación concreta, hablamos normalmente de estudios experimentales.

1.2. Tipos de datos

A modo de breve repaso, vamos a ver en esta sección las definiciones de tipo de datos, tipo abstracto, estructura de datos, las distintas clases de tipos y algunos de los principales

[3]O no debería. Desafortunadamente, estamos tan acostumbrados a ver programas que se cuelgan, como coches que fallan justo al salir del garaje, ¡exactamente por lo mismo: falta de pruebas!

tipos de datos usados en programación. Se supone que el alumno ya está familiarizado con todos estos conceptos. Es más, sería un interesante ejercicio pararse en este punto, cerrar el libro y tratar recordar todo lo que se pueda de los anteriores temas.

1.2.1. Definición de tipo de datos, tipo abstracto y estructura

Posiblemente, la primera duda del programador no ingeniero sería: ¿qué necesidad hay de tantos términos para referirse a lo mismo? Los conceptos de tipo abstracto de datos, tipo de datos y estructura de datos no son, en absoluto, redundantes ni incoherentes. Como ya hemos visto, aparecen en los distintos niveles de desarrollo. Un tipo abstracto es un concepto de diseño y por lo tanto previo e independiente de cualquier implementación. Un tipo de datos es un concepto de programación y se puede ver como la realización o materialización de un tipo abstracto. La estructura de datos es la organización o disposición en memoria de la información almacenada para cierto tipo de datos. Veamos, por partes, las definiciones.

Definición 1.1 Un **Tipo Abstracto de Datos** (**TAD**) está compuesto por un dominio abstracto de valores y un conjunto de operaciones definidas sobre ese dominio, con un comportamiento específico.

Por ejemplo, los booleanos constituyen un TAD, formado por el dominio de valores {*verdadero*, *falso*} y las operaciones sobre ese dominio: NO, Y, O, XOR, IMPLICA, etc. Decimos que el dominio de valores es *abstracto* porque no se le supone ninguna representación ni ningún significado. Es decir, el valor *verdadero* no tiene por qué ser un 1, ni debe almacenarse necesariamente en un bit. Pero, es más, ¿qué significa *verdadero*? Es un concepto hueco, vacío, sin ningún significado explícito. El único significado le viene dado implícitamente por su relación con las operaciones del tipo. Por ejemplo, *verdadero* Y *cualquier cosa* es siempre igual a *cualquier cosa*.

En el otro extremo de complejidad, otro ejemplo de TAD podrían ser las ventanas de un entorno Windows o XWindows. En este caso, el dominio de valores estaría compuesto por el conjunto –prácticamente infinito– de todas las ventanas en sus distintas formas, tamaños, colores y contenidos. Las operaciones sobre el dominio serían del tipo: crear ventana, mover, cambiar de tamaño, añadir botones, etc.

Las operaciones del TAD son abstractas, en el sentido de que sabemos lo que hacen pero no cómo lo hacen. A la hora de programar el TAD, esto da lugar a un principio conocido como **ocultación de la implementación**: lo importante es que se calcule el resultado esperado y no cómo se calcule. Volveremos a insistir en esto más adelante.

En cuanto que el TAD es un concepto abstracto, reside exclusivamente como un ente etéreo en la mente del programador. La realización del TAD en un lenguaje concreto, en cierto entorno de programación y por determinado programador, es lo da lugar a un tipo de datos, ahora sin el apellido "abstracto".

Definición 1.2 El **tipo de datos** de una variable, un parámetro o una expresión, determina el conjunto de valores que puede tomar esa variable, parámetro o expresión, y las operaciones que se pueden aplicar sobre el mismo.

El tipo de datos es un concepto de programación, de implementación, mientras que el TAD es un concepto matemático previo a la programación. Un tipo de datos debe ajustarse al comportamiento esperado de un TAD. Por ejemplo, el tipo `integer` de Pascal o el tipo `int` de C son dos tipos de datos, que corresponden al TAD de los enteros, **Z**.

Además de los tipos de datos elementales existentes en un lenguaje de programación, los lenguajes suelen ofrecer mecanismos para que el usuario se defina sus propios tipos. Estos mecanismos pueden ser más o menos avanzados. En lenguajes como C o Pascal, la definición del tipo indica los atributos que se almacenan para las variables de ese tipo. Las operaciones se definen por separado. Por ejemplo, en C una pila de enteros sería:

```
struct
  PilaEnteros
    {
      int datos[];
      int tope, maximo;
    };
```

En lenguajes orientados a objetos, como C++ o Java, los tipos definidos por el usuario se llaman **clases**, y su definición incluye tanto los atributos almacenados como las operaciones sobre los mismos. Por ejemplo, en C++ la definición de las pilas podría ser la siguiente:

```
class
  PilaEnteros
    {
      private:                // Atributos
        int datos[];
        int tope, maximo;
      public:                 // Operaciones
        PilaEnteros (int max);
        Push (int valor);
        int Pop();
    };
```

Volveremos a tratar los mecanismos de definición de tipos en el siguiente capítulo.

En la programación de un tipo de datos surgen dos cuestiones: cómo almacenar en memoria los valores del dominio y cómo programar las operaciones del tipo de datos. La primera cuestión implica diseñar una estructura de datos para el tipo. La segunda está relacionada con la implementación de las operaciones de manipulación y, evidentemente, estará en función de la primera.

Definición 1.3 La **estructura de datos** de un tipo de datos es la disposición en memoria de los datos necesarios para almacenar los valores de ese tipo.

Por ejemplo, para representar las pilas podemos usar una estructura de arrays como la que aparece antes, o una lista enlazada de enteros, o una lista de arrays de enteros, o un array de listas de enteros, y así sucesivamente. Por lo tanto, un tipo de datos puede ser implementado utilizando diferentes estructuras.

De forma análoga, una misma estructura de datos puede corresponder a diferentes tipos. Por ejemplo, una estructura de arrays puede usarse para implementar listas, pilas o colas de tamaño limitado. Será la forma de actuar de las operaciones la que determine si el array almacena una lista, una pila o una cola.

La utilización de estructuras de datos adecuadas para cada tipo de datos es una cuestión fundamental en el desarrollo de programas. La elección de una estructura tendrá efectos no sólo en la memoria utilizada por el programa, sino también en el tiempo de ejecución de los algoritmos. Por ejemplo, la representación de colas con arrays puede producir desperdicio de memoria, ya que el tamaño del array debe ser el tamaño de la máxima ocupación prevista de la cola. En una representación con punteros se podrá usar menos memoria, pero posiblemente el tiempo de ejecución será sensiblemente mayor.

1.2.2. Tipos de tipos

Dentro de este apartado vamos a hacer un repaso de terminología, clases y propiedades relacionadas con los tipos de datos.

Podemos hacer una primera clasificación de los tipos teniendo en cuenta si el tipo es un tipo estándar del lenguaje o no. Encontramos dos categorías:

- **Tipos primitivos o elementales**. Son los que están definidos en el lenguaje de programación. Por ejemplo, en C son tipos primitivos los tipos `int`, `long`, `char` y `float`, entre otros.

- **Tipos definidos por el usuario**. Son todos los tipos no primitivos, definidos por la misma persona que los usa, o bien por otro programador.

Idealmente, un buen lenguaje debería hacer transparente el uso de tipos de una u otra categoría. Es decir, al usar una variable de cierto tipo `T` debería ser lo mismo que `T` sea un tipo primitivo o definido por el usuario. Ambos deberían ser conceptualmente equivalentes. Pero esto no siempre ocurre. Por ejemplo, en C es posible hacer asignaciones entre tipos primitivos, pero no entre ciertos tipos definidos por el usuario. La asignación "`a= b;`" funcionará o no, dependiendo de que los tipos de `a` y `b` sean primitivos o no.

Por otro lado, podemos hacer otra clasificación –según el número de valores almacenados– en tipos simples y compuestos:

- **Tipo simple**. Una variable de un tipo simple contiene un único valor en cierto instante. Por ejemplo, los enteros, reales, caracteres y booleanos son tipos simples.

- **Tipo compuesto**. Un tipo compuesto es aquel que se forma por la unión de varios tipos simples o compuestos. Por lo tanto, una variable de tipo compuesto contiene varios valores de tipo simple o compuesto.

Los lenguajes de programación suelen ofrecer distintos mecanismos de composición para crear tipos compuestos. Los más habituales son los **arrays** y los **registros**. Un array contiene una serie de posiciones consecutivas de variables del mismo tipo, cada una identificada con un índice dentro del rango del array. Los registros se definen enumerando los campos que forman el tipo compuesto, cada uno de los cuales tiene un nombre y un

tipo (que, a su vez, puede ser simple o compuesto). Las **clases** de lenguajes orientados a objetos también se pueden ver como una manera de crear tipos compuestos, en la cual se pueden añadir atributos y operaciones al tipo.

Normalmente los tipos simples son tipos elementales y los tipos compuestos son definidos por el usuario. No obstante, también pueden existir tipos compuestos definidos en el lenguaje de programación, y tipos simples definidos por el usuario. Un mecanismo que permite al usuario definir nuevos tipos simples son los **enumerados**. Un enumerado se define listando de forma completa el dominio del tipo. Por ejemplo, el tipo Sexo se puede definir como un tipo enumerado con el dominio {masculino, femenino}.

Un **tipo contenedor**, o **colección**, es un tipo compuesto que puede almacenar un número indefinido de valores de otro tipo cualquiera. Por ejemplo, una lista de enteros o un array de Sexo son tipos contenedores. Un registro con dos enteros no sería un contenedor.

En general, interesa que los tipos contenedores puedan almacenar valores de otro tipo cualquiera; por ejemplo, que las listas puedan ser de enteros, de reales, de registros o de cualquier otra cosa. Un tipo se dice que es **genérico** o **parametrizado** si su significado depende de otro tipo que es pasado como parámetro. Por ejemplo, una lista genérica tendría la forma Lista[T], donde T es un parámetro que indica el tipo de los objetos almacenados. Podríamos tener una variable a de tipo Lista[entero] y otra variable b de tipo Lista[real]; estos casos concretos se llaman **instanciaciones** del tipo genérico.

Realmente son muy pocos los lenguajes que permiten la definición de tipos parametrizados. Entre ellos se encuentra C++, que permite parametrizar un tipo mediante el uso de plantillas `template`. Por ejemplo, un tipo genérico Pila[T] sería:

```
template <class T>
class
  Pila
    {
      private:           // Atributos
        T datos[];
        int tope, maximo;
      public:            // Operaciones
        Pila (int max);
        Push (T valor);
        T Pop();
    };
```

En la parte de prácticas se profundizará más en el uso de plantillas como una manera de crear tipos genéricos.

Finalmente, podemos hacer una distinción entre tipos mutables e inmutables. Decimos que un tipo de datos es **inmutable** cuando los valores del tipo no cambian después de haber sido creados. El tipo será **mutable** si los valores pueden cambiar. ¿Qué implicaciones tiene que un tipo sea mutable o inmutable? Supongamos que queremos definir el tipo Lista[T]. Si el tipo se define con una representación mutable, podemos incluir operaciones que añaden o suprimen elementos de la lista.

operación Inserta (**var** *lista*: Lista[T]; *elemento*: T)
operación SuprimePrimero (**var** *lista*: Lista[T])

Donde *lista* es un parámetro que se puede modificar dentro de las operaciones. Pero si el tipo es inmutable no sería posible que un parámetro de entrada se modificara. En ese caso, las operaciones de inserción y eliminación deberían devolver una nueva lista, con el elemento correspondiente insertado o eliminado.

operación Inserta (*lista*: Lista[T]; *elemento*: T): Lista[T]
operación SuprimePrimero (*lista*: Lista[T]): Lista[T]

Según nos interese, definiremos los tipos como mutables o inmutables. Normalmente, la mayoría de los tipos serán mutables, aunque en algunos casos nos interesará usar tipos inmutables: la suposición de que los valores del tipo no cambian puede simplificar, o hacer más eficiente, la implementación de la estructura de datos para el tipo.

1.2.3. Repaso de tipos y pseudolenguaje de definición

La mayoría de los lenguajes ofrecen al programador un conjunto similar de tipos de datos simples (enteros, reales, booleanos, etc.), con las operaciones necesarias para manejarlos. Estos tipos tienen una correspondencia con los tipos manejados por el procesador, por lo que su implementación es muy eficiente. Vamos a ver los que nos podemos encontrar más habitualmente. Para cada uno de ellos, indicaremos también la notación que se usará en el resto del libro para referirse a ese tipo, cuando usemos el pseudolenguaje[4].

- **Enteros.** Con signo o sin signo, con precisión simple, doble o reducida (un byte). En nuestro pseudocódigo supondremos un único tipo entero, con las operaciones habituales.
- **Reales.** Con precisión simple o doble. Para los números reales se suelen usar representaciones mantisa-exponente. Cuando lo usemos, pondremos el tipo real.
- **Caracteres.** Lo denotaremos como caracter.
- **Cadenas.** En algunos lenguajes las cadenas de caracteres son tipos elementales. En otros, como en C, las cadenas no son un tipo elemental sino que son simplemente arrays de caracteres. En nuestro caso, suponemos que tenemos el tipo cadena con operaciones para concatenar o imprimir por pantalla.
- **Booleanos.** Igual que antes, no existen en C –donde se usan enteros en su lugar–, aunque sí en C++ y Pascal. Supondremos que tenemos el tipo booleano.

En cuanto a los tipos contenedores (listas, pilas, colas, árboles, etc.), normalmente no forman parte del lenguaje de programación, sino que se definen en librerías de utilidades que se pueden incluir en un programa o no. No obstante, sí que se suelen incluir mecanismos de composición de tipos como arrays, registros y tipos enumerados. En nuestro pseudolenguaje tenemos la posibilidad de definir nuevos tipos, mediante una cláusula **tipo**. Además, permitimos que se definan tipos parametrizados. Por ejemplo, podemos tener los siguientes tipos:

[4]Este pseudolenguaje de definición de tipos sólo indica la estructura de datos del tipo. Las operaciones son descritas aparte. Al implementar los tipos –sobre todo usando un lenguaje orientado a objetos– no hay que olvidar que las estructuras y las operaciones deben ir juntos, en el mismo módulo o clase.

```
tipo
    DiasMeses = array [1..12] de entero
    Pila[T] = registro
        datos: array [1..MAXIMO] de T
        tope, maximo: entero
    finregistro
    Sexo = enumerado (masculino, femenino)
```

Para los arrays, se indica entre corchetes, [*min..max*], el rango mínimo y máximo de los índices del array[5]. Para los registros, se listan sus atributos, cada uno con su nombre y tipo. Y, por otro lado, los enumerados se forman listando los valores del dominio.

En algunos sitios usaremos también registros con variantes. Un **registro con variantes** es un registro que contiene un campo *especial*, en función del cual una variable podrá tener unos atributos u otros (pero nunca ambos a la vez). Por ejemplo, si en una aplicación que almacena empleados de una empresa, queremos almacenar diferente información para los hombres y las mujeres podemos tener algo como lo siguiente:

```
tipo
    Empleado = registro
        nombre: cadena
        edad: entero
        según sexo: Sexo
            masculino: (dirección: cadena; sueldo: entero)
            femenino: (salario: entero; teléfono: entero)
        finsegún
    finregistro
```

En cuanto a los tipos colecciones, además de los arrays daremos por supuesto que disponemos de los siguientes:

- **Listas.** Tendremos listas parametrizadas, Lista[T], y que guardan una posición actual. De esta forma, la inserción, eliminación, etc., se hacen siempre sobre la posición actual. Por conveniencia, en algunos sitios no se darán por supuestas las listas.
- **Pilas.** Una pila es una lista LIFO: last in, first out (último en entrar, primero en salir). Suponemos el tipo genérico Pila[T], con las operaciones push, pop y tope.
- **Colas.** Una cola es una lista FIFO: first in, first out (primero en entrar, primero en salir). Suponemos el tipo genérico Cola[T], con las operaciones meter, sacar y cabeza.

Por último, pero no menos importante, tenemos el tipo de datos **puntero**. Considerado como un tipo *non-grato* por algunos autores[6] por tratarse de un concepto cercano al bajo nivel, lo cierto es que los punteros son esenciales en la definición de estructuras de datos. Los punteros son un tipo parametrizado. Una variable de tipo Puntero[T] contiene una referencia, o dirección de memoria, donde se almacena una variable de tipo T.

[5] Ojo, hay que tener en cuenta que en lenguajes como C/C++ el índice mínimo es siempre 0, así que sólo se indica el tamaño del array.

[6] Por ejemplo, según C.A.R. Hoare, "su introducción en los lenguajes de alto nivel fue un paso atrás del que puede que nunca nos recuperemos". En el lenguaje Java se decidió no permitir el uso de punteros, aunque dispone de **referencias**, que vienen a ser lo mismo.

En nuestro pseudolenguaje suponemos que tenemos el tipo Puntero[T] con las siguientes operaciones y valores especiales:

- **Operador de indirección.** Si a es de tipo Puntero[T], entonces la variable apuntada por a se obtiene con el operador de indirección flecha, ↑, esto es: a↑ es de tipo T.
- **Obtención de la dirección.** Si t es una variable de tipo T, entonces podemos obtener la dirección de esa variable con la función PunteroA(t: T), que devuelve un puntero de tipo Puntero[T].
- **Puntero nulo.** Existe un valor especial del tipo Puntero[T], el valor predefinido NULO, que significa que el puntero no tiene una referencia válida o no ha sido inicializado.
- **Creación de una nueva variable.** Para crear una nueva variable apuntada por un puntero, suponemos la función genérica Nuevo(T: Tipo), que devuelve un Puntero[T] que referencia a una nueva variable creada. Esta variable se borrará con la función Borrar(a: Puntero[T]).
- **Comparación.** Suponemos que hay definidos operadores de comparación de igualdad y desigualdad entre punteros ($=$, $\neq$).

No consideramos otras operaciones sobre punteros, al contrario de lo que ocurre con C/C++, donde los punteros son considerados como enteros, pudiendo aplicar sumas, restas, etc. Por esta razón, entre otras, decimos que lenguajes como C/C++ son lenguajes **débilmente tipados**: existen pocas restricciones en la asignación, mezcla y manipulación de tipos distintos. Como contraposición, los lenguajes **fuertemente tipados**, como Pascal, tienen reglas más estrictas en el sistema de compatibilidad de tipos. En la práctica, esta flexibilidad de C/C++ le proporciona una gran potencia pero es fuente de numerosos errores de programación.

1.3. Algoritmos y algorítmica

Como ya hemos visto, los algoritmos junto con las estructuras de datos constituyen los dos elementos imprescindibles en el proceso de resolución de problemas. Los primeros definen el componente manipulador y los segundos el componente almacenado, y se combinan estrechamente para crear soluciones a los problemas.

No está de más recordar que la historia de los algoritmos es mucho anterior a la aparición de los ordenadores. De hecho, podríamos datar la aparición de los algoritmos al primer momento en que los seres humanos se plantearon resolver problemas de forma genérica. Históricamente uno de los primeros algoritmos inventados, para la resolución de problemas de tipo matemático, es el algoritmo de **Euclides** de Alejandría, Egipto, propuesto alrededor del año 300 a.C. Euclides propone el siguiente método para calcular el máximo común divisor de dos números enteros.

Ejemplo 1.3 Algoritmo de Euclides para calcular el máximo común divisor de dos números enteros, a y b, siendo $a > b$.

1. Hacer r:= a módulo b
2. Si $r = 0$ entonces el máximo común divisor es b
3. En otro caso:
   3.1. Hacer a:= b
   3.2. Hacer b:= r
   3.3. Volver al paso 1

Por ejemplo, si aplicamos el algoritmo con $a = 20$ y $b = 15$, entonces $r = 5$. No se cumple la condición del paso 2 y aplicamos el paso 3. Obtenemos $a = 15$, $b = 5$. Volvemos al punto 1, llegando a $r = 0$. Al ejecutar el paso 2 se cumple la condición, con lo que el resultado es 5.

Pero el honor de dar nombre al término *algoritmo* lo recibe el matemático árabe del siglo IX **Muhammad ibn Musa al-Khwarizmi** (que significa algo así como Mohamed hijo de Moisés de Khorezm, en Oriente Medio). Sus tratados sobre la resolución de ecuaciones de primer y segundo grado ponen de relieve la idea de resolver cálculos matemáticos a través de una serie de pasos predefinidos. Por ejemplo, para resolver ecuaciones de segundo grado del tipo: $x^2 + bx = c$, define la siguiente serie de pasos[7].

Ejemplo 1.4 Algoritmo de al-Khwarizmi para resolver la ecuación $x^2 + bx = c$. Se muestra el ejemplo $x^2 + 10x = 39$, el mismo que al-Khwarizmi utiliza en su libro "Hisab al-jabr w'al-muqabala" ("El cálculo de reducción y restauración"), sobre el año 825 d.C.
1. Tomar la mitad de b. (En el ejemplo, $10/2 = 5$)
2. Multiplicarlo por sí mismo. (En el ejemplo, $5 * 5 = 25$)
3. Sumarle c. (En el ejemplo, $25 + 39 = 64$)
4. Tomar la raíz cuadrada. (En el ejemplo, $\sqrt{64} = 8$)
5. Restarle la mitad de b. (En el ejemplo, $8 - 10/2 = 8 - 5 = 3$)

El resultado es: $\sqrt{(b/2)^2 - c} - b/2$, expresado algorítmicamente. El matemático bagdadí es también considerado como el introductor del sistema de numeración arábigo en occidente, y el primero en usar el número 0 en el sistema posicional de numeración.

1.3.1. Definición y propiedades de algoritmo

¿Qué es un algoritmo? En este punto, el lector no debería tener ningún problema para dar su propia definición de algoritmo. Desde los ejemplos de algoritmos más antiguos –como los de Euclides y al-Khwarizmi, mostrados antes– hasta los más modernos algoritmos –como los usados por el buscador Google o en compresión de vídeo con DivX– todos ellos se caracterizan por estar constituidos por una serie de pasos, cuya finalidad es producir unos datos de salida a partir de una entrada. Además, para que ese conjunto de reglas tenga sentido, las reglas deben ser precisas y terminar en algún momento. Así pues, podemos dar la siguiente definición informal de algoritmo.

Definición 1.4 Un **algoritmo** es una serie de reglas que dado un conjunto de **datos de entrada** (posiblemente vacío) produce unos **datos de salida** (por lo menos una) y cumple las siguientes propiedades:

[7]La notación es adaptada, ya que al-Khwarizmi no utiliza símbolos, expresiones ni variables, sino una descripción completamente textual. Por ejemplo, a b lo llama "las raíces" y a c "las unidades".

- **Definibilidad.** El conjunto de reglas debe estar definido sin ambigüedad, es decir, no pueden existir dudas sobre su interpretación.

- **Finitud.** El algoritmo debe constar de un número finito de pasos, que se ejecuta en un tiempo finito y requiere un consumo finito de recursos.

En la figura 1.3 se muestra una interpretación de los algoritmos como cajas negras, donde se destaca la visión del algoritmo como algo que dado unos datos de entrada produce una salida. La propiedad de definibilidad se refiere a los algoritmos "en papel", que también son algoritmos si están expresados sin ambigüedades. Un trozo de código implementado en un lenguaje de programación no tendrá ambigüedad, pero también deberá tener una duración finita para ser considerado un algoritmo. El problema general de demostrar la terminación de un conjunto de reglas es un problema complejo y no computable, es decir no puede existir ningún programa que lo resuelva.

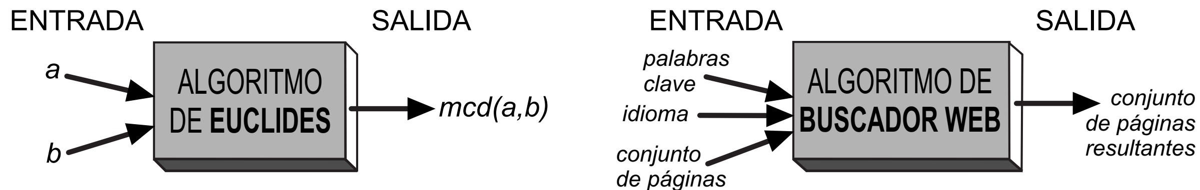


Figura 1.3: Interpretación de los algoritmos como cajas negras.

No todo lo que aparece en informática son algoritmos. Un ejemplo típico de algo que no es un algoritmo es un sistema operativo. El sistema operativo consta de una serie de instrucciones, pero no se le prevé una ejecución finita, sino que debería poder ejecutarse mientras no se requiera lo contrario.

Algoritmos deterministas y no deterministas

Según un algoritmo produzca o no siempre los mismos valores de salida para las mismas entradas, clasificamos los algoritmos en dos tipos:

- **Algoritmos deterministas.** Para los mismos datos de entrada producen siempre los mismos datos de salida.

- **Algoritmos no deterministas.** Para los mismos datos de entrada pueden producir diferentes salidas.

El no determinismo puede chocar un poco con la idea de definibilidad. Sin embargo, ambas propiedades no son contradictorias. Por ejemplo, el resultado de un algoritmo paralelo, que se ejecuta en distintos procesadores, puede depender de cuestiones de temporización que no forman parte de los datos del problema.

La existencia del no determinismo hace que un algoritmo no siempre se pueda ver como una función –en sentido matemático– de la forma f: $ENTRADA \rightarrow SALIDA$.

Definición formal de algoritmo

Donald E. Knuth, uno de los grandes pioneros en el estudio de los algoritmos y la programación, propone la siguiente definición formal de algoritmo.

Definición 1.5 Un **método de cálculo** es una cuaterna (Q, I, W, f), en la cual:

- I es el conjunto de **estados de entrada**;
- W es el conjunto de **estados de salida**;
- Q es el conjunto de **estados de cálculo**, que contiene a I y W; y
- f es una función: $f : Q \rightarrow Q$, con $f(w) = w; \forall\ w \in W$.

Definición 1.6 Una **secuencia de cálculo** es una serie de estados: $x_0, x_1, x_2, ..., x_n, ...$, donde $x_0 \in I$ y $\forall k \geq 0; f(x_k) = x_{k+1}$. Se dice que la secuencia de cálculo acaba en n pasos si n es el menor entero con $x_n \in W$.

Una definición formal como esta nos puede servir para comprobar, de manera rigurosa, si un algoritmo cumple las propiedades de finitud y definibilidad. Sin embargo, intentar resolver un problema definiendo la función f puede resultar una tarea bastante compleja. Y, lo que es peor, el resultado puede que nos aporte pocas ideas sobre cómo implementar el algoritmo en un lenguaje de programación. En lo sucesivo, intentaremos seguir una visión más práctica, pero formalizando en aquellos sitios donde sea conveniente.

La disciplina de la algorítmica

Si bien, como hemos visto, la historia de los algoritmos se remonta muy atrás, el estudio de los algoritmos no se concibió como una disciplina propia hasta bien entrada la mitad del pasado siglo XX. Actualmente, entendemos la **algorítmica** como la disciplina, dentro del ámbito de la informática, que estudia técnicas para construir algoritmos eficientes y técnicas para medir la eficiencia de los algoritmos. En consecuencia, la algorítmica consta de dos grandes áreas de estudio: el análisis y el diseño de algoritmos.

Pero, ¿cuál es el objetivo último de la algorítmica?, ¿qué es lo que motiva su estudio? El objetivo último es dado un problema concreto ser capaz de resolverlo de la mejor forma posible, de forma rápida, corta, elegante y fácil de programar. Y recordemos que en esta resolución entran también en juego las estructuras de datos, que son manejadas por los algoritmos. En definitiva, todos los ejemplos, técnicas, métodos y esquemas que vamos a estudiar tienen como finalidad última servir de herramientas útiles en el momento de afrontar la resolución de un problema completamente nuevo y desconocido.

1.3.2. Análisis de algoritmos

El análisis de algoritmos es la parte de la algorítmica que estudia la forma de medir la **eficiencia** de los algoritmos. Pero ¿cuándo decimos que un algoritmo es más o menos eficiente? Utilizando un punto de vista empresarial, podemos definir la eficiencia como la relación entre recursos consumidos y productos obtenidos. Se trata, por lo tanto, de una

nueva formulación de la bien conocida *ley del mínimo esfuerzo*: maximizar los resultados, minimizando el esfuerzo. Los **resultados** que ofrece un algoritmo pueden ser de distintos tipos:

- Un algoritmo puede resolver el problema de forma muy precisa, si es de tipo matemático, o garantizar el óptimo, en problemas de optimización, o encontrar siempre la solución, si es de satisfacción de restricciones.
- Otro algoritmo puede ser que sólo encuentre soluciones aproximadas, o más o menos cercanas al óptimo, pero siempre encuentre alguna.
- Finalmente, otro algoritmo puede que encuentre soluciones sólo en determinados casos, buenas o malas, y en otros casos acabe sin devolver ninguna respuesta.

Por otro lado, los **recursos** que consume un algoritmo pueden ser de distintos tipos. Entre los más importantes tenemos:

- Tiempo de ejecución, desde el inicio del programa hasta que acaba.
- Utilización de memoria principal.
- Número de accesos a disco, a la red o a otros periféricos externos.
- Número de procesadores usados, en el caso de los algoritmos paralelos, etc.

En aplicaciones distintas puede que el recurso crítico sea diferente. En la mayoría de los casos, el recurso clave es el tiempo de ejecución, por lo que muchas veces se asocia eficiencia con tiempo. Pero, recordemos, ni el tiempo es el único recurso, ni el tiempo por sí solo mide la eficiencia, si no que se debe contrastar con los resultados obtenidos.

En conclusión, un algoritmo será mejor cuanto más eficiente sea. No obstante, el sentido común nos dice que habrán otros muchos criterios para decidir lo bueno que es un algoritmo. Por ejemplo, será importante: que sea fácil de entender y de programar, que sea corto en su extensión, que sea robusto frente a casos extraños, que se pueda reutilizar en otros problemas, que se pueda adaptar, etc.

Factores en la medición de recursos

Supongamos el siguiente algoritmo sencillo, que realiza una búsqueda secuencial con centinela dentro de un array de enteros. La cuestión que se plantea es ¿cuántos recursos, de tiempo de ejecución y memoria, consume el algoritmo?

```
operación BusquedaSec (x, n: entero; a: array [1..n+1] de entero): entero
    i:= 0
    a[n+1]:= x
    repetir
        i:= i + 1
    hasta a[i] = x
    devolver i
```

Por muy perdido que ande el alumno, difícilmente habrá contestado algo como “tarda 3 segundos” o “requiere 453 Kbytes...”. Como mínimo, está claro que el tiempo y la

memoria dependen de n, de los datos de entrada, del procesador, de que estemos grabando un CD al mismo tiempo y de otras mil cosas más. Podemos clasificar todos los factores que intervienen en el consumo de recursos en tres categorías.

- **Factores externos.** No indican nada relevante sobre las características del algoritmo; son, más bien, cuestiones externas al algoritmo, relacionadas con el entorno en el que se implementa y ejecuta. Entre los principales factores externos tenemos: el ordenador donde se ejecuta el algoritmo, el uso de procesador por parte de otras tareas, el sistema operativo, el lenguaje de programación usado, el compilador (incluyendo las opciones de compilación usadas), la implementación concreta que se haga del algoritmo, etc.
- **Tamaño del problema.** El tamaño es el volumen de datos de entrada que debe manejar el algoritmo. Normalmente, el tamaño viene dado por uno o varios números enteros. La relación del tiempo con respecto al tamaño sí que resulta interesante sobre las características del algoritmo.
- **Contenido de los datos de entrada.** Para un tamaño de entrada fijo y bajo las mismas condiciones de ejecución, un algoritmo puede tardar distinto tiempo para diferentes entradas. Hablamos de diferentes casos: **mejor caso**, es el contenido de los datos de entrada que produce la ejecución más rápida del algoritmo (incluso para tamaños de entrada grandes); **peor caso**, es el que da lugar a la ejecución más lenta para un mismo tamaño; **caso promedio**, es el caso medio de todos los posibles valores de entrada.

Por ejemplo, en el anterior algoritmo de búsqueda secuencial con centinela, el tamaño del problema viene dado por n. No es lo mismo aplicar el algoritmo con $n = 5$ que con $n = 5.000.000$. Pero la ejecución con $n = 5.000.000$ puede tardar menos que con $n = 5$, si el x buscado se encuentra en la primera posición. Así pues, el mejor caso será que $a[1] = x$, independientemente de lo que valga n. El peor caso será que x no se encuentre en el array, con lo cual la búsqueda acabará al llegar a $a[n + 1]$. El caso promedio sería la media de todas las posibles entradas. Supongamos que x está en a con probabilidad p y que, en caso de estar, todas las posiciones tienen la misma probabilidad. Entonces se recorrerán $n/2$ posiciones con probabilidad p, y $n + 1$ posiciones con probabilidad $1 - p$.

Notaciones en el análisis de algoritmos

Nos interesa que el análisis de los recursos consumidos por un algoritmo sea independiente de los factores externos. En consecuencia, el objetivo es estudiar la variación del tiempo y la memoria utilizada, en función del tamaño de la entrada y, posiblemente, para los casos mejor, peor y promedio, en caso de ser distintos.

Esta dependencia se expresa como una o varias funciones matemáticas. El tiempo será normalmente una función $t : N \rightarrow R^+$, y de forma parecida la memoria $m : N \rightarrow R^+$. Pero si el algoritmo tarda distinto tiempo para la misma n, entonces no se puede decir que t sea una función de N en R^+. En ese caso, definimos los tiempos en los casos mejor, peor y promedio; denotaremos estos tiempos como t_m, t_M y t_p, respectivamente.

En concreto, ¿qué se mide con $t(n)$? Pues lo que más nos interese.

- Podemos medir **tiempos de ejecución**, en unidades de tiempo, asignando constantes indefinidas a cada instrucción o línea de código. En el algoritmo de búsqueda secuencial, tendríamos los siguientes tiempos:
  $t_m(n) = c_1 + c_2 + c_4 + c_5 + c_6$; $t_M(n) = c_1 + c_2 + c_6 + (c_4 + c_5)(n + 1)$;
  $t_p(n) = c_1 + c_2 + c_6 + (c_4 + c_5)(1 + (1 - p)n + p\ n/2)$
  Siendo cada c_i el tiempo de ejecución de la instrucción de la línea i-ésima.

- También podemos medir simplemente el **número de instrucciones**. Tendríamos:
  $t_m(n) = 5$; $t_M(n) = 5 + 2n$; $t_p(n) = 5 + 2((1 - p)n + p\ n/2)$
  No todas las instrucciones tardan el mismo tiempo, pero sabemos que todas ellas tardan como máximo un tiempo que está acotado por una constante.

- Puede que lo que nos interese medir sea únicamente **algún tipo de instrucción**, que sabemos que es la más relevante. Por ejemplo, si contamos el número de comparaciones en el algoritmo de búsqueda secuencial tenemos:
  $t_m(n) = 1$; $t_M(n) = 1 + n$; $t_p(n) = 1 + (1 - p/2)n$

Las tres versiones de cada una de las funciones t_m, t_M y t_p anteriores, nos vienen a decir lo mismo: en el mejor caso el tiempo es constante, en el peor es proporcional a n, y en promedio depende de n y de p. Pero seguro que la última forma, contando sólo comparaciones, se puede interpretar más fácilmente, porque es la más sencilla.

Para simplificar las expresiones de $t(n)$ usamos las notaciones asintóticas. Las **notaciones asintóticas** son notaciones que sólo tienen en cuenta el crecimiento asintótico de una función –es decir, para valores tendiendo a infinito– y son independientes de las constantes multiplicativas. Tenemos las siguientes notaciones:

- **O-grande: orden de complejidad.** Establece una cota superior de la función, asintóticamente e independiente de constantes multiplicativas.

- **Ω: omega.** Establece una cota inferior, asintótica e independiente de constantes.

- **Θ: orden exacto.** Se usa cuando la cota inferior y superior de una función coinciden.

- **o-pequeña.** Es similar a la notación Θ, pero teniendo en cuenta el factor que multiplica al término de mayor grado.

Podemos encontrar diferentes utilidades a las notaciones asintóticas:

- **Simplificar la notación** de una fórmula con expresión larga y complicada. Por ejemplo, dado un $t(n) = 2n^2 + 7{,}2n\ log_3 n - 21\pi n + 2\sqrt{n+3}$, podemos quedarnos simplemente con que $t(n) \in \Theta(n^2)$, o también $t(n) \in o(2n^2)$.

- **Acotar una función** que no se puede calcular fácilmente. Esto puede ocurrir en algunos algoritmos cuyo comportamiento es difícil de predecir. Supongamos, por ejemplo, la función $t(n) = 8n^{5-cos(n)}$, similar a la mostrada en la figura 1.4a). Se puede acotar superiormente con $t(n) \in O(n^6)$ e inferiormente con $t(n) \in \Omega(n^4)$.

- **Delimitar el rango** de variación del tiempo de un algoritmo, cuando no siempre tarda lo mismo. Por ejemplo, en el algoritmo de búsqueda secuencial tenemos distintos casos mejor y peor. Podemos decir que tiene un $O(n)$ y un $\Omega(1)$.

En la figura 1.4 se muestran dos ejemplos de aplicación de las notaciones asintóticas, para los dos últimos casos.

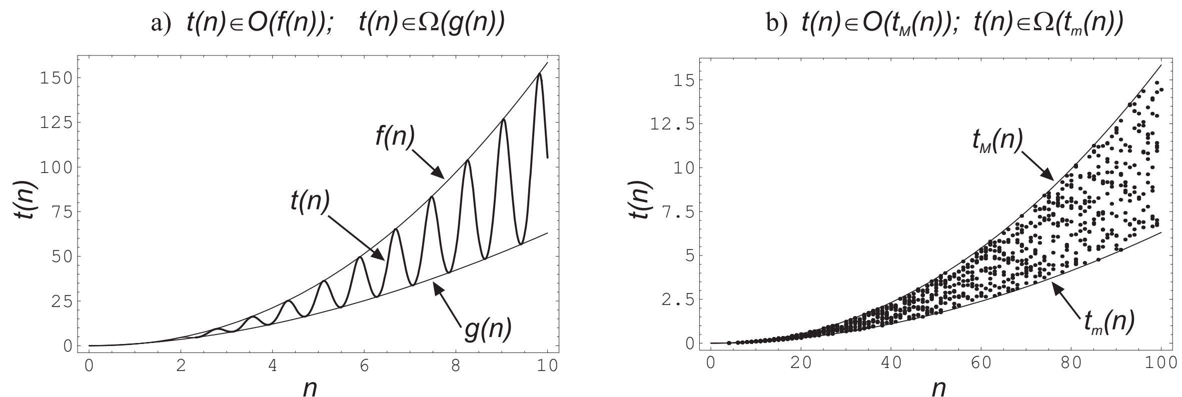


Figura 1.4: Distintas aplicaciones de las notaciones asintóticas. a) Para acotar una función oscilante o difícil de calcular con precisión. b) Para acotar un algoritmo que no siempre tarda lo mismo para el mismo n.

Las notaciones asintóticas pueden servir también para comparar fácilmente los tiempos de dos algoritmos. Pero hay que tener en cuenta sus limitaciones, derivadas de la independencia de las constantes y de los términos de menor grado. Por ejemplo, un algoritmo puede tener un tiempo $t_1(n) = 10n^{0,01} + 2$ y otro $t_2(n) = 2\log_2 n + 30$. Las notaciones nos dicen que $O(\log\ n) < O(n^{0,01})$. Sin embargo, ¡$t_1(n)$ sólo será mayor que $t_2(n)$ a partir de valores de n mayores que $4{,}5 * 10^{216}$!

1.3.3. Diseño de algoritmos

El diseño de algoritmos es la parte de la algorítmica que estudia **técnicas generales de construcción de algoritmos** eficientes. El objetivo no es, por lo tanto, estudiar unos cuantos algoritmos dados, sino desarrollar técnicas generales que se pueden aplicar sobre diversos tipos de problemas. Por ejemplo, divide y vencerás no es un algoritmo, sino una técnica que puede ser aplicada sobre una amplia variedad de problemas.

En primer lugar, cada técnica de diseño de algoritmos propone una **interpretación** particular del problema, es decir, una forma de ver o modelar el objetivo del problema. Esta interpretación consta de una serie de **elementos genéricos**, que deben ser concretados en cada caso. Por ejemplo, divide y vencerás interpreta que el problema trabaja con una serie de datos, que se pueden dividir para tener problemas más pequeños. Los elementos genéricos de la técnica serían: la manera de hacer la división de los datos, la forma de juntar las soluciones, una forma de resolver casos de tamaño pequeño, etc. Los componentes pueden ser tanto operaciones como estructuras de datos.

Habiendo resuelto los componentes genéricos del problema, la técnica propone un **esquema algorítmico**, que es una guía de cómo combinar los elementos para construir algoritmos basados en esa técnica. El esquema algorítmico puede ser más o menos detallado. Idealmente, el esquema algorítmico sería como un "algoritmo con huecos", correspondientes a los elementos genéricos. El programador simplemente debería insertar el código y los tipos adecuados en los huecos que correspondan, y ya tendría un algoritmo que resuelve el problema. La idea se muestra en la figura 1.5.

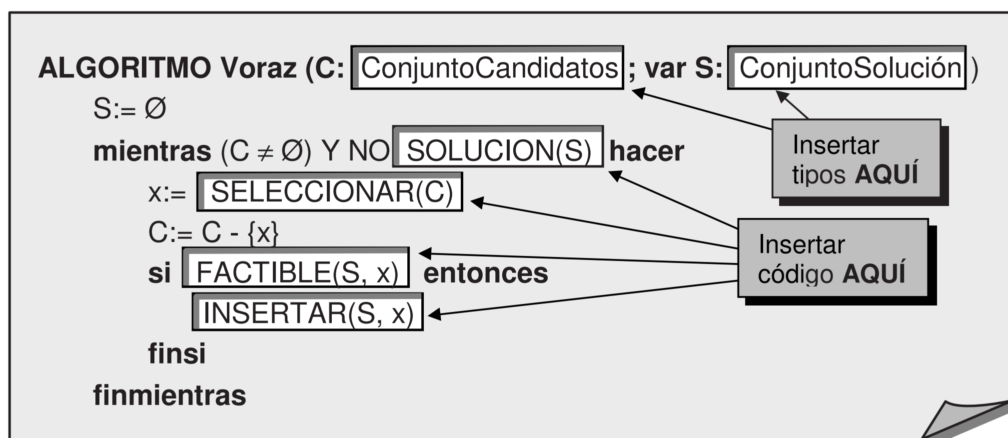


Figura 1.5: Interpretación de los esquemas algorítmicos como "algoritmos con huecos". Esquema de algoritmo voraz.

En la práctica, esta idea de construir algoritmos "rellenando huecos de un esquema" no suele ser tan simple ni extendida, por varios motivos. En primer lugar, la variedad de situaciones que nos podemos encontrar en distintos problemas es, normalmente, mucho mayor que la prevista en los esquemas de algoritmos. Pero, por otro lado, en esquemas excesivamente flexibles el problema suele ser la ineficiencia; un algoritmo escrito desde cero para un problema particular puede incluir código optimizado para ese caso dado. De todas formas, los esquemas algorítmicos son una ayuda muy útil en la construcción de algoritmos, por lo menos para la obtención de una primera versión.

Otra cuestión fundamental en la resolución de problemas es decidir qué técnica se puede aplicar en cada caso. Aunque puede que ocurra en un examen, en un problema de la vida real nadie nos dirá "prográmame esto utilizando backtracking" o "calcula aquello con programación dinámica"; el que lo resuelve debe decidir también cómo. Puede que se puedan aplicar varias técnicas, que alguna de ellas se pueda aplicar de diferentes modos, que distintas técnicas den diferentes resultados –por ejemplo, una aproxima la solución y otra encuentra el óptimo– o que no se pueda aplicar ninguna técnica. En el último caso, entrará en juego la habilidad de la persona para enfrentarse a situaciones nuevas.

Vamos a adelantar algunas de las principales técnicas generales de diseño de algoritmos, que veremos a lo largo de este libro. Se supone que el alumno está un poco familiarizado con alguna de ellas –como divide y vencerás, o backtracking–, aunque no resulta imprescindible.

Divide y vencerás

La técnica de **divide y vencerás** aporta una idea muy general: para resolver un problema primero divídelo en partes, luego resuélvelas por separado y finalmente junta las soluciones de cada parte. La idea es tan abierta que se podría aplicar casi en cualquier ámbito[8]. En programación, los subproblemas suelen ser del mismo tipo que el problema original, por lo que se resuelven aplicando recursividad.

Cualquier recursividad necesita un caso base (o varios), que actúe como punto de finalización. Los casos base en divide y vencerás serán problemas de tamaño pequeño, para los cuales se aplica una solución directa.

Por lo tanto, para que se pueda aplicar divide y vencerás: el problema se debe poder descomponer en partes, las soluciones de las partes deben ser independientes, tiene que haber algún método directo de resolución y debe haber una manera de combinar las soluciones parciales. Habrá muchos casos donde no se cumplan estas condiciones.

Algoritmos voraces

La interpretación de un problema bajo la perspectiva de los **algoritmos voraces** es básicamente la siguiente: tenemos un conjunto de candidatos y la solución se construye seleccionando algunos de ellos en cierto orden. Con este modelo del problema, el esquema voraz propone que la solución se construya paso a paso. Se empieza con una solución vacía. En cada paso se selecciona un candidato y se decide si se añade a la solución o no. Una vez tomada una decisión (añadir o descartar un candidato) nunca se puede deshacer. El algoritmo acaba cuando tengamos una solución o cuando no queden candidatos.

El esquema corresponde al viejo algoritmo de "comprar patatas en el mercado": tenemos un montón de patatas candidatas y una bolsa vacía; lo que hacemos es ir llenando la bolsa con las mejores patatas, una a una, hasta completar 5 kilos. Los componentes genéricos que habría que estudiar en cada problema son: cuáles son los candidatos, qué criterio se usa para seleccionar un candidato del montón, el criterio para decir si se añade el candidato seleccionado, el criterio de finalización, etc.

Por la propia estructura del algoritmo, se puede intuir que los algoritmos voraces serán normalmente bastante rápidos. Como contrapartida, puede que no siempre encuentren solución óptima, aunque algunos de ellos sí que la garantizan.

Programación dinámica

La **programación dinámica** coincide con divide y vencerás en la idea de definir un problema recursivamente en base a subproblemas de menor tamaño. Sin embargo, en programación dinámica no se utiliza la recursividad de forma directa. En su lugar, se resuelven primero los problemas de tamaño pequeño, anotando la solución en una tabla. Usando las soluciones de la tabla, vamos avanzando hacia problemas cada vez más grandes, hasta llegar al problema original. La memoria ocupada por la tabla se convierte en uno de los factores críticos de los algoritmos de programación dinámica.

[8]De hecho, la expresión "divide y vencerás" se atribuye a Napoleón Bonaparte, que fue fiel a su aplicación en sus estrategias militares, hasta que en Waterloo, en 1815, se convirtió en "divide y perderás".

Para obtener la descomposición recursiva, el problema se puede interpretar como una serie de toma de decisiones. Después de cada decisión, nos queda un problema de menor tamaño. Por ejemplo, el problema de comprar patatas podría interpretarse como: decidir para cada patata del montón si se toma o no (*true* o *false*). La decisión se hace resolviendo los subproblemas de usar el montón con una patata menos, suponiendo que la otra ya se ha tomado o no[9]. El resultado será la mejor de ambas posibilidades.

Un algoritmo de programación dinámica encontrará la solución óptima a un problema si cumple el principio de **optimalidad de Bellman**: una secuencia de decisiones óptimas se forma uniendo subsecuencias óptimas. No en todos los problemas se puede aplicar el principio, en muchos de ellos es difícil encontrar la definición recursiva y en otros demasiado ineficiente.

Backtracking

La técnica de **backtracking** se puede ver como la aplicación de una estrategia opuesta a los algoritmos voraces. Si en los algoritmos voraces nunca se deshacen las decisiones tomadas, en backtracking se pueden deshacer, quitar elementos que hemos añadido, probar otros. Y así hasta haber comprobado virtualmente todas las posibilidades.

En un algoritmo de backtracking, las soluciones al problema suelen representarse mediante tuplas $(s_1, s_2, ..., s_n)$, donde cada s_i significará algo concreto y podrá tomar ciertos valores. El esquema genérico de backtracking consiste básicamente en generar todos los posibles valores consistentes de la tupla anterior –haciendo un recorrido exhaustivo y sistemático– y quedarse con la óptima o la que solucione el problema.

El orden de generación de los distintos valores de la tupla se puede representar mediante un árbol –el árbol implícito de backtracking, no necesariamente almacenado– donde la raíz es el comienzo del algoritmo y los hijos de un nodo son las tuplas que se generan a partir de una solución parcial. La técnica de backtracking es muy flexible y se puede aplicar sobre una gran variedad de problemas. Desafortunadamente, los algoritmos de backtracking suelen ser extremadamente costosos en tiempo de ejecución.

Ramificación y poda

En cierto sentido, la **ramificación y poda** se puede ver como una generalización y mejora de backtracking. Realiza la misma interpretación del problema, a través del uso de una tupla solución $(s_1, s_2, ..., s_n)$, cuyos valores son generados mediante un recorrido implícito en un árbol. Pero difiere de backtracking en dos aspectos. ¿Cuáles?

Pues, obviamente, en la ramificación y en la poda. En cuanto a la ramificación, permite definir distintas estrategias para recorrer el árbol, es decir, determinar qué ramas explorar primero. En cuanto a la poda, define un proceso de eliminación de nodos del árbol por los cuales sabemos que no se encuentra la solución óptima.

La técnica se basa en ir haciendo estimaciones, para cada solución parcial, de la mejor solución que podemos encontrar a partir de la misma. Si la mejor solución que podemos obtener para un nodo no es mejor que lo que ya tengamos por otro lado, entonces será posible eliminar ese nodo.

[9]Notar que un algoritmo voraz decidiría si se usa cada patata de forma independiente.

Estrategia minimax y poda alfa-beta

Más que una técnica general de diseño de algoritmos, la **estrategia minimax** es una estrategia para la resolución de problemas de juegos. Consideramos exclusivamente los juegos de tablero, donde participan dos rivales –que llamamos A y B– de forma alternativa y no interviene el azar. Por ejemplo, dentro de esta categoría podemos encontrar las tres en raya, las damas, el ajedrez, el Nim, etc.

La evolución de todos los posibles estados del juego es representada mediante un árbol. La raíz representa el tablero inicial. Los hijos de la raíz son todos los posibles movimientos de A. Los hijos de estos son los movimientos de B, y así sucesivamente. Las hojas son situaciones donde acaba el juego y gana uno u otro jugador.

La estrategia minimax hace un recorrido del árbol de juego, intentando plasmar una idea bastante razonable: en los movimientos de A intenta ganar él y que pierda B, y en los de B intentará ganar él y que pierda A.

1.3.4. Descripción del pseudocódigo utilizado

Las técnicas de diseño de algoritmos son independientes del lenguaje e incluso del paradigma de programación que se utilice. Un algoritmo diseñado en papel puede ser implementado en Java, Pascal, Basic, C, Smalltalk o cualquier otro lenguaje. Por este motivo, en el estudio teórico de los algoritmos y las estructuras de datos utilizaremos un pseudocódigo independiente del lenguaje de programación[10]. El pseudocódigo usado es bastante intuitivo y su traducción a un lenguaje concreto resulta directa. A continuación vamos a comentar sus principales características.

Para los tipos de datos elementales, o definidos por el usuario, se utiliza el formato presentado en la sección 1.2.3. Las funciones y procedimientos se definen con la palabra **operación**, seguida por el nombre, y la lista de los parámetros formales entre paréntesis. Si un parámetro se puede modificar, se indicará colocando **var** delante de su nombre. Si la operación devuelve un valor, pondremos dos puntos, ':', y a continuación el tipo del valor devuelto.

Por ejemplo, podemos tener las siguientes cabeceras de funciones y procedimientos:

operación SuprimeDos (**var** *C*: Conjunto[T]; *t*, *p*: T)
operación Miembro (*C*: Conjunto[T]; *t*: T): booleano

Normalmente, los nombres de las variables aparecerán en cursiva (*C*, *x*) y las palabras reservadas en negrita (**operación**, **var**).

Incluimos en nuestro pseudolenguaje las siguientes instrucciones:

- **Asignación.** Se expresa mediante ':='. Por ejemplo, *C*:= 4+3. Para la comparación se usa '=' y '≠'.

- **Condicional.** La instrucción tendrá la forma: "**si** Condición **entonces** Acción **finsi**". También puede haber una acción que se ejecute en caso de no cumplirse la condición. La notación sería:

  si Condición **entonces**
  AcciónSiCierto

[10] Aunque, como se podrá ver, la estructura general tiene ciertas similitudes con Pascal.

```
sino
    AcciónSiFalso
finsi
```

- **Valor devuelto.** En las funciones, se devolverá el resultado con "**devolver** Valor".
- **Iteradores.** Utilizaremos los iteradores **para**, **mientras** y **repetir**. El formato será como el siguiente:

```
para v:= Rango hacer        mientras Condición hacer        repetir
    Acción                      Acción                          Acción
finpara                     finmientras                     hasta Condición
```

Dentro del código se podrán utilizar variables locales que no hayan sido definidas previamente. Normalmente estará claro el tipo de esas variables. En caso contrario, se expresará la definición de la variable en una sección **var**, después de la cabecera de la operación y antes del código. También podrán encontrarse operaciones que no estén definidas previamente, pero cuyo significado está bastante claro, como **error**(Texto) para indicar un error o **escribir**(Texto) para mostrar un resultado por pantalla. Para los comentarios dentro del código usaremos: *// Comentario* o *(* Comentario *)*.

1.4. Consejos para una buena programación

Aun con la corta experiencia de programación que se le supone al alumno, seguro que habrá vivido una situación similar a la siguiente. Se empieza a escribir un programa, que previsiblemente no será muy largo, así que se coloca todo dentro del procedimiento principal. Poco a poco, se van añadiendo cosas, nuevos casos, más funcionalidad o un interfaz más completo. Cuando el programa principal es demasiado grande, se pasan algunos trozos de código como procedimientos. Pero no está claro dónde se hace cada cosa, mientras el programa continúa creciendo.

Al final el código se convierte en un *gigante con pies de barro*. Ocupa unas decenas de páginas pero carece de una estructura lógica clara. A posteriori se detectan cosas que se repiten y algunos trozos que deberían modificarse. Pero tratar de hacerlo bien requeriría tantos cambios que se va montando chapuza sobre chapuza. Pero eso no es nada. Cuando vuelves a retomar el programa, varios meses después, resulta imposible saber dónde, por qué y cómo se hacía cada cosa. Aun así, lo intentas modificar. Pero el programa ya no compila; ni volverá a hacerlo nunca más.

Por lo menos, la experiencia habrá servido para conocer qué se debe hacer y qué no, para construir un programa de tamaño mediano-grande. Vamos a ver, a continuación, una serie de consejos prácticos que debería observar todo buen programador.

1.4.1. Importancia del análisis y diseño previos

Usualmente, el gran error de los programadores inexpertos es empezar a teclear código cuando ni si quiera se ha acabado de entender el enunciado de un problema. La

programación, como una disciplina ingenieril –frente a la programación “como arte”[11]– requiere plantearse de forma metódica la construcción de programas. El informático debe hacer un diseño previo de sus programas, de la misma forma que un arquitecto hace los planos antes de construir la casa.

Ya hemos visto, en el punto 1.1, los pasos que deben aplicarse sistemáticamente en la resolución de problemas. El estudio de los métodos, procesos, técnicas y herramientas para el análisis[12] y diseño de programas es lo que constituye el área de investigación de la **ingeniería del software**. Aun sin haber estudiado ingeniería del software, resulta siempre conveniente hacer un minucioso estudio previo del problema y de la solución propuesta, antes de empezar a programar. A nivel de análisis y diseño se trabajará con conceptos abstractos, olvidando los detalles menos relevantes.

Consejo práctico

Para un futuro ingeniero en informática, puede resultar un hábito interesante tomar medidas cuantitativas de su propio proceso de desarrollo de programas. Para ello, se puede utilizar una tabla como la mostrada en la figura 1.6. Dentro de esta tabla, para cada proyecto concreto se va anotando la dedicación temporal –en número de minutos, por ejemplo– que se emplea en cada fase de desarrollo, desde el análisis hasta la validación.

Proyecto: *Algoritmo de búsqueda secuencial con centinela*				**Fecha de inicio:** *3/5/2003*	
Programador: *Ginés García Mateos*				**Fecha de fin:** *4/5/2003*	
Día/Mes	**Análisis**	**Diseño**	**Implement.**	**Validación**	**TOTAL**
3/5	*4*	*6*			*10*
"		*3*	*5*	*6*	*14*
4/5			*2*	*9*	*11*
TOTAL (minutos)	*4*	*9*	*7*	*15*	*35*
MEDIAS (porcentaje)	*11,4%*	*25,7%*	*20%*	*42,9%*	*100%*

Figura 1.6: Tabla para contar la dedicación temporal de una persona a un proyecto software, en las diferentes fases de desarrollo. Las cantidades son minutos, medidos de forma aproximada.

Muchas veces la distinción entre una u otra fase no está muy clara. De modo orientativo, podemos diferenciarlas de la siguiente forma:

[11] “El proceso de escribir programas [...] puede llegar a ser una experiencia estética similar a componer música o escribir poesía”, Donald E. Knuth, *The Art of Computer Programming*, 1997.

[12] No confundir análisis de problemas con análisis de algoritmos. Analizar un problema es estudiar sus necesidades y características. Analizar un algoritmo es medir su consumo de recursos.

- **Análisis.** Cuenta desde que se empieza a leer el enunciado del problema. Incluye la comprensión del problema, búsqueda de información (libros de teoría, apuntes, manuales, código reutilizado, etc.) y modelado conceptual del problema.

- **Diseño.** Dentro de esta fase podemos contar todo el desarrollo "en papel". Incluye: definir los bloques en los que se divide el problema, escribir un esquema de los algoritmos, simular mentalmente el esquema, etc.

- **Implementación.** Abarca desde que comenzamos a teclear en el ordenador hasta que se acaba el programa y se compila por primera vez, produzca fallos o no.

- **Validación.** Empieza cuando se hace la primera compilación. Incluye la corrección de errores de compilación, la realización de pruebas, detección de fallos en los resultados y su corrección. Acaba cuando se finaliza el proyecto.

Seguir un control detallado de este tipo puede tener varios objetivos. En primer lugar, sirve para ser conscientes del propio proceso de desarrollo de software, analizando a qué cosas se dedica proporcionalmente más y menos tiempo. En segundo lugar, puede ser útil para estudiar comparativamente la evolución del proceso a lo largo del tiempo. Esto podría utilizarse para detectar deficiencias o puntos débiles. Finalmente, es interesante para extraer conclusiones fundadas: ¿hasta qué punto ayuda dedicar más tiempo a análisis y diseño a reducir el tiempo de implementación y validación? ¿Es fácil distinguir entre las distintas fases? ¿Qué fase requiere más tiempo y por qué? ¿Cómo reducirlo?

1.4.2. Modularidad: encapsulación y ocultamiento

El resultado de un buen diseño de software debe ser la definición de una estructura general de la aplicación. Esta estructura consta de una serie de partes separadas, que se relacionan entre sí a través de los interfaces definidos. Por lo tanto, resulta conveniente utilizar los mecanismos que ofrezca el entorno de programación para agrupar funcionalidades: **módulos**, **paquetes**, **clases**, **unidades**.

Un módulo, clase o paquete, agrupa un conjunto de funcionalidades relacionadas. Esto es lo que llamamos **encapsulación**: el módulo tiene un nombre descriptivo, bajo el cual se agrupan todas las utilidades relacionadas con el mismo. Por ejemplo, una clase Pila contiene tanto las estructuras de datos como las operaciones necesarias para manejar las pilas. La ventaja de encapsular es que sabemos dónde se encuentra cada cosa. En caso de haber fallos podemos encontrar quién es el responsable, y corregirlo. Si la funcionalidad se utiliza en otras aplicaciones, sólo tendremos que usar el mismo módulo.

Los paquetes son, normalmente, un nivel de encapsulación superior a los otros. Un paquete puede contener varias clases o módulos relacionados. Por ejemplo, un paquete Colecciones podría contener las clases Pila, Lista, Cola, Conjunto, etc.

Otro principio fundamental asociado a la modularidad es la **ocultación de la implementación**. Cuando se crea un módulo, una clase o un paquete, se definen una serie de operaciones para manejar la funcionalidad creada. Esto es lo que se llama la **interface** del módulo, clase o paquete. Los usuarios del módulo deben utilizar las operaciones del interface y sólo esas operaciones. Todo lo demás es inaccesible, está oculto. Es más, el

usuario sólo conoce cuál es el resultado de las operaciones, pero no cómo lo hacen. Por ejemplo, para el usuario de la clase Pila, el interface contiene las operaciones push, pop, etc., de las cuales conoce su significado, pero no cómo se han implementado.

Consejo práctico

El concepto de **software** se entiende normalmente como algo mucho más amplio que simplemente un programa; incluye tanto el programa como el documento de requisitos del problema, el diseño realizado y las especificaciones de cada parte. Sería adecuado que todos estos documentos se fueran completando en las mismas fases en las que deben hacerse, y no después de acabar la implementación del programa.

No estudiaremos aquí la forma de crear documentos de requisitos y diseños de software, que caen dentro de las asignaturas de ingeniería del software. El alumno puede utilizar la notación que le parezca más adecuada. En cuanto a la especificación, en el siguiente capítulo haremos un repaso de las especificaciones informales e introduciremos dos métodos de especificación formal. Es conveniente documentar siempre, con alguno de estos métodos, todas las abstracciones (tipos y operaciones) que se desarrollen.

1.4.3. Otros consejos

Las recomendaciones anteriores tienen más sentido cuando se trata de desarrollar aplicaciones de tamaño medio o grande; aunque siempre deberían aplicarse en mayor o menor medida. En este apartado vamos a ver algunos otros consejos más generales, aplicables tanto a problemas pequeños como de mayor tamaño.

- **Reutilizar programas existentes.** Un nuevo programa no debe partir desde cero, como si no existiera nada fuera del propio desarrollo. La reutilización se puede dar a distintos niveles. Se pueden reutilizar librerías, implementaciones de TAD, esquemas de programas, etc.
  Reutilizar implica no sólo pensar qué cosas tenemos hechas que podamos volver a usar, sino también qué cosas que estamos haciendo podremos usar en el futuro. Una buena reutilización no se debe basar en "copiar y pegar" código, sino en compartir módulos, clases o paquetes, en distintos proyectos.

- **Resolver casos generales.** Si no supone un esfuerzo adicional, se deberá intentar resolver los problemas más generales en vez de casos particulares. Por ejemplo, en lugar de implementar un tipo pilas de enteros, sería más útil definir pilas que puedan almacenar cualquier cosa. De esta manera, se favorece la posibilidad de que las funcionalidades desarrolladas se puedan reutilizar en el futuro.

- **Repartir bien la funcionalidad.** A todos los niveles (a nivel de paquetes, de módulos y de procedimientos) se debe intentar repartir la complejidad del problema de manera uniforme. De esta forma, hay que evitar crear procedimientos muy largos y complejos. En su lugar, sería mejor detectar los componentes que lo forman y usar subrutinas. Un procedimiento corto, bien estructurado y tabulado, usando funciones y variables con nombres descriptivos adecuados, resulta más legible y, por lo tanto,

más fácil de comprender y mantener. De forma similar, si un módulo es muy grande, puede que estemos mezclando funcionalidades separadas y sea mejor crear dos o más módulos.

- **Simplificar.** No es mejor un programa cuanto más largo y complejo, sino más bien todo lo contrario. Una solución corta y directa es, a menudo, la solución más elegante y adecuada. Una solución enrevesada suele incluir código repetido, casos innecesarios, desperdicio de memoria y falla con más facilidad. Para simplificar se debe aplicar la lógica y el sentido común. Por ejemplo, suponiendo una función que devuelve un booleano, es muy típico encontrar cosas del siguiente estilo o similares:

  ```
  si Condición = verdadero entonces
      devolver verdadero
  sino
      devolver falso
  finsi
  ```

  Toda la construcción anterior no es más que un largo rodeo para decir simplemente:
  devolver Condición
  O, por ejemplo, cuando tenemos una variable i que debe ir oscilando entre dos valores, pongamos por caso los valores 1 y 2, también se suelen hacer análisis de casos *if-else*. El problema se soluciona con una simple asignación del tipo: i:= 3-i.

Consejo práctico

No perdamos de vista que el interés principal es que los programas desarrollados sean correctos y eficientes. Dentro de eso, resulta muy conveniente que se cuiden los otros criterios de una buena programación: legibilidad, simplicidad, reparto de la funcionalidad, reutilización. La mejor forma de evaluarse uno mismo en estos parámetros es tomar medidas cuantitativas y que sean significativas, al estilo de lo que se propone en el apartado 1.4.1.

En la figura 1.7 se propone un formato para el tipo de información que sería interesante recoger en cada proyecto realizado. El número de operaciones de la *interface* se refiere a las operaciones que son accesibles fuera del módulo o clase. Las *privadas* son las que no son accesibles desde fuera. Por simplicidad, la longitud del código se debe expresar en una unidad que sea fácil de medir, como por ejemplo líneas de código. Por otro lado, dentro del campo *reutilizado* se indica si el módulo ha sido reutilizado de un proyecto anterior, o si se han reutilizado partes con modificaciones.

Está claro que una tabla como la de la figura 1.7 sólo tendrá sentido cuando el programa esté completamente acabado y además sea correcto. Una vez con la tabla, cabría plantearse diferentes cuestiones. ¿Es uniforme el reparto de operaciones entre los módulos? ¿Hay módulos que destaquen por su tamaño grande o pequeño? ¿Está justificado? ¿Es razonable la longitud media de las operaciones? ¿Hay alguna operación demasiado larga? ¿Se ha reutilizado mucho o poco? E, intentado ir un poco más allá, ¿existe algún otro parámetro de calidad que hayamos encontrado, pero que no está recogido en la tabla? En ese caso, ¿cómo se podría cuantificar?

Proyecto: *Algoritmo ficticio*					**Fecha de inicio:** *5/5/2003*			
Programador: *Ginés García Mateos*					**Fecha de fin:** *7/5/2003*			
Nombre del Módulo/Clase	**Número de operaciones**			**Líneas de código**	**Reutili -zada**	**Líneas por operación**		
	Interface	**Privadas**	**Total**			**Min**	**Max**	**Media**
Pilas	*4*	*2*	*6*	*53*	*no*	*5*	*13*	*8,8*
Principal	*1*	*3*	*4*	*39*	*no*	*7*	*24*	*9,8*
TOTAL	*5*	*5*	*10*	*92*	*0*	*5*	*24*	*9,2*
MEDIAS	*2,5*	*2,5*	*5*	*46*	*0*	*6*	*18,5*	*9,2*

Figura 1.7: Tabla para tomar diferentes medidas relacionadas con la calidad del software.

Ejercicios propuestos

Ejercicio 1.1 Busca uno o varios programas –preferiblemente de tamaño más o menos grande– que hayas escrito con anterioridad. Rellena una tabla como la de la figura 1.7. Si el programa no utiliza módulos, unidades o clases, intenta agrupar las operaciones por funcionalidad. Extrae conclusiones de los resultados obtenidos, respondiendo las cuestiones que se plantean al final del apartado 1.4.3. Haz una valoración crítica y comparativa (respecto a algún compañero o respecto a otros programas).
Consejo: Para facilitar el rellenado de la tabla, es adecuado utilizar alguna herramienta como una hoja de cálculo. Normalmente, en este tipo de tablas las celdas claras las introduce el usuario y las oscuras se calculan automáticamente a partir de otras (aunque en el ejemplo no siempre es así).

Ejercicio 1.2 A partir de ahora, para los siguientes problemas de tamaño mediano que resuelvas, intenta hacer un seguimiento del tiempo que dedicas, utilizando una tabla como la de la figura 1.6. Igual que antes, se aconseja automatizarlo usando una hoja de cálculo. Estos documentos, junto con las especificaciones y las medidas de calidad de la tabla 1.7, deberían acompañar la documentación del programa final.

Ejercicio 1.3 Anota en una hoja todos los tipos de datos, estructuras y técnicas de diseño que conozcas. Para cada tipo indica las variantes que puede tener y las formas de implementarlos. Por ejemplo, se supone que debes conocer el tipo lista, que pueden ser circulares, enlazadas simple o doblemente, se pueden implementar con punteros o con arrays, etc.

Cuestiones de autoevaluación

Ejercicio 1.4 A lo largo de todo el capítulo, se hace énfasis en lo importante que resulta no empezar el desarrollo de programas directamente tecleando código. ¿Estás de acuerdo con esta afirmación? ¿Qué otras cosas deberían hacerse antes? ¿En qué fases?

Ejercicio 1.5 ¿Qué relación existe entre los algoritmos y las estructuras de datos? ¿Cuál va primero? ¿Cuál es más importante en la resolución de problemas? ¿Qué papel juega cada uno?

Ejercicio 1.6 ¿Cuáles son las diferencias entre tipo abstracto de datos, tipo de datos y estructura de datos? ¿Son conceptos redundantes? Por ejemplo, una lista ¿qué es?

Ejercicio 1.7 Enumera las propiedades fundamentales de un algoritmo. Dado un trozo de código, ¿se pueden demostrar formalmente las propiedades? ¿Y si es pseudocódigo? Un programa de edición de texto, al estilo *Word*, ¿se puede considerar un algoritmo? ¿En qué se diferencian un algoritmo, un programa, un problema y una aplicación?

Ejercicio 1.8 En el análisis de algoritmos, identificamos distintos factores que influyen en el consumo de recursos. Algunos resultan de interés y otros no. Distínguelos y justifican por qué se consideran o no interesantes.

Ejercicio 1.9 Todas las notaciones asintóticas (excepto la o-pequeña) son independientes de constantes multiplicativas. Comprueba que los factores externos estudiados en el consumo de recursos de un algoritmo solo influyen en términos multiplicativos.

Ejercicio 1.10 ¿Para qué sirve un esquema algorítmico? ¿En qué se diferencia de un algoritmo? ¿Cumple las propiedades de definibilidad y finitud? ¿Por qué no? Enumera los distintos esquemas algorítmicos que conozcas.

Referencias bibliográficas

Antes de entrar de lleno en el contenido del libro, sería adecuado repasar los conceptos fundamentales que el alumno debe conocer de programación de primer curso. Para ello la mejor referencia es, sin duda, la que se usara entonces. Algunos libros de algoritmos y estructuras de datos incluyen capítulos de repaso de los tipos fundamentales –listas, pilas, colas y árboles– como los capítulos 2 y 3 de [Aho88], y los capítulos 3 y 4 de [Weis95], y el libro [Wirth80]; además de las definiciones relacionadas con estructuras y algoritmos, en los capítulos iniciales.

El proceso de resolución de problemas, planteado en el primer punto, es una versión resumida y simplificada del ciclo clásico de desarrollo de software. Pocos libros de la bibliografía inciden en este tema, que está más relacionado con el ámbito de la ingeniería del software. Una breve introducción a estos temas, así como a otros conceptos tratados en este libro, se pueden consultar en el capítulo 1 de [Aho88].

Algunos de los consejos de programación de la sección 1.4, han sido extraídos del apartado 1.6 de [Aho88]. Las tablas de las figuras 1.6 y 1.7 para medir el proceso personal de desarrollo de software, son una adaptación del proceso propuesto en [Humphrey01]. Son una simplificación, ya que este libro está más orientado hacia las cuestiones de ingeniería del software.

Además de la bibliografía en papel, que se detallará en cada uno de los capítulos sucesivos, Internet resulta una valiosa fuente de información adicional. Por ejemplo, un

completo compendio de todos los temas relacionados con los algoritmos y las estructuras de datos es el "Dictionary of Algorithms and Data Structures":

`http://www.nist.gov/dads/`

Además de la documentación escrita, se pueden encontrar proyectos dirigidos a crear animaciones interactivas de algoritmos y estructuras de datos. Un portal interesante con numerosos enlaces es:

`http://www.cs.hope.edu/~alganim/ccaa/`

Capítulo 2

Abstracciones y especificaciones

Una abstracción es una simplificación de un objeto o fenómeno del mundo real. Las abstracciones son básicas en la construcción de aplicaciones de cierto tamaño, donde es necesario distinguir entre distintos niveles de abstracción, desde la visión global y genérica del sistema hasta, posiblemente, el nivel de código máquina. Una especificación, en este contexto, es una descripción del significado o comportamiento de la abstracción. Las especificaciones nos permiten comprender el efecto de un procedimiento o el funcionamiento de un tipo abstracto de datos, sin necesidad de conocer los detalles de implementación. En este capítulo se tratan los mecanismos de abstracción soportados por los lenguajes de programación, y cómo estas abstracciones son descritas mediante especificaciones más o menos rigurosas.

Objetivos del capítulo:

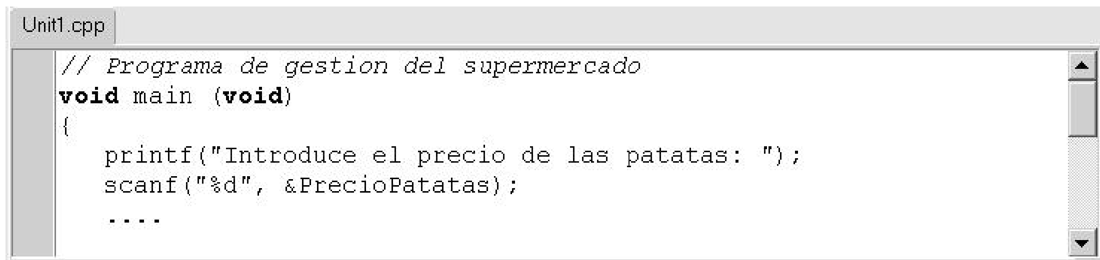
- Reconocer la importancia de la abstracción y conocer los tipos de abstracciones que aparecen en programación: funcional, de datos y de iteradores.
- Conocer los distintos mecanismos de los lenguajes de programación que dan soporte al uso de abstracciones, desde los módulos hasta las clases y objetos.
- Concienciarse de la utilidad de desarrollar especificaciones completas y precisas, entendiendo la especificación como un punto de acuerdo entre el usuario y el implementador de una abstracción.
- Estudiar una notación informal para la especificación de abstracciones.
- Comprender el método de especificación formal algebraico o axiomático (basado en una definición mediante axiomas) y el método constructivo u operacional (basado en el uso de precondiciones y postcondiciones).
- Aplicar las notaciones analizadas en la especificación de una amplia variedad de tipos abstractos de datos.

Contenido del capítulo:

2.1. Las abstracciones en programación . . . 37
2.1.1. Diseño mediante abstracciones . . . 38
2.1.2. Mecanismos de abstracción: especificación y parametrización . . . 39
2.1.3. Tipos de abstracciones: funcional, de datos e iteradores . . . 40
2.1.4. Mecanismos de abstracción en los lenguajes de programación . . . 42
2.2. Especificaciones informales . . . 47
2.2.1. Especificación informal de abstracciones funcionales . . . 47
2.2.2. Especificación informal de abstracciones de datos . . . 48
2.2.3. Especificación informal de abstracciones de iteradores . . . 51
2.3. Especificaciones formales algebraicas . . . 52
2.3.1. Propiedades, notación y ventajas de las especificaciones formales . . . 53
2.3.2. Especificaciones algebraicas o axiomáticas . . . 54
2.3.3. Taxonomía de las operaciones de un TAD . . . 56
2.3.4. Completitud y corrección de la especificación . . . 57
2.3.5. Reducción de expresiones algebraicas . . . 59
2.4. Especificaciones formales constructivas . . . 61
2.4.1. Precondiciones y postcondiciones . . . 61
2.4.2. Especificación como contrato de una operación . . . 62
2.4.3. Necesidad de un modelo subyacente . . . 64
2.4.4. Ejecución de especificaciones constructivas . . . 67
Ejercicios resueltos . . . 68
Ejercicios propuestos . . . 75
Cuestiones de autoevaluación . . . 77
Referencias bibliográficas . . . 78

2.1. Las abstracciones en programación

Supongamos que recibimos el encargo de crear una aplicación para la gestión de una gran cadena de supermercados, que incluye las compras a los proveedores, el control de las cajas, la gestión de empleados, el inventario del almacén, la contabilidad de la empresa, las campañas publicitarias, la comunicación entre los distintos centros, etc. Los documentos de la aplicación ocupan unos 2.000 folios –aunque no están muy claros– y hay en torno a quince personas implicadas en el proyecto. Ni cortos ni perezosos, empezamos a escribir el programa:



¡Imposible seguir por ese camino! ¡Debe de haber una manera más razonable de abordar la construcción de la aplicación! ¿Cuál es el problema? Uno no puede tener todo en la cabeza al mismo tiempo: los 2.000 folios de requisitos, los doce departamentos de la empresa –cada uno con sus necesidades e intereses–, la estructura final de la aplicación, los tipos de datos necesarios, el flujo de información, etc. Debe haber un método o un proceso, que nos permita ir poco a poco, desde el punto de vista más amplio y genérico del sistema hasta llegar a los pequeños subproblemas.

El estudio de estas técnicas para el desarrollo de grandes aplicaciones es parte del área conocida como *ingeniería del software*, como ya comentamos en el capítulo 1. Prácticamente todas ellas se basan en la aplicación del **concepto de abstracción**: dejar a un lado lo irrelevante y centrarse en lo importante, en la esencia de las cosas.

Pero, ¿qué es lo importante? Obviamente, en cada momento será una cosa distinta. En cierto estado inicial de desarrollo lo que importa será la estructura general de la aplicación. En otra fase, será la implementación de una operación concreta y, en una posible fase de optimización de código, lo importante puede ser el código máquina del programa. Así, tenemos distintos **niveles de abstracción**: según el nivel de detalle, decimos que estamos en un nivel de abstracción mayor o menor.

La abstracción no sólo aparece en el desarrollo de programas, sino que es usada en todos los órdenes de la vida, cuando intentamos abordar un problema complejo. Ser capaz de distinguir los aspectos superfluos de los que forman la esencia de un problema es una de las claves de la inteligencia humana. En programación, las abstracciones son aplicadas de forma continua, dando lugar a conceptos básicos como las rutinas, funciones, procedimientos y los tipos abstractos de datos. Pero la cosa no acaba ahí, y la aplicación de mecanismos de abstracción en los lenguajes de programación da lugar a conceptos y principios fundamentales, como los módulos, las clases, la genericidad, la ocultación de información, la encapsulación. Gracias a ellos, es posible construir aplicaciones cada vez más complejas y fiables, con un menor coste. El propio proceso de diseño de programas está basado en el uso de abstracciones.

2.1.1. Diseño mediante abstracciones

Consideremos la aplicación de gestión de la gran cadena de supermercados, planteada en la introducción. Vamos a ver cómo usando abstracciones se puede abordar su resolución de forma ordenada y sistemática.

Un ejemplo de diseño usando abstracciones

Dejando a un lado todos los detalles irrelevantes, podemos distinguir los grandes bloques en los que se divide el problema. Por un lado tenemos una base de datos, en la que estará almacenada toda la información que maneje la empresa. Por otro lado, tendremos una aplicación de administración y mantenimiento del sistema. Y finalmente tendremos una o varias aplicaciones de usuarios, que acceden a la parte que les corresponde de la base de datos.

De esta forma, se puede establecer una descomposición inicial del problema en tres grandes subproblemas: *Base de Datos*, *Administración* y *Usuarios*. Cada uno de los subproblemas es una abstracción de una parte del sistema. Existe una relación entre ellos, pero básicamente se pueden abordar de forma independiente. La estructura global del sistema se muestra en la figura 2.1.

Figura 2.1: Una visión muy abstracta de un sistema informático genérico.

¿Cómo resolver ahora los subproblemas? Pues volviendo a aplicar la abstracción, evidentemente. Para ello será necesario especificar, de forma abstracta, la funcionalidad que es responsabilidad de cada parte. Por ejemplo, si nos centramos en la Base de Datos, su misión es almacenar toda la información manejada en la empresa, permitiendo su consulta y modificación. Podemos distinguir, a su vez, cuatro grandes partes: *productos*, *proveedores*, *clientes* y *empleados*. De esta manera podemos seguir descomponiendo en partes cada vez más pequeñas, hasta llegar al nivel de programación. Luego se van combinando las soluciones de cada parte y se compone la solución para el problema original.

El proceso de diseño mediante abstracciones

Analizando el ejemplo anterior, se observa que existe un modo ordenado y sistemático de abordar el problema, que se va repitiendo en los distintos niveles de abstracción. Este proceso aparece siempre en la resolución de problemas complejos, y es lo

que se denomina **diseño mediante abstracciones**. El proceso está compuesto básicamente por los siguientes cuatro pasos.

Pasos en el proceso de diseño mediante abstracciones

1. Identificar los subproblemas en los que se divide el problema original.
2. Especificar cada subproblema de forma abstracta, usando alguna notación.
3. Resolver cada subproblema de forma separada.
4. Unir las soluciones y verificar la solución para el problema original.

En conclusión, se puede apreciar que se trata de un proceso cíclico de especificar / implementar / verificar. Tras el paso de verificación puede ser necesario volver al paso 3 (corregir la implementación), al paso 2 (cambiar la especificación) o incluso al 1 (hacer otra descomposición más adecuada de los subproblemas). Este ciclo constituye la idea básica del proceso clásico de desarrollo de programas, como vimos en el capítulo 1.

Por otro lado, existe una clara analogía con el **método científico** –aplicado en las ciencias experimentales– compuesto por los pasos de: observación, proposición de hipótesis, experimentación y verificación de la hipótesis. El proceso se puede ver, por lo tanto, como una aplicación del método científico a la disciplina de la programación. En nuestro caso, las abstracciones de objetos del mundo real juegan el papel de hipótesis que deben ser especificadas, implementadas y verificadas.

2.1.2. Mecanismos de abstracción: especificación y parametrización

Las abstracciones utilizadas en programación (procedimientos, funciones y tipos de datos) son descritas mediante especificaciones, que determinan su comportamiento esperado. Además, esas abstracciones pueden estar parametrizadas, de forma que su significado está dado en función de ciertos parámetros. La especificación y la parametrización son mecanismos de abstracción independientes, pueden aparecer juntos o no.

Abstracción por especificación

Como vimos en la introducción, la abstracción consiste en distinguir lo que es relevante de lo que no lo es. En la abstracción por especificación lo relevante es la descripción del efecto o resultado, y lo irrelevante es la forma de calcularlo.

La mayor parte de las abstracciones usadas en el desarrollo de programas son abstracciones por especificación. La especificación de un procedimiento, o de un tipo de datos, describe su comportamiento, pero no dice nada acerca de cómo funciona internamente, de cómo está implementado.

Consideremos, por ejemplo, las librerías del sistema operativo para crear procesos, abrir ventanas, pedir memoria, etc. La especificación describe el significado, el modo de uso y el efecto de las funciones existentes, pero las cuestiones de implementación quedan

completamente ocultas. Un programador que use esas librerías no sólo no necesita conocer la implementación, sino que conocerla supondría una complejidad excesiva e innecesaria.

Esto es lo que se conoce como el **principio de ocultación de la implementación**: el usuario de una abstracción no puede ni debe conocer la implementación de la misma. Este principio implica que el lenguaje tenga mecanismos que soporten la ocultación, y que el programador de la abstracción los use de forma adecuada para garantizar la ocultación.

La abstracción por especificación también da lugar al concepto de **encapsulación**: un conjunto de operaciones, o tipos, relacionados son agrupados en un mismo *módulo* o *paquete*. En consecuencia, el módulo se puede considerar como una abstracción de nivel superior, compuesta por varias abstracciones que proporcionan un conjunto de funcionalidades relacionadas. Por ejemplo, las librerías del sistema operativo podrían ser separadas en distintos módulos: uno para la parte gráfica, otro para el manejo de ficheros, otro para los procesos, etc.

Abstracción por parametrización

Las abstracciones pueden estar parametrizadas, de manera que el significado de la abstracción no es fijo, sino que depende del valor de uno o más parámetros. Una clase muy importante es la *parametrización de tipo*, en la cual el parámetro es un tipo de datos.

Supongamos, por ejemplo, que queremos implementar un conjunto de funciones para ordenar los elementos de un array, sean del tipo que sean. Deberíamos definir una función por cada posible tipo de entrada:

operación OrdenaEntero (A : **array** [min..max] **de** entero)
operación OrdenaReal (A : **array** [min..max] **de** real)
operación OrdenaCaracter (A : **array** [min..max] **de** caracter)
....

Resultaría mucho más interesante crear una abstracción más genérica, que se pueda aplicar sobre cualquier tipo de datos del array. Para ello, *factorizamos* todas las operaciones anteriores creando una nueva operación, donde el tipo de datos de los elementos del array es un parámetro más de la función:

operación Ordena[T: tipo_comparable](A : **array** [min..max] **de** T)

La operación Ordena[T] es una abstracción de nivel superior, que permite ordenar un array de cosas de cualquier tipo. Se dice que Ordena[T] es una abstracción genérica o parametrizada.

La abstracción por parametrización se aplica también sobre abstracciones de datos, permitiendo definir tipos de datos genéricos. Por ejemplo, un tipo Conjunto[T: tipo] podría almacenar un conjunto de elementos de tipo T, mientras que Pareja[K: tipo] almacenaría un par de valores de cualquier tipo K.

2.1.3. Tipos de abstracciones: funcional, de datos e iteradores

De forma consciente o inconsciente, los mecanismos de abstracción son usados constantemente en programación. Podemos distinguir entre distintos tipos o categorías de abstracciones, en función de qué es lo que se abstrae. Los dos tipos fundamentales son la *abstracción de datos* y la *funcional*. Adicionalmente, encontramos las *abstracciones de*

iteradores, que permiten definir un recorrido abstracto sobre los elementos de una colección.

Abstracciones funcionales

En la abstracción funcional abstraemos un trozo de código, más o menos largo, al cual le asignamos un nombre. El concepto resultante es la idea de **rutina**, **procedimiento** o **función**. La especificación de un procedimiento determina cuáles son los parámetros de entrada, los de salida y el efecto que tendrá su ejecución, pero no se indica nada sobre la implementación. Para el usuario del procedimiento es indiferente cómo se haya implementado, siempre que se garantice el efecto descrito en la especificación.

Por ejemplo, el procedimiento de ordenación OrdenaEntero(*A*: **array** [min..max] **de** entero) es una abstracción funcional, de la cual sabemos que su resultado es que los elementos del array de enteros *A* son ordenados en la salida de menor a mayor. Pero, respetando el *principio de ocultación de la implementación*, no sabemos nada acerca de cómo lo hace. Los algoritmos de ordenación por selección, inserción, burbuja, mezcla, etc., son posibles implementaciones de la abstracción OrdenaEntero.

Los procedimientos y funciones son *generalizaciones* del concepto de operador de un lenguaje de programación. En lugar de usar los operadores existentes en el lenguaje, un programador puede definir su propio conjunto de operadores, con comportamientos más complejos y adaptados a sus necesidades.

Abstracciones de datos

En la abstracción de datos se abstrae un dominio de valores, junto con un conjunto de operaciones sobre ese dominio, que poseen un significado particular. Esta abstracción da lugar al concepto de **tipo abstracto de datos** (**TAD**). La especificación del TAD determina el comportamiento de las operaciones definidas sobre el tipo, pero no dice nada sobre cómo son almacenados los valores del tipo o cómo están implementadas las operaciones.

Por ejemplo, los números naturales forman un TAD, al que le asignamos el nombre Natural. Cada natural puede tomar valores dentro de un dominio $\{0, 1, 2, 3, ...\}$ y puede ser manipulado a través de un conjunto de operaciones definido (suma, multiplicación, resta, etc.) con un significado conocido. En este caso el principio de ocultación de la implementación es aplicado tanto sobre las operaciones como sobre la estructura que se usa para almacenar los datos del dominio.

Igual que con los procedimientos, los TAD se pueden considerar como generalizaciones del concepto de tipo de datos primitivo. En lugar de usar los tipos primitivos definidos en un lenguaje (enteros, reales, caracteres, etc.) el usuario se puede definir sus propios tipos, combinando los tipos primitivos. O puede usar los TAD implementados por otro programador y que sean de utilidad para su problema.

Tipos contenedores y parametrizados

Una clase importante de TAD son los tipos **contenedores** o **colecciones**. Un tipo contenedor es un TAD que está compuesto por un número no fijo de valores de otro tipo.

Por ejemplo, un array, una cola, una lista o un árbol son tipos contenedores. Las implementaciones de listas mediante cursores, punteros o tablas, son ejemplos de *estructuras de datos* usadas para almacenar los valores del tipo. Se puede decir que la estructura de datos es la *concretización* del tipo *abstracto*.

Para conseguir una definición lo más general posible, se debe permitir que el tipo de valores almacenados en el contenedor sea un TAD cualquiera, es decir, tener listas, arrays o colas de cualquier cosa. En consecuencia, los TAD colecciones deben estar *parametrizados*, incluyendo un parámetro que indica el tipo de objetos almacenados. Por ejemplo, el TAD parametrizado Lista [T: tipo] define las listas de valores de tipo T. Dos posibles instanciaciones del tipo serían: Lista [Natural], o Lista [Lista [Natural]].

Abstracciones de iteradores

Un tipo interesante de abstracción –aunque menos frecuente– es la abstracción de iteradores. Un **iterador** permite realizar un recorrido sobre los elementos de una colección de forma abstracta. En un iterador lo relevante es que se recorren todos los elementos de una colección, independientemente de cuál sea la forma de almacenar los valores del tipo.

Los iteradores son una generalización de los iteradores elementales de los lenguajes de programación. Por ejemplo, un iterador **para**, o **for**, tiene la forma:

para *i*:= 1, ..., *n* **hacer** Acción sobre *i*

Esta instrucción permite recorrer todos los enteros entre 1 y *n*, aplicando determinada operación sobre cada valor. Si en lugar de enteros tenemos listas, pilas, árboles o cualquier otra colección, sería muy útil disponer de una operación similar a la anterior, que nos permita recorrer todos los elementos de forma abstracta. La operación podría tener la forma:

para cada elemento *i* de la lista *L* **hacer** Acción sobre *i*
para cada elemento *i* del árbol *A* **hacer** Acción sobre *i*

Los dos ejemplos anteriores son abstracciones de iteradores: permiten procesar todos los elementos de una colección, independientemente de los detalles de representación del tipo de datos. Otros ejemplos de iteradores pueden incluir una condición sobre los elementos a iterar, devolver otra colección con los elementos procesados o fijar un orden concreto de recorrido. El formato podría ser del siguiente tipo:

para cada elemento *i* de la colección *C* **que cumpla** *P(i)* **hacer** Acción sobre *i*
res:= **Seleccionar** los elementos *i* de la colección *C* **que cumplan** *P(i)*
Eliminar los elementos *i* de la colección *C* **que cumplan** *P(i)*
para cada elemento *i* del árbol *A* **en pre-orden hacer** Acción sobre *i*

2.1.4. Mecanismos de abstracción en los lenguajes de programación

La evolución de los lenguajes de programación se puede considerar, en gran medida, como un intento por incluir y dar soporte a mecanismos de abstracción cada vez más avanzados. En particular, la forma de proveer un mecanismo de definición y uso de Tipos Abstractos de Datos ha sido una de las principales motivaciones para la aparición de nuevos conceptos. La evolución va casi siempre en la dirección de ofrecer características

más próximas al concepto teórico de TAD: posibilidad de usar un TAD como un tipo cualquiera, principio de ocultación de la implementación, encapsulación y modularidad, genericidad, reutilización de TAD, posibilidad de usar iteradores.

Pero, ¿por qué es tan importante el concepto de Tipo Abstracto de Datos? Un lenguaje que no permite usar TAD –o un entorno de programación para el que no disponemos de TAD– es como una casa sin muebles y sin decoración. Es posible vivir en ella y realizar tareas rutinarias, aunque tendríamos que hacerlo de manera muy rudimentaria. Por ejemplo, deberíamos dormir en el suelo o beber agua chupando del grifo. Los TAD son como los muebles de la casa, no son estrictamente necesarios para vivir pero facilitan mucho las cosas. Los muebles, que hacen el papel de tipos abstractos:

- Aportan una funcionalidad extra fundamental, un valor añadido a la casa, permitiendo realizar las tareas comunes de manera más sencilla y cómoda.

- Pueden cambiarse por otros, moverse de sitio, arreglarlos en caso de rotura, etc., lo cual no es posible hacerlo –o resulta mucho más costoso– con la casa en sí.

- Los muebles y la decoración están adaptados a las necesidades, gustos y caprichos de cada uno, mientras que la casa es la misma para todo el bloque de apartamentos.

- Hay una separación clara entre la persona que fabrica el mueble y la que lo utiliza. Para usar un mueble no es necesario conocer nada de carpintería.

En definitiva, los TAD son uno de los conceptos más importantes en programación. Su uso permite construir aplicaciones cada vez más complejas, simplificando la utilización de funcionalidades adicionales. Los TAD pueden ser, por ejemplo, desde los enteros y los booleanos, hasta las ventanas de Windows, las conexiones a Internet o el propio sistema operativo.

Lenguajes de programación primitivos

Los lenguajes de programación primitivos (como Fortran o BASIC) ofrecían un conjunto predefinido de tipos elementales: entero, real, carácter, cadena, array, etc. Pero no existía la posibilidad de definir tipos nuevos. Si, por ejemplo, un usuario necesitaba usar una pila, debía definir y manejar las variables adecuadas de formal "manual". Por ejemplo, el programa en notación Pascal sería algo como lo siguiente.

```
var
  PilaUno, PilaDos: array [1..100] of integer;
  TopePilaUno, TopePilaDos: integer;

begin
  PilaUno[TopePilaUno]:= 52;
  Write(PilaDos[TopePilaDos]);
  TopePilaUno:= TopePilaUno + 1;
...
```

Utilizar las pilas de esta manera resulta difícil y engorroso, y el código generado no se puede reutilizar en otras aplicaciones. Además, el programador debería tener en cuenta que `PilaUno` se corresponde con `TopePilaUno` y `PilaDos` con `TopePilaDos`. En estas circunstancias, es muy fácil equivocarse o acceder a posiciones no válidas del array.

Tipos definidos por el usuario

Los *tipos definidos por el usuario* son un primer paso hacia los TAD. Aparecen en lenguajes como C y Pascal, y permiten agrupar un conjunto de variables bajo un nombre del tipo. Estos tipos se pueden usar en el mismo contexto que un tipo elemental, es decir, para definir variables, parámetros o expresiones de ese tipo nuevo. En el ejemplo anterior, podríamos definir las pilas de enteros como un nuevo tipo definido por el usuario.

```
type
  Pila= record
    Tope: integer;
    Datos: array [1..100] of integer;
  end;

var
  PilaUno, PilaDos: Pila;

procedure Push (valor: integer; var pila: Pila);
procedure Pop (var pila: Pila);
 ...
```

El gran inconveniente de este mecanismo es que no permite garantizar el ocultamiento de la implementación. En teoría, las pilas sólo deberían ser usadas con las operaciones `Push`, `Pop`, etc., definidas sobre ellas. Pero, en su lugar, un programador podría hacer sin problemas: `PilaUno.Datos[72]:=36;` o `PilaDos.Tope:=-12;`. El lenguaje debería impedir las anteriores instrucciones, que acceden directamente a la representación del tipo.

Módulos y encapsulación

Los módulos o paquetes son un mecanismo de encapsulación; permiten tener agrupados bajo un mismo nombre una serie de tipos y operaciones relacionados. Y lo que es más interesante, en algunos casos, como en el lenguaje Modula-2, los módulos ofrecen ocultación de la implementación: los tipos definidos dentro de un módulo sólo se pueden usar a través de las operaciones de ese mismo módulo. De esta forma se evitarían los problemas vistos en el apartado anterior. El programa sería algo del siguiente tipo.

```
MODULE ModuloPilasEnt;
EXPORT PilaEntero, Push, Pop, Top, Nueva;

TYPE
  PilaEntero= RECORD
```

```
    Tope: INTEGER;
    Datos: ARRAY [1..100] OF INTEGER;
  END;
PROCEDURE Push (valor: integer; VAR pila: PilaEntero);
PROCEDURE Pop (var pila: PilaEntero);
...

USE ModuloPilasEnt;
VAR PilaUno, PilaDos: PilaEntero;
  ...
  Push(23, PilaUno);
  Pop(PilaDos);
  ...
  PilaUno.Datos[72]:= 36;  ----> ERROR EN TIEMPO DE COMPILACIÓN
  PilaDos.Tope:= -12;      ----> ERROR EN TIEMPO DE COMPILACIÓN
  ...
```

El compilador impide las dos últimas instrucciones, indicando que los atributos `Datos` y `Tope` no son accesibles.

Clases y objetos

Hoy por hoy, las clases son el mecanismo más completo y extendido para definir y utilizar TAD. De hecho, su utilización implica tantos conceptos nuevos que hablamos de un paradigma de programación propio: la *programación orientada a objetos*. Una **clase** es, al mismo tiempo, un tipo definido por el usuario y un módulo donde se encapsulan los datos y operaciones de ese tipo. Un **objeto** es una instancia particular de una clase, un valor concreto.

Las clases permiten usar ocultamiento de la implementación. Para ello, en la definición de la clase es posible decir qué partes son visibles desde fuera (sección `public`) y cuáles no (sección `private`). En el ejemplo de las pilas, los atributos `Datos` y `Tope` deben ser privados, y `Push`, `Pop`, etc., públicos. La definición del tipo, usando el lenguaje ObjectPascal[1], sería como la siguiente.

```
type
  PilaEntero= class
    private:
      Tope: integer;
      Datos: array [1..100] of integer;
    public:
      procedure Inicializar;
      procedure Push (valor: integer);
      procedure Pop;
      function Top: integer;
  end;
```

[1]Utilizamos aquí ObjectPascal para facilitar la comparación con las definiciones de los apartados anteriores. En las prácticas se usará el lenguaje C++.

```
...
var PilaUno, PilaDos: PilaEntero;
  ...
  PilaUno.Push(23);
  PilaDos.Pop;
  Write(PilaUno.Top);
  ...
  PilaUno.Datos[72]:= 36;  ----> ERROR EN TIEMPO DE COMPILACIÓN
  PilaDos.Tope:= -12;      ----> ERROR EN TIEMPO DE COMPILACIÓN
  ...
```

Las pilas, y en general cualquier objeto de una clase, sólo pueden ser manipuladas a través de las operaciones públicas de su definición. De esta forma, es posible conseguir la ocultación de la implementación.

Hay que notar que en la definición de las operaciones con pilas, la propia pila no aparece como un parámetro. Por ejemplo, donde antes teníamos: `Pop(PilaUno)` ahora tenemos: `PilaUno.Pop`. Esto es lo que se conoce como el **principio de acceso uniforme**: los atributos y las operaciones de una clase están conceptualmente al mismo nivel, y se accede a ellos usando la notación punto ".". En la llamada `PilaUno.Pop`, el valor `PilaUno` (llamado "objeto receptor") es un parámetro implícito de la operación `Pop`.

Algunos lenguajes orientados a objetos permiten aplicar *parametrización de tipo*, dando lugar a la definición de TAD genéricos o parametrizados. Por ejemplo, usando el lenguaje C++ es posible definir el tipo genérico `Pila<T>`, de pilas de cualquier tipo `T`. La definición se hace a través de la sentencia `template`.

```
template <class T>
  class Pila {
    private:
      T Datos [ ];
      int Maximo, Tope;
    public:
      Pila (int max);    // Operación de inicialización de una pila
      ~Pila ();          // Operación de eliminación de una pila
      Push (T valor);
      Pop ();
      T Tope ();
  };
  ...
  Pila<int> PilaUno(100);  // Instanciación a pila de enteros, con tamaño 100
  Pila<char> PilaDos(200); // Instanciación a pila de caracteres, con tamaño 200
  ...
  PilaUno.Push(5);
  PilaDos.Push('w');
  ...
```

Dentro de las prácticas profundizaremos en el uso de las clases como un mecanismo de definición de tipos abstractos, y en la utilización de patrones para conseguir clases parametrizadas.

2.2. Especificaciones informales

En una especificación informal, normalmente la descripción de una abstracción se realiza textualmente utilizando el lenguaje natural. En consecuencia, las especificaciones informales pueden resultar ambiguas e imprecisas. Lo que una persona entiende de una forma, otra lo puede entender de manera distinta. Es más, puede que la especificación no sea completa, quedando parte del comportamiento sin especificar.

Una **notación de especificación**, formal o informal, define las partes que debe tener la especificación, las partes que pueden ser opcionales, y el significado y tipo de descripción de cada parte. Existe una gran variedad de notaciones de especificación informal, muchas de ellas adaptadas a lenguajes de programación y aplicaciones concretos. En este punto se estudia un formato no ligado a ningún lenguaje particular.

2.2.1. Especificación informal de abstracciones funcionales

La especificación informal de una abstracción funcional –es decir, de una rutina, función o un procedimiento– debe contener la siguiente información:

- Nombre de la operación.
- Lista de parámetros, tipo de cada parámetro y del valor devuelto, en su caso.
- Requisitos necesarios para ejecutar la operación.
- Descripción del resultado de la operación.

La notación propuesta para la especificación informal de abstracciones funcionales tiene el siguiente formato:

Operación <nombre> (**ent** <id>:<tipo>; <id>:<tipo>;... ; **sal** <tipo_resultado>)
Requiere: Descripción textual, donde se establecen los requisitos y restricciones de uso de la operación.
Modifica: Lista de parámetros de entrada que se modifican o pueden modificarse.
Calcula: Descripción textual del resultado de la operación.

Las cláusulas **Requiere** y **Modifica** son opcionales, al igual que es posible que no existan parámetros de entrada o de salida. Si la función es parametrizada, entonces el formato sería del siguiente tipo:

Operación <nombre>[<id>:<tipo>;...] (**ent** <id>:<tipo>; ... ; **sal** <tipo_result>)
...

Aunque no resulta muy frecuente, la función puede estar parametrizada por más de un tipo genérico. Además, la cláusula Requiere puede contener restricciones sobre estos tipos genéricos. A continuación se muestran algunos ejemplos de especificaciones informales de abstracciones funcionales, usando el formato definido.

Ejemplo 2.1 Especificación informal de una operación QuitarDuplicados, para eliminar los elementos repetidos de un array de enteros.

Operación QuitarDuplicados (**ent** *A*: **array** [] **de** entero; **sal** entero)
  Requiere: El tamaño de *A* debe ser mayor que cero.
  Modifica: *A*
  Calcula: Elimina los elementos duplicados del array *A*, manteniendo el orden de los elementos que no se repiten. Devuelve el número de los elementos no repetidos existentes.

Ejemplo 2.2 Especificación informal de una operación parametrizada RecorridoPreorden, para hacer el recorrido en preorden de un árbol binario de elementos de cualquier tipo.
Operación RecorridoPreorden [T: tipo] (**ent** *A*: ArbolBinario[T]; **sal** Lista[T])
  Calcula: Devuelve como resultado una lista con los elementos del árbol *A* recorridos en preorden.

Ejemplo 2.3 Especificación informal de una operación parametrizada BuscarEnArray, para buscar un valor dentro de un array de cualquier tipo T.
Operación BuscarEnArray [T: tipo] (**ent** *A*: **array** [] **de** T; *valor*: T; **sal** entero)
  Requiere: El tipo T debe tener definida una operación de comparación *Igual(ent T, T; sal booleano)*.
  Calcula: Devuelve el menor valor de índice *i*, tal que *Igual(valor, A[i])* = *verdadero*. Si no existe tal valor, entonces devuelve un número fuera del rango del array *A*.

Normalmente, las características del formato usado (partes de la especificación, nivel de detalle de la explicación, cosas que se describen más o menos) suelen variar mucho de una aplicación a otra. En la figura 2.2 se muestran dos ejemplos concretos de especificaciones informales en aplicaciones reales. En ambos casos, se puede decir que la descripción está *centrada en los parámetros*; se hace especial énfasis en el efecto de cada uno de los parámetros sobre el resultado final de la operación.

2.2.2. Especificación informal de abstracciones de datos

La especificación informal de un TAD describe el significado y uso del tipo, enumerando las operaciones que se pueden aplicar. La especificación debería contener al menos la siguiente información:

- Nombre del TAD.
- Si el TAD es un tipo parametrizado, número de parámetros. En ese caso, se pueden imponer restricciones sobre los tipos que se usen como parámetros en la instanciación.
- Lista de operaciones definidas sobre los valores del TAD.
- Especificación de cada una de las operaciones de la lista anterior.

La notación propuesta para la especificación informal de abstracciones de datos tiene la siguiente forma.

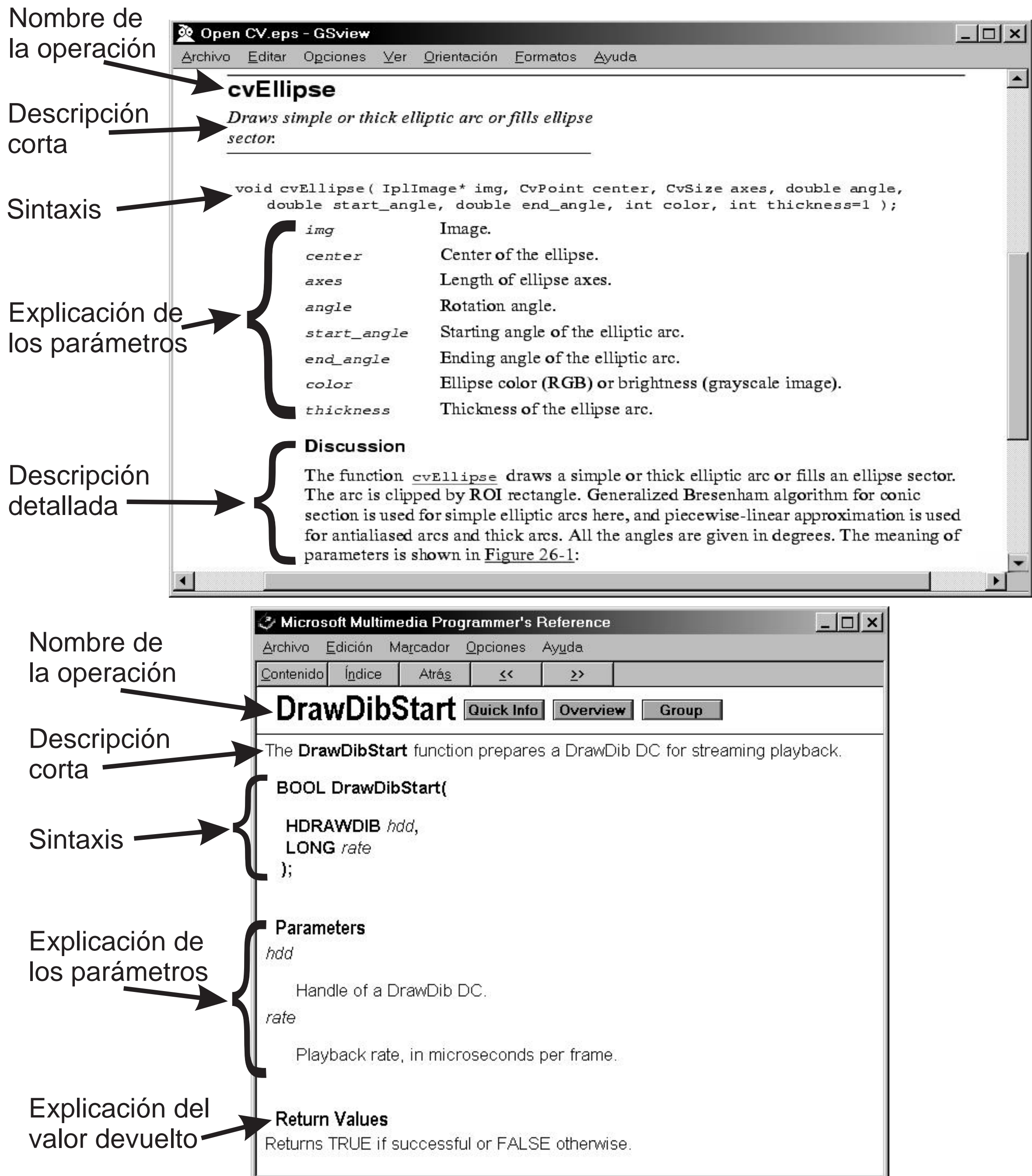


Figura 2.2: Especificaciones informales como documentación del código. Arriba: procedimiento "cvEllipse" de la librería Intel OpenCV. Abajo: función "DrawDibStart" de las librerías Microsoft Multimedia API.

TAD <nombre> **es** <lista de operaciones>
Descripción
Descripción textual del significado, comportamiento y modo de uso del tipo abstracto.
Operaciones
Especificación informal de cada una de las operaciones de <lista de operaciones>.
Fin <nombre>.

Vamos a ver algunas especificaciones de ejemplo, sobre tipos parametrizados y no parametrizados, con representaciones mutables o inmutables. Esta última distinción se puede realizar teniendo en cuenta la forma de las operaciones: en la representación inmutable los parámetros de entrada nunca se modifican.

Ejemplo 2.4 Especificación informal del TAD Polinomio.
TAD Polinomio **es** Crear, Grado, Coef, Sumar, Prod, Igual
Descripción
Los valores de tipo Polinomio son polinomios de coeficientes enteros no modificables.
Operaciones
Operación Crear (**ent** *c*, *n*: entero; **sal** Polinomio)
Requiere: $n \geq 0$.
Calcula: Devuelve el polinomio cx^n.
Operación Grado (**ent** *P*: Polinomio; **sal** entero)
Calcula: Devuelve el grado del polinomio *P*.
Operación Coef (**ent** *P*: Polinomio; *n*: entero; **sal** entero)
Calcula: Devuelve el coeficiente del polinomio *P* asociado al término x^n. Si no existe ese término, devuelve 0.
Operación Sumar (**ent** *P*, *Q*: Polinomio; **sal** booleano)
Calcula: Devuelve la suma de los polinomios $P+Q$.
Operación Prod (**ent** *P*, *Q*: Polinomio; **sal** booleano)
Calcula: Devuelve el producto de los polinomios $P \times Q$.
Operación Igual (**ent** *P*, *Q*: Polinomio; **sal** booleano)
Calcula: Devuelve verdadero si $P = Q$ y falso en caso contrario.
Fin Polinomio.

Ejemplo 2.5 Especificación informal del TAD parametrizado Asociacion, que almacena un par de valores $(clave, valor)$.
TAD Asociacion[tclave, tvalor: tipo] **es** Crear, Clave, Valor, PonValor, Igual
Requiere
Los tipos tclave y tvalor deben tener definidas sendas operaciones de comparación *Igual(ent tclave, tclave; sal booleano)* e *Igual(ent tvalor, tvalor; sal booleano)*.
Descripción
Los valores de tipo *Asociación[tclave, tvalor]* son tuplas modificables, compuestas por el par *(clave: tclave, valor: tvalor)*, que asocia una clave con un valor correspondiente.
Operaciones
Operación Crear (**ent** *c*: tclave; *v*: tvalor; **sal** Asociacion[tclave,tvalor])
Calcula: Devuelve la asociación compuesta por el par *(c, v)*.
Operación Clave (**ent** *A*: Asociacion[tclave,tvalor]; **sal** tclave)

Calcula: Devuelve el primer elemento de la tupla *A*, es decir si *A* = *(c, v)*, devuelve *c*.

Operación Valor (**ent** *A*: Asociacion[tclave,tvalor]; **sal** tvalor)

Calcula: Devuelve el segundo elemento de la tupla *A*, es decir si *A* = *(c, v)*, devuelve *v*.

Operación PonValor (**ent** *A*: Asociacion[tclave,tvalor]; valor: tvalor)

Requiere: El nuevo valor debe ser distinto del antiguo, es decir *Igual(Valor(A), valor)* = *falso*.

Modifica: *A*

Calcula: Cambia el segundo elemento de la tupla *A*, poniendo *valor*.

Operación Igual (**ent** *A1*, *A2*: Asociacion[tclave,tvalor]; **sal** booleano)

Calcula: Devuelve *verdadero* si *A1* y *A2* contienen la misma clave y valor.

Fin Asociacion.

2.2.3. Especificación informal de abstracciones de iteradores

A diferencia de lo que ocurre con las abstracciones funcionales y de datos, no existe un claro consenso sobre el formato de las abstracciones de iteradores y la forma de implementarlas en los lenguajes de programación. Recordemos que la abstracción de iteradores implica un procesamiento repetido sobre los elementos de una colección.

Si utilizamos un lenguaje de programación que permita pasar funciones como parámetros, entonces la abstracción de iteradores se puede expresar como una abstracción funcional, donde un parámetro es la colección sobre la que iterar y el otro es la acción a aplicar. Por lo tanto, la especificación tendrá un formato similar a la especificación de abstracciones funcionales.

Iterador <nombre> (**ent** <id>:<tipo>; <id>:<tipo>;... ; **sal** <tipo_resultado>)

Requiere: Descripción textual de los requisitos del iterador.

Modifica: Indicación de si se modifica la colección de entada o no.

Calcula: Descripción textual del resultado del iterador.

Ejemplo 2.6 Especificación informal de un iterador ParaTodoHacer, para iterar sobre los elementos de un conjunto de cualquier tipo.

Iterador ParaTodoHacer [T: tipo] (**ent** *C*: Conjunto[T]; *accion*: Operacion)

Requiere: *accion* debe ser una operación que recibe un parámetro de tipo *T* y no devuelve nada, *accion(ent T)*.

Calcula: Recorre todos los elementos *c* del conjunto *C*, aplicando sobre ellos la operación *accion(c)*.

Ejemplo 2.7 Especificación informal de un iterador Seleccionar, que devuelve todos los elementos de una colección que cumplen cierta condición.

Iterador Seleccionar [*T*: tipo; *Coleccion*: tipo_coleccion] (**ent** *C*: Coleccion[T]; *condicion*: Operacion; **sal** Coleccion[T])

Requiere: *condicion* debe ser una operación que recibe un parámetro de tipo *T* y devuelve un booleano, *condicion (ent T; sal booleano)*. *tipo_coleccion* debe ser un TAD parametrizado, que incluya un iterador *ParaTodoHacer*.

Calcula: Devuelve una colección que contiene todos los elementos *c* de *C* tal que *condicion(c)* es *verdadero*.

Si el lenguaje no permite pasar procedimientos como parámetros, la abstracción de iteradores puede ser realizada a través de un TAD que proporcione operaciones para iniciar la iteración, obtener el elemento actual, avanzar al siguiente y comprobar si hemos procesado todos los elementos. En este caso, la notación toma una forma similar a la especificación de abstracciones de datos.

TipoIterador <nombre> **es** <lista de operaciones> **Requiere** Requisitos del iterador. **Descripción** Descripción textual del significado y modo de uso del iterador. **Operaciones** Especificación informal de cada una de las operaciones de <lista de operaciones>. **Fin** <nombre>.

Ejemplo 2.8 Especificación informal de un iterador IteradorPreorden, que permite recorrer todos los elementos de un árbol binario en preorden.

TipoIterador IteradorPreorden [T: tipo] **es** Iniciar, Actual, Avanzar, EsUltimo

Descripción

Los valores de tipo *IteradorPreorden[T]* son iteradores definidos sobre árboles binarios de cualquier tipo *T*. Los elementos del árbol son devueltos en preorden. El iterador se debe inicializar con *Iniciar*.

Operaciones

Operación Iniciar (**ent** *A*: ArbolBinario[T]; **sal** IteradorPreorden)

Calcula: Devuelve un iterador nuevo, colocado sobre la raíz de *A*.

Operación Actual (**ent** *iter*: IteradorPreorden; **sal** T)

Calcula: Devuelve el elemento actual en la iteración.

Operación Avanzar (**ent** *iter*: IteradorPreorden)

Requiere: *EsUltimo(iter)* debe ser *falso*.

Modifica: *iter*

Calcula: Avanza el iterador *iter* al siguiente elemento en preorden, dentro del árbol binario sobre el que fue creado.

Operación EsUltimo (**ent** *iter*: IteradorPreorden; **sal** booleano)

Calcula: Devuelve *verdadero* si y sólo si el elemento actual de la iteración es el último.

Fin IteradorPreorden.

2.3. Especificaciones formales algebraicas

En una especificación formal, el comportamiento de una operación o de un TAD se describe de forma rigurosa utilizando una notación matemática no ambigua. En las