CAPÍTULO 5

MODELO DE REPRESENTACIÓN

En este capítulo queremos introducir una serie de conceptos necesarios para la comprensión del proceso de construcción de algoritmos y programas: la abstracción y la modularidad. También se presenta la noción de paradigma de programación.

El objetivo final del capítulo es lograr que se comprenda el modo en que se aborda un problema del que se busca una solución informática. Qué información del problema es importante, cómo se codifica, cómo se plantea la definición de tareas que deberán ser resueltas mediante diferentes algoritmos.

Introducción

La creación de un programa no es una tarea lineal. Primero se debe definir el problema que se desea solventar y sobre el que buscamos diseñar una solución: un algoritmo o conjunto de algoritmos que

conducen a solventar el problema planteado. No existe habitualmente una única solución, y muchas veces tampoco se puede destacar a una como mejor que las demás. Además, la solución adoptada debe ser eficiente, que sepa hacer buen uso de los recursos de los que se dispone. Hay muchos problemas para los que aún no se ha obtenido una solución aceptable. Uno de esos recursos es el tiempo: no todas las soluciones son igualmente rápidas.

Además, los programas necesitan con mucha frecuencia modificaciones en sus instrucciones o en las definiciones de sus datos. Los problemas evolucionan y sus soluciones también. Poco a poco mejora la comprensión de los problemas que se abordan, y por tanto soluciones antes adoptadas necesitan pequeñas o no tan pequeñas modificaciones. Los algoritmos cambian a medida que se conoce mejor el problema y su entorno, y en la medida en que se aprenden caminos más eficientes para alcanzar soluciones parciales o globales de nuestro problema.

El primer paso cuando se pretende resolver un problema mediante medios informáticos consiste en la abstracción del problema en busca de un modelo que lo represente. Así, mediante la creación de una representación simplificada, se consideran sólo aquellos detalles que nos interesan para poder tratar el problema. Primeramente hay que determinar cuál es exactamente el resultado que se busca y se desea obtener; luego cuáles son los valores de entrada de los que disponemos, identificando aquellos que sean realmente necesarios para la consecución de nuestro objetivo. Por último, hay que determinar con precisión los pasos que deberá seguir el proceso para alcanzar el resultado final buscado.

Al conjunto formado por el problema, con todos sus elementos y la solución buscada, y todo su entorno, es a lo que llamaremos **sistema**.

Abstracción

La **abstracción** es la capacidad de identificar los elementos más significativos de un sistema que se está estudiando, y las relaciones entre esos elementos: permite separar lo esencial de lo accesorio. La correcta abstracción del sistema que se aborda capacita para la construcción de modelos que permiten luego comprender la estructura del sistema estudiado y su comportamiento. La correcta abstracción, si de verdad está bien hecha, permite centrar el trabajo en los aspectos esenciales de los problemas que se deben abordar.

La abstracción es un paso previo en la construcción de cualquier programa. Fundamentalmente hablaremos de dos formas de abstracción:

1. Por un lado se deben determinar los **tipos de datos** que interviene en el sistema, es decir, cuál es el conjunto de parámetros que definen su estado en todo momento y su rango de valores posibles, y las operaciones que pueden realizarse con esos valores. También interesa determinar cuáles son los valores iniciales y los resultados finales que resumen los **estados inicial y final** del sistema.
2. Por otro lado, se debe también determinar las **funciones** o **procedimientos** del sistema. Los procedimientos que definen su comportamiento.

Modularidad

La **modularidad** es la capacidad de dividir el sistema sobre el que estamos trabajando en sus correspondientes partes diferenciadas (módulos), cada una de ellas con sus propias responsabilidades y subtareas. En una buena modularización de un sistema, para cada uno de los módulos deben quedar bien definidas sus relaciones con todos los demás módulos, su modo de comunicación con todo el resto del sistema.

Qué sea lo que se considera por módulo depende del paradigma de programación que se utilice. En el lenguaje C, que es un lenguaje del paradigma imperativo y estructurado (ya veremos más adelante en este capítulo estos conceptos) a cada módulo lo llamaremos **función** o **procedimiento**. En Java, que es un lenguaje de paradigma de programación orientado a objetos, un módulo puede ser una **clase**.

La modularidad permite convertir un problema en un conjunto de problemas menores, más fáciles de abordar. Así se logra la división del trabajo entre programadores o equipos de programadores, se aumenta la claridad del software que se desarrolla y se favorece la reutilización de parte del software desarrollado para problemas distintos para los que pudiera haber algún módulo semejante a los ya desarrollados. Además, en muchas ocasiones, este modo de trabajar reduce los costes de desarrollo del software y de su posterior mantenimiento.

Esta tarea, tan ventajosa, no es en absoluto trivial. Determinar correctamente los módulos que describen el funcionamiento del sistema y lograr definir las relaciones entre todos ellos puede llegar a ser muy complicado. De hecho esta modularización de un sistema se realiza siempre mediante técnicas de refinamientos sucesivos, pasando de un problema general a diferentes módulos que, a su vez, pueden considerarse como otro problema a modularizar; y así sucesivamente, hasta llegar a partes o módulos muy simples y sencillos de implementar.

Para mejor comprender este concepto de modularidad proponemos un ejemplo sencillo. Todos conocemos el **Triángulo de Tartaglia**: está formado por números combinatorios, ordenados de una forma concreta, tal y como se muestra en la figura 5.1., donde la expresión de los números combinatorios también es conocida:

$$\binom{m}{k} = \frac{m!}{n! \times (m-n)!}$$

Queremos hacer un programa que muestre una cualquiera de las filas del triángulo de Tartaglia. El programa preguntará al usuario qué fila

desea que se muestre, y entonces el programa, mediante el algoritmo que definamos, calculará los valores de esa fila, que mostrará luego por pantalla.

fila 1 → $\binom{0}{0}$

fila 2 → $\binom{1}{0}\ \binom{1}{1}$

fila 3 → $\binom{2}{0}\ \binom{2}{1}\ \binom{2}{2}$

fila 4 → $\binom{3}{0}\ \binom{3}{1}\ \binom{3}{2}\ \binom{3}{3}$

fila k → $\binom{k-1}{0}\ \binom{k-1}{1}\ \dots\ \binom{k-1}{k-2}\ \binom{k-1}{k-1}$

Figura 5.1.: Distribución de valores en el Triángulo se Tartaglia

El algoritmo que logra resolver nuestro problema tendría la siguiente forma inicial:

Algoritmo de Tartaglia:

1. [Entrada del nº de la fila a mostrar]: **Leer** $fila$.
2. **Para** $I = 0$ **Hasta** $fila - 1$ **Repetir**:
   2.1. [Mostrar resultado parcial]: **Mostrar** binomio: $\binom{fila-1}{I}$
3. **Fin**.

El flujograma de este algoritmo quedaría como se recoge en la figura 5.2., donde se ve que hemos utilizado el diagrama de "**Llamada a Procedimiento**". Esta llamada a procedimiento significa y exige la creación de un nuevo programa o módulo que calcule el binomio o número combinatorio.

Hay, pues, que diseñar un nuevo algoritmo, que se habrá de emplear repetidamente en nuestro primer algoritmo diseñado para mostrar una línea del triángulo de Tartaglia. Este segundo algoritmo podría tener la siguiente forma:

Algoritmo del Binomio:

1. [Entrada de los valores de m y k]: **Leer** m y k
2. [Calcular]: $B = factorial(m)$.

4. [Calcular]: $aux = factorial(k)$.
3. [Calcular]: $B = B \,/\, aux$.
4. [Calcular]: $aux = factorial(m - k)$.
5. [Calcular]: $B = B \,/\, aux$.
6. [Devolver valor]: **Devolver** B.
7. **Fin**.

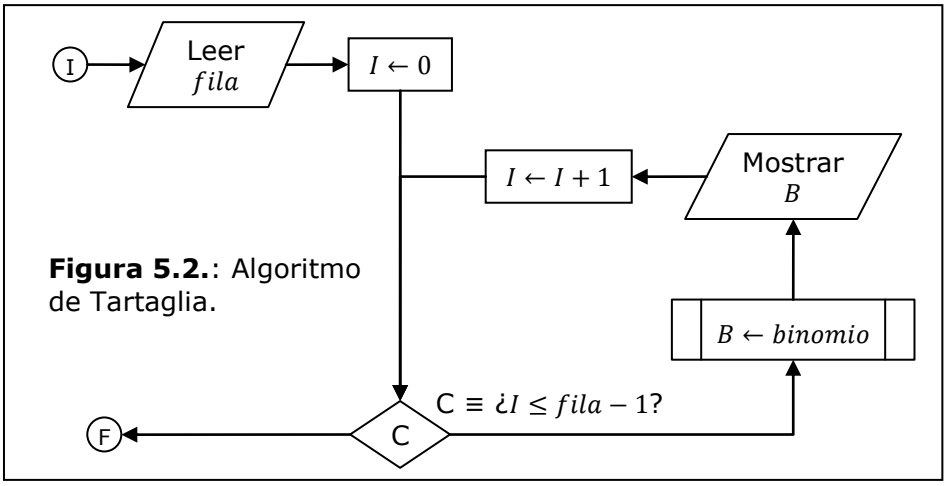


Figura 5.2.: Algoritmo de Tartaglia.

El flujograma de este nuevo algoritmo queda recogido en la figura 5.3., donde de nuevo nos encontramos con varias llamadas a un mismo procedimiento: la que se emplea para calcular el valor factorial de los diferentes valores. Y es que, como ya se dijo en el capítulo anterior, un algoritmo debe estar compuesto por operaciones simples: el ordenador no sabe calcular el valor factorial de un entero y hay, por tanto, que definir también un algoritmo que calcule ese valor para un entero dado. (Este algoritmo lo hemos dejado definido en la Figura 4.2.)

En el paso 6 de nuestro Algoritmo del Binomio ya no hemos puesto [Mostrar resultado], sino [Devolver valor], puesto que el valor del binomio no es la solución de mi problema y no se tiene interés alguno en conocer su valor: es un cálculo intermedio necesario para lograr encontrar nuestra información verdaderamente buscada: la fila del Triángulo de Tartaglia indicada como entrada de todo el proceso que estamos afrontando.

Por lo mismo, el algoritmo del cálculo del Factorial presentado en el

Capítulo anterior deberá sufrir una pequeña modificación, pues no se espera que "muestre" resultado alguno, sino simplemente que "devuelva" o facilite el valor del factorial del entero recibido, pues de nuevo no hay interés alguno en conocer ningún valor factorial calculado.

Con este ejemplo hemos visto que para afrontar y resolver satisfactoriamente un problema es conveniente descomponerlo en partes menores, independientes y más sencillas, que hacen el proceso más inteligible y fácil de resolver. Es a este proceso al que llamamos de creación de módulos, o modularización.

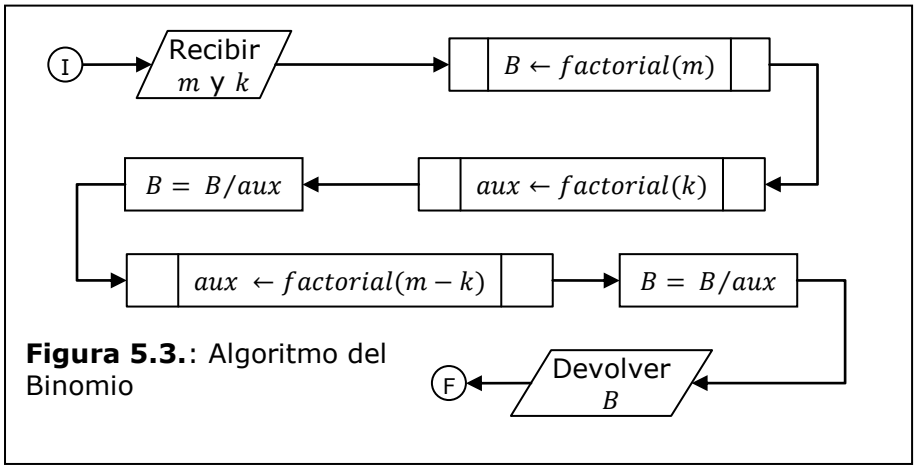


Figura 5.3.: Algoritmo del Binomio

Otro ejemplo: supongamos que queremos hacer un programa que sirva para operar con números complejos. Queremos definir las operaciones de suma, de resta y de producto de dos complejos. Vamos a representar aquí un número complejo como un par de reales: $a + b \cdot i = (a, b)$ donde a y b son reales.

Habrá que definir tres módulos o procedimientos que definan cada una de las tres operaciones. Cada uno de ellos recibe como datos de entrada dos números complejos, o lo que es lo mismo dos pares de números reales; y ofrece como salida un número complejo.

La definición de los tres módulos queda definido por los algoritmos representamos en los flujogramas de la Figura 5.4.

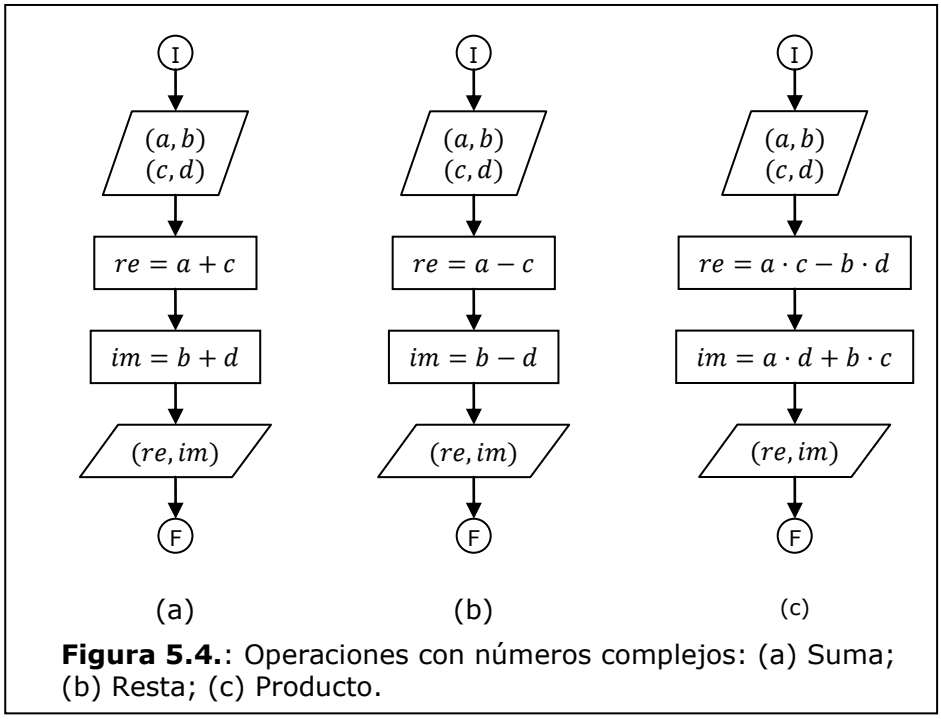


Figura 5.4.: Operaciones con números complejos: (a) Suma; (b) Resta; (c) Producto.

El algoritmo de la aplicación (lo desarrollamos ahora en pseudocódigo) podría tener la siguiente definición. Suponemos que la aplicación recibe como entrada, además de los complejos a operar, un carácter que indica cuál es la operación que se desea realizar.

1. [Entrada de datos]: $V1 \leftarrow (a,b)$, $v2 \leftarrow (c,d)$, op: $(+|-|*)$.
2. **Si** $op = '+'$, **Entonces** $R = Suma(V1,V2)$.
3. **Sino Entonces:**
   3.1. **Si** $op = '-'$ **Entonces** $R = Resta(V1,V2)$
   3.2. **Sino Entonces**
      3.2.1. **Si** $op = '*'$ **Entonces** $R = Producto(V1,V2)$
      3.2.2. **Sino Entonces**
         [Mostrar Mensaje]: **Mostrar**: "op. inválido".
4. **Si** $op = '+'$ **ó** $op = '-'$ **ó** $op = '*'$ **Entonces**
   [Mostrar Resultado]: **Mostrar** R.
5. **Fin**.

Evidentemente, si el usuario solicita una operación no definida, el algoritmo deberá advertirlo y no realizar nada.

Como ya hemos vislumbrado en ambos ejemplos, los módulos que se obtienen deben gozar de algunas propiedades, necesarias si se desea

que la modularización resulte finalmente útil:

1. **Independencia funcional**. Es decir, cada módulo definido debe realizar una función concreta o un conjunto de funciones afines (alta cohesión), sin apenas ninguna relación con el resto (bajo acoplamiento).

   En el ejemplo de las operaciones con valores complejos, tenemos definidas tres módulos que en nada interfieren el uno con el otro.

   En el ejemplo del triángulo de Tartaglia tenemos dos niveles de independencia: el módulo *Binomio* necesita del módulo *Factorial*. Para que el módulo *Factorial* se ejecute correctamente no necesita de nada del resto del proceso definido excepto, claro está, de una valor de entrada. Y para que el módulo *Binomio* ejecute correctamente no necesita de nada del resto del proceso definido en su mismo nivel excepto, de nuevo, de dos valores de entrada.

2. **Comprensibilidad**. Es decir, cada módulo debe ser comprensible de forma aislada. Para lograr eso, desde luego se requiere la independencia funcional; y también establecer correctamente las relaciones entre cada módulo definido y los restantes.

   En nuestros dos ejemplos, cada uno de los módulos realiza una tarea concreta y bien definida.

3. **Adaptabilidad**. Es decir, los módulos se deben definir de manera que permitan posteriores modificaciones o extensiones, realizadas tanto en ese módulo como en los módulos con los que se relaciona.

   En nuestro ejemplo del triángulo de Tartaglia, el módulo *Factorial*, una vez definido, ya es válido para todo proceso que necesite el cálculo de ese valor. De hecho ni siquiera ha sido necesario presentar este módulo en este capítulo porque ha bastado hacer referencia a la descripción que ya se hizo en el capítulo anterior: en nada importa el entorno concreto en el que queremos que se ejecute el algoritmo de cálculo del factorial.

4. Una advertencia importante para la correcta definición y construcción de un nuevo módulo: la correcta descripción del modo en que se utiliza: lo que podríamos llamar como una correcta definición de su **interfaz**. Las tres propiedades previas de la modularidad presentadas, exigen que cuando se defina un módulo o procedimiento quede perfectamente determinado el modo en que se hará uso de él: qué valores y en qué orden y forma espera recibir ese módulo como entrada para realizar correctamente el proceso descrito con su algoritmo. Y qué valores ofrece como resultado (y de nuevo en qué orden y forma) el módulo al finalizar las sentencias del algoritmo. Para que un módulo se considere debidamente definido debe ocurrir que, para cualquiera que desee usarlos, le sea suficiente conocer únicamente cuál es su interfaz. Cuál sea la definición del módulo, sus sentencias y su orden, es cuestión que en nada importa al usuario de ese módulo.

   Allí donde se desee realizar una operación suma de complejos se podrá acudir al módulo definido con el algoritmo recogido en la Figura 5.4. (a). Pero siempre que se haga uso de ese módulo será necesario que quien lo utilice facilite como entrada los dos pares de reales que definen los dos complejos, operandos de la operación suma que se desea realizar.

El proceso para la construcción de un programa pasa, por tanto, por estos pasos: abstracción → modularización → diseño de algoritmos → implementación. Entendemos por **implementación** el proceso de escribir cada uno de los algoritmos diseñados en un lenguaje de programación concreto.

De nuevo, y sin pretensión de que se entienda ahora el código que se presente, mostramos un programa escrito en C que realizaría la tarea indicada.

```
#include <stdio.h>

/* Declaración de módulos o funciones definidas. --------- */
```

```
void Tartaglia(unsigned short);
unsigned long Binomio(unsigned short, unsigned short);
unsigned long Factorial(unsigned short);

/* Función Principal. ----------------------------------- */
void main(void)
{
      unsigned short fila;
      printf("Aplicación que muestra una fila");
      printf(" del triángulo de Tartaglia.\n\n");
      printf("Indique número de la fila a visualizar ... ");
      scanf("%hu", &fila);

      Tartaglia(fila);
}

void Tartaglia(unsigned short f)
{
      unsigned short I;
      for(I = 0 ; I < f ; I++)
            printf("%lu\t", Binomio(f - 1 , I ));
}

unsigned long Binomio(unsigned short m, unsigned short k)
{
      return Factorial(m) / (Factorial(k)*Factorial(m - k));
}

unsigned long Factorial(unsigned short n)
{
      unsigned long Fact = 1;
      while(n) Fact *= n--;
      return Fact;
}
```

El programa, escrito en Java, podría tomar, por ejemplo, la siguiente forma:

```
import herramientas.Teclado;

public class Tartaglia {
      public long bin(short m, short k)   {
            return fact(m)/(fact(k)*fact((short)(m - k)));
      }

      public long fact(short n)     {
            long Fact = 1;
            while(n != 0) Fact *= n--;
            return Fact;
```

```
	}

	public Tartaglia(short f)	{
		short I;
		for(I = 0 ; I < f ; I++)
		    System.out.print("\t"+bin((short)(f-1), I));
	}


	public static void main(String[] args) {
		short fila;
		System.out.println("Programa: muestra 1 fila ");
		System.out.println("del triángulo de Tartaglia");
		System.out.println("Indique número de la fila");

		fila = Teclado.readShort();

		Tartaglia t = new Tartaglia(fila);
	}
}
```

Donde Teclado es una clase (que no mostramos aquí) y que permite tomar una entrada de teclado.

Queda pendiente presentar la forma en que podemos expresar los módulos o las funciones.

Para definir una función hay que especificar un nombre y varios conjuntos: el nombre de la función, y los conjuntos de posibles valores de entrada y el conjunto de los posibles valores que ofrece la función como salida.

Las formas en que expresaremos los métodos Binomio y Factorial son las siguientes:

Función Binomio (m y k Enteros) → Entero
Constantes:
	$\emptyset$
Variables:
	aux Entero
Acciones:
	1. $Binomio \leftarrow factorial(m)$.
	2. $aux \leftarrow factorial(k)$.
	3. $Binomio \leftarrow Binomio/aux$.
	4. $aux \leftarrow factorial(m-k)$.
	5. $Binomio \leftarrow Binomio/aux$.
	6. **FIN**.

Función Factorial (n Entero) → Entero
Constantes:
$\emptyset$
Variables:
$\emptyset$
Acciones:
1. **Mientras** $\neq 0$ **Repetir**
   1.1. $factorial \leftarrow factorial \times n$.
   1.2. $n \leftarrow n - 1$.
2. **FIN**.

La abstracción de la información: los datos.

En este proceso de abstracción de la realidad que hemos presentado, se ha seleccionado la información necesaria para resolver el problema planteado. Esta información viene recogida mediante unas entidades que llamamos datos. Entendemos por **dato** cualquier objeto manipulable por el ordenador: un carácter leído por el teclado, una información numérica almacenada en el disco de ordenador o que llega a través de la red, etc. Estos datos pueden estar vinculados entre sí mediante un conjunto de relaciones.

Un dato puede ser tanto una **constante** definida dentro del programa y que no altera su valor durante su ejecución; o dato **variable**, que puede cambiar su valor a lo largo de la ejecución el programa.

El conjunto de valores posibles que puede tomar una variable se llama **rango** o **dominio**. Por ejemplo, podemos definir un rango que sea todos los enteros comprendidos entre 0 y $2^8 - 1 = 255$ (256 valores posibles). Los diferentes valores recogidos en un determinado rango se denominan literales. Un **literal** es la forma en que se codifica cada uno de los valores posibles de un rango o dominio determinado. Entendemos por literal cualquier símbolo que representa un valor. Por ejemplo, *3* representa el número 3 (literal numérico), y "*Este es un texto de 7 palabras*" representa un texto (literal alfanumérico).

Los datos deberán ser codificados y ubicados en el programa mediante la reserva, para cada una de ellos, de un **espacio de memoria**. Los

diferentes valores que pueden tomar esos datos variables quedarán codificados en los distintos estados que puede tomar esa memoria. El estado físico concreto que tome en un momento concreto un determinado espacio de memoria significará un valor u otro dependiendo de cuál sea el código empleado.

Cada espacio de memoria reservado para codificar y ubicar un dato (lo llamaremos variable) deberá ser identificado de forma inequívoca mediante un nombre. Para la generación de esos nombres hará falta echar mano de un alfabeto y de unas reglas de construcción (cada lenguaje tiene sus propias reglas). A estos nombres los llamamos identificadores. Los **identificadores** son símbolos empleados para representar objetos. Cada lenguaje debe tener definidas sus reglas de creación. Si un identificador representa un literal, es decir, si se define para un valor concreto, no variable, entonces lo llamamos **constante**. Por ejemplo, se podría llamar PI a la constante que guarde el literal *3.14159*.

Tipo de dato

Un **tipo de dato** define de forma explícita un conjunto de valores, denominado dominio (ya lo hemos definido antes), sobre el cual se pueden realizar un conjunto de operaciones.

Cada espacio de memoria reservado en un programa para almacenar un determinado tipo de valores debe estar asociado a un tipo de dato. La principal motivación es la organización de nuestras ideas sobre los objetos que manipulamos.

Un lenguaje de programación proporciona un conjunto de tipos de datos simples o predefinidos (que se llaman los **tipos de dato primitivos**) y además proporciona mecanismos para definir nuevos tipos de datos, llamados compuestos, combinando los anteriores.

Distintos valores pertenecientes a diferentes tipos de datos pueden

tener la misma representación en la memoria. Por ejemplo, un byte con el estado 01000001 codificará el valor numérico *65* si este byte está empleado para almacenar valores de un tipo de dato entero; y codificará la letra *'A'* si el tipo de dato es el de los caracteres y el código empleado es el ASCII.

Por eso, es muy importante, al reservar un espacio de memoria para almacenar valores concretos, indicar el tipo de dato para el que se ha reservado ese espacio.

Variable

Una **variable** es un elemento o espacio de la memoria que sirve de almacenamiento de un valor, referenciada por un nombre, y perteneciente a un tipo de dato.

Podemos definir una variable como la cuádrupla

$$V = <N, T, R, K> \qquad (5.1.)$$

Donde N es el nombre de la variable (su identificador); T el tipo de dato para el que se creó esa variable (que le indica el dominio o rango de valores posibles); R es la referencia en memoria, es decir, la posición o dirección de la memoria reservada para esa variable (su ubicación); y K es el valor concreto que adopta esa variable en cada momento y que vendrá codificado mediante un estado físico de la memoria.

Por ejemplo, mediante <x, entero, 10001, 7> nos referimos a una variable que se ha llamado x, creada para reservar datos de tipo entero, que está posicionada en la dirección de memoria 10001 y que en este preciso instante tiene codificado el valor entero 7. Al hacer la asignación x ← 3, se altera el estado de esa memoria, de forma que pase a codificar el valor entero 3.

En la mayoría de los lenguajes (y también en C) es preciso que toda

variable se declare antes de su utilización. En la declaración se realiza la asociación de tipo de dato (que explicita el dominio de valores válidos) con el nombre de la variable.

En la declaración de la variable también queda definida la asociación entre el nombre y el espacio de memoria reservado: antes de la ejecución de un programa el ordenador ya conoce los requerimientos de espacio que ese programa lleva consigo.

Variable - Tipo de dato - Valor

Una definición intuitiva de **valor** es: elemento perteneciente a un conjunto. Ese conjunto se explicita mediante la declaración del tipo de dato. A este conjunto es al que hemos llamado rango o dominio.

Una vez introducidos estos tres conceptos, es útil pensar en ellos conjuntamente.

Una variable ocupa un porción de memoria. Que esa porción sea más o menos extensa dependerá de para qué haya sido reservada esa variable. Y eso lo determina el tipo de dato para el que se ha creado esa variable. Por ejemplo, si se crea una variable para almacenar enteros que pueden llegar a tomar valores muy grandes, entonces se necesitará al menos 32 bits para codificar esos valores. Si los enteros son pequeños, quizá baste con 16 bits.

Qué valor codifica una variable en un momento determinado también depende del tipo de dato. Dos estados iguales pueden codificar valores distintos si se trata de espacios de memoria que codifican tipos de dato diferentes.

Paradigmas de programación. Programación estructurada

Un lenguaje de programación refleja cierto paradigma de programación.

Un **paradigma** de programación es una colección de conceptos que guían el proceso de construcción de un programa y que determinan su estructura. Dichos conceptos controlan la forma en que se piensan y formulan los programas. Existen distintas formas de abordar un problema y darle solución mediante un programa informático, distintas enfoques que se pueden adoptar para resolver un problema de programación. A cada enfoque, a cada forma de actuar, se le llama paradigma.

La clasificación más común distingue tres tipos de paradigmas: imperativo, declarativo y orientado a objetos (que es una extensión del imperativo). Vamos a ceñirnos aquí a los lenguajes imperativos, que es el paradigma del lenguaje C.

Las **características** más importantes de los **lenguajes imperativos** son:

(1) el concepto de **variable** como medio para representar el estado del proceso computacional, y la instrucción de **asignación** como medio para cambiar el valor de una variable,

(2) las acciones y **funciones** como medio de descomposición de los programas, y

(3) las **instrucciones de control** que permiten establecer el orden en que se ejecutan las instrucciones.

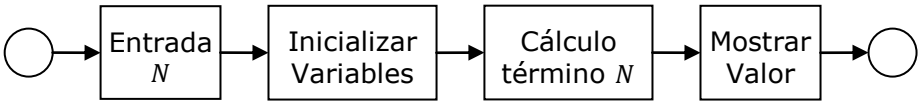
Dentro de estos lenguajes imperativos cabe distinguir a su vez los estructurados y los no estructurados. Los lenguajes estructurados incluyen estructuras de control que permiten determinar el orden de ejecución de las sentencias del algoritmo.

La programación estructurada establece las siguientes reglas:

1. Todo programa consiste en una serie de acciones o **sentencias que se ejecutan en secuencia**, una detrás de otra.

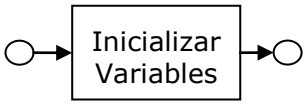
   Si tomamos el ejemplo del capítulo anterior del cálculo del término N

de la serie de Fibonacci, podemos esquematizar el proceso en estos pasos:

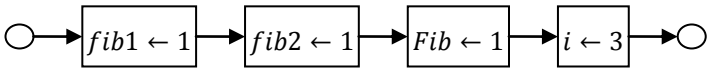


2. Cualquier acción puede ser sustituida por dos o más acciones en secuencia. Esta regla se conoce como la de **apilamiento**.

   Por ejemplo, el paso antes visto:

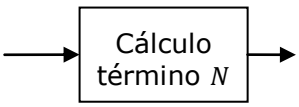


   puede representarse mejor como:

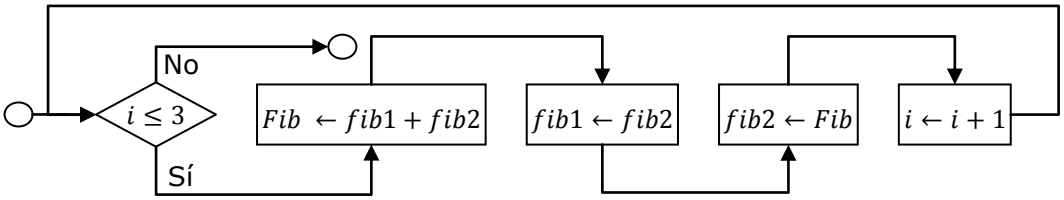


3. Cualquier acción puede ser sustituida por cualquier **estructura de control**; y sólo se consideran tres estructuras de control: la **secuencia** (ejecución secuencias de sentencias, una detrás de otra), la **selección** (o decisión) y la **repetición**. Esta regla se conoce como **regla de anidamiento**. Todas las estructuras de control de la programación estructurada tienen un solo punto de entrada y un solo punto de salida. (Estas estructuras de control ya las hemos visto en el capítulo 4 del manual.)

   Por ejemplo, la sentencia:



   puede representarse mejor como:



Las reglas de apilamiento y de anidamiento pueden aplicarse tantas veces como se desee y en cualquier orden.

El lenguaje C es un lenguaje imperativo y estructurado.

Recapitulación

En este capítulo hemos presentado una serie de conceptos muy necesarios para comprender el trabajo de la programación. La abstracción y la modularidad, como pasos previos e imprescindibles en todo trabajo de construcción de una aplicación: muchas horas de trabajo y de pensar y de hablar, con un bloque de papeles en la mesa, sin teclados ni pantallas, previas a la selección o definición de algoritmos, y previas a la implementación a uno u otro lenguaje de programación.

Y hemos presentado el concepto de paradigma de programación, que determina distintos modos de trabajar y de formas de plantear soluciones. Los lenguajes de programación quedan clasificados en función de su paradigma. Para cada modo de plantear y resolver un problema existe un conjunto de lenguajes aptos para expresar el modelo abstraído y modularizado.

Ejemplo de lenguaje de paradigma imperativo y estructurado es el lenguaje C. El hecho de que el lenguaje C pertenezca a este paradigma implica un modo de trabajar y de programar. Ya ha quedado dicho en este capítulo y en el anterior, pero no viene mal repetirlo: en lenguaje C no se deben emplear sentencias de salto: todo el control del flujo del programa debe organizarse mediante estructuras de control.

Ejemplos de lenguaje para programación orientada a objetos son los conocidos C++ y Java, entre otros. Aprender Java, por ejemplo, no es una tarea que se limite a conocer las reglas sintácticas de este lenguaje: requiere conocer a fondo el planteamiento que lleva consigo el paradigma de la programación orientada a objetos. Aprender Java sin conocer su paradigma es como enseñar a hablar a un mono.