Tema 1

Introducción

El objetivo de este tema es introducir los conceptos generales, y dar una panorámica de la programación que permita posteriormente situar en el contexto adecuado las técnicas y metodologías que se expondrán en el resto del libro.

Especialmente importante es la presentación de diferentes modelos abstractos de cómputo, para poner de manifiesto que la programación imperativa, aunque sea la más extendida, no constituye la única manera de representar programas, y que no hay que identificar el concepto de programa con el de secuencia de órdenes.

1.1 Máquinas y programas

Intuitivamente podemos asociar el concepto de *máquina* a un dispositivo o instrumento físico capaz de realizar un cierto trabajo u operación. El concepto puede extenderse incluyendo máquinas que, aunque no existan físicamente, pueden concebirse y describirse con precisión y predecir su comportamiento. Estas máquinas se denominan *máquinas virtuales*.

1.1.1 Máquinas programables

En general, las máquinas operan a lo largo del tiempo, por lo que el concepto de máquina lleva asociado el de un proceso de funcionamiento en el cual diferentes operaciones se van realizando sucesiva o simultáneamente. Desde el punto de vista del control de su funcionamiento, podemos clasificar las máquinas en diferentes tipos.

Las *máquinas no automáticas*, o de control manual, son gobernadas por un operador o agente externo que desencadena unas determinadas operaciones en cada momento. Por ejemplo, una máquina de escribir imprime las letras o mueve el papel de acuerdo con las teclas pulsadas por el mecanógrafo.

Las *máquinas automáticas* actúan por sí solas, sin necesidad de operador, aunque pueden responder a estímulos externos. Por ejemplo, un ascensor automático gobierna por sí mismo los movimientos de subida y bajada incluyendo cambios de velocidad, apertura y cierre de puertas, etc., de forma coordinada, respondiendo a los estímulos de los botones de llamada o envío a un piso dado.

El funcionamiento de una máquina automática puede depender de la forma en que está construida, es decir, de los elementos que la componen y la manera en que están conectados entre sí. En este caso el comportamiento de la máquina será fijo, en el sentido de que a unos determinados estímulos externos responderá siempre de la misma manera. Esto ocurre en el ejemplo del ascensor.

Otras máquinas automáticas se denominan programables , y su comportamiento no es siempre el mismo. Una *máquina programable* (figura 1.1) se puede concebir como una máquina base, de comportamiento fijo, que se completa con una parte modificable que describe el funcionamiento de la máquina base. Esta parte modificable se denomina *programa*.

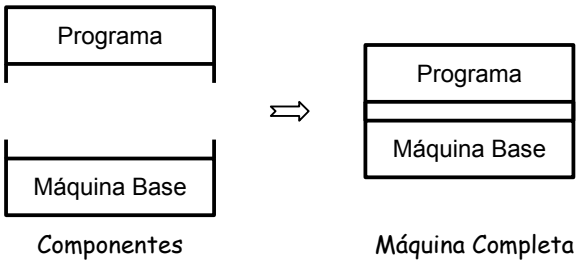

Figura 1.1 Componentes de una máquina programable.

Aunque habitualmente no se considere como tal, podemos analizar un reproductor de CD como una máquina programable, identificando el CD (reemplazable) con el programa. Incluso podemos establecer la siguiente serie de ejemplos:

- *Piano*: máquina manual de producir música.
- *Caja de música*: máquina automática de producir música (fija).
- *Reproductor de CD*: máquina programable de producir música (variable).

Dependiendo de cuál sea el programa que gobierne su funcionamiento, una máquina programable responderá a los estímulos externos de una forma o de otra. Una máquina programable se comporta, por tanto, como diferentes máquinas particulares, en función del programa utilizado.

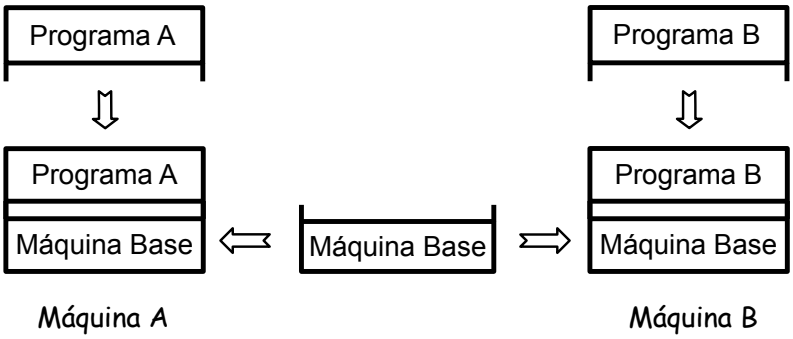

Figura 1.2 Una máquina programable puede comportarse como diferentes máquinas.

Cuando una máquina programable opera bajo control de un programa determinado, se dice que el programa se *ejecuta* en dicha máquina.

1.1.2 Concepto de cómputo

La palabra *cómputo* es sinónimo de cuenta o cálculo. Si consultamos un diccionario podemos encontrar una definición más elaborada:

> **Cómputo** (del latín *computum*). Determinación indirecta de una cantidad mediante el cálculo de ciertos datos.

En esta definición se puede apreciar que un cómputo es una operación de tratamiento de información. A partir de una información conocida se obtiene otra nueva como resultado de unos cálculos. En informática y de una forma general puede identificarse el concepto de cómputo con el de *tratamiento de la información*.

Un cómputo puede expresarse de diferentes maneras. Por ejemplo, mediante una fórmula o expresión matemática, tal como

$$34 \times 5 + 8 \times 7$$

Un cómputo se concibe también como un proceso a lo largo del cual se van realizando operaciones o cálculos elementales hasta conseguir el resultado final. En el ejemplo anterior se encuentra implícito dicho proceso. El resultado se podría obtener mediante los siguientes cálculos elementales:

1. Producto de 34 por 5, obteniendo 170.
2. Producto de 8 por 7, obteniendo 56.
3. Suma de los resultados anteriores, obteniendo 226.

En la expresión matemática usada como ejemplo están implícitos estos cálculos elementales, así como el orden en que pueden ser realizados. Los cálculos 1º y 2º podrían realizarse en cualquier orden, pero el cálculo 3º ha de realizarse necesariamente después de los otros dos.

1.1.3 Concepto de computador

La máquina programable por excelencia es el *computador*. Un computador se define como una máquina programable para tratamiento de la información, es decir, un computador es (¡obviamente!) una máquina para realizar cómputos.

Un programa de computador es, por tanto, una descripción de un cómputo. Al mismo tiempo nos encontramos con que un programa es también una descripción del comportamiento de una máquina, y podemos así considerarlo como una *máquina virtual* cuando convenga.

Un computador, como máquina programable que es, posee unos elementos fijos (máquina base) y otros modificables (programa). De forma simplificada podemos asociar los elementos fijos a los dispositivos físicos del computador, que constituyen el *hardware*, y los elementos modificables a las representaciones de los programas en sentido amplio, que constituyen el *software*.

Los computadores actuales corresponden a un tipo particular de máquinas programables que se denominan *máquinas de programa almacenado*. En estas máquinas la modificación del programa no implica un cambio de componentes físicos de la máquina, sino que estas máquinas poseen una memoria en la cual se puede almacenar información de cualquier tipo, debidamente codificada, y esta información incluye tanto los datos con los que opera la máquina como la representación codificada del programa. El programa es, por tanto, pura información, no algo material.

La estructura general de un computador se puede representar como se muestra en la figura 1.3. La memoria almacena datos y programas. Los dispositivos de entrada/salida permiten intercambiar información con el exterior, y el procesador es el elemento de control, que realiza operaciones elementales de tratamiento de la información interna, u operaciones de entrada o salida de información al exterior, de acuerdo con los códigos del programa que están almacenados en la memoria.

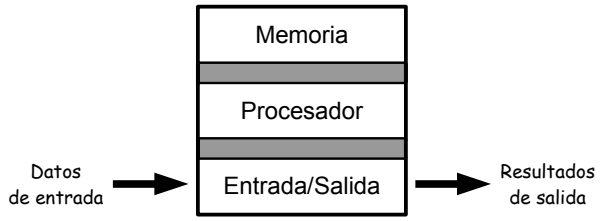

Figura 1.3 Esquema general de un computador.

1.2 Programación e ingeniería de software

De las explicaciones anteriores se deduce que una máquina programable, y en particular un computador, es totalmente inútil si no dispone del programa adecuado. Para realizar un determinado tratamiento de información con ayuda de un computador habrá sido necesario:

(a) Construir el computador (*hardware*).
(b) Idear y desarrollar el programa (*software*).
(c) Ejecutar dicho programa en el computador.

Sólo la última fase (c) es habitualmente realizada por el usuario. Las dos primeras corresponden a los profesionales de la informática: la fase (a) a los fabricantes de *hardware* y la (b) a los de *software*. En los siguientes apartados se analiza la actividad de desarrollo de software.

1.2.1 Programación

La labor de desarrollar programas se denomina en general *programación*. En realidad este término se suele reservar para designar las tareas de desarrollo de programas en pequeña escala, es decir, realizadas por una sola persona. El desarrollo de programas complejos, que son la mayoría de los usados actualmente, exige un equipo más o menos numeroso de personas que debe trabajar de manera organizada. Las técnicas para desarrollo de software a gran escala constituyen la *ingeniería de software*.

Programación e ingeniería de software no son disciplinas independientes, sino complementarias. El desarrollo de software en gran escala consiste esencialmente en descomponer el trabajo total de programación en partes independientes que pueden ser desarrolladas por miembros individuales del equipo. La ingeniería de software se limita a añadir técnicas o estrategias organizativas a las técnicas básicas de programación. El trabajo en equipo es, en último extremo, la suma de los trabajos realizados por los individuos.

1.2.2 Objetivos de la programación

La ingeniería de software excede del ámbito de este libro. En él nos centraremos sólo en la labor de programación, correspondiente a la preparación de programas medianos o pequeños, realizables por una sola persona. No obstante, las técnicas de programación han de establecerse con el objetivo de ser una base adecuada para la ingeniería de software. Entre los objetivos particulares de la programación podemos reconocer los siguientes:

- CORRECCIÓN: Es evidente que un programa debe realizar el tratamiento esperado, y no producir resultados erróneos. Esto tiene una consecuencia inmediata que no siempre se considera evidente: antes de desarrollar un programa debe especificarse con toda claridad cuál es el funcionamiento esperado. Sin dicha especificación es inútil hablar de funcionamiento correcto.
- CLARIDAD: Prácticamente todos los programas han de ser modificados después de haber sido desarrollados inicialmente. Por esta razón es fundamental que sus descripciones sean claras y fácilmente inteligibles por otras personas distintas de su autor, o incluso por el mismo autor al cabo de un cierto tiempo, cuando ya ha olvidado los detalles del programa.
- EFICIENCIA: Una tarea de tratamiento de información puede ser programada de muy diferentes maneras sobre un computador determinado, es decir, habrá muchos programas distintos que producirán los resultados deseados. Algunos de estos programas serán más eficientes que otros. Los programas eficientes aprovecharán mejor los recursos disponibles y, por tanto, su empleo será más económico en algún sentido.

Estos y otros objetivos resultan a veces contrapuestos. Quizá el ejemplo más intuitivo sea la dualidad entre claridad y eficiencia. Para ser claros los programas han de ser sencillos, pero para aprovechar los recursos de manera eficiente en muchos casos hay que introducir complicaciones que hacen el programa más difícil de entender.

Si se trata de establecer una importancia relativa entre los distintos objetivos, habría que considerar como prioritaria la *corrección*. Piénsese, por ejemplo, que un programa de contabilidad no es aceptable si no calcula correctamente los saldos de las cuentas.

A continuación debe perseguirse la *claridad*, que como ya se ha indicado es necesaria para poder realizar modificaciones, o simplemente para poder certificar que el programa es correcto. En realidad el objetivo de claridad va ligado al de corrección. Es prácticamente imposible asegurar que un programa es correcto si no puede ser entendido claramente por la persona que lo examina.

Tal como se ha dicho antes, la claridad facilita la tarea de realizar modificaciones cuando las necesidades así lo exijan. Puede afirmarse que esto ocurre siempre con todos los programas que tienen un cierto interés.

Finalmente ha de atenderse a la *eficiencia*. Este objetivo, aunque importante, sólo suele ser decisivo en determinados casos. En muchas situaciones el aumento de capacidad de los computadores a medida que avanza la tecnología va permitiendo utilizar de manera aceptable, desde el punto de vista económico, programas relativamente menos eficientes.

1.3 Lenguajes de programación

Ya se ha explicado que un computador funciona bajo control de un programa que ha de estar almacenado en la unidad de memoria. El programa contiene una descripción codificada del comportamiento deseado del computador.

Cada modelo de computador podrá utilizar una forma particular de codificación de programas, que no coincidirá con la de otros modelos. La forma de codificar programas de una máquina en particular se dice que es su *código de máquina* o *lenguaje de máquina*. La palabra "lenguaje" utilizada habitualmente en el vocabulario informático en español es, en realidad, una transcripción directa del término inglés "*language*", cuyo significado correcto es "idioma".

Un programa codificado en el lenguaje de un modelo de máquina (figura 1.4) no podrá ser ejecutado, en general, en otro distinto. Si queremos que un programa funcione en diferentes máquinas tendremos que preparar versiones particulares en el lenguaje de máquina de cada una de ellas. Evidentemente con ello se multiplica el costo de desarrollo si cada versión se prepara de manera independiente.

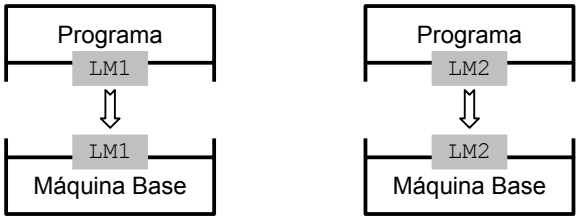

Figura 1.4 Cada computador necesita programas en su lenguaje de máquina (LM) particular.

Por otra parte, los programas en código de máquina son extraordinariamente difíciles de leer por una persona. Normalmente contienen códigos numéricos

(figura 1.5) sin ningún sentido nemotécnico, y compuestos por millones de operaciones elementales muy sencillas que en conjunto pueden realizar los tratamientos complejos que vemos a diario.

```
88 94 50 FF 76 0A FF 76 08 9A BA CD 3A 16 B8 01
00 EB E8 B8 88 94 50 2B C0 50 9A FA C5 3A 16 EB
ED B8 88 94 50 B8 01 00 EB EF B8 88 94 50 9A 48
D1 3A 16 EB D9 5D CA 0A 00 55 8B EC 83 EC 08 57
56 B8 01 00 50 9A 97 41 9B 34 8B D8 8B 47 14 89
```

Figura 1.5 Fragmento de programa en código de máquina.

Para facilitar la tarea de programación resulta muy deseable disponer de formas de representación de los programas que sean adecuadas para ser leídas o escritas por personas. En particular los *lenguajes de programación* sirven precisamente para representar programas de manera simbólica, en forma de un texto (figura 1.6) que puede ser leído con relativa facilidad por una persona. Además los lenguajes de programación son formas de representación prácticamente independientes de las máquinas particulares que se vayan a usar.

```
void PintarPlazas(const TipoPlazas P) {

  printf("\n\n");
  printf("      A    B    C       D    E    F\n\n");
  for (int i = 0; i < NumFilas; i++) {
    printf("%3d",i+1);
    for (int j = 0; j < AsientosFila; j++) {
      if ( j == Pasillo ) {
        printf("   ");
      }
      if (P[i].AsientosOcupa[j] == ocupado) {
        printf("  (*)");
      } else if (P[i].AsientosOcupa[j] == reservado) {
        printf("  (R)");
      } else if (P[i].AsientosOcupa[j] == vacio) {
        printf("  ( )");
      }
    }
    printf("\n");
  }
  printf("\n");
}
```

Figura 1.6 Fragmento de programa en lenguaje **C±**.

La comparación de los fragmentos de programa de las Figuras 1.5 y 1.6 pone de manifiesto sin necesidad de más explicaciones la ventaja de usar lenguajes de programación simbólicos.

1.4 Compiladores e Intérpretes

Un programa escrito en un lenguaje de programación simbólico puede ser ejecutado en máquinas muy diferentes. Pero para ello se necesita disponer de los mecanismos adecuados para transformar ese programa simbólico (figura 1.7) en un programa en el lenguaje particular de cada máquina.

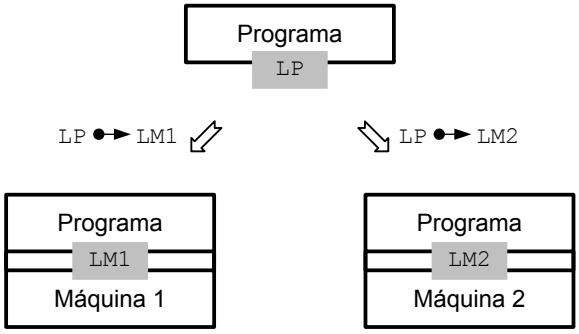

Figura 1.7 Un programa en un lenguaje de programación simbólico ha de adaptarse al lenguaje de cada máquina.

Existen diferentes estrategias para conseguir ejecutar en una máquina determinada un programa escrito en un lenguaje de programación simbólico. Normalmente se basan en el uso de programas especiales que realizan un tratamiento de la información en forma de texto que representa el programa en el lenguaje de programación simbólico. Estos programas para manipular representaciones de programas los denominaremos *procesadores de lenguajes*.

Un *compilador* es un programa que traduce programas de un lenguaje de programación simbólico a código de máquina. A la representación del programa en lenguaje simbólico se le llama *programa fuente*, y su representación en código de máquina se le llama *programa objeto*. Análogamente al lenguaje simbólico y al lenguaje máquina se les llama también *lenguaje fuente* y *lenguaje objeto*, respectivamente.

La ejecución del programa mediante compilador exige al menos dos etapas separadas, tal como se indica en la figura 1.8. En la primera de ellas se traduce el programa simbólico a código de máquina mediante el programa compilador.

En la segunda etapa se ejecuta ya directamente el programa en código de máquina, y se procesan los datos y resultados particulares. La compilación del programa ha de hacerse sólo una vez, quedando el programa en código de máquina disponible para ser utilizado de forma inmediata tantas veces como se desee.

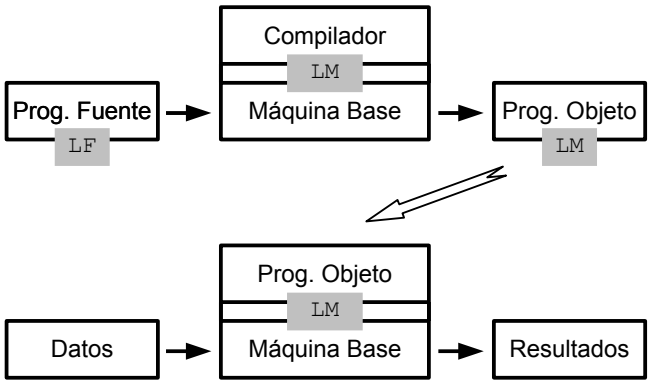

Figura 1.8 Proceso de un programa en lenguaje fuente (LF) mediante compilador.

Un *intérprete* es un programa que analiza directamente la descripción simbólica del programa fuente y realiza las operaciones oportunas. El intérprete debe contener dentro de él los fragmentos de código de máquina de todas las operaciones posibles que se puedan usar en el lenguaje de programación simbólico. Puede decirse que el intérprete es (o simula) una *máquina virtual* (figura 1.9) cuyo lenguaje de máquina coincide con el lenguaje fuente.

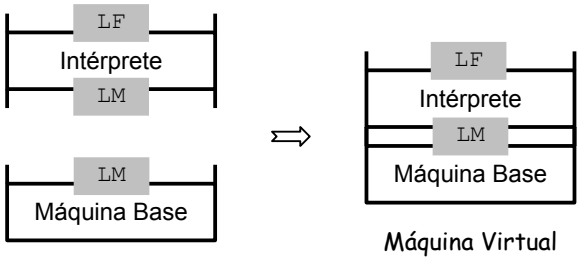

Figura 1.9 Un intérprete se comporta como una máquina virtual cuyo lenguaje es el lenguaje fuente.

El proceso de un programa mediante intérprete (figura 1.10) se limita a ejecutar directamente el programa en la máquina virtual, es decir, sobre el intérprete.

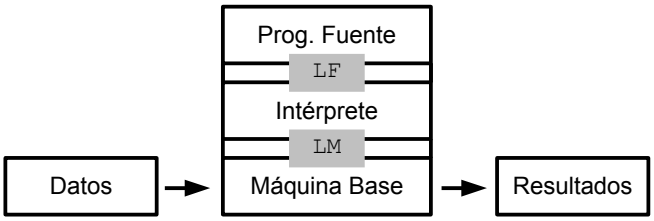

Figura 1.10 Proceso de un programa en lenguaje fuente (LF) mediante intérprete.

El proceso mediante intérprete es más sencillo, en conjunto, que mediante compilador, ya que no hay que realizar dos fases separadas. Su principal inconveniente es que la velocidad de ejecución es más lenta, ya que al tiempo que se van tratando los datos de la aplicación hay que ir haciendo el análisis e interpretación de las operaciones descritas en el programa fuente.

1.5 Modelos abstractos de cómputo

Los lenguajes de programación permiten describir programas o cómputos de manera formal, y por tanto simbólica y rigurosa. La descripción se hace, naturalmente, basándose en determinados elementos básicos y formas de combinación de estos elementos simples para construir programas tan complicados como sea necesario.

Existen muchísimos lenguajes de programación distintos que unas veces difieren en aspectos generales y otras simplemente en detalles. Si analizamos estos lenguajes podremos observar que muchos de ellos utilizan elementos básicos y formas de combinación similares, aunque representándolos con símbolos diferentes.

Si de un conjunto de lenguajes de programación basados en elementos computacionales similares extraemos los conceptos comunes, obtendremos un *modelo abstracto de cómputo*. Este modelo abstracto recoge los elementos básicos y formas de combinación de una manera abstracta, prescindiendo de la notación concreta usada en cada lenguaje de programación para representarlos.

Existen diversos modelos abstractos de cómputo, o modelos de programación, que subyacen en los lenguajes de programación actuales. Entre ellos están la *programación funcional*, *programación lógica*, *programación imperativa*, *modelo de flujo de datos*, *programación orientada a objetos*, etc. Todos estos modelos son modelos universales, en el sentido de que pueden utilizarse para describir cualquier cómputo intuitivamente posible.

Quizá el aspecto más interesante a destacar en este análisis de modelos abstractos de cómputo es que un programa que permita resolver un determinado problema puede adoptar formas muy diferentes, dependiendo del modelo de cómputo que se utilice para desarrollarlo. El modelo de programación imperativa es el más extendido, y eso induce a quienes empiezan a estudiar informática a identificar el concepto de programa con el de secuencia o lista de órdenes. Sin embargo, como veremos a continuación, existen otras formas igualmente válidas de representar un programa.

1.5.1 Modelo funcional

El modelo de *programación funcional* se basa casi exclusivamente en el empleo de funciones. El concepto de función se corresponde aquí de manera bastante precisa con el concepto de función en matemáticas. Una función es una aplicación, que hace corresponder un elemento de un conjunto de destino (resultado) a cada elemento de un conjunto de partida (argumento) para el que la función esté definida.

Por ejemplo, la operación de suma de números enteros es una función en que el conjunto de partida es el de las parejas de números enteros y el de destino es el conjunto de los números enteros. A cada pareja de enteros se le hace corresponder un entero, que es su suma.

De forma convencional, representaremos como $f(x)$ al resultado que se obtendrá al aplicar la función f al argumento x. Por ejemplo, podemos suponer definidas las funciones de suma, resta y producto de la forma:

Función	**Resultado**
Suma(a, b)	a + b
Diferencia(a, b)	a - b
Producto(a, b)	a × b

Para describir cómputos complejos, las funciones pueden combinarse unas con otras, de manera que el resultado obtenido en una función se use como argumento para otra. De esta manera un cómputo tal como

$$34 \times 5 + 8 \times 7$$

puede representarse de manera funcional de la forma

Suma(Producto(34, 5), Producto(8, 7))

Este es el aspecto que tiene un programa funcional, que será siempre en último extremo una aplicación de una función a unos argumentos, para obtener un resultado. El proceso de cómputo, llamado *reducción*, se basa en reemplazar

progresivamente cada función por el resultado de la misma. Este sistema de evaluación por sustitución es la base del llamado *cálculo-λ*. Aplicado al ejemplo se tendría:

Cómputo parcial	**Expresión / Resultado**
	Suma(Producto(34, 5), Producto(8, 7))
34×5	Suma(170, Producto(8, 7))
8×7	Suma(170, 56)
$170 + 56$	226

Las explicaciones anteriores se refieren a cómputos en los que sólo intervienen funciones primitivas, que son las que el computador o máquina abstracta que ejecuta el programa puede evaluar de forma directa. La programación funcional permite la definición por parte del programador de nuevas funciones a partir de las ya existentes. Utilizando de manera convencional el símbolo ::= para indicar definición, podremos crear una nueva función, *Cuadrado*, para obtener el cuadrado de un número basándose en el uso del producto de dos números.

$$\text{Cuadrado(x)} ::= \text{Producto(x, x)}$$

Cuando en un cómputo intervienen funciones definidas, la evaluación se sigue haciendo por sustitución. El proceso, llamado *reescritura*, consiste en reemplazar una función por su definición, sustituyendo los argumentos simbólicos en la definición por los argumentos reales en el cómputo. Por ejemplo, para evaluar $(5 + 3)^2$ tendremos:

Cómputo parcial	**Expresión / Resultado**
	Cuadrado(Suma(5, 3))
reducir Suma	Cuadrado(8)
reescribir Cuadrado	Producto(8, 8)
reducir Producto	64

1.5.2 Modelo de flujo de datos

En este modelo de cómputo, un programa corresponde a una *red de operadores* interconectados entre sí. Cada operador lo representaremos gráficamente mediante un cuadrado con *entradas* y *salidas*, y dentro de él el símbolo de la operación que realiza. Un operador espera hasta tener valores presentes en sus entradas, y entonces se activa él solo, consume los valores en las entradas, calcula el resultado, y lo envía a la salida. Después de esto vuelve a esperar que le lleguen nuevos valores por las entradas.

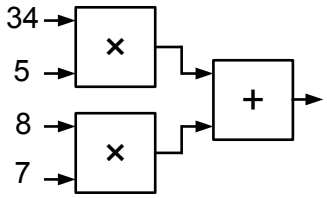

Figura 1.11 Red de flujo de datos.

Por ejemplo, la expresión $34 \times 5 + 8 \times 7$ puede ser calculada por la red de la figura 1.11.

El cómputo se realiza durante la evolución de la red, tal como se indica en la figura 1.12. Cuando la evolución termina, el resultado está presente en la salida de la derecha.

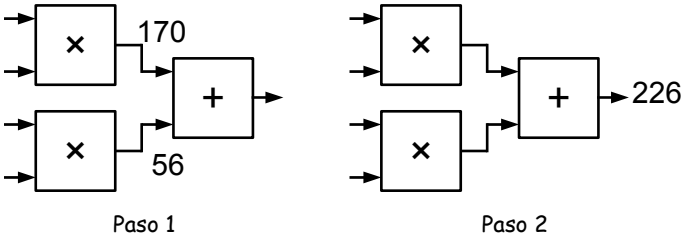

Figura 1.12 Evolución de la red de flujo de datos.

Una red de flujo de datos puede organizarse de manera que opere de forma iterativa, obteniendo no ya un resultado sino una serie de ellos. Por ejemplo, la red de la figura 1.13 produce la serie de números naturales, a base de reciclar sobre sí mismo un operador de incremento. Las líneas verticales representan operadores de duplicación o mezcla de valores.

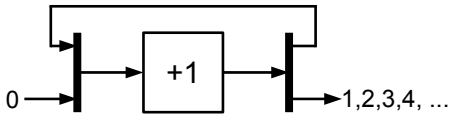

Figura 1.13 Generación de una serie de números.

Esta red no termina nunca de evolucionar. Añadiendo operadores especiales de bifurcación se puede conseguir que se detenga al llegar a un resultado adecuado.

1.5.3 Modelo de programación lógica

Este modelo abstracto de cómputo corresponde plenamente a lo que se denomina *programación declarativa*. Un programa consiste en plantear de manera formal un problema a base de declarar una serie de elementos conocidos, y luego preguntar por un resultado, dejando que sea la propia máquina la que decida cómo obtenerlo.

En *programación lógica* los elementos conocidos que pueden declararse son hechos y reglas. Un *hecho* es una relación entre objetos concretos. Una *regla* es una relación general entre objetos que cumplen ciertas propiedades. Una relación entre objetos la escribiremos poniendo el nombre de dicha relación y luego los objetos relacionados entre paréntesis. Por ejemplo:

Hijo(Juan, Luis)

significaría que Juan es hijo de Luis. Si tenemos el árbol genealógico como el de la figura 1.14

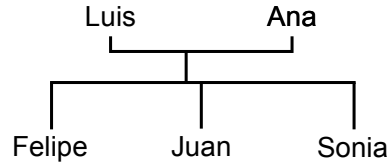

Figura 1.14 Árbol genealógico.

podremos declarar (prescindiendo del sexo) los hechos siguientes:

Hechos
Hijo(Felipe, Luis)
Hijo(Juan, Luis)
Hijo(Sonia, Luis)
Hijo(Felipe, Ana)
Hijo(Juan, Ana)
Hijo(Sonia, Ana)

Para realizar una consulta escribiremos el esquema de un hecho en que alguno de los elementos sea desconocido. Esto lo indicaremos usando nombres de incógnitas en minúscula, para distinguirlos de los nombres de elementos conocidos, que en este caso se habían escrito en mayúsculas (en lenguaje Prolog se usa el convenio contrario). La consulta será respondida indicando todos los valores posibles que puedan tomar las incógnitas. Por ejemplo:

Consulta	Respuesta
Hijo(x, Ana)	x = Felipe
	x = Juan
	x = Sonia

La verdadera potencia de la programación lógica aparece cuando declaramos reglas. Al realizar consultas basadas en reglas la máquina realiza automáticamente las inferencias (deducciones) necesarias para responderla. Por ejemplo, usando el símbolo ":−" para definir una regla, escribiremos:

Reglas
Padre(x, y) :− Hijo(y, x)
Hermano(x, y) :− Hijo(x, z), Hijo(y, z)

La primera regla se limita a nombrar la relación inversa de "hijo". La segunda expresa que dos personas son hermanas si son hijas de un mismo padre o madre. Ahora pueden realizarse consultas como las siguientes:

Consulta	Respuesta
Padre(x, Sonia)	x = Luis
	x = Ana
Hermano(x, Felipe)	x = Felipe
	x = Juan
	x = Sonia

La segunda consulta tiene una respuesta aparentemente errónea. Un análisis detallado nos permite comprobar que el error está en la formulación de la regla, ya que no se le ha informado a la máquina de que una persona no se considera hermana de sí misma. Modificando la regla se obtendrá la respuesta deseada.

Reglas
Hermano(x, y) :− Hijo(x, z), Hijo(y, z), ≠(x, y)

Consulta	Respuesta
Hermano(x, Felipe)	x = Juan
	x = Sonia

1.5.4 Modelo imperativo

El modelo de programación imperativa responde a la estructura interna habitual de un computador, que se denomina *arquitectura Von Neumann*. Un

programa en lenguaje de máquina aparece como una lista de *instrucciones* u órdenes elementales que han de ejecutarse una tras otra, en el orden en que aparecen en el programa. El nombre de *programación imperativa* deriva del hecho de que un programa aparece como una lista de órdenes a cumplir.

El orden de ejecución puede alterarse en caso necesario mediante el uso de instrucciones de control. Con ello se consigue ejecutar o no, o repetir, determinadas partes del programa dependiendo de ciertas condiciones en los datos.

Las instrucciones de un programa imperativo utilizan datos almacenados en la memoria del computador. Esta capacidad de almacenamiento de valores se representa en los programas imperativos mediante el uso de *variables*. Una variable no tiene aquí el mismo significado que en matemáticas, sino que representa un dato almacenado bajo un nombre dado. Una variable contiene un valor que puede ser usado o modificado tantas veces como se desee.

Un programa imperativo se plantea como el cálculo o modificación de sucesivos valores intermedios hasta obtener el resultado final. Las instrucciones típicas de un programa imperativo son las de asignación, que consisten en obtener un resultado parcial mediante un cálculo elemental que puede ser realizado por la máquina, y que se almacena en una variable para ser utilizado posteriormente.

En los lenguajes de programación simbólicos las instrucciones u órdenes se denominan *sentencias*. En **C±** y otros lenguajes similares la sentencia de asignación se representa de la forma

$$variable = expresión$$

La parte derecha de esta sentencia es una expresión aritmética que puede usar variables o valores constantes, así como operadores que estén definidos en el lenguaje, tales como los correspondientes a las operaciones aritméticas habituales: suma, resta, etc. Una sentencia de asignación representa una orden de calcular el resultado de la expresión y luego almacenar dicho resultado como nuevo valor de la variable.

Usando sólo expresiones simples y variables auxiliares, podremos expresar el cálculo de

$$34 \times 5 + 8 \times 7$$

mediante las sentencias siguientes

1. $a = 34 \times 5$
2. $b = 8 \times 7$
3. $c = a + b$

que obtendrán el resultado final en la variable *c*.

En realidad los lenguajes de programación permiten escribir directamente expresiones complejas. El cálculo anterior podría haberse hecho con una sola sentencia

1. c = 34 × 5 + 8 × 7

Para mostrar cómo las variables en programación imperativa pueden ir modificando su valor paso a paso hasta obtener el resultado deseado, analizaremos el siguiente fragmento de programa, que calcula el valor de 5! = 1×2×3×4×5. En este programa se ha supuesto que existen instrucciones REPETIR y HASTA que permiten programar la repetición controlada de una parte del programa.

1. f = 1
2. k = 1
3. REPETIR
4. k = k + 1
5. f = f × k
6. HASTA k == 5

Este programa obtiene el resultado final en la variable f. La variable k se utiliza para ir disponiendo de los sucesivos valores 2, 3, 4, etc. Las instrucciones **3.** y **6.** controlan la repetición de **4.** y **5.** La instrucción:

4. k = k + 1

tiene el significado siguiente: Se calcula el resultado de la expresión $k + 1$, usando el valor de k al iniciarse la ejecución de esa instrucción, y el valor obtenido se almacena de nuevo en k reemplazando el valor anterior. La primera vez que se ejecuta esa instrucción k tiene el valor 1, asignado inicialmente por la instrucción **2.** Al sumarle 1 se obtiene el valor 2, y la variable k pasa a tener ahora este nuevo valor. Cada vez que se vuelva a ejecutar la instrucción **4.** la variable k incrementará su valor en 1 unidad.

El análisis detallado del funcionamiento de un programa imperativo puede hacerse mediante una traza en la que se van anotando las sentencias o instrucciones de asignación que se van ejecutando sucesivamente, y los valores que toman las variables inicialmente y tras cada instrucción. En este ejemplo se tendrá:

Instrucción	k	f
	?	?
1.	?	1
2.	1	1
4.	2	1
5.	2	2
4.	3	2
5.	3	6
4.	4	6
5.	4	24
4.	5	24
5.	5	120

El programa comienza con valores no definidos (?) para las variables. Termina cuando k ha tomado el valor 5, en cuyo caso finalizan las repeticiones y f tiene el valor $120 = 5!$.

1.6 Elementos de la programación imperativa

La mayoría de los lenguajes de programación actualmente en uso siguen el modelo de programación imperativa. Por esta razón se ha optado en este primer nivel de enseñanza de programación por seguir dicho modelo. El lenguaje de programación **C±** se utilizará en este libro como herramienta para desarrollar las ideas generales en ejemplos prácticos realizables en máquina. **C±** es un subconjunto de los lenguajes C y C++ que se utilizan habitualmente en desarrollos reales. Además, **C±** presenta importantes ventajas para la enseñanza, por ser un lenguaje bien estructurado, permitir el uso de programación modular, permitir la implementación de tipos abstractos de datos, y ser un primer paso hacia el uso de lenguajes más evolucionados (y más complicados) como ocurre con los lenguajes C++, Ada o Java.

En el resto de este texto se irán desarrollando las ideas abstractas o generales de programación, junto con su realización en lenguaje **C±**. Todo ello en el marco de la programación imperativa, cuyos elementos abstractos se describen a continuación.

1.6.1 Procesador, entorno, acciones

Al introducir el modelo de programación imperativa se ha definido un programa, de manera intuitiva, como una lista de órdenes o instrucciones que han de ir siendo ejecutadas por la máquina en el orden preciso que se indique.

La idea abstracta correspondiente al concepto físico de máquina es el *procesador*. Definiremos como procesador a todo agente capaz de entender las órdenes del programa y ejecutarlas.

El procesador es esencialmente un elemento de control. Para ejecutar las instrucciones empleará los recursos necesarios, que formarán parte del sistema en el cual se ejecute el programa. Por ejemplo, se necesitarán dispositivos de almacenamiento para guardar datos que habrán de ser utilizados posteriormente, o dispositivos de entrada-salida que permitirán tomar datos de partida del exterior y presentar los resultados del programa. Todos estos elementos disponibles para ser utilizados por el procesador constituyen su *entorno*.

Las órdenes o instrucciones del programa definen determinadas *acciones* que deben ser realizadas por el procesador. Un programa imperativo aparece así como la descripción de una serie de acciones a realizar en un orden preciso. Las acciones son la idea abstracta equivalente a las instrucciones de un programa real.

1.6.2 Acciones primitivas. Acciones compuestas

Las acciones que son directamente realizables por el procesador se denominan *acciones primitivas*. Estas acciones suelen ser bastante sencillas, incluso en el caso de programas descritos en lenguajes de programación simbólicos. Entender un programa descrito enteramente a base de acciones primitivas suele ser muy difícil, por el nivel de detalle con el que hay que analizar cada una de sus partes, y todas ellas en conjunto.

El planteamiento razonable de un programa complejo debe pasar por usar la idea de *abstracción* para limitar la complejidad de detalles al estudiar el programa en su conjunto. Es muy útil usar la idea de *acción compuesta* como abstracción equivalente a un fragmento de programa más o menos largo que realiza una operación bien definida.

La descripción de un programa en términos de acciones compuestas puede facilitar su comprensión. Por supuesto, al desarrollar el programa habrá sido preciso describir o descomponer las acciones compuestas en otras más sencillas, hasta llegar finalmente a acciones primitivas, que son las que realmente podrá ejecutar el procesador. Las acciones compuestas son un elemento de descripción que facilita la comprensión del programa en su conjunto, o de partes importantes de él, pero no reduce el tamaño total del programa, que necesariamente deberá ser largo si se han de realizar operaciones complicadas.

1.6.3 Esquemas de acciones

Una acción compuesta consistirá, tal como se ha indicado, en la ejecución combinada de otras acciones más sencillas. La manera en la que varias acciones sencillas se combinan para realizar una acción complicada se denomina *esquema* de la acción compuesta.

Una buena metodología de programación exige usar esquemas sencillos y fáciles de entender a la hora de desarrollar acciones compuestas. A lo largo de este libro se irán introduciendo los principales esquemas de programación imperativa, junto con recomendaciones para su aplicación en el desarrollo de programas.

En particular, la llamada *programación estructurada* sugiere el uso de tres esquemas generales denominados *secuencia*, *selección* e *iteración*, con los cuales (junto con la definición de operaciones abstractas) se puede llegar a desarrollar de forma comprensible un programa tan complicado como sea necesario.

1.7 Evolución de la programación

Las ideas sobre cuál es la manera apropiada de desarrollar programas han ido evolucionando con el tiempo. Ha habido diversos motivos para ello, que pasaremos a analizar brevemente.

1.7.1 Evolución comparativa Hardware/Software

Los primeros computadores eran máquinas extraordinariamente costosas, y con una capacidad que hoy día consideraríamos ridículamente limitada. Sin embargo en su momento representaban el límite en la capacidad de tratamiento de información, y hacían posibles determinados trabajos de cálculo inabordables hasta entonces.

Como consecuencia de ello la finalidad principal de la programación era obtener el máximo rendimiento de los computadores. Los programas se escribían directamente en el lenguaje de la máquina, o a lo sumo en un lenguaje *ensamblador* en que cada instrucción de máquina se representaba simbólicamente mediante un código nemotécnico para facilitar la lectura del programa.

No existían ideas abstractas sobre el significado u objetivo preciso de un programa. Simplemente se consideraba que el mejor programa era el que realizaba el trabajo en menos tiempo y usando el mínimo de recursos de la máquina. La programación era, por tanto, una labor artesana, basada en la habilidad

personal del programador para conseguir que el programa cumpliera con esos objetivos de eficiencia, para lo cual era imprescindible conocer en detalle el funcionamiento interno del computador, y poder así emplear ciertos "trucos" de codificación que permitían ahorrar algunas instrucciones o usar menos elementos de memoria para almacenar datos intermedios.

El costo de desarrollo del software resultaba, en todo caso, muy inferior al costo del equipo material (*hardware*), por lo que pocas personas se paraban a considerar las posibilidades de reducir los costos de desarrollo de programas.

Por otra parte, y dada la limitación de capacidad de las máquinas, los programas eran necesariamente sencillos, en términos relativos, y se consideraba que podían llegar a depurarse de errores mediante ensayos o pruebas en número suficiente.

Los avances en la tecnología electrónica han ido suministrando computadores cada vez más capaces y baratos, en términos relativos. La necesidad de preparar sistemáticamente programas muy eficientes ha ido disminuyendo, y poco a poco se ha ido haciendo rentable utilizar programas no tan eficientes.

La mayor capacidad de los computadores ha permitido abordar aplicaciones cada vez más complejas. En los primeros computadores el software de una aplicación podía contener algunos cientos o quizá miles de instrucciones. En la actualidad las aplicaciones consideradas sencillas tienen decenas de miles de instrucciones, y en aplicaciones de gran envergadura el volumen del software desarrollado se cuenta por millones de instrucciones y en ellos trabajan centenares de programadores.

Con estos volúmenes de programa el costo del desarrollo del software supera ampliamente al costo de los equipos hardware utilizados. Ya no tiene sentido dedicar un gran esfuerzo a conseguir programas eficientes. Es más barato desarrollar programas relativamente más simples, aunque no aprovechen muy bien los recursos de la máquina, y comprar un computador de mayor potencia de proceso que compense esa posible falta de eficiencia. Los lenguajes de programación simbólicos han facilitado extraordinariamente las tareas de programación, al poder invocar en un programa operaciones cada vez más complejas como acciones primitivas. De esta manera se reduce el volumen total del programa medido en número de instrucciones, que ahora pasan a ser sentencias del lenguaje simbólico.

1.7.2 Necesidad de metodología y buenas prácticas

Los programas actuales son tan complicados que ya no es posible desarrollarlos de una manera artesanal. Es necesario aplicar técnicas de desarrollo muy

precisas para controlar el producto obtenido. Ya se ha indicado que estas técnicas aplicables a proyectos desarrollados en equipo constituyen la *ingeniería de software*.

A nivel individual hay que promover el empleo de una *metodología de programación* apropiada, que satisfaga los objetivos de corrección y claridad mencionados anteriormente. Para aplicaciones grandes la claridad se convierte en un objetivo prioritario, ya que resulta imposible analizar y modificar un programa si no se comprende suficientemente su funcionamiento.

Para facilitar la obtención de programas correctos, sin fallos, se pueden emplear técnicas formales, que permitan en lo posible garantizar la corrección del programa mediante demostraciones lógico-matemáticas, y no mediante ensayos en busca de posibles errores. La técnica de ensayos sólo resulta útil si consigue descubrir fallos, pues así demuestra que el programa contiene errores que hay que corregir, pero es más bien inútil si no se descubre ningún fallo, porque eso no garantiza que el programa no contenga errores, los cuales pueden manifestarse (y lamentablemente se manifiestan con harta frecuencia) cuando el programa ya está en explotación y los usuarios lo emplean en multitud de situaciones nuevas que no habían sido ensayadas nunca.

Lamentablemente las técnicas formales son tan complejas que sólo son utilizables en programas críticos de pequeño tamaño. Para grandes aplicaciones en las que intervienen cientos de ingenieros es absolutamente necesario emplear técnicas de ingeniería de software basadas en las *buenas prácticas*. Las buenas prácticas son un conjunto de normas, basadas en la experiencia de desarrollos anteriores, que se autoimponen todos los miembros de un equipo. El *Manual de Estilo* es el documento que compendia el conjunto de buenas prácticas que todos los miembros del equipo deben utilizar de una manera disciplinada durante el desarrollo de una aplicación compleja.