INTRODUCCIÓN A LA PROGRAMACIÓN. TEORÍA Y PRÁCTICA

Jesús Javier Rodríguez Sala

Título: Introducción a la programación. Teoría y práctica.

Autor: © Jesús Javier Rodríguez Sala
Laureano Santamaría Arana
Alejandro Rabasa Dolado
Oscar Martínez Bonastre

Profesores de la Universidad Miguel Hernández
Departamento de Estadística y Matemática Aplicada
Área de Lenguajes y Sistemas Informáticos.

I.S.B.N.: 84-8454-274-2
Depósito legal: A-905-2003

Edita: Editorial Club Universitario Telf.: 96 567 61 33
C/. Cottolengo, 25 - San Vicente (Alicante)
www.ecu.fm

Printed in Spain
Imprime: Imprenta Gamma Telf.: 965 67 19 87
C/. Cottolengo, 25 - San Vicente (Alicante)
www.gamma.fm
gamma@gamma.fm

Índice

Prólogo

"*!!Otro libro sobre programación¡¡*"; puede que éste sea el pensamiento de algún programador al topar con este libro que ahora presentamos, y es que, efectivamente, la literatura disponible a tal efecto es muy extensa, más aún si tenemos en cuenta la cantidad de páginas web que hay colgadas en Internet que comentan, con mayor o menor rigor, diferentes aspectos sobre programación en general o sobre algún lenguaje de programación en particular.

El objetivo que los autores de este libro perseguimos no es otro que el de ofrecer una obra que pueda ser utilizada por cualquier persona no iniciada que quiera aprender a programar. Obviamente, queda claro que esta obra no está pensada para programadores (que ya lo son), sino más bien para los que lo quieren ser; está especialmente dirigida a estudiantes de primeros cursos de cualquier ingeniería en la que la programación sea una de las materias importantes.

Como aspecto característico, este libro presenta claramente diferenciados los aspectos puramente teóricos de los prácticos. Observando el índice, se ve como la obra está dividida en dos grandes módulos, un primer módulo teórico, que abarca los temas del 1 al 9 y un segundo módulo práctico que comprende desde el tema 10 hasta el 19. Por último, podemos encontrara al final del libro una serie de apéndices que comentaremos más adelante.

Sobre el módulo teórico cabe decir que pretende sentar los conceptos teóricos básicos relacionados con la programación y los lenguajes de programación en general. Se ha pretendido deliberadamente que la exposición de dichos conceptos fuera independiente de cualquier lenguaje de programación. Es por ello que estos podrían ser aplicados a cualquier lenguaje de programación imperativo como C, Pascal, Cobol o Fortram (por citar algunos).

Ahora bien, en el aprendizaje de cualquier disciplina práctica, de poco serviría la mejor de las lecciones teóricas si después de ésta el alumno no tuviera la oportunidad de poner en práctica la teoría aprendida. Si se nos permite el símil, en este sentido, podríamos comparar el hecho de aprender a programar con el de aprender a montar en bicicleta. Del mismo modo que si los mejores ciclistas profesionales nos dieran unas lecciones teóricas sobre como montar en bicicleta, nadie podría decir tras asistir a dichas lecciones que sabe montar en bicicleta si no se sube a una y pone en práctica lo que ha aprendido. Con el aprendizaje de la programación ocurre lo mismo, con la teoría no basta. Para aprender a montar en bicicleta hay que subirse a una, pedalear y probablemente caerse en más de una ocasión. Para aprender a programar hay que sentarse ante un ordenador y ponerse a ello, y durante el proceso seguro que se incurrirá en algún error que otro, pero esto también forma parte del aprendizaje. Esta es la razón de ser del segundo módulo de este libro, pretende dar al lector la oportunidad de poner en práctica la teoría aprendida en el primer módulo.

Si estamos decididos a programar lo primero es optar por un lenguaje, pues bien, en este libro se ha optado por el C. En el segundo módulo se muestra como empezar a hacer nuestro primer programa en C y poco a poco, con suficientes ejemplos bien comentados, se va introduciendo la forma de aplicar o implementar los conceptos vistos en la parte teórica. Queremos hacer una advertencia al lector que se esté planteando aprende a programar con este libro; los ejemplos que aparecen en el segundo módulo deben ser probados, no basta con leer dicho módulo, si nos limitamos a esto estaremos leyendo un segundo módulo teórico.

Cada ejemplo debería ser escrito, compilado y ejecutado, y aún más, animamos al lector a que realice cuantas modificaciones se le ocurran y las vuelva a compilar y ejecutar, verificando después si obtiene el resultado deseado. Ésta es la única forma de aprender a programar, practicando. Y algo muy importante, queremos dejar claro que este módulo no pretende ser un manual de C. Para profundizar en al programación en este lenguaje sería recomendable que el lector adquiera un manual lo más completo posible sobre este leguaje.

Una posible crítica a este libro podría ser "*¿por qué en la parte práctica se ve el C habiendo lenguajes más modernos?*". Pues bien, las razones son varias, el C es un lenguaje que permite aplicar prácticamente todos los conceptos teóricos que se presentan en el libro; también puede considerarse como un lenguaje del que otros han copiado algunas características, por ejemplo: Java, JavaScript y PHP son lenguajes que tienen una sintaxis prácticamente idéntica a C, por lo que aprendiendo C se puede facilitar el posterior aprendizaje de otros lenguajes. Al hablar de '*lenguajes modernos*' es casi inevitable no pensar en el mencionado Java; sobre este lenguaje decir sólo una cosa; una persona sin nociones de programación que aprende Java, puede con el tiempo llegar a ser un buen programador en Java sin tener muy claros algunos conceptos como qué es un puntero o una dirección de memoria y, en general, todo lo relacionado con la gestión dinámica de memoria. En cambio es imposible ser un buen programador de C y pasar por alto estos conceptos. En opinión de quien escribe estas líneas, y como toda opinión es siempre discutible, los mencionados conceptos deberían ser, no sólo conocidos, sino dominados por cualquier programador serio. Con esto no queremos decir que no deba aprenderse Java; pensamos que el Java es un lenguaje muy importante hoy día, pero como dijimos antes, este libro va dirigido a quien no sabe programar y quiere aprender y para empezar pensamos que es más idóneo el C.

Un último argumento a favor de este lenguaje pese a su antigüedad (fue creado a principios de los 70) es que ni mucho menos es un lenguaje en desuso, en investigación o en aplicaciones donde el tiempo es un factor crítico se sigue aplicando. También existen entornos de desarrollo profesionales como '*Visual C++*' de la compañía *Microsoft* o '*C++ Builder*' de *Borland* que utilizan el lenguaje C y C++ (C++ es una extensión del lenguaje C que permite realizar programación orientada a objetos).

En los apéndices se han incluido una serie de datos o informaciones no estrictamente necesarias para el aprendizaje de la programación en C aunque sí pueden resultar de interés. En el apéndice A se muestra una guía de estilo sobre como programar en C, esta guía no son más que un pequeño conjunto de normas que la mayoría de los programadores siguen a la hora de programar para que el código escrito sea fácil de seguir e interpretar. Recomendamos encarecidamente a todo aquel que empiece a programar que siga esta guía (todos los ejemplos descritos en el libro lo hacen). En el apéndice B se muestra una forma empezar a programar (no es la única); de poco serviría insistir en la importancia de la práctica a la hora de aprender a programar si no se proporciona un medio para ponerse a ello, en este apéndice se muestran algunas direcciones de Internet de las que es posible obtener algún compilador de C de libre distribución y su funcionamiento básico. Los apéndices C y D muestran unas tablas, el primero con la correspondencia entre los códigos ASCII y sus caracteres asociados, y el segundo con los valores enteros del 0 (cero) al 255 y sus correspondientes conversiones a las bases hexadecimal, octal y binaria.

Por último, decir al lector no iniciado que tras leer este prólogo no haya entendido algunos términos utilizados (¿qué será eso de '*gestión dinámica de memoria*' o '*código*

ASCII'?) que no se preocupe en absoluto, a medida que avance en la lectura de esta obra verá como todo eso va quedando claro.

Los Autores

Tema 1

Conocimientos básicos

1.1 Conceptos iniciales

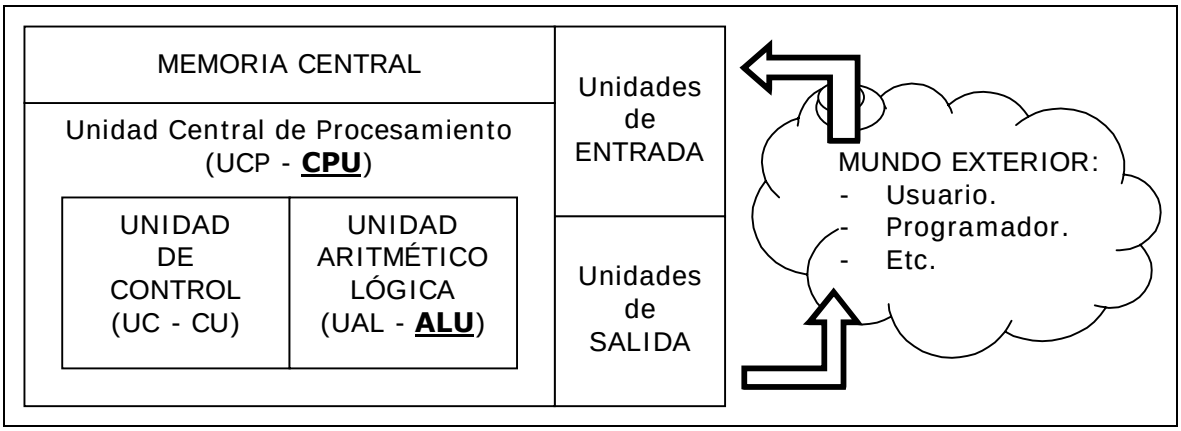
1.1.1 Conceptos de Informática y ordenador

Informática: Término formado por la contracción de las palabras '*información*' y '*automática*' que hace referencia al conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de los ordenadores (computadores, PC's, ...).

La informática es la ciencia que trata la adquisición, representación, tratamiento y transmisión de la información. Las tareas básicas para realizar dicho tratamiento son:

1. Entrada de información.
2. Tratamiento o proceso de la información.
3. Salida de resultados.

Ordenador (PC, computador, ...): Máquina programable que permite realizar el tratamiento automático de la información. Por tanto, un ordenador estará compuesto de lo siguientes elementos:



- Unidades de entrada y salida (E/S).- Dispositivos que permiten la comunicación entre el ordenador y el mundo exterior (usuarios, programadores, otros ordenadores, ...). Los primeros recogen datos de entrada para ser procesados y los segundos muestran los resultados de dicho proceso (Ejemplos: monitor, teclado, ratón, módem, ...).

- Memoria central.- También llamada *memoria principal* o *memoria interna*. Es donde se almacena la información proveniente del mundo exterior a la espera de ser procesada y donde se van guardando los resultados del procesamiento antes de ser mostrados.

- Unidad central de procesamiento (CPU – Central Proces Unit).- Es donde se procesa la información que hay guardada en memoria central. En la CPU destacan estos dos componentes:

  a) Unidad de control (CU – Control Unit).- Lleva el control de todas las operaciones que ocurren en el ordenador. Da ordenes al resto de dispositivos

para que la información se procese adecuadamente así como para que se realicen las operaciones de entrada y salida.

b) Unidad aritmético–lógica (ALU – Aritmetic-Logit Unit).- Realiza operaciones aritméticas y lógicas según se lo va ordenado la unidad de control.

1.1.2 Hardware y Software

Hardware (HW). Parte física: Conjunto de componentes físicos o dispositivos de un ordenador que se han descrito anteriorment:

- Unidades de entrada/salida (E/S)

- Memoria:
  a) Memoria central.
     - RAM (Random Acces Memory).- Memoria de acceso aleatorio. Volátil, se pierde al apagar el ordenador.
     - ROM (Read Only Memory).- Memoria de sólo lectura. Permanente, no se pierde al apagar el ordenador.
  b) Memoria auxiliar (secundaria o externa).- Diferentes tipos de dispositivos que permiten leer y/o escribir información (discos duros, disquetes, CD's, DVD's, cintas, ...).

- Unidad central de procesamiento.
  a) Unidad de control.
  b) Unidad aritmético–lógica.

Software (SW). Parte lógica: Conjunto de programas que pueden ser ejecutados en un ordenador. Podemos distinguir entre:

- Software de aplicación.- Contabilidad, facturación, diseño, etc.
- Software de sistemas.- Sistemas operativos, compiladores, etc.

1.1.3 Sistema Operativo

El sistema operativo (SO) es un conjunto de programas que facilitan y optimizan el empleo del ordenador. Actúa como intermediario entre el usuario y el hardware liberando a aquel de tareas de bajo nivel, en lo referente a operaciones entre las diferentes aplicaciones y la propia máquina. Además, si consideramos el ordenador como un conjunto de recursos necesarios para la ejecución de programas, el SO también se encarga de gestionar dichos recursos. Por ejemplo:

- Al imprimir un trabajo desde cualquier aplicación, será el SO quién se encargue de controlar el manejo de la impresora.

- Si se guarda un trabajo en disco, el SO se encargará de realizar la escritura física del mismo en el soporte que le indiquemos (disquete, disco duro, a través de una red, ...).

- Cuando un programa debe proporcionar un resultado en pantalla es el SO el encargado de controlar el monitor.

- El SO decide cuánta memoria del sistema puede utilizar un programa, que ficheros del disco puede utilizar o si debe detener su ejecución para que se ejecute otro programa.

1.2 Lenguajes de Programación

1.2.1 Concepto de lenguaje de programación

Lenguaje artificial que se utiliza para expresar programas de ordenador.

Cada ordenador, según su diseño, '*entiende*' un cierto conjunto de instrucciones elementales (lenguaje máquina). No obstante, para facilitar la tarea del programador, se dispone también de lenguajes de alto nivel más fáciles de manejar y que no dependen del diseño específico de cada ordenador. Los programas escritos en un lenguaje de alto nivel no podrán ser ejecutados por un ordenador mientras no sean traducidos al lenguaje propio de éste.

Para definir un lenguaje de programación es necesario especificar:

- Conjunto de símbolos y palabras clave utilizables.

- Reglas gramaticales para construir sentencias (instrucciones, ordenes) sintáctica y semánticamente correctas.
  a) Sintaxis: Conjunto de normas que determinan cómo escribir las sentencias del lenguaje.
  b) Semántica: Interpretación de las sentencias. Indica el significado de las mismas.

1.2.2 Paradigmas de programación

Un paradigma de programación es una colección de patrones conceptuales que moldean la forma de razonar sobre problemas, de formular soluciones y de estructurar programas. Los paradigmas de programación son:

- Programación imperativa
- Programación funcional
- Programación lógica
- Programación orientada a objetos

Programación imperativa:

En este paradigma, un *programa* es una secuencia finita de instrucciones, que se ejecutan una tras otra. Los datos utilizados se almacenan en memoria principal y se referencian utilizando *variable*s.

```
leer(x)
leer(y)
resultado = x + y
escribir(resultado)
```

Ejemplo de lenguajes que utilizan este paradigma: Pascal, Ada, Cobol, C, Modula-2 y Fortran.

Programación funcional:

Paradigma en el que todas las sentencias son funciones en el sentido matemático del término. Un *programa* es una función que se define por composición de funciones más simples.

La misión del ordenador será evaluar funciones.

```
predecesor(x)=x-1, si x>0
sucesor(x)=x+1
suma(x,0)=x
suma(x, y)=sucesor(suma(x, predecesor(y)))

?- suma(3,2)
```

Ejemplo de lenguaje: LISP.

Programación lógica:

En este paradigma un *programa* consiste en declarar una serie de hechos (elementos conocidos, relación de objetos concretos) y reglas (relación general entre objetos que cumplen unas propiedades) y luego preguntar por un resultado.

```
Mujer(Rosa)
Mujer(Marta)
Mujer(Laura)
Padres(Rosa, Carlos, Pilar)
Padres(Marta, Carlos, Pilar)
Padres(Laura, Carlos, Pilar)
Hermanas(X, Y):- mujer(X), mujer(Y), padres(X, P, M), padres(Y, P, M)

?- hermanas(Rosa, Marta)
?- hermanas(Rosa, X)
```

Ejemplo: Prolog.

Programación orientada a objetos (POO):

El paradigma orientado a objetos (OO) se refiere a un estilo de programación. Un lenguaje de programación orientado a objetos (LOO) puede ser tanto imperativo como funcional o lógico. Lo que caracteriza un LOO es la forma de manejar la información que está basada en tres conceptos:

- Clase.- Tipo de dato con unas determinadas propiedades y una determinada funcionalidad (ejemplo: clase '*persona*').

- Objeto.- Entidad de una determinada clase con un determinado estado (valores del conjunto de sus propiedades) capaz de interactuar con otros objetos (ejemplos: '*Pedro*', '*Sonia*', ...).

- Herencia.- Propiedad por la que es posible construir nuevas clases a partir de clases ya existentes (ejemplo: la clase '*persona*' podría construirse a partir de la clase '*ser vivo*').

Ejemplos de LOO: Smalltalk, C++, Java.

1.2.3 Desarrollo histórico de los lenguajes de programación

Lenguajes máquina (código máquina):

Es el lenguaje que comprende la máquina de forma directa. Internamente, el ordenador representa la información utilizando únicamente unos y ceros. Por tanto, un programa escrito en lenguaje máquina (o código máquina) estará formado por una secuencia finita de unos y ceros.

01011010 10101010 ...

Este lenguaje rara vez se emplea para programar ya que tiene muchos inconvenientes:

- Difícil de escribir y entender.
- Laboriosa modificación y corrección de errores.
- Depende del hardware (distintos ordenadores $\Rightarrow$ distintos lenguajes máquina).
- Repertorio reducido de instrucciones.

Lenguajes simbólicos:

Estos lenguajes utilizan símbolos para la construcción de sentencias de forma que son más fáciles de entender y corregir.

Leguaje de bajo nivel (ensamblador). Características:

- Las instrucciones se representan utilizando mnemotécnicos.

- Los datos se referencian por un nombre.

```
MOV 7,SP
ADD X
```

- Se mantiene una relación 1 a 1 respecto al lenguaje máquina (una instrucción en ensamblador representa una instrucción en código máquina).

- Sigue dependiendo de la máquina y por tanto el programador debe conocer el procesador utilizado.

Lenguajes de alto nivel. Características:

- Lenguajes más '*naturales*'. Estructura próxima a los lenguajes naturales.

- Repertorio de instrucciones amplio, potente y fácilmente utilizable.

```
a:=b+4
if a>b then b=0
```

- Independientes de la máquina.

- Programas legibles y más fáciles de entender.

- Mantenimiento y corrección de errores más sencilla.

Generaciones de los lenguajes:

1. Primera generación:

   - Lenguajes máquina y lenguaje ensamblador.
   - Dependen totalmente de la máquina.

2. Segunda generación (finales de los 50 y principios de los 60):

   - Fortran: Científico y de ingeniería.
   - Cobol: Aplicaciones de procesamiento de datos.
   - Algol: Predecesor de lenguajes de 3ª generación.
   - Basic: Originalmente para enseñar a programar.

3. Tercera generación (hacia los años 70 – crisis del software):

   - Lenguajes de programación estructurada.
   - Posibilidades procedimentales y de estructura de datos.

     a) De propósito general:
        - Pascal: Bloques estructurados, tipificación de datos.
        - C: Originalmente para sistemas, gran flexibilidad.
        - Ada: para aplicaciones de tiempo real.

     b) Orientados a Objetos:
        - Smalltalk.

- Eiffel.
- C++.
- Java.

c) Especializados (sintaxis diseñada para una aplicación particular):
- LISP: Demostración de teoremas.
- Prolog: inteligencia artificial.
- Apl: tratamiento de vectores y matrices.

4. Cuarta generación (finales de los años 80):

- Alto nivel de abstracción.
- No son necesarios detalles algorítmicos.
- Ejemplo: Sql (Structured Query Language) orientados a tratamiento de datos.

1.3 Traductores

1.3.1 Introducción

En el apartado 1.2.3 se ha visto que el lenguaje que entiende la máquina directamente es el código máquina. Cualquier programa escrito en un lenguaje diferente a éste debe ser traducido antes de que el ordenador pueda ejecutarlo.

Un traductor es un programa que toma como entrada un programa escrito en un lenguaje fuente y lo transforma en un programa escrito en lenguaje máquina.

El proceso de conversión se denomina *traducció*n, que puede realizarse de dos formas diferentes: por *interpretación* o por *compilación*.

1.3.2 Compiladores e intérpretes

Intérprete:

Es un programa que toma como entrada un programa escrito en lenguaje fuente y lo va traduciendo y ejecutando instrucción por instrucción (de una en una).

Compilador:

Es un programa que toma como entrada un programa fuente y genera un programa equivalente llamado programa objeto o código objeto.

Interpretación vs Compilación:

En el desarrollo de software, el programador deberá determinar qué tipo de herramienta utilizará para realizar esta tarea; un interprete o un compilador. Normalmente se empleará un compilador por las siguientes razones:

- La fase de traducción utilizando un compilador sólo se realiza una vez (la definitiva). Con un interprete hay que traducir cada vez que se ejecuta el programa, lo que hace que dicha ejecución sea más lenta.

- El código generado por un compilador puede optimizarse, siendo así más eficiente.

- Diferentes módulos de un programa se pueden compilar por separado y después ser enlazados (linkados) para generar el programa ejecutable final. Si se modifica un módulo, para compilar el programa completo bastará con traducir este módulo y volver a linkarlo con el resto; no es necesario volver a traducir todos los módulos, por lo que también se ahorra tiempo de compilación.

1.3.3 El proceso de traducción

Se divide en dos grandes fases: análisis y síntesis.

Fase de análisis:

Consiste en ver si el código del programa fuente está escrito de acuerdo a las reglas sintácticas y semánticas que define el lenguaje fuente. Se realizan tres tipos de análisis:

a) Análisis léxico
   - Elimina del programa fuente toda la información innecesaria (espacios y líneas en blanco, comentarios, etc.).
   - Comprueba que los símbolos del lenguaje (palabras clave, operadores, ...) se han escrito correctamente.

b) Análisis sintáctico
   - Comprueba si lo obtenido de la fase anterior es sintácticamente correcto (obedece a la gramática del lenguaje).

c) Análisis semántico
   - Comprueba el significado de las sentencias del programa.

Fase de síntesis:

Consiste en generar el código objeto equivalente al programa fuente. Sólo se genera código objeto cuando el programa fuente está libre de errores de análisis, lo cual no quiere decir que el programa se ejecute correctamente, ya que un programa puede tener errores de concepto o expresiones mal calculadas.

Tema 2

Algoritmos y programas

2.1 Introducción

Los términos '*algoritmo*' y '*programa*' se emplean para referir dos conceptos similares cuya diferencia se debe aclarar:

Algoritmo:

Conjunto de instrucciones que especifican la secuencia ordenada de operaciones a realizar para resolver un problema. En otras palabras, un algoritmo es un método o fórmula para la resolución de un problema. Un algoritmo es independiente tanto del lenguaje de programación en que se exprese como del ordenador en el que se ejecute.

Ejemplo: Una receta de cocina (especificada de forma precisa y sin decisiones subjetivas) puede ser considerada un algoritmo: indica los pasos a realizar para resolver el problema (cocinar un plato) y es independiente tanto del idioma en que se escriba como del cocinero que la ejecute.

Las principales características que debe tener un algoritmo son:

- Debe ser comprensible y preciso (sin ambigüedades), e indicar el orden de realización de cada paso.
- Debe ser predecible. Si se aplica partiendo de la misma situación inicial, se debe obtener siempre el mismo resultado.
- Debe ser finito. El algoritmo debe terminar en algún momento (debe tener un número finito de pasos).

Curiosidad: El término algoritmo es muy anterior a la era informática: proviene de *Mohammed al-Khowârizmî* (apellido que se tradujo al latín empleando la palabra '*algoritmus*'), matemático persa del siglo IX que enunció paso a paso las reglas para sumar, restar, multiplicar y dividir números decimales.

Programa:

Secuencia de operaciones especificadas en un determinado lenguaje de programación, cada una de las cuales determina las operaciones que debe realizar el ordenador para la resolución de un problema. Se trata pues de una implementación concreta (en un tipo de ordenador concreto y con un lenguaje de programación concreto) de un algoritmo diseñado con anterioridad.

Ejemplo: Si en una receta de cocina (algoritmo) se especifica en uno de sus pasos que hay que batir un huevo, este se batirá de forma diferente según los medios de que se disponga (lenguaje de programación y/o ordenador), bien empleando un tenedor, o bien con una batidora.

2.2 Resolución de problemas

La resolución de un problema mediante un ordenador consiste en el proceso que a partir de la descripción de un problema, expresado habitualmente en lenguaje natural y en

términos propios del dominio del problema, permite desarrollar un programa que resuelva dicho problema.

Este proceso exige los siguientes pasos:

- Análisis del problema.
- Diseño o desarrollo de un algoritmo.
- Transformación del algoritmo en un programa (codificación).
- Ejecución y validación del programa.

Este apartado se centrará en los dos primeros pasos, que son los más difíciles del proceso. Una vez analizado el problema y obtenido un algoritmo que lo resuelva, su transformación a un programa de ordenador es una tarea de mera traducción al lenguaje de programación deseado.

2.2.1 Análisis del problema

Cuando un usuario plantea un *programador* un problema a resolver mediante su ordenador, por lo general ese usuario tendrá conocimientos más o menos amplios sobre el dominio del problema, pero no es habitual que tenga conocimientos de informática. Por ejemplo, un contable que necesita un programa para llevar la contabilidad de una empresa será un experto en contabilidad (dominio del problema), pero no tiene porque ser experto en programación.

Del mismo modo, el informático que va a resolver un determinado problema puede ser un experto programador, pero en principio no tiene porque conocer el dominio del problema; siguiendo el ejemplo anterior, el informático que hace un programa no tiene porque ser un experto en contabilidad.

Por ello, al abordar un problema que se quiere resolver mediante un ordenador, el programador necesita de la experiencia del experto del dominio para entender el problema. Al final, si se quiere llegar a una solución satisfactoria es necesario que:

- El problema esté bien definido si se quiere llegar a una solución satisfactoria.
- Las especificaciones de las entradas y salidas del problema, deben ser descritas con detalle:
  a) ¿Qué datos son necesarios para resolver el problema?
  b) ¿Qué información debe proporcionar la resolución del problema?

2.2.2 Diseño del algoritmo

Como ya se ha descrito, un algoritmo consiste en una especificación clara y concisa de los pasos necesarios para resolver un determinado problema, pero para poder diseñar algoritmos es necesario disponer de una notación, que llamaremos '*notación algorítmica*', que permita:

- Describir las operaciones puestas en juego (acciones).
- Describir los objetos manipulados por el algoritmo (datos/informaciones).
- Controlar la realización de las acciones descritas, indicando la forma en que estas se organizan en el tiempo.

Nota: la notación algorítmica no es en ningún caso un lenguaje de programación. Lo esencial de la notación algorítmica es que las acciones elegidas sean las necesarias y suficientes para expresar todo algoritmo.

Para poder describir cualquier tipo de acción de las que intervienen en un algoritmo, diversos autores propusieron del uso de un conjunto de construcciones lógicas (secuencia, decisión e iteración) con las que es posible escribir cualquier programa. Lo que sigue a continuación es la descripción de las diferentes construcciones disponibles para el diseño de algoritmos.

1. Acción elemental

Se entiende por *acciones elementales* aquellas que el ordenador es capaz de realizar y que serán de dos tipos:

- Aritmético – lógicas: Operaciones que, a partir de unos determinados datos, realizan un cálculo aritmético (suma, resta, multiplicación, ...) o un cálculo lógico (mayor que, menor que, igual que, ...).Las primeras devuelven un valor numérico (4, -5.67, ...) y las segundas un valor lógico (verdadero o falso).
- De entrada – salida: Acciones que permiten capturar datos para su posterior tratamiento (las de entrada) y guardar los resultados de dicho tratamiento (las de salida).

2. Secuencia de acciones elementales (composición secuencial)

Cuando en un algoritmo se deben ejecutar varias acciones sucesivamente, éstas se describen una detrás de otra según el orden en que deban ejecutarse. Si se desea se puede emplear algún tipo de símbolo para separar dos acciones consecutivas. En el siguiente ejemplo se nuestra la descripción de *n* acciones separadas por *punto y coma* (símbolo que habitualmente se emplea como separador).

```
Acción 1;
Acción 2;
...
Acción n;
```

3. Ejecución condicional de una acción (composición condicional)

Cuando en un algoritmo se quiere indicar que cierta acción sólo se debe ejecutar bajo cierta condición se indica del siguiente modo:

```
Si Condición Entonces
        Acción;
FinSi
```

Sólo si la '*Condición*' (operación lógica) es verdadera se ejecutará la '*Acción*'. En este caso, la '*Acción*' puede referirse tanto a una acción elemental como a un conjunto de ellas.

4. Ejecución alternativa de una de dos acciones (composición alternativa)

En ocasiones, se deben ejecutar unas acciones u otras dependiendo de la ocurrencia de una determinada condición. Esta especificación se realiza del siguiente modo:

```
Si Condición Entonces
        Acción A;
SiNo
        Acción B;
FinSi
```

Dependiendo de si la '*Condición*' es verdadera o falsa se ejecutará la '*Acción A*' o la '*Acción B*' respectivamente. De forma análoga a como ocurría en el caso anterior, tanto la '*Acción A*' como la '*Acción B*' pueden referirse a una acción elemental o a un conjunto de ellas.

5. Ejecución condicional de una de varias opciones (composición selectiva)

También es posible que a la hora de especificar la ejecución de una acción haya que escoger ésta entre varias dependiendo del valor de una determinada variable (o indicador). Este caso se expresa del siguiente modo:

```
Seleccionar Indicador
        Caso Valor 1:
                Acción 1;
        Caso Valor 2:
                Acción 2;
        ...
        Caso Valor n:
                Acción n;
        [EnOtroCaso:
                Acción X;]
FinCaso
```

En esta construcción '*Indicador*' debe tener un determinado valor que en caso de coincidir con alguno de los *n* valores provocará la ejecución de la acción asociada a dicho valor. Si el valor del '*Indicador*' no coincidiera con ninguno de los especificados se ejecutará la '*Acción X*'. No tiene porque haber una '*Acción X*' para cuando el '*Indicador*' no coincida con ninguno de los *n* valores; en ese caso, si el '*Indicador*' no coincide con ningún valor no se ejecutaría ninguna acción.

Al igual que en los casos anteriores, todas las acciones que aparecen en esta estructura ('*Acción 1*', '*Acción 2*', ..., '*Acción n*' y '*Acción X*') pueden referirse a una única acción o a un conjunto de ellas.

6. Ejecución múltiple de una acción (composición iterativa o bucle)