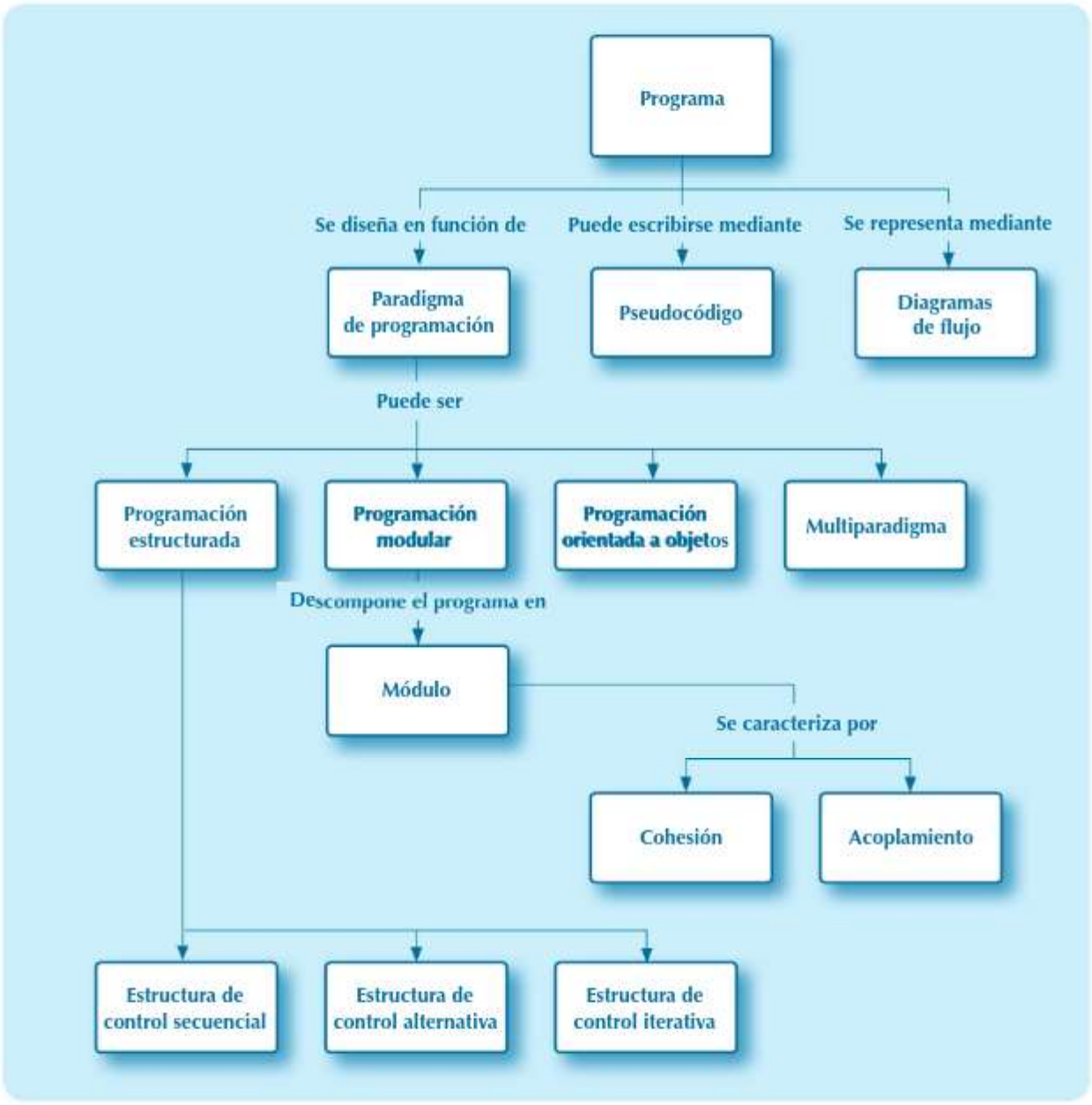
1

Introducción a la programación

Objetivos

- ✓ Conocer los conceptos básicos relacionados con la programación y el diseño de aplicaciones.
- ✓ Describir los paradigmas de programación más utilizados.
- ✓ Aprender a utilizar sistemas de descripción de programas de alto nivel.

Mapa conceptual



Glosario

Acoplamiento. Grado y forma de dependencia entre los módulos que comparten información.

ANSI. Instituto Nacional Americano de Estándares. Es la organización encargada del desarrollo de estándares para productos, servicios y procesos.

Argumento. Valor transferido a un parámetro correspondiente a un procedimiento o función cuando este es invocado.

Aritmómetro. Evolución de la antigua calculadora mecánica de Leibniz, capaz de realizar las cuatro operaciones aritméticas elementales (suma, resta, multiplicación y división) con resultados de hasta 12 cifras.

Booleano. Es el tipo que puede tomar una variable o condición. Se limita a 2 valores: verdadero o falso.

Código espagueti. Código de un programa cuya estructura es tan compleja que hace muy difícil su seguimiento y modificación.

Cohesión. Nivel de relación existente entre los elementos software contenidos en un módulo.

Diagrama. Representación gráfica de un conjunto de elementos y las relaciones existentes entre sí.

Máquina analítica. Computador moderno de naturaleza mecánica ideado por Charles Babbage, que dispone de la mayor parte de los elementos lógicos de un ordenador tal como hoy en día se conoce.

Paradigma. Guía de trabajo a seguir para la obtención de un resultado, elaborada y consensuada por un grupo de expertos en la materia.

Parámetro. Propiedad de un procedimiento o función que se define junto a él y que establece la forma y el tipo del valor que espera recibir cuando es invocado.

Programación. Proceso de diseño y codificación de las instrucciones que se ejecutan sobre un computador.

Pseudocódigo. Lenguaje cercano a un lenguaje de programación cuyo objetivo es el desarrollo de algoritmos fácilmente interpretables por un programador.

Teorema. Término proveniente del latín *theorēma* que referencia a una verdad que, sin ser obvia, es totalmente demostrable.

1.1. Introducción

En este primer capítulo se realizará una breve introducción a la programación desde sus orígenes hasta los tiempos modernos, se presentarán conceptos generales de programación y se abordarán los diferentes paradigmas de programación, explicando en detalle la programación estructurada y la programación modular. Por último, se expondrán los diagramas de flujo y el pseudocódigo como técnicas que ayudarán a introducirse en el mundo de la programación.

1.2. Orígenes de la programación

Se hace uso del término *programación* para hacer referencia al proceso mediante el cual se diseña y codifica un conjunto de instrucciones que implementan un determinado algoritmo.

Surgió ante la necesidad de automatizar y facilitar las tareas que inicialmente se realizaban de forma manual. La programación tiene sus raíces en las primeras calculadoras de operaciones matemáticas elementales, las cuales llevaban los algoritmos implementados en su estructura física, y en los telares automáticos, que recibían los datos sobre el trabajo a realizar a través de tarjetas perforadas. Sin embargo, no es hasta la aparición del diseño de la *máquina analítica* de Charles Babbage (considerado el primer ordenador de la historia) y la posterior aportación de Augusta Ada Byron (considerada la primera programadora de la historia) cuando puede hablarse de programación como tal.

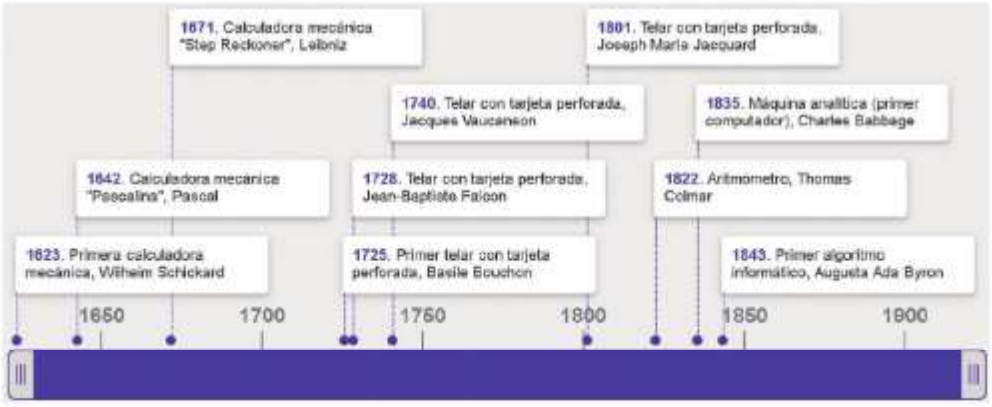


Figura 1.1
Orígenes de la programación.

Hoy en día el concepto de *programación* es sustancialmente diferente, y ello se debe a que ha ido evolucionando de forma pareja a como lo han ido haciendo los dispositivos y tecnologías de cada época (figura 1.1). En las primeras computadoras las instrucciones de los programas eran escritas en código binario (ceros y unos). Dada la complicación asociada a esta técnica de programación, los científicos que estudiaban esta nueva ciencia decidieron sustituir determinadas secuencias binarias por palabras que permitieran representarlas más fácilmente; este proceso dio lugar al conocido *lenguaje ensamblador*. Conforme las necesidades fueron creciendo y se fue depurando la técnica aparecieron los lenguajes de alto nivel, que añadían una nueva capa de abstracción al proceso de programación y que permitían el desarrollo de programas cada vez más sofisticados al mismo tiempo que más fáciles de desarrollar.

1.3. Paradigmas de programación

A lo largo de la historia de la computación se han empleado diversas técnicas de programación, presentando cada una de ellas unas características concretas que las hace diferenciarse del resto y mostrarse como candidatas ideales para según qué problema. Cuando un conjunto de reglas y costumbres son aceptadas como modelo de referencia para generar un código de calidad, se puede comenzar a hablar de un *paradigma de programación*.

Pero, a pesar de que cada paradigma de programación tiene unas características concretas, en la vida real los programadores pueden basarse en varios de ellos al mismo tiempo para el desarrollo de sus programas. La inclusión de dos o más paradigmas de programación en el mismo programa se denomina *programación multiparadigma* y generalmente es la que mejores resultados obtiene.

Utilizar un paradigma de programación, aceptando y siguiendo sus reglas, derivará en un código fuente fácil de mantener, entender y corregir.

Son muchos los paradigmas de programación que se pueden enumerar, aunque a continuación serán descritos con detalle aquellos más empleados y que mayor relación guardan con los contenidos del curso. En este capítulo se estudiará en detalle el *paradigma de programación estructurada* y el *paradigma de programación modular*. El *paradigma de programación orientada a objetos* será estudiado en el capítulo 2.

Actividad propuesta 1.1

Busca ejemplos de otros paradigmas de programación diferentes a los mencionados anteriormente y establece una línea de tiempo que muestre el momento en el que fueron surgiendo.

1.3.1. Programación estructurada

La *programación estructurada* tiene sus orígenes en la década de los sesenta y pretende mejorar la calidad del software, así como agilizar su desarrollo a la vez que reduce su coste. Deriva del *paradigma de programación imperativa*, en el cual se especifica cómo debe resolverse el problema mediante un conjunto de instrucciones que son ejecutadas y que van variando el estado del programa.

La programación estructurada se basa en el *teorema del programa estructurado* propuesto por Corrado Böhm y Giuseppe Jacopini, según el cual todo programa se puede escribir empleando únicamente tres tipos básicos de estructuras de control:

1. *Secuencial*: las instrucciones se ejecutan una detrás de otra siguiendo un orden (secuencialidad).
2. *Alternativa:* las expresiones son evaluadas y, dependiendo del resultado, se decide cuál será la siguiente instrucción a ejecutar.
3. *Iterativa:* se repetirá un conjunto de instrucciones hasta que una condición sea cierta, permitiéndose de esta manera el salto a otra instrucción.

Aunque los lenguajes de programación ofrecen un abanico más amplio de estructuras de control, bien es cierto que todas ellas finalmente se basan en las anteriormente descritas.

INVESTIGA

Edsger Dijkstra escribió "La sentencia *Go To* considerada dañina". Busca información acerca de ese escrito y extrae las ideas principales.

Los lenguajes de programación también ofrecen instrucciones de *salto incondicional* (*go to*), pero su uso está totalmente desaconsejado porque complica el seguimiento y trazabilidad de los programas, generando un *código espagueti* más difícil de modificar y corregir. Es por ello por lo que este tipo de estructuras deben evitarse en la medida de lo posible.

Por último, hay que reseñar que la programación estructurada presenta grandes dificultades a la hora de abordar proyectos de gran tamaño, para los cuales es más eficiente el empleo de la programación orientada a objetos.

RECUERDA

✓ En la programación estructurada los datos y las funciones que los manejan se definen separados.

La programación estructurada no ofrece garantías para abordar proyectos de gran tamaño, en cuyo caso es más recomendable emplear programación orientada a objetos.

1.3.2. Programación modular

La programación estructurada establece unas directrices base para realizar un código de calidad y facilitar el mantenimiento. Aun así, no tiene en cuenta que el código puede crecer hasta llegar a complicar mucho el desarrollo.

La *programación modular* propone como solución la descomposición del problema en subproblemas de menor tamaño, los cuales serán descompuestos sucesivamente en otros de menor tamaño hasta que el resultado de dichas descomposiciones permita obtener problemas fáciles de resolver. A esta técnica de resolución de problemas se le denomina *divide y vencerás* y permite poner en práctica un proceso de desarrollo software *top-down*.

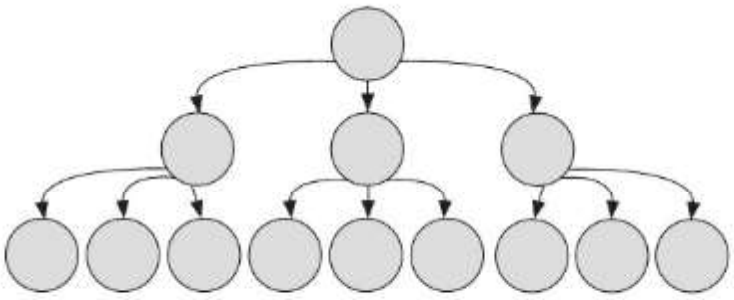


Figura 1.2
Top-down.

Cada bloque de código que se ocupará de resolver un problema de menor tamaño se denomina *módulo* o *subprograma*, y estará compuesto por un conjunto de sentencias físicamente unidas y delimitadas. Dicho módulo será referenciado por un nombre y podrá hacer uso de otros módulos. La comunicación entre módulos se realizará a través de interfaces de comunicación claramente definidas.

Los elementos empleados para llevar a cabo la modularización pueden ser muy distintos dependiendo del nivel de abstracción:

1. A un alto nivel serán las *unidades*, en diseños estructurados, y los *paquetes* y *librerías*, en diseños orientados a objetos.
2. A más bajo nivel serán los *procedimientos* y las *funciones*, en diseños estructurados, y las *clases* con sus correspondientes *métodos*, en diseños orientados a objetos.

Para llevar a cabo una correcta modularización del programa, es necesario realizar un diseño de calidad que tenga en cuenta los siguientes criterios:

- El módulo debe tener un único punto de entrada y un único punto de salida.
- Cada módulo debe realizar una única función bien definida.
- Cada módulo debe comportarse como una caja negra, de manera que dependa únicamente de las entradas.
- Módulos trazables en una pantalla (tamaño ideal 30-50 líneas de código por módulo).
- Máxima *cohesión* y mínimo *acoplamiento*. Ambos criterios están relacionados entre sí de manera que, a mayor cohesión en los módulos, menor será el acoplamiento entre ellos.

A) Cohesión

Se denomina *cohesión* de un módulo al nivel de relación existente entre los elementos software contenidos en él, entendiéndose como elementos software las instrucciones, definiciones de datos o las llamadas a otros módulos.

Existen diferentes tipos de cohesión:

1. *Cohesión funcional:* los elementos del módulo contribuyen a la realización de una única tarea. Dicho de otra manera, cada elemento es una parte integral de la estructura del módulo. El módulo se representa mediante un identificador simple. Por ejemplo: módulos matemáticos *suma, seno, potencia…*
2. *Cohesión secuencial:* se da cuando el módulo realiza varias tareas según una secuencia que establece que la salida de una actividad es la entrada necesaria de la siguiente. Por ejemplo:

Progresivamente el módulo se ocupará de buscar información a partir de un dato de entrada y, con ella, generar un informe formateado que devolverá como dato de salida. Podrían encadenarse más acciones.

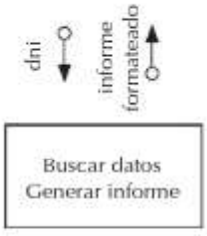


Figura 1.3
Cohesión secuencial.

3. *Cohesión comunicacional:* contiene actividades paralelas (sin orden) que comparten los mismos datos (de entrada o salida). Se recomienda su descomposición en módulos independientes de cohesión funcional. Por ejemplo:

El módulo se ocupará de buscar información a partir de un dato de entrada y retornará como salida información de diverso tipo. El orden de las actividades no es relevante.

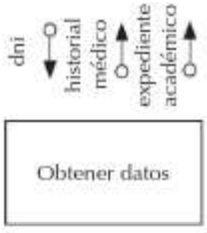


Figura 1.4
Cohesión comunicacional.

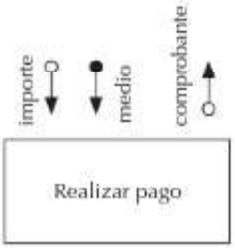
4. *Cohesión procedural:* los elementos realizan diferentes actividades posiblemente no relacionadas entre sí. También es posible que no exista relación alguna entre los datos de entrada y de la salida de los módulos. El control fluye de una actividad a la siguiente. Por ejemplo:

El módulo eliminará del sistema los datos relativos al libro cuyo código recibe como dato de entrada; posteriormente retornará como datos de salida un listado de los socios de la biblioteca.

Figura 1.5
Cohesión procedural.

5. *Cohesión temporal:* los elementos están implicados en la realización de actividades relacionadas por el momento en el cual se llevan a cabo. Ejemplos típicos son los módulos de inicialización y finalización de sistema (arrancar sistema, apagar sistema, resetear sistema, inicializar sistema…). Es posible que no manejen ningún dato de entrada o salida.

6. *Cohesión lógica:* los elementos están destinados a que realicen actividades de una misma categoría general, pero la selección de la actividad concreta tiene lugar desde fuera del módulo. Por ejemplo:



El módulo se encargará de efectuar un pago que podrá realizarse en efectivo o por transferencia (el medio de pago se seleccionará desde fuera del módulo).

Figura 1.6
Cohesión lógica.

7. *Cohesión casual:* los elementos no guardan ninguna relación observable y son fruto de una organización caótica. Por ejemplo:

Dentro del módulo se realizarán tareas de diverso tipo pero sin relación alguna. La selección de la tarea en cuestión se realiza mediante un *flag* creado para tal efecto. Algunas tareas pueden devolver datos mientras que otras no.

Figura 1.7
Cohesión casual.

Aunque siempre se tiende a pensar que, a mayor cohesión, mejor será el diseño de un programa, hay que tener en cuenta que no todos los tipos de cohesión son deseables. En la siguiente escala se pueden apreciar los diferentes tipos de cohesión y cómo estos afectan al mantenimiento.

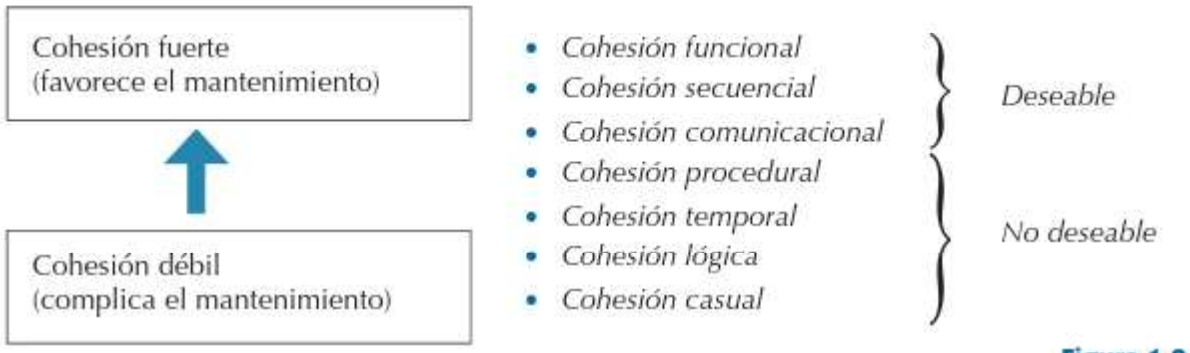


Figura 1.8
Escala de cohesión por Stevens, Myers, Constantine y Yourdon.

Mediante el siguiente árbol de cohesión se podrá identificar el tipo de cohesión ante el que un programador puede encontrarse:

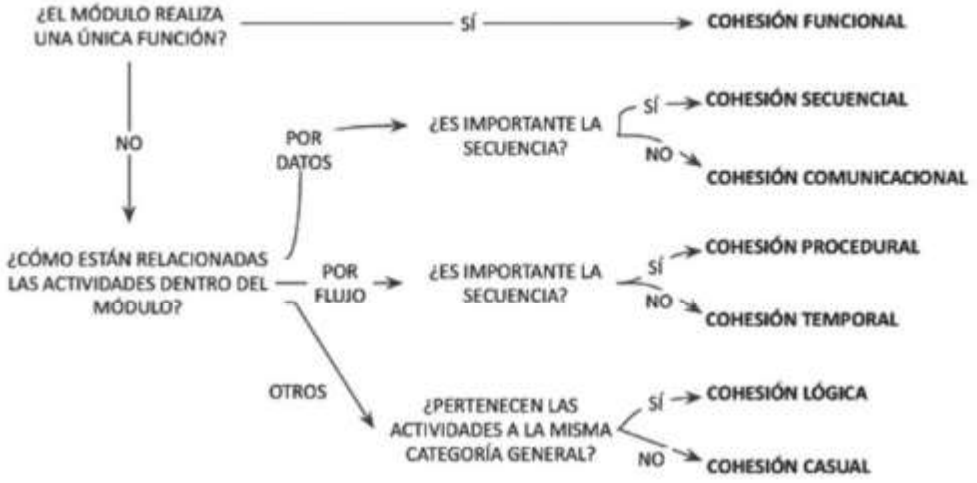


Figura 1.9
Árbol de cohesión.

Actividad propuesta 1.2

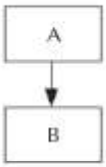
Propón dos ejemplos de cada uno de los tipos de cohesión vistos anteriormente.

B) Acoplamiento

El término *acoplamiento* hace referencia al grado y forma de dependencia entre los módulos. A menor cantidad de información compartida entre diferentes módulos, menor acoplamiento y por tanto mejor será el diseño.

Existen varios tipos de acoplamiento que son descritos a continuación:

1. *Acoplamiento normal:* se da en aquellos casos en los que un módulo *A* invoca a otro módulo *B*. Por ejemplo:



El módulo *A* invoca al módulo *B*, el cual, tras realizar su función, retorna el control al módulo *A*.

Figura 1.10
Acoplamiento normal.

De intercambiarse información, esta únicamente estará presente en los parámetros de llamada. En ese caso, y atendiendo al tipo de la información, se definen tres subtipos:

a) Acoplamiento normal por datos: si los parámetros intercambiados son datos elementales (tipos básicos). Por ejemplo:

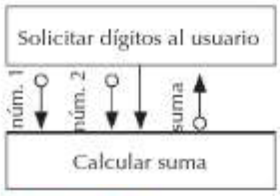


Figura 1.11
Acoplamiento normal por datos.

El módulo "Solicitar dígitos al usuario" pide datos simples al usuario (números en este caso) y los almacena. Posteriormente invoca al módulo "Calcular suma" pasándole como parámetros los datos solicitados.

El módulo "Calcular suma" hace la operación indicada y devuelve el resultado.

b) Acoplamiento normal por marca o estampado: si los parámetros intercambiados son un dato compuesto (registro) de datos de tipos básicos. Por ejemplo:

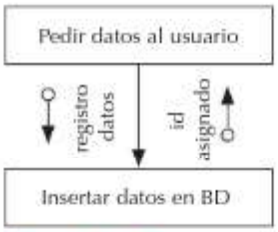


Figura 1.12
Acoplamiento normal por marca o estampado.

El módulo "Pedir datos al usuario" solicita datos simples al usuario y los almacena en forma de dato compuesto (registro). Posteriormente invoca al módulo "Insertar datos en BD" pasándole como parámetro el dato compuesto.

El módulo "Insertar datos en BD" inserta los datos del registro en la base de datos y retorna el identificador asignado a la inserción realizada.

c) Acoplamiento normal de control: cuando un módulo le pasa al otro un parámetro con la intención de controlar su lógica de funcionamiento interna. Por ejemplo:

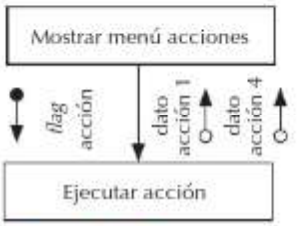


Figura 1.13
Acoplamiento normal de control.

El módulo "Mostrar menú acciones" presenta las acciones que se pueden realizar y recoge la que el usuario seleccione. Posteriormente invoca al módulo "Ejecutar acción" pasándole como parámetro el identificador de la acción que el usuario seleccionó previamente.

El módulo "Ejecutar acción" lleva a cabo la acción correspondiente y, dependiendo de esta, puede que retorne algún dato.

2. *Acoplamiento externo*: se da cuando dos o más módulos utilizan las mismas fuentes externas de datos (interfaces de dispositivos o de programas externos).
3. *Acoplamiento global*: presente cuando los módulos utilizan los mismos datos globales (variables globales, memoria compartida, ficheros o bases de datos).
4. *Acoplamiento patológico o por contenido*: un módulo lee o modifica los datos internos de otro módulo o bien salta al interior de su código.

En la figura 1.14 se pueden apreciar los diferentes tipos de acoplamiento y cómo estos afectan al mantenimiento:

- *Acoplamiento normal*
  - *Acoplamiento normal por datos*
  - *Acoplamiento normal por marca o estampado*
  - *Acoplamiento normal de control*
- *Acoplamiento externo*
- *Acoplamiento global*
- *Acoplamiento patológico o por contenido*

Figura 1.14
Tipos de acoplamiento.

RECUERDA

✓ La programación modular intenta solventar el crecimiento desmesurado que se plantea con la programación estructurada mediante la *descomposición* en módulos o subprogramas de menor tamaño.

Cuanto mayor sea la relación existente entre los elementos contenidos en un módulo, mayor será la *cohesión* y, por tanto, más exitosa la modularización llevada a cabo.

Existen varios tipos de cohesión. Steven, Myers, Constantine y Yourdon (1974) establecieron la "escala de cohesión", que identifica qué tipos son los más deseables así como aquellos que deben evitarse.

Mediante el *árbol de cohesión* es fácil conocer el tipo de cohesión.

Cuanto menor sea la relación existente entre los elementos de un módulo con los elementos de otro módulo, menor será el *acoplamiento* y, por tanto, más exitosa la modularización realizada.

Existen varios tipos de acoplamiento y los módulos pueden estar relacionados por varios tipos de ellos, en cuyo caso el acoplamiento que caracterizará la relación entre estos será la del peor tipo.

Actividad propuesta 1.3

Propón dos ejemplos de cada uno de los tipos de acoplamiento vistos anteriormente.

1.4. Pseudocódigo

El *pseudocódigo* es un lenguaje cercano a un lenguaje de programación cuyo objetivo es el desarrollo de algoritmos fácilmente interpretables por un programador, independientemente del lenguaje de programación del que provenga. En sí mismo no se trata de un lenguaje de programación, pero sí que utiliza un conjunto limitado de expresiones que permiten representar las estructuras de control y los módulos descritos en los paradigmas de programación estructurada y modular.

Mediante pseudocódigo se puede escribir aquellos algoritmos que tengan solución finita y que comiencen desde un único punto de partida. La escritura de un algoritmo o programa en pseudocódigo debería favorecer la posterior traducción al lenguaje de programación elegido.

SABÍAS QUE...

Dado que no existe una sintaxis estandarizada para la escritura de pseudocódigo, es posible encontrar diferencias sustanciales en los pseudocódigos escritos por diferentes programadores.

1.4.1. Operadores, palabras reservadas y tipos de datos

A pesar de que no existe una norma rígida que establezca cómo realizar la escritura de programas en pseudocódigo, es recomendable seguir una serie de recomendaciones que permitan transcribir el programa al lenguaje de programación que va a usarse con la mayor facilidad. A continuación, se muestran las tablas resumen de los operadores, palabras reservadas y tipos de datos empleados para la escritura de pseudocódigo en el capítulo.

CUADRO 1.1
Resumen operadores

Aritméticos		Relacionales (Usados para formar condiciones)		Lógicos (Usados para formar condiciones)	
+	Suma	=	Igual	and	y lógico (conjunción)
-	Resta	<	Menor	or	o lógico (disyunción)
*	Multiplicación	≤	Menor o igual	no	Negación lógica
/	División real	>	Mayor	**Especiales**	
div	División entera	≥	Mayor o igual	←	Asignación
mod o ÷	Resto o módulo	<>	Distinto	//	Comentario
^	Potencia				

CUADRO 1.2
Palabras reservadas

Inicio	Si no	Otro	Para	En
Fin	Según	Mientras	Hasta	Procedimiento
Si	Hacer	Repetir	Incremento	Función
Entonces	Caso	Hasta que	Cada	Imprimir
Leer	Retornar			

CUADRO 1.3
Tipos de datos

Carácter	Cadena	Entero	Real	Booleano

1.4.2. Estructuras de control

Tal como se ha comentado en el apartado anterior, el pseudocódigo utiliza las estructuras de control propias de la programación estructurada. Por este motivo se emplearán secuencias que representen las estructuras de control *secuencial*, *alternativa* e *iterativa*.

A) Estructuras de control secuencial

Describen bloques de instrucciones que son ejecutadas en orden de aparición (secuencialmente). Los bloques pueden estar delimitados por las expresiones `Inicio-Fin` o bien estar contenidos en otras estructuras. A continuación, se muestra un ejemplo de estructura secuencial:

```
Inicio
   <instrucción1>
   …
   <instrucciónN>
Fin
```

B) Estructuras de control alternativa

La estructura de control *alternativa* o *selectiva* encauza el flujo de ejecución hacia un bloque de instrucciones u otro en función de la evaluación que se realiza sobre una condición determinada.

Hay diferentes subtipos de este tipo de estructura de control:

1. *Alternativa simple*: establece un conjunto de instrucciones que se ejecutarán si se cumple una condición que retornará un valor *booleano*. Ejemplo:

```
Si <condición> entonces
   <instrucción1>
      …
   <instrucciónX>
Fin Si
```

2. *Alternativa doble*: añade otro bloque de instrucciones que se ejecuta en caso de que no se cumpla la condición. Ejemplo:

```
Si <condición> entonces
   <instrucciones1>
Si no
   <instrucciones2>
Fin Si
```

3. *Alternativa múltiple*: permite ejecutar diferentes bloques de instrucciones según el valor que tome una expresión que es comparada con los valores de cada caso o bien mediante el anidamiento de diferentes estructuras de alternativa doble cuyas condiciones son excluyentes. Ejemplo de bloques equivalentes en resultado:

Equivalentes →

```
Según <expresión> hacer
   Caso <valor1>
        <instrucciones1>
   Caso <valor2>
        <instrucciones2>
   Caso <valor3>
        <instrucciones3>
   Otro caso
        <instruccionesN>
Fin Según
```

```
Si <condición1> entonces
   <instrucciones1>
Si no
   Si <condición2> entonces
        <instrucciones2>
   Si no
        Si <condición3> entonces
             <instrucciones3>
        Si no
             <instruccionesN>
        Fin Si
   Fin Si
Fin Si
```

C) Estructuras de control iterativa

La estructura de control *iterativa* permite que un bloque de instrucciones sea ejecutado mientras se cumpla una condición.

Hay diferentes subtipos de este tipo de estructura de control:

1. *Iteración con salida al principio (while)*: primeramente, evalúa la condición y en caso de cumplirse ejecuta el bloque de instrucciones. La condición deberá cambiar de valor según las instrucciones contenidas para evitar bucles infinitos. Es posible que nunca llegue a ejecutarse el bloque de instrucciones. Ejemplo:

```
Mientras <condición> Hacer
   <instrucciones>
Fin Mientras
```

2. *Iteración con salida al final (repeat y do while):* primeramente, ejecuta el bloque de instrucciones y posteriormente evalúa la condición. La condición deberá cambiar de valor según las instrucciones contenidas para evitar bucles infinitos. El bloque de instrucciones se ejecutará, como mínimo, una vez. La versión que hace uso de *repeat* ejecutará las instrucciones hasta que se cumpla la condición mientras que la variante *do while* ejecutará las instrucciones mientras se cumpla la condición. A continuación, se muestran ejemplos de pseudocódigo para *repeat* y *do while*:

```
Repetir
   <instrucciones>
Hasta Que <condición>
```

```
Hacer
   <instrucciones>
Mientras <condición>
```

3. *Iteración con contador (for):* ejecutará el bloque de instrucciones un número determinado de iteraciones. Hace uso de una variable que irá incrementando o decrementando por cada iteración hasta que se cumpla la condición de salida. La condición de control, que se traducirá a `i==N o i==1` según sea el incremento, deberá ser falsa para que sigan ejecutándose las instrucciones. Ejemplos de *for incremental* y *decremental:*

```
entero i
Para i ← 1 Hasta N Incremento 1 Hacer
   <instrucciones>
```

Incremento positivo

```
entero i
Para i ← N Hasta 1 Incremento -1 Hacer
   <instrucciones>
```

Incremento negativo

4. *Iteración para cada (for each):* ejecutará el bloque de instrucciones para cada elemento contenido en un conjunto. Ejemplo:

```
entero i
Para Cada elemento En conjunto Hacer
   <instrucciones>
Fin Para Cada
```

D) Estructuras modulares

También es posible describir mediante pseudocódigo la descomposición modular de los programas representando los procedimientos y las funciones encargados de realizar las tareas.

1. Procedimientos

Cada bloque de código contenido en un procedimiento se ocupará de llevar a cabo un conjunto de instrucciones cuando este sea invocado. Podrá recibir *argumentos* con los que trabajar a la hora de ser llamado si se ha definido el *parámetro* correspondiente, pero en ningún caso retornará un valor de salida. A continuación, se muestra un ejemplo de pseudocódigo para un procedimiento:

```
Procedimiento nombre (tipo parámetro1, tipo parámetro2, …)
   <instrucciones>
Fin Procedimiento
```

A continuación, se muestran dos ejemplos típicos de programación correspondientes a *HolaMundo* y a un programa *Saludo*:

```
Procedimiento HolaMundo ()
   imprimir ("Hola Mundo")
   imprimir ("----------")
Fin Procedimiento

Inicio
   HolaMundo ()
Fin
```

```
Procedimiento Saludo (cadena nombre)
   imprimir ("Hola" + nombre)
Fin Procedimiento
Inicio
   cadena nombre
   imprimir ("Introduzca su nombre: ")
   leer (nombre)
   Saludo (nombre)
Fin
```

2. Funciones

Al igual que los procedimientos, una función podrán recibir *argumentos* con los que trabajar a la hora de ser llamada si se ha definido el *parámetro* correspondiente, pero, a diferencia de los procedimientos, retornará un valor de salida. Ejemplo de pseudocódigo para una función:

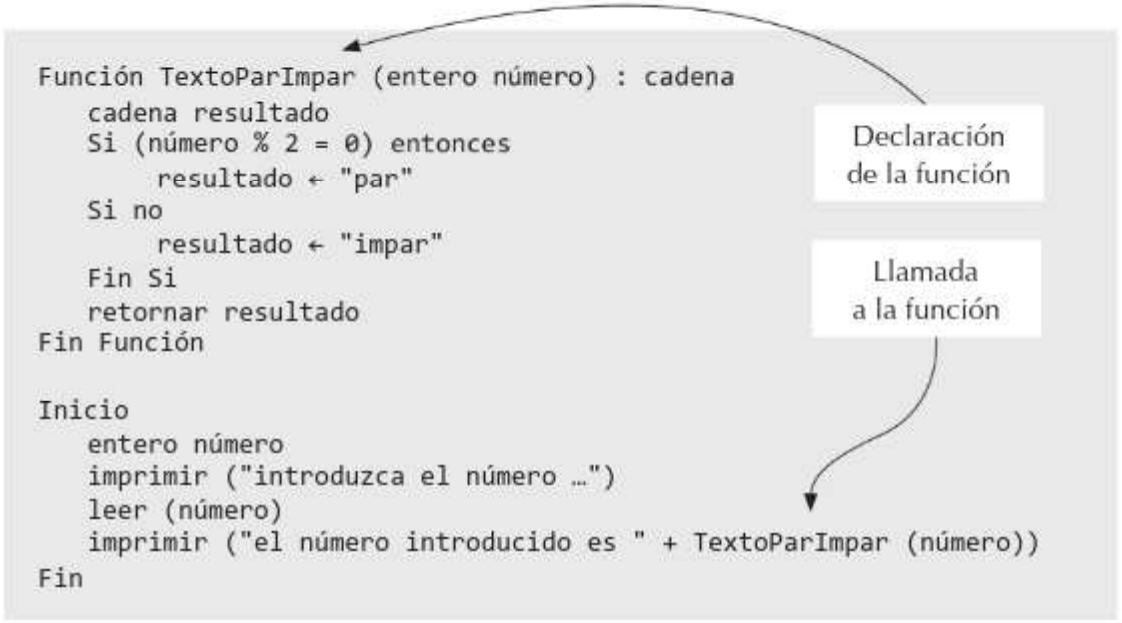
```
Función nombre (tipo parámetro1, tipo parámetro2, …) : tipoRetorno
   <instrucciones>
   …
   retornar X
Fin Función
```

RECUERDA

✓ El pseudocódigo es un lenguaje que permite realizar una aproximación muy cercana a la implementación de un problema determinado en un lenguaje real de programación (C, Pascal, Algol…).

No se ajusta a ningún estándar y, por lo tanto, puede presentar diferencias dependiendo de qué programador lo escriba.

A continuación, se muestra un ejemplo típico de programación:




```
Función TextoParImpar (entero número) : cadena
    cadena resultado
    Si (número % 2 = 0) entonces
        resultado ← "par"
    Si no
        resultado ← "impar"
    Fin Si
    retornar resultado
Fin Función

Inicio
    entero número
    imprimir ("introduzca el número …")
    leer (número)
    imprimir ("el número introducido es " + TextoParImpar (número))
Fin
```

Actividad propuesta 1.4

Empleando pseudocódigo desarrolla un programa que solicite la introducción de dos números por teclado (*núm. 1* y *núm.* 2) y muestre por pantalla la suma, resta, multiplicación y división entera (siendo *núm.* 2 el divisor). Ten en consideración que si *núm.* 2 es 0, el resultado será infinito (∞).

1.5. Diagramas de flujo

Un *diagrama de flujo*, *ordinograma* o *flujograma* es una representación gráfica de un algoritmo o proceso. Son utilizados en informática, aunque también se emplean en otros ámbitos diferentes.

Facilita la comprensión del algoritmo gracias a la descripción visual que aporta sobre el flujo de ejecución de este.

PARA SABER MÁS

Otra técnica empleada para la representación de procesos son los diagramas *Nassi-Shneiderman o Chapin*. Este tipo de diagramas permite representar, de forma análoga a los diagramas de flujo, el flujo de ejecución de un programa o algoritmo.

1.5.1. Simbología

Los diagramas de flujo se construyen utilizando un conjunto de símbolos que se van enlazando entre sí para dar significado al proceso. En el cuadro 1.4 se muestra cada uno de los símbolos empleados, así como una descripción de estos:

- ✓ Todo diagrama de flujo comienza y finaliza con un *terminal* representado mediante un óvalo o elipse.
- ✓ El trapecio rectángulo representa la *entrada de datos desde teclado*. Indica la detención del proceso a la espera de que el usuario teclee los datos.
- ✓ La comunicación con los periféricos para la *entrada* o *salida* de datos se representa mediante un paralelogramo.
- ✓ En todo algoritmo existe un orden en el que se realizan sus operaciones. En los diagramas de *flujo* ese orden se indica mediante una flecha.
- ✓ Las *decisiones* se representan mediante un rombo del que salen tantas líneas de flujo como alternativas sean posibles.
- ✓ Si el algoritmo a representar es muy grande, puede ser complejo diseñar su diagrama en un único bloque y lo más conveniente será dividir el diagrama en bloques más pequeños. Para ello se pueden emplear *conectores* que representarán el punto al cual saltar una vez se llegue a ese punto. Los saltos se representarán mediante una circunferencia si son en la misma página, o con un pentágono irregular si son en otra.
- ✓ Los *procesos* que llevan a cabo las operaciones internas de cálculo se representan mediante un rectángulo.
- ✓ Es posible que haya determinadas tareas que, por su complejidad, no puedan ser representadas como un proceso. En ese caso serán descompuestas en *subprocesos* que estarán definidos en otro lugar y que están representados por un rectángulo con doble línea en cada lado.
- ✓ Las *bases de datos* con las que se intercambia información tienen su propio símbolo (cilindro).
- ✓ Los *documentos impresos* también tienen sus propios símbolos de representación.

CUADRO 1.4
Simbología propuesta por la American National Standards Institute (ANSI)

Hasta este punto se han representado los símbolos más comunes. Se ha considerado innecesario describir la simbología al completo debido a que algunos de los símbolos existentes han quedado obsoletos al representar elementos que ya no se utilizan (cintas magnéticas, tarjetas perforadas…).

1.5.2. Estructuras de control

Mediante los diagramas de flujo se podrán representar las estructuras de control anteriormente descritas en pseudocódigo.

Figura 1.15
Secuencial.

Figura 1.16
Alternativa simple.

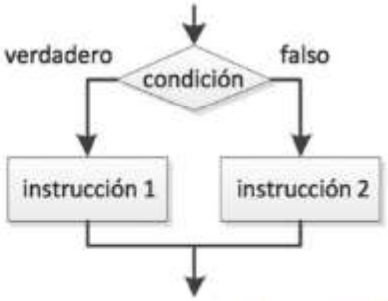


Figura 1.17
Alternativa doble.

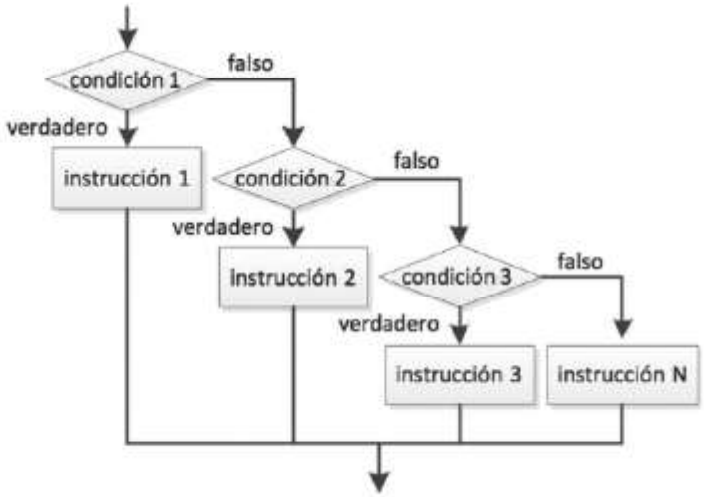


Figura 1.18
Alternativa múltiple con simples anidadas.

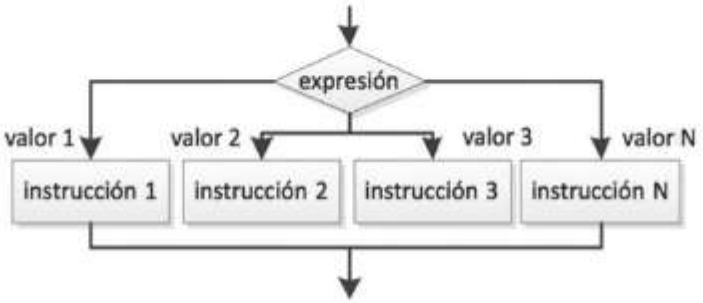


Figura 1.19
Alternativa múltiple II.

Figura 1.20
Iterativa *while*.

Figura 1.21
Iterativa *repeat*.

Figura 1.22
Iterativa *do while*.

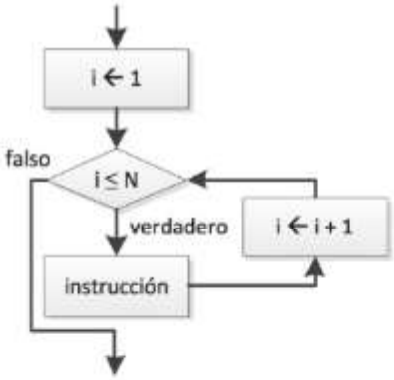


Figura 1.23
Iterativa *for incremental*.

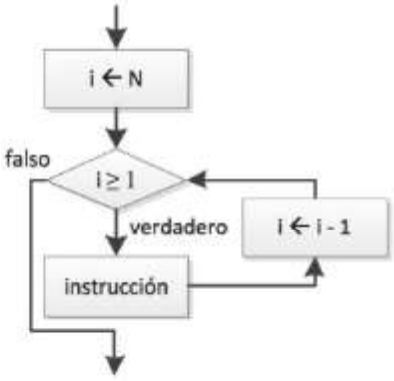


Figura 1.24
Iterativa *for decremental*.

Cada función o procedimiento (estructuras modulares) quedará descrito por un diagrama independiente y la invocación a dicho subprograma se realizará desde el diagrama principal.

- En el caso de invocar un procedimiento, lo representaremos mediante una llamada al subproceso que queda identificado por el rectángulo de lados dobles. A continuación, se muestran los diagramas correspondientes a los ejemplos realizados en pseudocódigo:

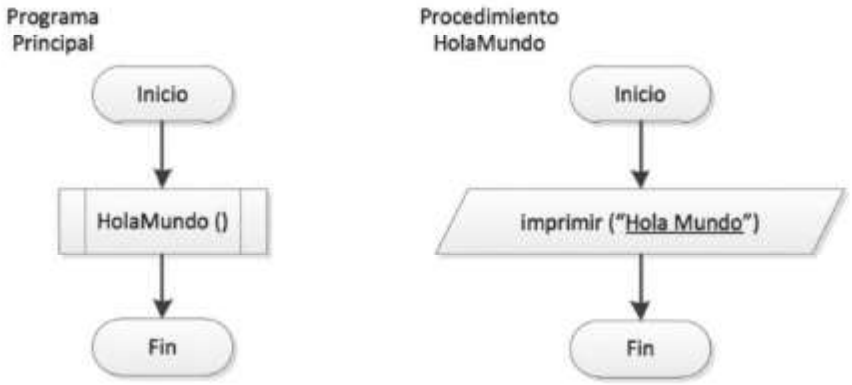


Figura 1.25
Ejemplo *HolaMundo*.

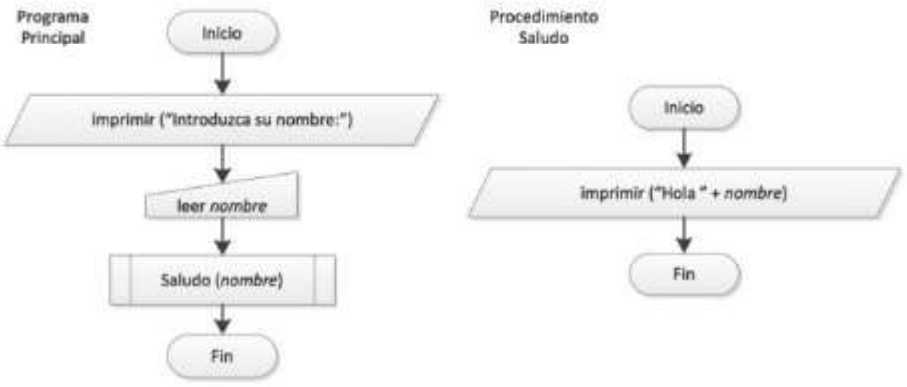


Figura 1.26
Ejemplo *Saludo*.

- Para invocar una función bastará con indicar la llamada en el proceso:

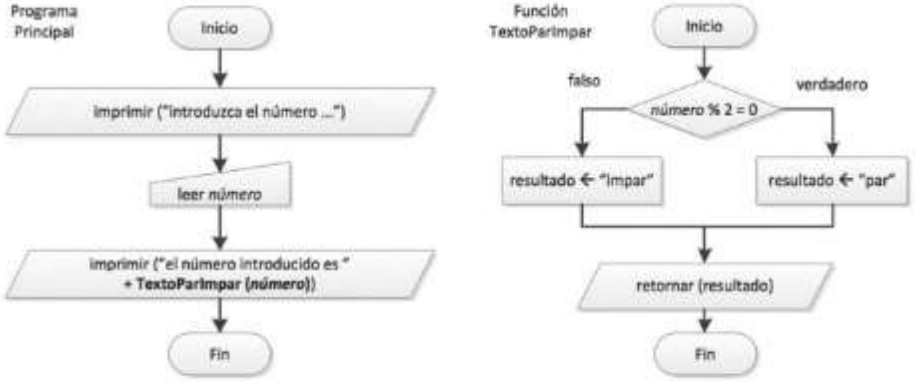


Figura 1.27
Ejemplo de función.

RECUERDA

✓ Los diagramas de flujo son la representación gráfica de un algoritmo, proceso o programa.
La simbología empleada será la propuesta por ANSI.

Actividad propuesta 1.5

Realiza el diagrama de flujo para un programa similar al del enunciado 1.4, en el que se solicite la introducción de dos números por teclado (*num1* y *num2*) y se muestre por pantalla la suma, resta, multiplicación y división entera (siendo num2 el divisor). Ten en cuenta que si num2 es 0, el resultado será infinito (∞).