

Capítulo 2 - Fundamentos básicos de la programación y algoritmos

[Introducción](#)

[2.1 ¿Qué es programar?](#)

[2.2 Procesos de desarrollo del software](#)

[Modelo en cascada](#)

[Análisis de requisitos](#)

[Diseño de la solución](#)

[Codificación, traducción y pruebas unitarias](#)

[Integración, validación y verificación del programa](#)

[Mantenimiento](#)

[Comentarios sobre la aplicación modelo en cascada](#)

[2.3 Algoritmos](#)

[Fundamentos y definiciones](#)

[Datos](#)

[Entradas y salidas](#)

[Variables](#)

[Tipos de datos](#)

[Numéricos](#)

[Lógicos](#)

[Texto](#)

[Otros tipos especiales](#)

[Arreglos](#)

[Constantes](#)

[Operadores](#)

[Operador de asignación](#)

[Tabla 2.1: operadores de asignación \(los operadores en rojo no son válidos en lenguaje C\)](#)

[Operadores aritméticos](#)

[Tabla 2.2: operadores aritméticos \(los operadores en rojo no son válidos en lenguaje C\)](#)

[Operadores relacionales](#)

[Tabla 2.3: operadores relacionales. \(los operadores en rojo no son válidos en lenguaje C\)](#)

[Operadores lógicos](#)

[Tabla 2.4: tabla de verdad de las operaciones lógicas](#)

[Tabla 2.5: operadores lógicos \(los operadores en rojo no son válidos en lenguaje C\)](#)

[Precedencia](#)

[Funciones predefinidas](#)

[Representación de Algoritmos](#)

[Pseudocódigo](#)

[Diagrama de flujo](#)

[2.4 Programación estructurada](#)

[Estructura secuencial](#)

[Estructuras de control selectivas](#)

[Selectiva simple](#)

[Selectiva doble](#)

[Selectiva múltiple](#)

[Estructuras de control iterativas](#)

[Mientras](#)

[Hacer-Mientras](#)

[PARA](#)

[2.5 Programación modular](#)

[Concepto de función](#)

[Definición de una función](#)

[Uso de una función](#)

Introducción

En este capítulo se introduce al lector en los aspectos más básicos de la programación, como fase previa al aprendizaje del lenguaje C.

En la primera parte se explica el esquema tradicional (en cascada) para la resolución de un problema mediante un programa informático. Se expone el concepto de algoritmo, sus elementos básicos (datos de E/S, variables, instrucciones ejecutables) y distintas maneras de representación de alto nivel (diagrama de flujo, pseudo-código).

En una segunda parte se desarrollan las distintas estructuras de control de flujo como bloques elementales del paradigma de “programación estructurada” y el concepto de función como herramienta fundamental para la “programación modular”.

2.1 ¿Qué es programar?

Programar es, en un sentido general, planificar y ordenar las acciones necesarias para realizar un proyecto. En nuestro caso, que nos ocuparemos de la programación de computadoras, programar es una actividad (para algunos es un arte) mediante la cual le indicamos la *secuencia de acciones* que debe realizar una máquina para cumplir con algún objetivo determinado.

Pero aunque parezca que programar una computadora es simplemente escribir las *instrucciones* que queremos que esta cumpla, la resolución de problemas mediante la creación de *programas informáticos (software)* suele ser un *proceso* arduo que comienza siempre con una necesidad o problema a resolver y, al contrario de lo que se cree, no finaliza con la obtención de un *programa ejecutable*, sino que el mismo estará sujeto a pruebas, modificaciones y actualizaciones.

2.2 Procesos de desarrollo del software

El proceso de desarrollo o “ciclo de vida” del software es una de las ramas de estudio de la Ingeniería de Software. Existen distintos modelos que describen el proceso de desarrollo, los cuales plantean distintas etapas y formas de proceder para que un programador o un equipo de programadores lleve a cabo el desarrollo de un programa.

Modelo en cascada

El modelo de *desarrollo en cascada* (ver fig. 1) es el más clásico, el primero en aparecer [] y a partir del cual surgen las técnicas más modernas.

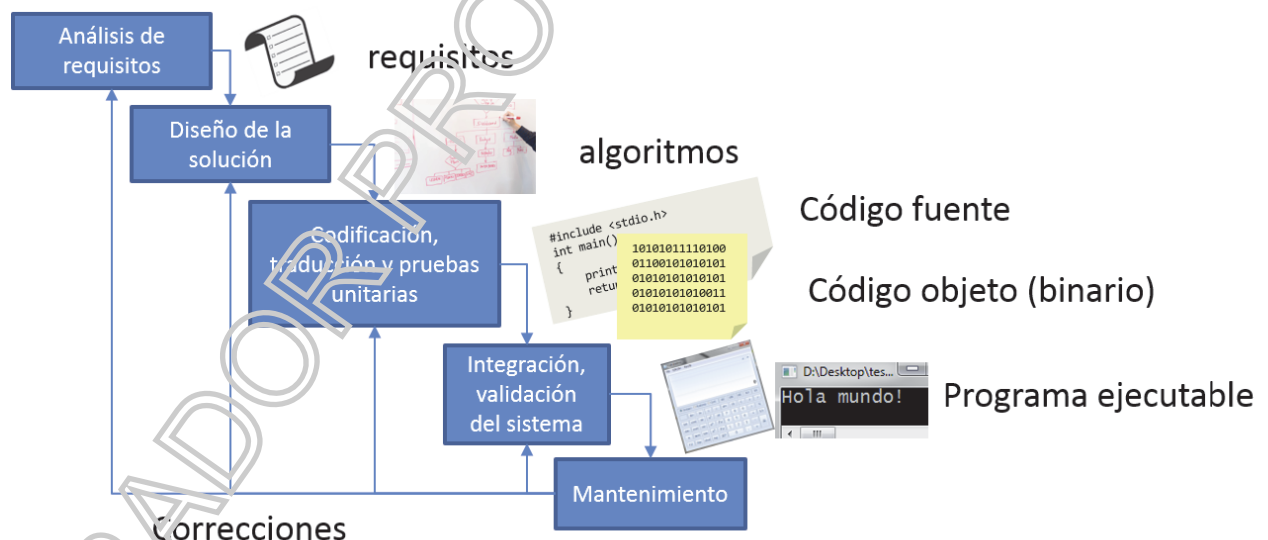


Fig 1: Modelo de desarrollo en cascada -

En este modelo se plantea seguir una serie de etapas de manera secuencial, las cuales resumimos como:

Análisis de requisitos

Se analiza la problemática a resolver y las necesidades de los futuros usuarios del programa, convirtiéndolos en *requisitos* concretos que debe cumplir el programa.

Diseño de la solución

Se realizan los esquemas de las distintas partes del software (*módulos, funciones*). Se determina la manera en que se representará y manejará la información de interés (*variables, constantes, estructuras de datos*). Se establece la forma de interacción del usuario (*interfaz, entradas y salidas*). Se diseñan los *algoritmos* de cada módulo.

En lo que resta de este capítulo se abordan los fundamentos para poder llevar a cabo la fase de diseño.

Codificación, traducción y pruebas unitarias

Se escribe el *código fuente* de los algoritmos de los distintos módulos en algún *lenguaje de programación*.

Se “traduce” el código escrito en un *lenguaje de programación* (comprensible y cercano a las personas) al *código de máquina* compuesto por el *conjunto de instrucciones* del procesador de la computadora, que sólo comprende y puede ejecutar este último. Existen dos tipos de “traductores”: *compiladores* e *intérpretes*, los cuales se explicarán en el siguiente capítulo.

Luego de la compilación (o interpretación) se prueba cada módulo de manera independiente del resto, verificando las salidas obtenidas para entradas conocidas.

Integración, validación y verificación del programa

Se integran los distintos módulos ya depurados, conformando el *programa ejecutable*. Se ejecuta el programa y se verifica el cumplimiento de los distintos requisitos generados en la fase de análisis. Se corrigen los errores encontrados.

Finalmente, se instala o despliega el programa en el sistema informático del usuario, que debe comprobar que el programa sea acorde con sus necesidades originales.

Mantenimiento

Corrección de fallas detectadas durante el uso y de necesidades cubiertas parcialmente.

Comentarios sobre la aplicación modelo en cascada

Seguir a rajatabla el modelo anterior no es eficiente, especialmente cuando esperamos concluir toda la fase de codificación para comenzar la traducción y las pruebas. Lo cual retrasa la identificación de errores de funcionamiento y dificulta su identificación en el código.

Suele ser conveniente entremezclar estas dos fases, de manera tal de ir codificando, ejecutando y probando de a partes pequeñas de código. Así es posible identificar con más facilidad las secciones de código con errores y solucionarlos a la brevedad.

2.3 Algoritmos

Según el diccionario de la RAE un algoritmo es un “*Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.*”. Prácticamente convivimos con distintos algoritmos toda la vida, que permiten resolver desde los más triviales problemas de la vida cotidiana, como la secuencia de pasos para atarnos los cordones o las instrucciones para seguir una receta de cocina hasta procedimientos rutinarios de trabajo o protocolos de evacuación de un edificio ante una contingencia.

Como hemos visto en la sección anterior, para la informática, programar implica definir las acciones que deberá ejecutar una máquina (y el orden de las mismas) para cumplir un objetivo. Esto no es ni más ni menos que “decirle” a la computadora que debe ejecutar un algoritmo.

En general se deben cumplir las siguientes condiciones:

- En el algoritmo debe existir un único punto de *inicio* y (al menos) un fin.
- El algoritmo debe tener una *cantidad finita* de pasos.
- Los pasos del algoritmo deben ejecutarse en *tiempo finito*.
- Las órdenes deben ser *ejecutables*: Tienen que ser operaciones básicas que la computadora pueda realizar. También pueden ser procedimientos complejos, pero definidos en base a operaciones básicas y otros procedimientos, que puedan ser ejecutados.
- Las instrucciones deben ser *precisas*, no pueden ser ambiguas: Una orden o instrucción dada a una computadora solo puede ser interpretada de una única manera.

En el proceso de desarrollo de software, el diseño de los algoritmos es una parte fundamental ya que establece la secuencia de pasos lógicos, acciones y decisiones que posteriormente deberá ejecutar la computadora. Los algoritmos diseñados son independientes de los distintos *lenguajes de programación* y de las *arquitecturas de cómputo*, y constituyen descripciones de alto nivel de abstracción, que en una etapa posterior pueden codificarse en el lenguaje deseado. Esto permite

trabajar sobre lo que debe hacer un programa sin que importen los detalles de cómo está codificado ni con qué lenguaje, yendo de lo más general a lo más particular¹.

Fundamentos y definiciones

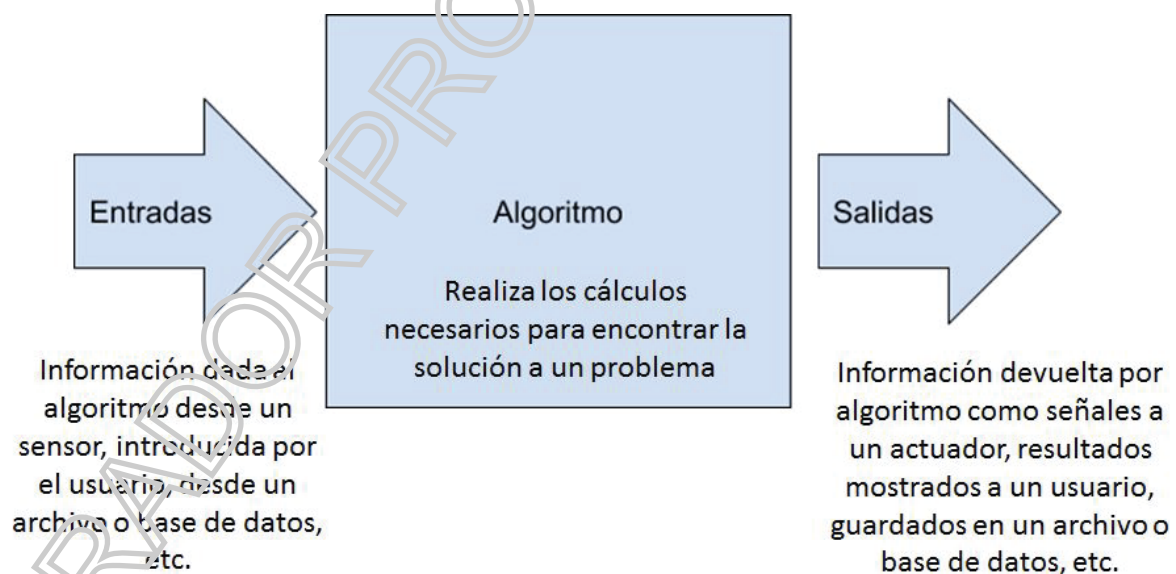
Antes de profundizar en el diseño y representación de algoritmos existen tres conceptos fundamentales con los que el lector deberá familiarizarse, estos son los *Datos*, los *Operadores* y las *Funciones de E/S*.

Datos

Los conceptos vistos sobre los algoritmos hasta el momento giran en torno a las instrucciones o pasos que se deben ejecutar para cumplir un determinado objetivo. Sin embargo, la figura principal en el mundo de los algoritmos informáticos es la *información*, o los *datos* a través de los cuales esta se materializa.

Entradas y salidas

Suele ser común entonces pensar los algoritmos como bloques que procesan información, que tienen como materia prima un conjunto de *datos de entrada*, los cuales son manipulados por el algoritmo, y a partir de los cuales entrega otro conjunto de *datos de salida*. La Fig. 2.2 representa gráficamente este modelo.



¹ A esta modalidad de diseño desde lo más general (y abstracto) a lo más específico (y concreto) se lo conoce como “top-down”, como se explica en el Capítulo 1.

Fig. 2.2: Modelo del algoritmo como procesador de información

Cuando se requiere diseñar o analizar un algoritmo es importante identificar cuales son los datos de entrada y de salida.

Los datos de entrada pueden provenir de diversas fuentes, pueden ser valores previamente almacenados en la memoria principal o cualquier dato proveniente de los distintos periféricos de entrada del sistema de cómputo (como por ejemplo teclado, mouse, red, disco, scanner, sensor de temperatura, sensor de velocidad, etc.). En este curso trataremos principalmente con datos que provienen de tres fuentes distintas: datos ingresados por el usuario a través del teclado, parámetros de entrada de una función o programa y archivos en disco.

Los datos de salida son los que el algoritmo envía a un periférico de salida del sistema de cómputo (por ejemplo monitor, impresora, parlantes, red, disco, motores, luces, etc.); y también cualquier dato producido por el algoritmo que luego de finalizado el mismo queda en memoria principal a disposición de otros programas.

Variables

Tanto los datos de entrada/salida como el resto de los datos auxiliares que requiera un algoritmo para cumplir su objetivo, se almacenan en la memoria principal de la computadora, cada uno en un lugar (*dirección de memoria*) particular de la misma y ocupando un bloque de una cantidad determinada de esta memoria (medida en bytes). A cada uno de estos bloques de memoria donde se guarda un dato se lo conoce como *variable*. En el diseño del algoritmo, en principio no debemos preocuparnos por aspectos tales como las direcciones de memoria, la cantidad de bytes que ocupa cada dato, o su representación en memoria, sino que basta con referirnos a la variable con un nombre o *identificador*.

Por ejemplo, en la Fig. 2.3 se esquematiza como dos datos distintos, que son almacenados en posiciones arbitrarias de la memoria principal, abarcando bloques de varios bytes cada una, para el algoritmo son representadas por las variables reales x y r .

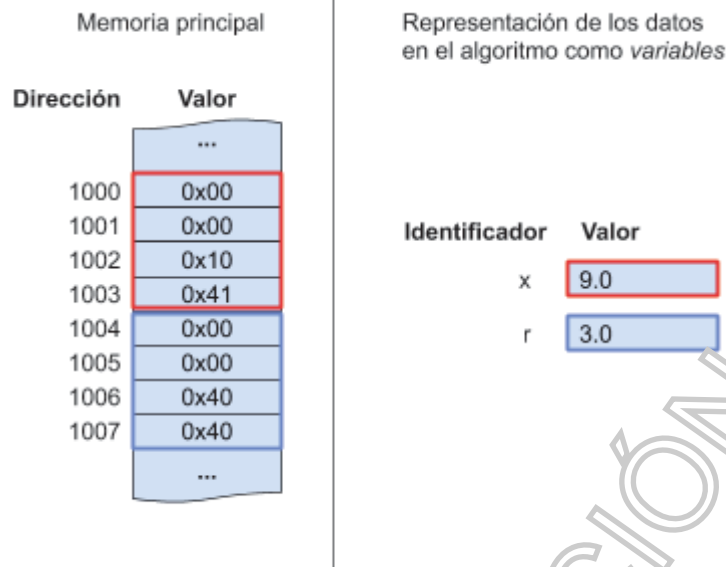


Fig. 2.3. Representación en memoria principal de las variables reales x y r utilizadas por el algoritmo.

Tipos de datos

Las computadoras son esencialmente máquinas que realizan operaciones aritméticas entre números y operaciones lógicas entre valores lógicos (verdadero/falso), también llamados *booleanos*. Sin embargo, la naturaleza de la información que la computadora debe procesar puede ser de diversos tipos. Para diseñar un algoritmo de manera independiente del lenguaje de programación que se use posteriormente para su codificación, puede considerarse que prácticamente todos los lenguajes permitirán representar datos de tipo numérico, texto y variables lógicas.

Numéricos

Muchas arquitecturas básicas de hardware sólo pueden operar con *números naturales* y *números enteros* siendo la representación y la operación con *números reales* muy costosa computacionalmente, ya que se resuelve por software (mediante algoritmos específicos). Por lo tanto (casi)² todos los lenguajes de programación diferencian los tipos de datos numéricos entre reales y enteros, y para estos últimos suelen diferenciar entre enteros con signo y sin signo (naturales).

Lógicos

La gran mayoría de lenguajes de programación poseen un tipo de dato específico para las variables booleanas y los que no, como el lenguaje C, utilizan números enteros y convenciones como considerar “falso” al cero y “verdadero” a todo lo que no sea cero.

² Una excepción notable es el lenguaje JavaScript que tiene un tipo de dato único para representar números que, por supuesto, permite almacenar reales.

Texto

Más allá de trabajar solamente con datos lógicos y numéricos, las computadoras que son operadas por el ser humano deben interpretar las entradas de los usuarios y reportar resultados legibles por estos últimos. Por esto prácticamente cualquier lenguaje de programación posee el tipo *carácter* para representar letras, símbolos alfanuméricos y de puntuación. Agrupando varios caracteres es posible representar texto. Muchos lenguajes, además, poseen un tipo de dato específico llamado *cadena* para el almacenamiento de texto.

Otros tipos especiales

Es común que los lenguajes incluyan algún tipo de dato compuesto para representar *fechas*. Los programadores, a su vez, pueden crear sus propios *tipos de datos compuestos*, agrupando variables de distintos tipos en una única *estructura*. Este tema se profundizará en el **capítulo 7**.

Arreglos

Los arreglos son conjuntos de variables del mismo tipo que comparten un mismo identificador, y donde cada uno de los elementos que componen el arreglo se identifica por un índice.

Constantes

Las constantes son tratadas como variables cuyo valor no podrá modificarse durante la ejecución del programa.

Operadores

Son elementos de los lenguajes de programación que definen las operaciones más básicas que pueden realizarse sobre los datos (operandos). La combinación entre operadores y operandos forman *expresiones*, las cuales dan como resultado un **valor** de un **tipo** determinado.

Operador de asignación

Modifican el valor almacenado en una variable. El identificador de la variable se escribe a la izquierda del operador y a la derecha va la expresión a evaluar, cuyo resultado se almacenará en la variable mencionada.

Para su representación se usarán indistintamente cualquiera de dos símbolos: una flecha de derecha a izquierda, que indica el sentido de la asignación: \leftarrow , o el signo igual: $=$.

Operador	Ejemplos
----------	----------

<-	a <- 5.6 b <- raiz_cuadrada(a*10)
=	a = 5.6 b = raiz_cuadrada(a*10)

Tabla 2.1: operadores de asignación (los operadores en rojo no son válidos en lenguaje C)

Operadores aritméticos

Representan las operaciones matemáticas más comunes entre números. Los operandos y el resultado de la operación son numéricos.

Operador	Operación	Tipo operandos	Tipo resultado	Ejemplos	Resultado
+	suma	numéricos	numérico	5 + 14	19
-	resta	numéricos	numérico	5 - 14	-9
*	multiplicación	numéricos	numérico	5 * 14	70
/	división	numéricos	real	5 / 14 14 / 5	0.3571 2.8
mod (%)	módulo (resto de división entera)	enteros	entero	5 mod 14 14 mod 5	5 4
div	división entera	enteros	entero	5 div 14 14 div 5	0 2
^	potencia	numéricos	numérico	14 ^ 5	537824

Tabla 2.2: operadores aritméticos (los operadores en rojo no son válidos en lenguaje C)

Operadores relacionales

Comparan dos operandos numéricos y devuelven un resultado lógico.

Operador	Operación	Tipo operandos	Tipo resultado	Ejemplos	Resultado V: verdadero F: falso
==	igual a	numéricos	lógico	5 == 14	F

>	mayor que	numéricos	lógico	5 > 14	F
<	menor que	numéricos	lógico	5 < 14	V
>=	mayor o igual que	numéricos	lógico	5 >= 14	F
<=	menor o igual que	numéricos	lógico	5 <= 14	V
<> (!=)	distinto a	numéricos	lógico	5 <> 14	V

Tabla 2.3: operadores relacionales. (los operadores en rojo no son válidos en lenguaje C)

Operadores lógicos

Realizan las 3 funciones lógicas básicas (NO,Y,O) que pueden resumirse en la siguiente tabla:

A	B	no(A)	no(B)	A y B	A o B
F	F	V	V	F	F
F	V	V	F	F	V
V	F	F	V	F	V
V	V	F	F	V	V

Tabla 2.4: tabla de verdad de las operaciones lógicas

Operador	Operación	Tipo operandos	Tipo resultado	Ejemplos	Resultado
NO (!)	negación: opera sobre un único operando, invirtiendo el valor lógico	lógico	lógico	no(5 < 14) no(5 == 14)	F V
Y / AND (&&)	conjunción: el resultado solo es verdadero si ambos operandos son verdaderos, de lo contrario es falso.	lógicos	lógico	no(5 < 14) y no(5 == 14) 5 < 14 y no(5 == 14)	F V
O / OR ()	disyunción: el resultado es verdadero si al	lógicos	lógico	no(5 < 14) o no(5 == 14) 5 < 14 o no(5 == 14) no(5 < 14) o (5 == 14)	V V F

	menos uno de los operandos son verdaderos, de lo contrario es falso.				
--	--	--	--	--	--

Tabla 2.5: operadores lógicos (los operadores en rojo no son válidos en lenguaje C)

Precedencia

La precedencia o prioridad de los operadores determina el orden en que se van a evaluar las distintas operaciones en una expresión compleja en la que intervienen varios operadores y sus respectivos operandos. Los operadores de mayor prioridad se evaluarán antes que los de prioridades más bajas. Los paréntesis sirven para delimitar expresiones y tienen la prioridad más alta. En caso de haber varios operadores de un mismo nivel de prioridad, estos se evalúan según la asociatividad de cada nivel, de izquierda a derecha en general o de derecha a izquierda para la asignación.

Operador	Prioridad	Asociatividad
()	Más alta (se evalúa primero)	izquierda a derecha
^		izquierda a derecha
* / mod div		izquierda a derecha
+ -		izquierda a derecha
==, <>, <, >, <=, >=		izquierda a derecha
NO		izquierda a derecha
Y		izquierda a derecha
O		izquierda a derecha
=	Más baja (se evalúa último)	derecha a izquierda

Tabla 2.6: prioridad de evaluación para los operadores de una expresión algorítmica. Esto en general se cumple para todos los lenguajes.

Ejemplo 2.1. Analizar el orden en que se evalúan las operaciones en cada expresión, y los valores que quedarán almacenados en las variables a, b, c y d, y el tipo de dato de la variable. Nótese que en un algoritmo las instrucciones (*sentencias*) se ejecutarán en orden, y en este caso los números a la izquierda de cada línea indican dicho orden de ejecución.

1 a = 4,5 * 4 div 1 + 3

2	$b = 4,5 * 4 \text{ div } (1 + 3)$
3	$c = 4,5 * (4 \text{ div } (1 + 3))$
4	$d = a <> b \text{ Y } a <> c \text{ Y NO}(b==c)$

Resolución:

Resolveremos cada línea por separado, evaluando las operaciones según el orden de prioridad dado por la tabla 2.6.

Línea 1:

$a = 4,5 * 4 \text{ div } 1 + 3$ → se indican en rojo los operadores involucrados
 $a = 4,5 * 4 \text{ div } 1 + 3$ → en azul los operadores de mayor prioridad
 $a = 4,5 * 4 \text{ div } 1 + 3$ → como hay más de uno se evalúa de izquierda a derecha
 $a = 18 \text{ div } 1 + 3$
 $a = 18 + 3$
 $a = 21$ → Se determina el valor de la variable a , que puede ser **entera o real**.

Línea 2:

$b = 4,5 * 4 \text{ div } (1 + 3)$
 $b = 4,5 * 4 \text{ div } (1 + 3)$ → Lo que esté encerrado entre paréntesis se evaluará primero
 $b = 4,5 * 4 \text{ div } (4)$
 $b = 4,5 * 4 \text{ div } 4$
 $b = 18 \text{ div } 4$
 $b = 4$ → Se determina el valor de la variable b , que puede ser **entera o real**.

Línea 3:

$c = 4,5 * (4 \text{ div } (1 + 3))$
 $c = 4,5 * (4 \text{ div } (1 + 3))$ → Lo que esté encerrado entre paréntesis se evaluará primero
 $c = 4,5 * (4 \text{ div } (1 + 3))$ → Si hay paréntesis anidados se evaluarán primero
 $c = 4,5 * (4 \text{ div } (4))$
 $c = 4,5 * (4 \text{ div } 4)$
 $c = 4,5 * 1$
 $c = 4,5$ → Se determina el valor de la variable c , que será **real**.

Línea 4:

$d = a <> b \text{ Y } a <> c \text{ Y NO}(b==c)$ → Cuando se ejecuta la línea 4: $a = 21$, $b = 4$ y $c = 4,5$
 $d = a <> b \text{ Y } a <> c \text{ Y NO}(b==c)$

d = a <> b Y a <> c Y NO(F)

d = a <> b Y a <> c Y NO F

d = a <> b Y a <> c Y NO F

d = V Y a <> c Y NO F

d = V Y V Y NO F

d = V Y V Y V

d = V Y V Y V

d = V Y V Y V

d = V

→ Se determina el valor de la variable d, que será **lógica**.

Funciones predefinidas

Además de los operadores mencionados, que permiten expresar operaciones aritmético-lógicas elementales, en general los programadores contarán con un conjunto de funciones predefinidas para la plataforma en la cual desarrollan. Estas funciones se identifican mediante un nombre particular (al igual que las variables y constantes) y pueden ser *llamadas* o *invocadas* por el programador desde un algoritmo principal.

En lugar de realizar operaciones simples, como las que realizan los operadores, pueden requerir la ejecución de numerosos pasos para realizar una tarea concreta, es decir que son sub-algoritmos o sub-programas, *invocados* desde otro algoritmo. Las funciones, además, pueden recibir *parámetros* como datos de entrada desde el algoritmo que las invoca y devolver a este algún resultado como salida. Si bien estas funciones con que se cuenta dependen tanto de las herramientas de desarrollo, como del lenguaje y del hardware utilizados, en general se dispone de *bibliotecas* (conjuntos de funciones) para las siguientes categorías:

- **Entrada / salida:** Las funciones de entrada brindan los mecanismos para que un algoritmo pueda recibir información desde los periféricos de entrada (teclado, ratón, micrófono, cámara, red, sistema de archivos, sensores, etc.). Por su parte, las funciones de salida proporcionan al algoritmo una manera de enviar la información procesada a los periféricos de salida (pantalla, impresora, parlantes, red, sistema de archivos, actuadores, etc.). Para algoritmos que interactúan con usuarios a través de *consolas de texto*, la entrada típica de información al algoritmo es el ingreso de texto a través del teclado y la salida de información del algoritmo al usuario es mediante texto escrito en la pantalla. Para trabajar con estas entradas y salidas generalmente existen las funciones **Leer()** y **Escribir()** o equivalentes.
- **Matemáticas:** Permiten realizar operaciones matemáticas más complejas, como logaritmos, funciones trigonométricas, exponenciales, raíces, redondeos, etc.

- Manejo de texto: Facilitan el procesamiento de textos. Permiten unir y separar cadenas, pasar a mayúsculas o minúsculas, contar letras, comparar textos, etc.
- Manejo de fechas / horas: Asisten en el procesamiento de datos de fecha/hora.

Ejemplo 2.2: (a) Escribir las instrucciones que formarán parte de un algoritmo que calcule el perímetro y área de una circunferencia. El algoritmo debe interactuar con un usuario que ingresará el radio por teclado. Los resultados se mostrarán en pantalla.

(b) Mencione las variables que intervienen y si almacenan datos de entrada o de salida.

(c) Simular la ejecución instrucción por instrucción y el estado de las variables suponiendo que el usuario ingresa un radio de 1.5 m.

Resolución:

(a) El programa deberá esperar a que el usuario ingrese un valor numérico por teclado para el radio de la circunferencia (en metros), para eso se usará la función `Leer()`, que recibe como parámetro la variable donde se almacenará el valor ingresado:

```
1 Leer(radio)
```

Luego, se pueden calcular el área y el perímetro, los cuales se guardarán en dos variables distintas

```
1 Leer(radio)
2 área = radio*radio*3.1416
3 perímetro = 2*radio*3.1416
```

Finalmente, el algoritmo debe mostrar al usuario los resultados de su procesamiento, para esto usamos la función `Escribir()`, que recibe como parámetro la información a mostrar, según las siguientes convenciones:

- El texto literal se pasa entre comillas dobles:

`Escribir("texto")`

- Para mostrar el valor de una variable se pasa su identificador sin comillas:

`Escribir(variable)`

- Para escribir en la siguiente línea se pasa el parámetro NL (nueva línea):

`Escribir(NL)`

- Pueden pasarse varios parámetros separados por comas, se escribirán uno a continuación del otro:

`Escribir("valor 1 = ", var1, " valor 2 = ", var2)`


```

1 Leer(radio)
2 área = radio*radio*3.1416
3 perímetro = 2*radio*3.1416
4 Escribir("El área es ", área, " m2",NL)
5 Escribir("El perímetro es ", perímetro, " m")

```

Nótese que al comenzar el programa el usuario no recibe ninguna indicación del programa, lo cual puede ser confuso. Para corregir esto, se agrega una instrucción al principio para mostrar un cartel con indicaciones al usuario.

```

0 Escribir("Ingrese el radio en metros: ")
1 Leer(radio)
2 área = radio*radio*3.1416
3 perímetro = 2*radio*3.1416
4 Escribir("El área es ", área, " m2",NL)
5 Escribir("El perímetro es ", perímetro, " m")

```

(b) Las instrucciones escritas en (a) hacen uso de tres variables: *radio*, *área* y *perímetro*. La primera es una variable de entrada del algoritmo ya que almacena el dato ingresado por el usuario al ejecutarse el paso 1, mientras que las otras dos son variables de salida, ya que almacenan los resultados del procesamiento de los pasos 2 y 3, los cuales son mostrados al usuario en los pasos 4 y 5.

(c) Para simular la ejecución realizamos la "traza" del algoritmo, esto es una tabla de doble entrada con una instrucción por fila y una columna por cada variable que va mostrando la evolución de cada dato a medida que progresa la ejecución del algoritmo. Incluimos también una columna que muestra la evolución de la salida por pantalla.

Paso	radio	área	perímetro	Pantalla
0	?	?	?	Ingrese el radio en metros:
1	1.5	?	?	Ingrese el radio en metros: 1.5
2	1.5	7.069	?	Ingrese el radio en metros: 1.5
3	1.5	7.069	9.425	Ingrese el radio en metros: 1.5

4	1.5	7.069	9.425	Ingrese el radio en metros: 1.5 El área es 7.069 m2
5	1.5	7.069	9.425	Ingrese el radio en metros: 1.5 El área es 7.069 m2 El perímetro es 9.425 m

Representación de Algoritmos

Durante la fase de diseño de los algoritmos, se cuenta con distintas herramientas para poder representarlos, a continuación se abordarán dos de estas: los *diagramas de flujo* y el *pseudocódigo*.

Estas herramientas nos permiten documentar el diseño e intercambiar ideas con colegas. Al mismo tiempo, contar con un algoritmo correctamente representado permite analizar y predecir el comportamiento que tendrá el programa antes de codificarlo y así detectar errores en una etapa temprana del desarrollo.

Pseudocódigo

El pseudocódigo permite una representación de alto nivel de abstracción, donde usualmente se ocultan detalles de implementación irrelevantes para la comprensión del algoritmo, combinando texto común con estructuras y construcciones similares a las de los lenguajes de programación. De una manera distendida, los algoritmos en los **ejemplos 2.1 y 2.2** están representados en pseudocódigo, utilizando líneas de texto para representar cada paso o instrucción a ejecutarse en el algoritmo, ordenadas según la secuencia de ejecución.

Si bien no hay una sintaxis estándar para el pseudocódigo, los algoritmos se escriben en forma de listado de instrucciones ordenadas según su secuencia de ejecución y además suelen contar con una sección opcional donde se enumeran los datos (variables y/o constantes) involucrados en la ejecución del algoritmo, así como su tipo. A su vez pueden indicarse los puntos de inicio y de finalización del pseudocódigo, junto con un nombre que lo represente.

Por ejemplo, en el **Listado 2.1** se puede observar el algoritmo del **ejemplo 2.2**, representado en un pseudocódigo más detallado.

```

0  INICIO Área_y_Perímetro
1  Datos:
2    Variables:
3      reales: radio, área, perímetro
4    Constantes:
5      real: PI = 3.141592653
6  Algoritmo:
7    Escribir("Ingrese el radio en metros: ")

```

```

8      Leer(radio)
9      área = radio*radio*PI
10     perímetro = 2*radio*PI
11     Escribir("El área es ", área, " m2",NL)
12     Escribir("El perímetro es ", perímetro, " m")
13     FIN

```

Listado 2.1: Algoritmo del **ejemplo 2.2**.

Ejemplo 2.3: Escriba la representación en pseudocódigo del método numérico iterativo que calcula la raíz cuadrada de un número. El dato de entrada será ingresado por el usuario en la variable x . El resultado será mostrado en pantalla.

Método Numérico:

- Paso 1- en la variable r que se utilizará para devolver el resultado se asigna inicialmente x
- Paso 2- mientras que $r*r$ sea diferente de x (aún no se halló la raíz de x) se repite el Paso 3
- Paso 3- se actualiza el valor de r como el promedio entre su valor actual y x/r
- Paso 4- la raíz cuadrada de x quedó almacenada en r

Resolución:

Se comienza por los indicadores de inicio y fin, sabiendo que habrá solo dos variables, x y r , se indican en la sección de datos. Nótese que entre las marcas $/*$ y $*/$ se escriben comentarios, los cuales tienen aclaraciones que ayudan a la comprensión del algoritmo pero no son instrucciones ejecutables.

```

0      INICIO Raíz_cuadrada
1      Variables:
2      reales: x, r
3      Algoritmo:
4      /*
5      pendiente
6      */
7      FIN

```

En segundo lugar, se sabe que x será la variable de entrada ingresada por teclado, y r la variable de salida a mostrar en pantalla.

```

0      INICIO Raíz_cuadrada
1      Variables:
2      reales: x, r
3      Algoritmo:
4      Leer(x)
5      /*
6      método numérico r <-raiz(x)
7      */

```

```
8      Escribir(r, "es la raíz cuadrada de ", x)
9      FIN
```

Solo resta escribir las instrucciones correspondientes al método numérico como expresiones algorítmicas, teniendo en cuenta que el paso 4 no es necesario implementarlo.

```
0      INICIO Raíz_cuadrada
1      Variables:
2      reales: x, r
3      Algoritmo:
4      Leer(x)
5      r = x                      /* Paso 1 */
6      Mientras r*r <> x repetir: /* Paso 2 */
7      r = (x/r + r)/2           /* Paso 3 */
8      Fin_Mientras              /* esto indica el fin de la repetición */
9      Escribir(r, "es la raíz cuadrada de ", x)
10     FIN
```

Nótese que los pasos 2 y 3 se ejecutarán varias veces hasta que $r*r$ sea igual a x , y que "Fin_Mientras" se usó para delimitar las instrucciones que se repetirán mientras $r*r$ sea distinto de x . Esta construcción forma parte de las *estructuras de control* propias de la *programación estructurada*.

Diagrama de flujo

El diagrama de flujo (DF) representa gráficamente un algoritmo, mediante símbolos estandarizados por la ANSI y la ISO, facilitando la identificación de los distintos caminos o flujos de ejecución y la toma de decisiones.

Por ejemplo, en la FIG. 2.4 se representa el DF del **ejemplo 2.3**, que permite calcular la raíz cuadrada de un número ingresado por el usuario. En dicha figura se explican los símbolos utilizados.

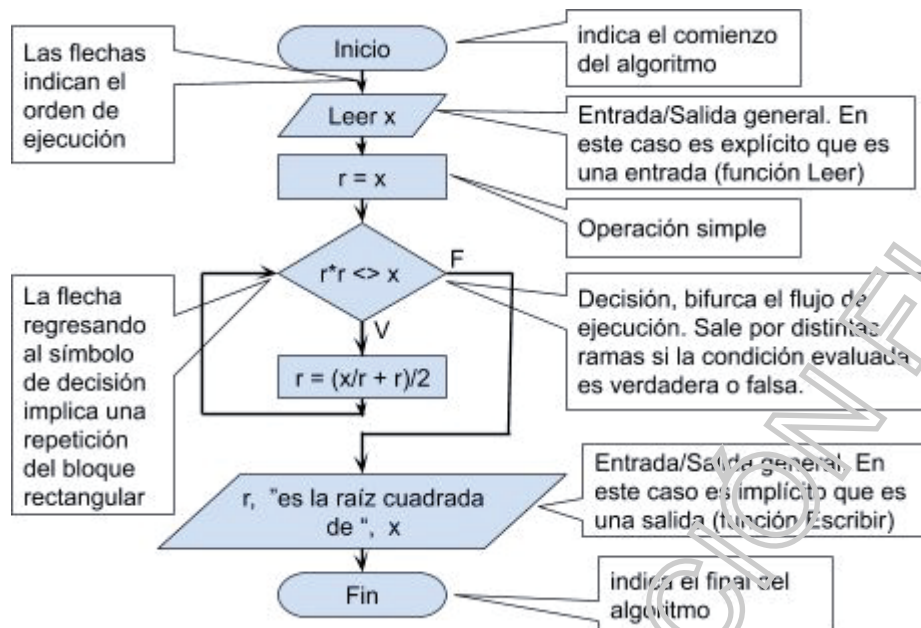


FIG. 2.4: Algoritmo del ejemplo 2.3 representado mediante DF.

A continuación, la **tabla 2.7** describe detalladamente todos los símbolos que se utilizarán para realizar diagramas de flujo.

	<p>Flechas: cualquier flecha en un DF indicará el <i>orden temporal</i> en que se ejecutarán distintas acciones (<i>flujo de ejecución</i>). Suelen graficarse con orientación descendente, con lo cual el DF comienza en su parte superior y finaliza en la inferior.</p>
	<p>Inicio del algoritmo: Aparece una única vez en el DF en su parte superior. No tiene ninguna flecha de ingreso y una única flecha de salida que dirige el flujo de ejecución hacia la primera instrucción del algoritmo. palabra “Inicio” en su interior, el nombre del algoritmo o <i>función</i> y parámetros de entrada/salida en el caso de <i>funciones</i>.</p>
	<p>Fin del algoritmo: Aparece al menos una vez en el DF, en la parte inferior del mismo. Tiene una flecha de ingreso y ninguna flecha de salida. Suele contener la palabra “Fin” en su interior.</p>

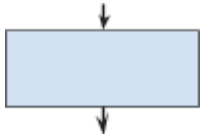
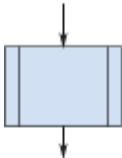

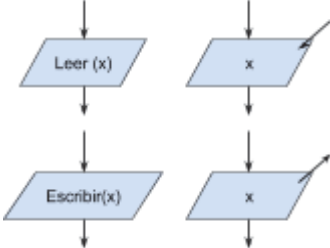
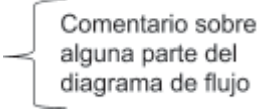

	Proceso: Operación <i>definida</i> o conjunto de operaciones. contienen <i>expresiones</i> simples (compuestas por <i>operadores</i> y <i>operandos</i>) y <i>asignaciones</i> que manipulan la información.
	Proceso definido: Ejecuta una subrutina o <i>función</i> definida en algún otro algoritmo. En este curso solemos utilizar el rectángulo normal (Proceso) en lugar del símbolo específico para el proceso definido .
	Decisión: Dentro del rombo se inscribe la <i>expresión</i> a evaluar. Se toma un camino diferente según el valor de la expresión. Si la misma es lógica hay un camino para el verdadero y otro para el falso, si es numérica los caminos posibles pueden ser varios.
	Entrada/Salida: Se representan con paralelogramos que en su interior indican la función de E/S que se ejecutará. Si no se indica el nombre de la función se sobreentiende que se están utilizando las funciones <i>Leer()</i> o <i>Escribir()</i> . Si bien no está estandarizado, el uso opcional de una flecha en el vértice superior derecho puede ayudar a distinguir si se trata de una entrada o salida.
	Comentario: brinda más información y ayuda a comprender ciertas partes del algoritmo. No se ejecutan.
	Repetición: Se utiliza para representar algunas estructuras de control repetitivas.

Tabla 2.7: Algunos de los símbolos estandarizados para diagramas de flujo.

2.4 Programación estructurada

El paradigma de la programación estructurada es uno de los dos paradigmas de programación sobre los cuales se basa este curso. Este establece que cualquier programa se puede implementar mediante un conjunto acotado de *estructuras de control de flujo* que pueden agruparse en las siguientes categorías: Estructura secuencial, estructuras selectivas y estructuras iterativas.

Esto tiene un impacto positivo en la legibilidad del código, en el tratamiento de fallos y mantenimiento, así como en el tiempo de desarrollo.

Estructura secuencial

Es la estructura que se forma naturalmente al establecer que un paso o instrucción se ejecutará a continuación de otro. En la tabla 2.8 se puede apreciar la estructura secuencial en DF y pseudocódigo con tres acciones sucesivas, las cuales pueden representar una simple instrucción o encapsular un conjunto de instrucciones.

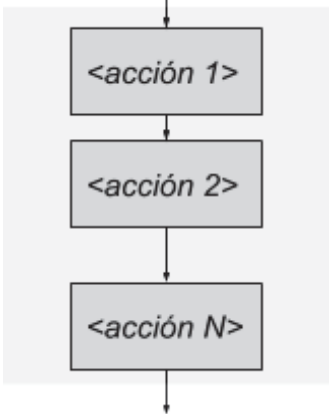
Diagrama de flujo	Pseudocódigo
	<pre><acción 1> <acción 2> <acción 3></pre>

Tabla 2.8: Representación de la estructura selectiva simple

Estructuras de control selectivas

Son estructuras donde el flujo de ejecución se bifurca según el valor de una expresión. Se consideran en este curso tres tipos de estructuras selectivas diferentes:

Selectiva simple

Evalúan una expresión lógica o condición y sólo en caso de ser verdadera se ejecuta una o un conjunto de acciones, mientras que si es falsa no se ejecuta nada. A la selectiva simple se la conoce como estructura *SI*. En la Tabla 2.9 puede observarse su implementación en pseudocódigo y en diagrama de flujo.

Diagrama de flujo	Pseudocódigo
-------------------	--------------

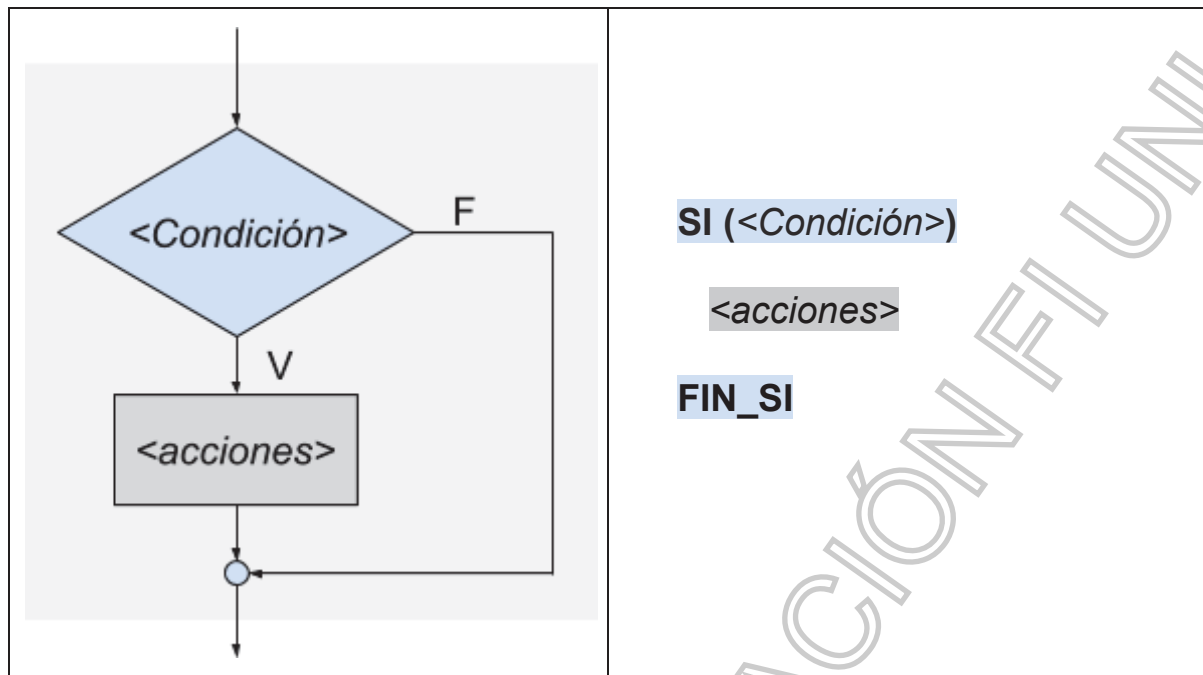


Tabla 2.9: Representación de la estructura selectiva simple

Selectiva doble

Evalúan una expresión lógica o condición y bifurcan el código en dos ramas distintas de ejecución según sea verdadera o falsa la expresión. A la selectiva doble se la conoce como estructura *SI/SINO*. En la Tabla 2.10 puede observarse su implementación en pseudocódigo y en diagrama de flujo.

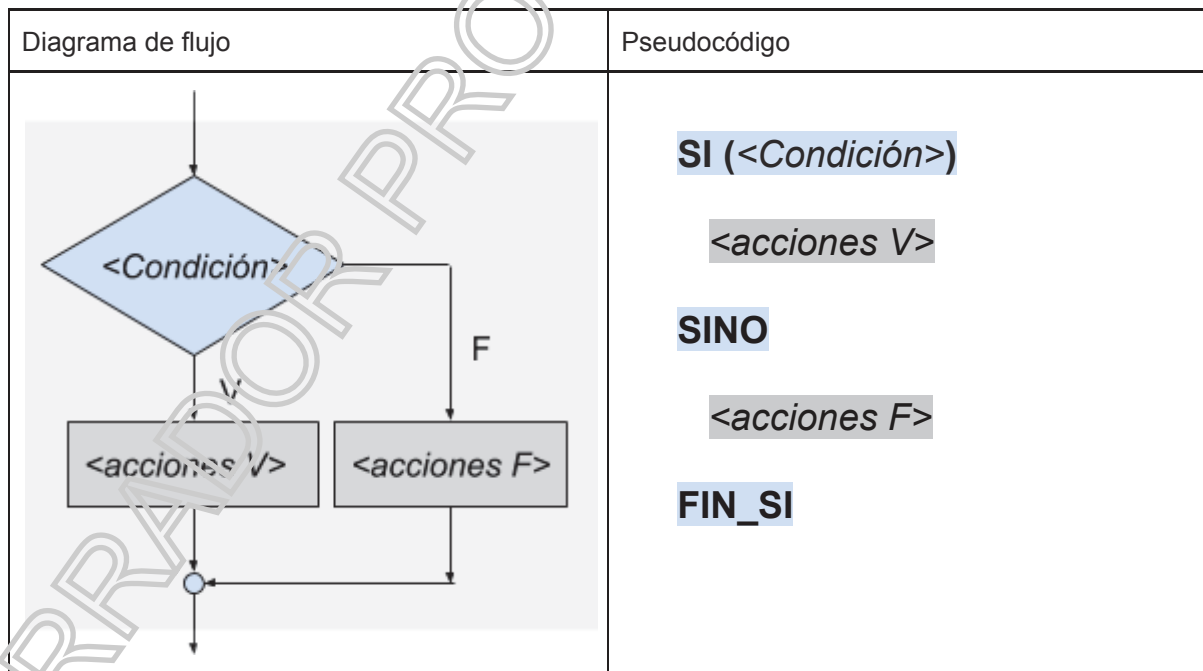


Tabla 2.10: Representación de la estructura selectiva doble

Selectiva múltiple

Evalúan una expresión entera (o de texto), que puede tomar un conjunto finito de N valores distintos. A la selectiva múltiple se la conoce como estructura *SEGÚN*. En la Tabla 2.9 puede observarse su implementación en pseudocódigo y en diagrama de flujo. La opción *otros* indica el camino de ejecución que se seguirá si el resultado de *<expresión>* no coincide con ninguno de los N valores evaluados.

Diagrama de flujo	Pseudocódigo
<pre> graph TD Entrada(()) --> Decisión{<expresión>} Decisión -- Valor1 --> Acc1[<acciones 1>] Decisión -- Valor2 --> Acc2[<acciones 2>] Decisión -- ValorN --> AccN[<acciones N>] Decisión -- otros --> AccO[<acciones o>] Acc1 --> Salida(()) Acc2 --> Salida AccN --> Salida AccO --> Salida </pre>	<pre> SEGÚN (<expresión>) Valor1: <acciones 1> Valor2: <acciones 2> ... ValorN: <acciones N> otros: <acciones o> FIN_SEGÚN </pre>

Tabla 2.11: Representación de la estructura selectiva múltiple

Ejemplo 2.4: Representar el algoritmo (en DF y pseudocódigo) de un programa que pide ingresar las notas de un alumno en los dos módulos de una materia. El programa deberá indicar si el alumno está desaprobado (alguna de las notas menor a 4), aprobado (ambas notas mayor o igual a 4 y promedio menor a 6) o promocionado (ambas notas mayor o igual a 4 y promedio mayor o igual a 6).

Resolución: se observa a continuación el algoritmo resultante, representado tanto en DF como en pseudocódigo, que consta de dos estructuras selectivas dobles. La primera en evaluarse detecta un caso de desaprobación si alguna de las dos notas es menor a 4. De ser esto verdadero, se le asigna la cadena “desaprobado” a la variable de salida *estado*, se sale de la estructura selectiva y se ejecuta la última instrucción del algoritmo que muestra el resultado en pantalla. En caso de ser falsa la condición mencionada, habrá que distinguir entre los estados de aprobación y promoción, y para esto hace falta una segunda estructura selectiva doble, que está *anidada* dentro de la primera. Con fines exclusivamente didácticos se indica con números en el DF la línea equivalente en el pseudocódigo.



```

0  INICIO Aprobación
1  Variables:
2      reales: M1,M2,promedio
3      cadena: estado
4  Algoritmo:
5      /* M1 y M2 son las notas en el
6       primero y en el segundo módulo
7       respectivamente */
8      Leer M1, M2
9      Si M1<4 o M2<4
10         estado = "desaprobado"
11     Sino
12         promedio = (M1+M2)/2
13         Si promedio <6
14             estado = "aprobado"
15         Sino
16             estado = "promocionado"
17     Fin_Si
18 Fin_Si
19 Escribir ("El alumno está ",estado)
20 FIN
  
```

Ejemplo 2.5: Representar el algoritmo en pseudocódigo de un programa que muestre en pantalla un menú con 4 opciones:

- 1) sumar 2 números
- 2) restar 2 números
- 3) multiplicar 2 números
- 4) dividir 2 números

El programa deberá pedir al usuario que ingrese la opción deseada y luego los valores con los cuales realizar la operación elegida. Por simplicidad, se asume que el usuario siempre ingresará una opción válida.

Resolución: El problema propuesto puede resolverse mediante el uso de una selectiva múltiple que en función de la opción elegida realice la operación aritmética correspondiente.

```

0  INICIO
1  Variables:
2      reales: a,b,res
3      entero: opción
4  Algoritmo:
5      Escribir "1)sumar 2 números",NL
6      Escribir "2)restar 2 números",NL
7      Escribir "3)multiplicar 2 números",NL
8      Escribir "4)dividir 2 números",NL
9      Escribir "Ingrese la opción deseada: "
10     Leer opción
11     Leer a,b
12
13     Según(opción)
14     1:
15         res = a + b
16     2:
17         res = a - b
18     3:
19         res = a * b
  
```

```

20      4:
21      res = a / b
22      Fin_Según
23      Escribir("El resultado es ", res)
24      FIN

```

Estructuras de control iterativas

Son las estructuras que permiten repetir la ejecución de determinada parte del código. Dentro de las estructuras iterativas trabajaremos con tres variantes distintas.

Mientras

Evalúa una expresión lógica o condición y en caso de ser verdadera se ejecuta una o un conjunto de acciones, luego de dicha ejecución se vuelve a evaluar la condición mencionada, produciendo que mientras dicha expresión lógica sea verdadera, las instrucciones contenidas por la estructura de control se repitan. Cuando el resultado de la expresión lógica es falso se sale de la estructura de control.

En la Tabla 2.11 puede observarse su implementación en pseudocódigo y en diagrama de flujo.

Diagrama de flujo	Pseudocódigo
<pre> graph TD Entry(()) --> Cond{<Condición>} Cond -- V --> Acciones[<acciones>] Acciones --> Cond Cond -- F --> Exit(()) </pre>	<pre> MIENTRAS (<Condición>) <acciones> FIN_MIENTRAS </pre>

Tabla 2.11: Representación de la estructura iterativa Mientras

Hacer-Mientras

Primero ejecuta una o un conjunto de acciones, contenidas en la estructura de control, y luego se evalúa una expresión lógica o condición. En caso de ser verdadera se ejecuta nuevamente el conjunto de acciones, produciendo que mientras dicha expresión lógica sea verdadera, las instrucciones contenidas por la estructura de control se repitan. Cuando el resultado de la expresión lógica es falso se sale de la estructura de control.

. En la Tabla 2.12 puede observarse su implementación en pseudocódigo y en diagrama de flujo.

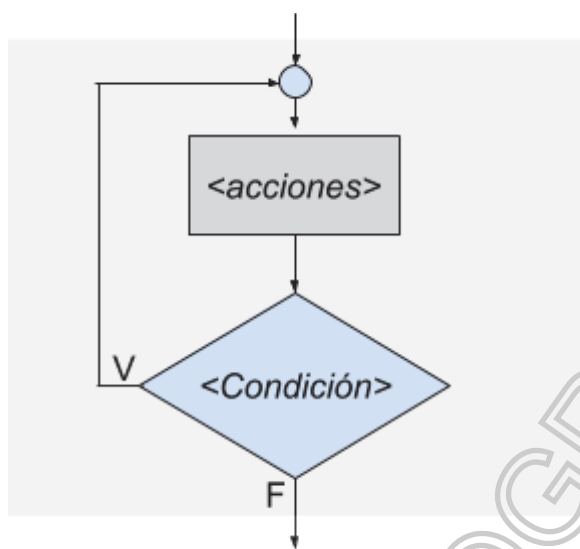
Diagrama de flujo	Pseudocódigo
	HACER <acciones> MIENTRAS (<Condición>)

Tabla 2.12: Representación de la estructura iterativa Hacer-Mientras

La diferencia entre las estructuras MIENTRAS y HACER-MIENTRAS está en que en el MIENTRAS, primero se verifica la condición y solo si esta es verdadera se ejecutan las acciones contenidas, pero en el HACER-MIENTRAS primero se ejecutan las acciones y luego se verifica la condición para repetir las acciones. Por lo tanto, el HACER-MIENTRAS asegura que al menos una vez las acciones contenidas serán ejecutadas, mientras que el MIENTRAS no.

PARA

La estructura PARA suele utilizarse para repetir una o un conjunto de acciones una cantidad de veces conocida. Tiene asociada una variable que se utiliza para controlar las iteraciones, para la cual se indica el valor inicial, el valor final y el incremento o decremento que esta sufre entre repeticiones.

En la Tabla 2.13 puede observarse su implementación en pseudocódigo y en diagrama de flujo.

Diagrama de flujo	Pseudocódigo
-------------------	--------------

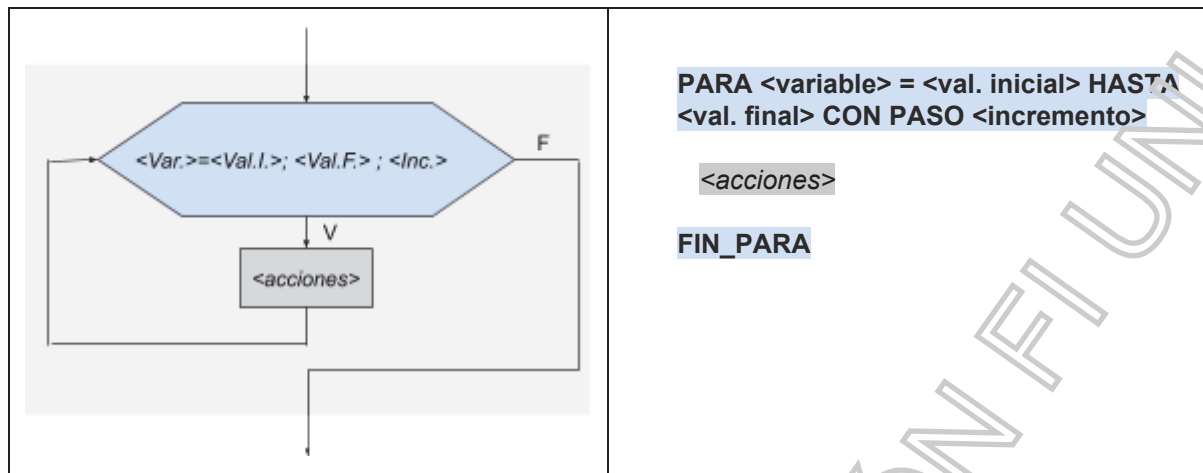


Tabla 2.13: Representación de la estructura iterativa Para

Ejemplo 2.6: Realice el pseudocódigo y el DF de un programa que escriba en pantalla los números pares entre 2 y 10, utilizando:

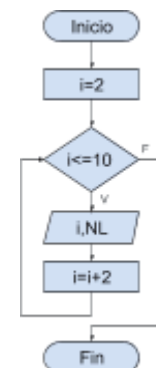
- MIENTRAS
- HACER-MIENTRAS
- PARA

Resolución:

a)

```

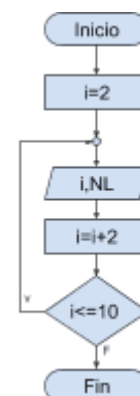
0 INICIO
1   Variables:
2       entero: i
3   Algoritmo:
4       i=2
5       Mientras i <= 10
6           Escribir(i,NL)
7           i=i+2
8       Fin_Mientras
9 FIN
  
```



b)

```

0 INICIO
1   Variables:
2       entero: i
3   Algoritmo:
4       i=2
5       Hacer
6           Escribir(i,NL)
7           i=i+2
8       Mientras i <= 10
9 FIN
  
```

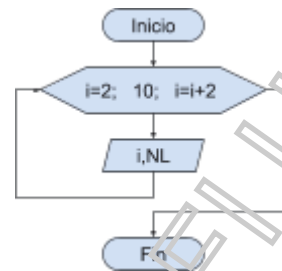


c)

```

0 INICIO
1   Variables:
2     entero: i
3   Algoritmo:
4     Para i=2 hasta 10 con paso 2
5       Escribir(i,NL)
6     Fin_Para
7 FIN

```



Ejemplo 2.7: Repita los pseudocódigos del **ejemplo 2.6** pero ahora imprima los números pares entre 2 y N, siendo N un valor ingresado por el usuario.

Resolución: Se muestran en rojo las modificaciones realizadas a la resolución del **ejemplo 2.6**

a)

```

0 INICIO
1   Variables:
2     entero: i,N
3   Algoritmo:
4     Leer(N)
5     i=2
6     Mientras i <= N
7       Escribir(i,NL)
8       i=i+2
9     Fin_Mientras
10  FIN

```

b)

```

0 INICIO
1   Variables:
2     entero: i
3   Algoritmo:
4     Leer(N)
5     i=2
6     Hacer
7       Escribir(i,NL)
8       i=i+2
9     Mientras i <= N
10  FIN

```

c)

```

0 INICIO
1   Variables:
2     entero: i,N
3   Algoritmo:
4     Leer(N)
5     Para i=2 hasta N con paso 2
6       Escribir(i,NL)

```


2.5 Programación modular

La programación modular es un paradigma de programación que también aplicaremos durante este curso, además de la ya mencionada programación estructurada.

Este paradigma se basa en la aplicación del principio “divide y reinarás” a problemas complejos, los cuales son divididos en un conjunto de problemas más sencillos. Cada uno de dichos problemas sencillos es considerado un “módulo”, el cual es resuelto individualmente mediante un algoritmo particular. La solución al problema complejo más general resulta entonces de juntar todos los “módulos” y ejecutar dichos algoritmos coordinadamente desde un *programa principal*.

Son varias las ventajas de aplicar el paradigma de la programación modular, entre ellas:

- Promueve el diseño *top-down*, permitiendo el diseño de soluciones generales que hacen uso de bloques abstractos, que son implementados posteriormente.
- Facilita el trabajo en equipo, haciendo que cada programador se centre en la resolución de un problema particular.
- Permite la reutilización de código.
- Facilita el análisis del código, su mantenimiento y depuración de errores.

Concepto de función

La herramienta que brindan los distintos lenguajes de programación para implementar los mencionados “módulos” se denomina *función*, aunque también suelen usarse los nombres *procedimiento*, *subrutina* o *subprograma*.

Como se muestra en la Fig. 2.5, cada función será un algoritmo, según lo hemos definido anteriormente, que puede recibir datos de entrada (parámetros o argumentos) y entregar datos de salida como resultado del procesamiento (valor de retorno).

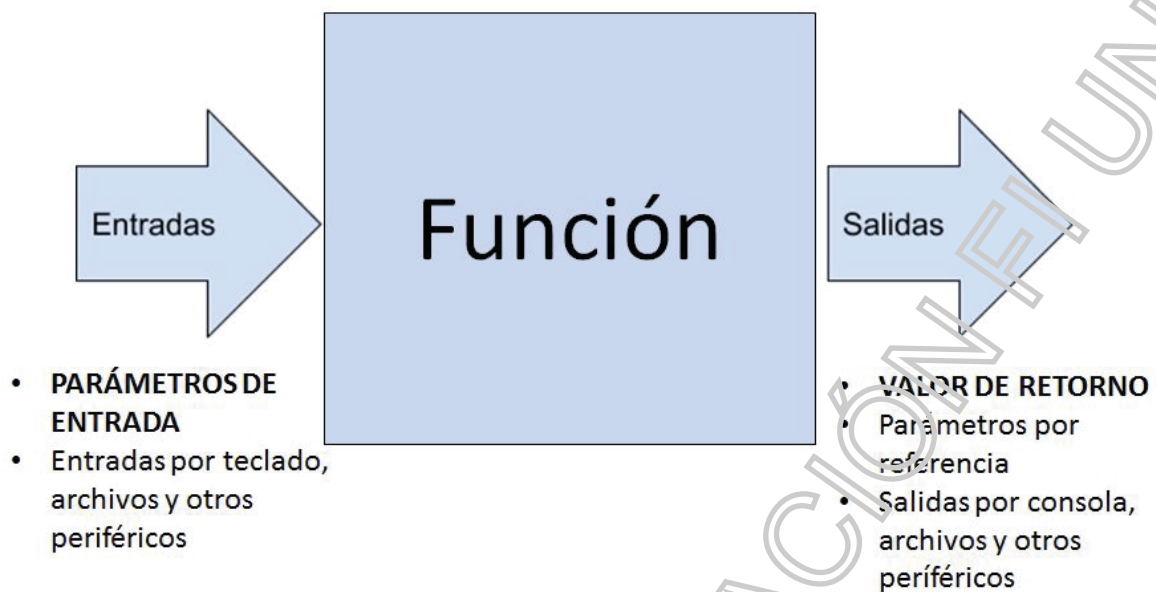


FIG. 2.5: Las funciones implementan algoritmos que, además de las entradas y salidas típicas, cuentan con entradas (parámetros) y salidas (parámetros/valor de retorno) especiales.

Una vez definido el algoritmo de una función, esta puede ser *utilizada (ejecutada, invocada o llamada)* desde otras partes del programa, cuantas veces se lo desee.

Definición de una función

Simplemente para distinguir el algoritmo de un programa principal de los demás sub-algoritmos o funciones, usaremos las palabras clave **FUNCIÓN** y **FIN_FUNCIÓN** para definir a estas últimas. Así mismo las funciones deben tener un *nombre* que las identifique y una lista de los parámetros o variables de entrada, si los hubiese. Si la función devolviese algún valor como resultado, se utiliza la palabra clave **RETORNAR** acompañada del valor de salida correspondiente, la cual hace que finalice el algoritmo de la función.

En el **listado 2.2** se muestra el algoritmo que calcula la raíz cuadrada de un número (**ejemplo 2.3**) convertido en una función. Pero, a diferencia del programa original, la función no tiene interacción con el usuario, sino que recibe la variable *x* como un parámetro de entrada, mientras que la raíz calculada en la variable *r* se retorna como resultado de la función.

```

0  FUNCION Raíz_cuadrada(real: x)
1      Variables:
2          real: r
3      Algoritmo:
4          r = x
5          Mientras r*r <> x
6              r = (x/r + r)/2
7          Fin_Mientras
8
9      RETORNAR r
10  FIN_FUNCION

```

Listado 2.2: Definición de la función Raíz_cuadrada()

Uso de una función

Para usar una función desde el algoritmo de otra solo se debe escribir su nombre acompañado por un par de paréntesis con los valores pasados como parámetros. Si la función no recibe parámetros de entrada los paréntesis irán vacíos.

Por ejemplo, para calcular la raíz cuadrada de 121 en cualquier parte de un programa solo será necesaria la siguiente línea de código:

```
Raíz_cuadrada(121)
```

Esto hará que se invoque la ejecución del algoritmo definido en el **listado 2.2** el cual se ejecutará de inicio a fin, asignando a la variable de entrada `x` el valor 121 pasado como parámetro. Una vez finalizada la ejecución de la función, la llamada es reemplazada por el valor retornado (11), por lo que la instrucción anterior por si sola no tiene ninguna utilidad, pero podría utilizarse como parte de otras expresiones, por ejemplo:

```

y = Raíz_cuadrada(121)      /* asigna 11 a una variable y */
Escribir(Raíz_cuadrada(121)) /* muestra el 11 en pantalla */

```

Suponga que existe una función llamada Hipotenusa() que recibe como parámetro la base y la altura de un triángulo rectángulo y devuelve la hipotenusa del mismo. Escriba el pseudocódigo de un programa que reciba por teclado la base y la altura de un triángulo rectángulo e imprima en pantalla el valor de la hipotenusa.

Aunque aún no esté definida la función Hipotenusa(), podemos usarla para escribir el pseudocódigo del programa principal, desde el cual será invocada. Esta forma de programar permite ir de lo más general a lo más particular haciendo que el programa principal sea más legible.

```

0  INICIO
1      Variables:
2          reales: b,a
3      Algoritmo:

```

```
4      Escribir("Ingrese base y altura:")
5      Leer(b)
6      Leer(a)
7      Escribir("La hipotenusa es ",Hipotenusa(b,a))
8  FIN
```

Ejemplo 2.10: Escriba el pseudocódigo de la función Hipotenusa() invocada en el ejemplo anterior

Resolución:

```
0  FUNCION Hipotenusa(real: base, real: altura)
1      RETORNAR Raíz_cuadrada(base^2 + altura^2)
2  FIN_FUNCION
```