CAPÍTULO 4

Flujo de control I: Estructuras selectivas

INTRODUCCIÓN

En la actualidad, dado el tamaño considerable de las memorias centrales y las altas velocidades de los procesadores —Intel Core 2 Duo, AMD Athlon 64, AMD Turion 64, etc.—, *el estilo de escritura de los programas se vuelve una de las características más sobresalientes en las técnicas de programación. La legibilidad de los algoritmos y posteriormente de los programas* exige que su diseño sea fácil de comprender y su flujo lógico fácil de seguir. La *programación modular* enseña la descomposición de un programa en módulos más simples de programar, y la *programación estructurada* permite la escritura de programas fáciles de leer y modificar. En *un programa estructurado* el flujo lógico se gobierna por las estructuras de control básicas:

1. *Secuenciales.*
2. *Repetitivas.*
3. *Selectivas.*

En este capítulo se introducen las estructuras selectivas que se utilizan para controlar el orden en que se ejecutan las sentencias de un programa. Las sentencias **`si`** (*en inglés,* "**`if`**") y sus variantes, **`si-entonces`**, **`si-entonces-sino`** y la sentencia **`según-sea`** (*en inglés*, "**`switch`**") se *describen* como parte fundamental de un programa. Las sentencias **`si`** anidadas y las sentencias de multibifurcación pueden ayudar a resolver importantes problemas de cálculo. Asimismo se describe la "tristemente famosa" sentencia **`ir-a`** (*en inglés* "**`goto`**"), cuyo uso se debe evitar en la mayoría de las situaciones, pero cuyo significado debe ser muy bien entendido por el lector, precisamente para evitar su uso, aunque puede haber una situación específica en que no quede otro remedio que recurrir a ella.

El estudio de las estructuras de control se realiza basado en las herramientas de programación ya estudiadas: diagramas de flujo, diagramas N-S y pseudocódigos.

4.1. EL FLUJO DE CONTROL DE UN PROGRAMA

Muchos avances han ocurrido en los fundamentos teóricos de programación desde la aparición de los lenguajes de alto nivel a finales de la década de los cincuenta. Uno de los más importantes avances fue el reconocimiento a finales de los sesenta de que cualquier algoritmo, no importaba su complejidad, podía ser construido utilizando combinaciones de tres estructuras de control de flujo estandarizadas (*secuencial*, *selección*, *repetitiva* o *iterativa*) y una cuarta denominada, *invocación* o *salto ("jump").* Las sentencias de *selección son:* **`si`** (**`if`**) y **`según-sea`** (**`switch`**); las *sentencias de repetición o iterativas* son: **`desde`** (**`for`**), **`mientras`** (**`while`**), **`hacer-mientras`** (**`do-while`**) o **`repetir-hasta que`** (**`repeat-until`**); las sentencias de salto o bifurcación incluyen **`romper`** (**`break`**), **`continuar`** (**`continue`**), **`ir-a`** (**`goto`**), **`volver`** (**`return`**) y **`lanzar`** (**`throw`**).

El término **flujo de control** se refiere al orden en que se ejecutan las sentencias del programa. Otros términos utilizados son *secuenciación* y *control del flujo*. A menos que se especifique expresamente, el flujo normal de control de todos los programas es el **secuencial**. Este término significa que las sentencias se ejecutan en secuencia, una después de otra, en el orden en que se sitúan dentro del programa. Las estructuras de selección, repetición e invocación permiten que el flujo secuencial del programa sea modificado en un modo preciso y definido con anterioridad. Como se puede deducir fácilmente, las estructuras de selección se utilizan para seleccionar cuáles sentencias se han de ejecutar a continuación y las estructuras de repetición (repetitivas o iterativas) se utilizan para repetir un conjunto de sentencias.

Hasta este momento, todas las sentencias se ejecutaban secuencialmente en el orden en que estaban escritas en el código fuente. Esta ejecución, como ya se ha comentado, se denomina *ejecución secuencial.* Un programa basado en ejecución secuencial, siempre ejecutará exactamente las mismas acciones; es incapaz de reaccionar en respuesta a condiciones actuales. Sin embargo, la vida real no es tan simple. Normalmente, los programas necesitan alterar o modificar el flujo de control en un programa. Así, en la solución de muchos problemas se deben tomar acciones diferentes dependiendo del valor de los datos. Ejemplos de situaciones simples son: cálculo de una superficie *sólo* ***si*** las medidas de los lados son positivas; la ejecución de una división se realiza, *sólo* ***si*** el divisor no es cero; la visualización de mensajes diferentes *depende* del valor de una nota recibida, etc.

Una ***bifurcación*** ("*branch*", en inglés) es un segmento de programa construida con una sentencia o un grupo de sentencias. Una *sentencia de bifurcación* se utiliza para ejecutar una sentencia de entre varias o bien bloques de sentencias. La elección se realiza dependiendo de una condición dada. Las *sentencias de bifurcación* se llaman también *sentencias de selección* o *sentencias de alternación o alternativas.*

4.2. ESTRUCTURA SECUENCIAL

Una **estructura secuencial** es aquella en la que una acción (instrucción) sigue a otra en secuencia. Las tareas se suceden de tal modo que la salida de una es la entrada de la siguiente y así sucesivamente hasta el final del proceso. La estructura secuencial tiene una entrada y una salida. Su representación gráfica se muestra en las Figuras 4.1, 4.2 y 4.3.

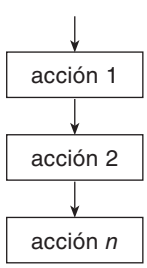


Figura 4.1. Estructura secuencial.

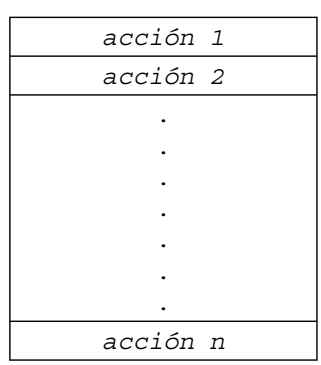


Figura 4.2. Diagrama N-S de una estructura secuencial.

```
inicio
   <acción 1>
   <acción 2>
fin
```

Figura 4.3. Pseudocódigo de una estructura secuencial.

EJEMPLO 4.1

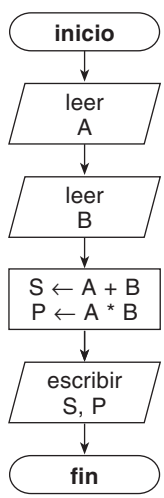
Cálculo de la suma y producto de dos números.

La suma `S` de dos números es `S = A+B` y el producto `P` es `P = A*B`. El pseudocódigo y el diagrama de flujo correspondientes se muestran a continuación:

Pseudocódigo

```
inicio
  leer(A)
  leer(B)
  S ← A + B
  P ← A * B
  escribir(S, P)
fin
```

Diagrama de flujo



EJEMPLO 4.2

Se trata de calcular el salario neto de un trabajador en función del número de horas trabajadas, precio de la hora de trabajo y, considerando unos descuentos fijos, el sueldo bruto en concepto de impuestos (20 por 100).

Pseudocódigo

```
inicio
  // cálculo salario neto
  leer(nombre, horas, precio_hora)
  salario_bruto ← horas * precio_hora
  impuestos ← 0.20 * salario_bruto
  salario_neto ← salario_bruto - impuestos
  escribir(nombre, salario_bruto, salario_neto)
fin
```

Diagrama de flujo

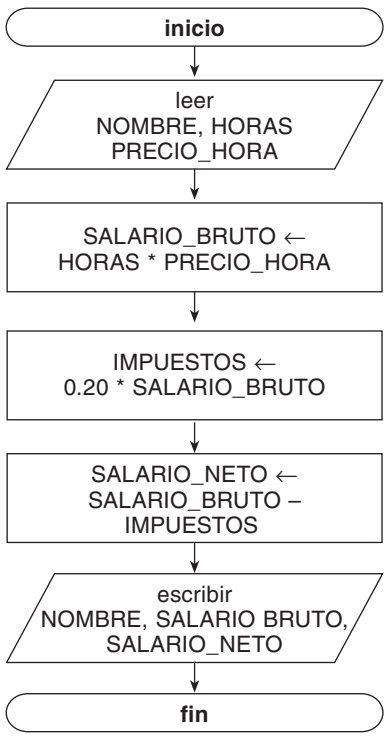


Diagrama N-S

```
leer
nombre, horas, precio
salario_bruto ← horas * precio
impuestos ← 0.20 * salario_bruto
salario_neto ← salario_bruto - impuestos
escribir
nombre, salario_bruto, salario_neto
```

4.3. ESTRUCTURAS SELECTIVAS

La especificación formal de algoritmos tiene realmente utilidad cuando el algoritmo requiere una descripción más complicada que una lista sencilla de instrucciones. Este es el caso cuando existen un número de posibles alternativas resultantes de la evaluación de una determinada condición. Las estructuras selectivas se utilizan para tomar decisiones lógicas; de ahí que se suelan denominar también *estructuras de decisión o alternativas*.

En las estructuras selectivas se evalúa una condición y en función del resultado de la misma se realiza una opción u otra. Las condiciones se especifican usando expresiones lógicas. La representación de una estructura selectiva se hace con palabras en pseudocódigo (**`if`**, **`then`**, **`else`** o bien en español **`si`**, **`entonces`**, **`si_no`**), con una figura geométrica en forma de rombo o bien con un triángulo en el interior de una caja rectangular. Las estructuras selectivas o alternativas pueden ser:

- *simples,*
- *dobles,*
- *múltiples*.

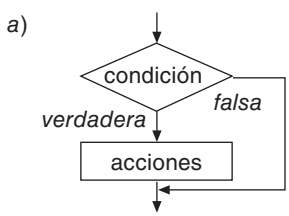
La estructura simple es **si** (**if**) con dos formatos: *Formato* Pascal, **si-entonces** (**if-then**) y *formato C,* **si** (**if**). La estructura selectiva doble es igual que la estructura simple **si** a la cual se le añade la cláusula `si-no` (**else**). La estructura selectiva múltiple es **según_sea** (**switch** en lenguaje **C**, **case** en **Pascal**).

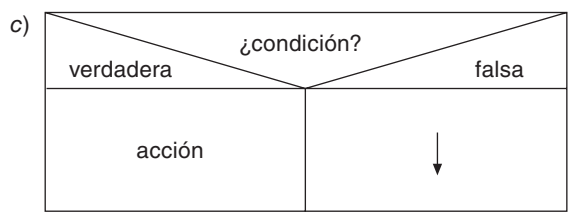
4.4. ALTERNATIVA SIMPLE (SI-ENTONCES/IF-THEN)

La estructura alternativa simple **si-entonces** (en inglés **if-then**) ejecuta una determinada acción cuando se cumple una determinada condición. La selección **si-entonces** evalúa la condición y

- si la condición es *verdadera*, entonces ejecuta la acción `S1` (o acciones caso de ser `S1` una acción compuesta y constar de varias acciones),
- si la condición es *falsa*, entonces no hacer nada.

Las representaciones gráficas de la estructura condicional simple se muestran en la Figura 4.4.





b) Pseudocódigo en español

```
si <condición> entonces
   <acción S1>
fin _ si
```

Pseudocódigo en inglés

```
if <condición> then
   <acción S1>
endif
```

Pseudocódigo en español

```
//S1 accion compuesta
si <condición> entonces
   <acción S11>
   <acción S12>
      .
      .
      .
   <acción S1n>
fin _ si
```

Figura 4.4. Estructuras alternativas simples: *a*) Diagrama de flujo; *b*) Pseudocódigo; *c*) Diagrama N-S.

Obsérvese que las palabras del pseudocódigo **si** y **fin_si** se alinean verticalmente *indentando* (sangrando) la <*acción*> o bloque de acciones.

Diagrama de sintaxis

Sentencia **if_simple::=**

```
1. si (<expresión_lógica>)
   inicio
      <sentencia>
   fin

   Sentencia_compuesta ::=
                       inicio
                            <Sentencias>
                       fin
```

```
2. si <expresión lógica> entonces
      <Sentencia_compuesta>
   fin-si
```

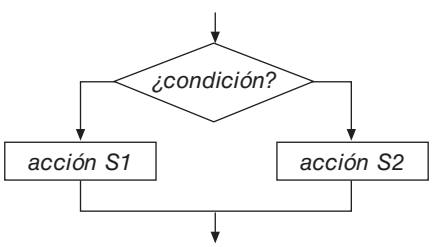
Sintaxis en lenguajes de programación

Pseudocódigo	Pascal	C/C++

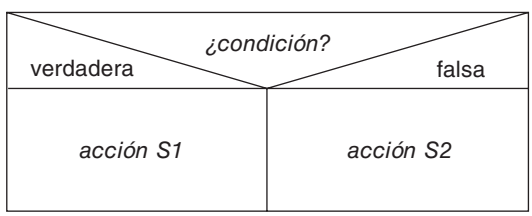
si (*condición*) **entonces**
  acciones
fin-si
	if *(condición)* **then**
	begin
	  sentencias
	end
		if *(condición)*
		{
		  sentencias
		}

4.4.1. Alternativa doble (`si-entonces-sino/if-then-else`)

La estructura anterior es muy limitada y normalmente se necesitará una estructura que permita elegir entre dos opciones o alternativas posibles, en función del cumplimiento o no de una determinada condición. Si la condición `C` es verdadera, se ejecuta la acción `S1` y, si es falsa, se ejecuta la acción `S2` (véase Figura 4.5).



a)



c)

Pseudocódigo en español

```
si <condicion> entonces
   <accion S1>
si _ no
   <accion S2>
fin _ si
```

Pseudocódigo en inglés

```
if <condicion> then
   <accion S1>
else
   <accion S2>
endif
```

Pseudocódigo en español

```
//S1 accion compuesta
si <condicion> entonces
   <accion S11>
   <accion S12>
   .
   .
   .
   <acción S1n>
si _ no
   <accion S21>
   <accion S22>
   .
   .
   .
   <acción S2n>
fin _ si
```

b)

Figura 4.5. Estructura alternativa doble: *a*) diagrama de flujo; *b*) pseudocódigo; *c*) diagrama N-S.

Obsérvese que en el pseudocódigo las acciones que dependen de **`entonces`** y **`si_no`** están *indentadas* en relación con las palabras **`si`** y **`fin_si`**; este procedimiento aumenta la legibilidad de la estructura y es el medio más idóneo para representar algoritmos.

EJEMPLO 4.3

Resolución de una ecuación de primer grado.

Si la ecuación es $ax + b = 0$, a y b son los datos, y las posibles soluciones son:

```
• a <> 0                x = -b/a
• a = 0 b <> 0          entonces "solución imposible"
• a = 0 b = 0           entonces "solución indeterminada"
```

El algoritmo correspondiente será

```
algoritmo RESOL1
var
  real : a, b, x
inicio
  leer (a, b)
  si a <> 0 entonces
     x ← -b/a
     escribir(x)
  si_no
     si b <> 0 entonces
         escribir ('solución imposible')
     si_no
         escribir ('solución indeterminada')
     fin_si
  fin_si
fin
```

EJEMPLO 4.4

Calcular la media aritmética de una serie de números positivos.

La media aritmética de n números es

$$\frac{x1 + x2 + x3 + ... + xn}{n}$$

En el problema se supondrá la entrada de datos por el teclado hasta que se introduzca el último número, en nuestro caso `-99`. Para calcular la media aritmética se necesita saber cuántos números se han introducido hasta llegar a `-99`; para ello se utilizará un contador *n* que llevará la cuenta del número de datos introducidos.

Tabla de variables
`real:` `s` (suma)
`entera:` `n` (contador de números)
`real:` `m` (media)

```
algoritmo media
var
  real: s, m
  entera: n
```

```
inicio
  s ← 0 // inicialización de variables : s y n
  n ← 0
datos:
    leer (x) // el primer número ha de ser mayor que cero
    si x < 0 entonces
      ir_a(media)
    si_no
      n ← n + 1
      s ← s + x
      ir_a(datos)
    fin_si
media:
    m ← s/n // media de los números positivos
    escribir (m)
  fin
```

En este ejemplo se observa una bifurcación hacia un punto referenciado por una etiqueta alfanumérica denominada media y otro punto referenciado por datos.

Trate el alumno de simplificar este algoritmo de modo que sólo contenga un punto de bifurcación.

EJEMPLO 4.5

Se desea obtener la nómina semanal —salario neto— de los empleados de una empresa cuyo trabajo se paga por horas y del modo siguiente:

- *las horas inferiores o iguales a 35 horas (normales) se pagan a una tarifa determinada que se debe introducir por teclado al igual que el número de horas y el nombre del trabajador,*
- *las horas superiores a 35 se pagarán como extras a un promedio de 1,5 horas normales,*
- *los impuestos a deducir a los trabajadores varían en función de su sueldo mensual:*
  - *sueldo <= 2.000, libre de impuestos,*
  - *las siguientes 220 euros al 20 por 100,*
  - *el resto, al 30 por 100.*

Análisis

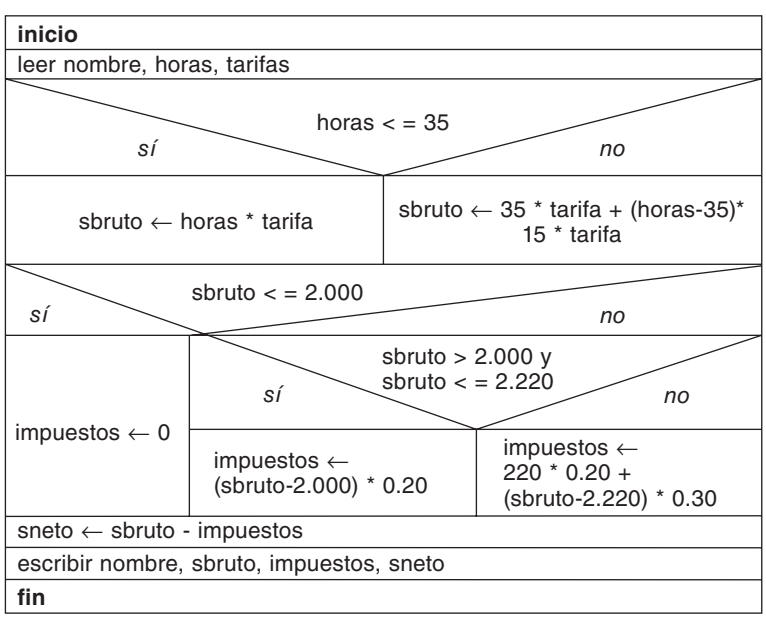
Las operaciones a realizar serán:

```
1. Inicio.
2. Leer nombre, horas trabajadas, tarifa horaria.
3. Verificar si horas trabajadas <= 35, en cuyo caso
      salario_bruto = horas * tarifa; en caso contrario,
      salario_bruto = 35 * tarifa + (horas - 35) * tarifa.
4. Cálculo de impuestos
      si salario_bruto <= 2.000, entonces impuestos = 0
      si salario_bruto <= 2.220 entonces
      impuestos = (salario_bruto - 2.000) * 0.20
      si salario_bruto > 2.220 entonces
      impuestos = (salario_bruto - 2.220) * 0.30 + (220 * 0.20)
5. Cálculo del salario_neto
      salario_neto = salario_bruto - impuestos.
6. Fin.
```

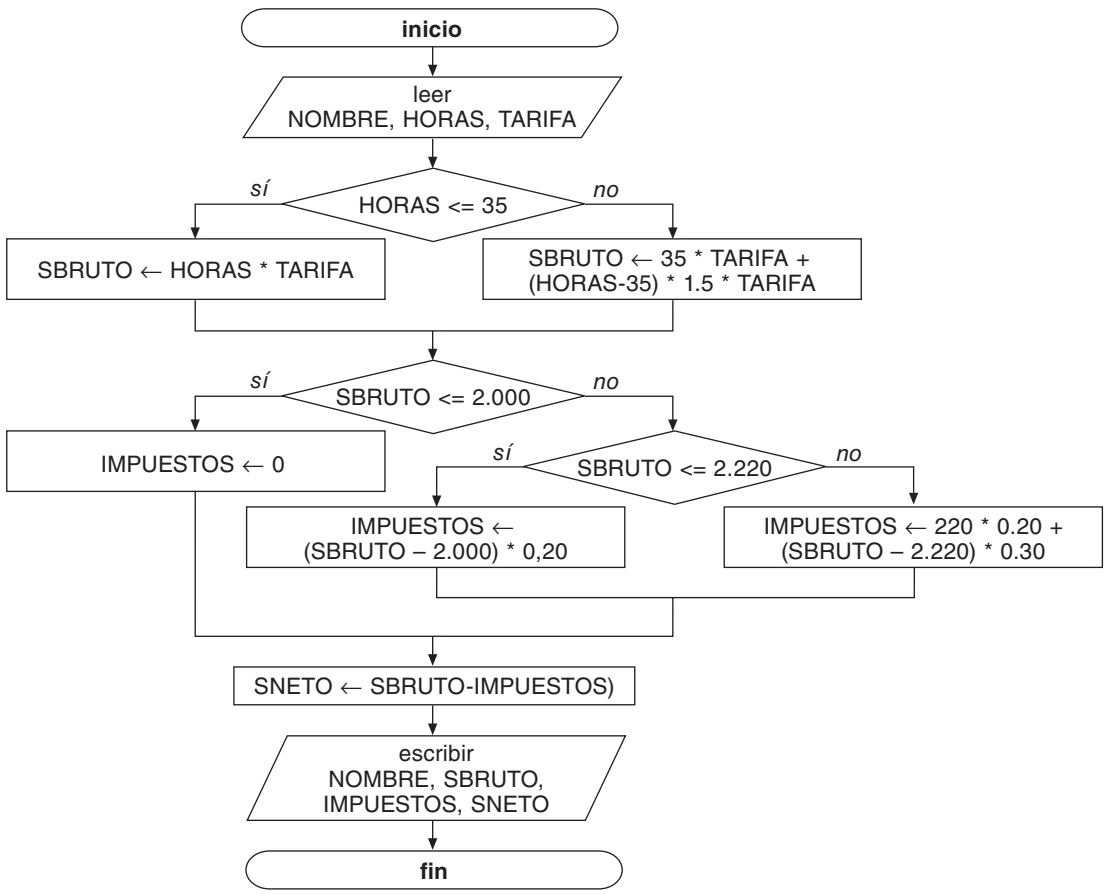
Representación del algoritmo en pseudocódigo

```
algoritmo Nómina
var
  cadena : nombre
  real : horas, impuestos, sbruto, sneto
inicio
  leer(nombre, horas, tarifa)
  si horas <= 35 entonces
    sbruto ← horas * tarifa
  si_no
    sbruto ← 35 * tarifa + (horas - 35) * 1.5 * tarifa
  fin_si
  si sbruto <= 2.000 entonces
    impuestos ← 0
  si_no
    si (sbruto > 2.000) y (sbruto <= 2.220) entonces
      impuestos ← (sbruto - 2.000) * 0.20
    si_no
      impuestos ← (220 * 0.20) + (sbruto - 2.220)
    fin_si
  fin_si
  sneto ← sbruto - impuestos
  escribir(nombre, sbruto, impuestos, neto)
fin
```

Representación del algoritmo en diagrama N-S



Representación del algoritmo en diagrama de flujo



EJEMPLOS 4.6

Empleo de estructura selectiva para detectar si un número tiene o no parte fraccionaria.

```
algoritmo Parte_fraccionaria
var
  real : n
inicio
  escribir('Deme numero ')
  leer(n)
  si n = trunc(n) entonces
      escribir('El numero no tiene parte fraccionaria')
  si_no
      escribir('Numero con parte fraccionaria')
  fin_si
fin
```

EJEMPLOS 4.7

Estructura selectiva para averiguar si un año leído de teclado es o no bisiesto.

```
algoritmo Bisiesto
var
```

```
  entero : año
inicio
  leer(año)
  si (año MOD 4 = 0) y (año MOD 100 <> 0) 0 (año MOD 400 = 0) entonces
    escribir('El año ', año, ' es bisiesto')
  si_no
    escribir('El año ', año, ' no bisiesto')
  fin_si
fin
```

EJEMPLOS 4.8

Algoritmo que nos calcule el área de un triángulo conociendo sus lados. La estructura selectiva se utiliza para el control de la entrada de datos en el programa.

Nota: $Area = \sqrt{p.(p-a)\cdot(p-b)\cdot(p-c)}$ $p = (a + b + c)/2$

```
algoritmo Area_triangulo
var
  real : a,b,c,p,area
inicio
  escribir('Deme los lados ')
  leer(a,b,c)
  p ← (a + b + c) / 2
  si (p > a) y (p > b) y (p > c) entonces
     area ← raiz2(p * (p - a) * (p - b) * (p - c))
     escribir(area)
  si_no
    escribir('No es un triangulo')
  fin_si
fin
```

4.5. ALTERNATIVA MÚLTIPLE (`según_sea, caso de/case`)

Con frecuencia —en la práctica— es necesario que existan más de dos elecciones posibles (por ejemplo, en la resolución de la ecuación de segundo grado existen tres posibles alternativas o caminos a seguir, según que el discriminante sea negativo, nulo o positivo). Este problema, como se verá más adelante, se podría resolver por estructuras alternativas simples o dobles, *anidadas* o *en cascada*; sin embargo, este método si el número de alternativas es grande puede plantear serios problemas de escritura del algoritmo y naturalmente de legibilidad.

La estructura de decisión múltiple evaluará una expresión que podrá tomar *n* valores distintos, 1, 2, 3, 4, ..., *n*. Según que elija uno de estos valores en la condición, se realizará una de las *n* acciones, o lo que es igual, el flujo del algoritmo seguirá un determinado camino entre los *n* posibles.

Los diferentes modelos de pseudocódigo de la estructura de decisión múltiple se representan en las Figuras 4.6 y 4.7.

***Sentencia* `switch` (C , C++, Java, C#)**

```
switch (expresión)
{
  case valor1:
    sentencia1;
    sentencia2;
    sentencia3;
```

Modelo 1:

```
según_sea expresion (E) hacer
     e1: accion S11
         accion S12
         .
         .
         accion S1a
     e2: accion S21
         accion S22
         .
         .
         accion S2b
     .
     .
     en: accion S31
         accion S32
         .
         .
         accion S3p
     si-no
         accion Sx
fin_según
```

Modelo 2 (simplificado):

```
según E hacer
       .
       .
       .
fin_según
```

Modelo 3 (simplificado):

```
opción E de
 .
 .
fin_opción
```

Modelo 4 (simplificado):

```
caso_de E hacer
 .
 .
 .
fin_caso
```

Modelo 5 (simplificado):

```
si E es n hacer
 .
 .
 .
fin_si
```

Figura 4.6. Estructuras de decisión múltiple.

Modelo 6:

```
según_sea (expresión) hacer
   caso expresión constante :
      [Sentencia
       sentencia
       ...
       sentencia de ruptura | sentencia ir_a ]
   caso expresión constante :
      [Sentencia
       sentencia
       ...
       sentencia de ruptura | sentencia ir_a ]
   caso expresión constante :
      [Sentencia
       ...
       sentencia
       sentencia de ruptura | sentencia ir_a ]
   [otros:
      [Sentencia
       ...
       sentencia
       sentencia de ruptura | sentencia ir_a ]
fin_según
```

Figura 4.7. Sintaxis de sentencia `según_sea`.

```
          .
          .
    break;
  case valor2:
    sentencia1;
    sentencia2;
    sentencia3;
          .
          .
    break;
  .
  .
  default:
    sentencia1;
    sentencia2;
    sentencia3;
          .
          .
}   // fin de la sentencia compuesta
```

Diagrama de flujo

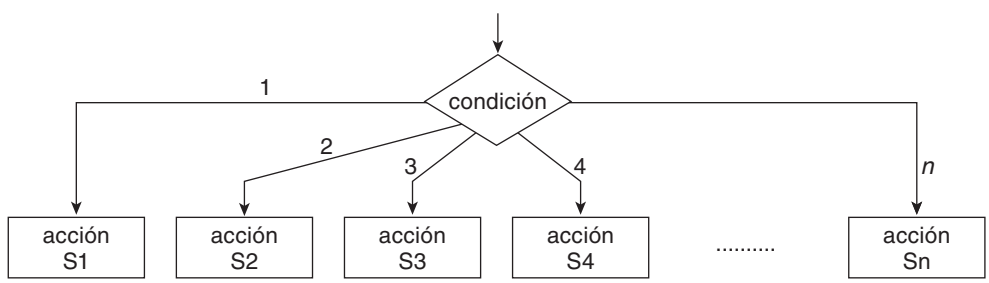
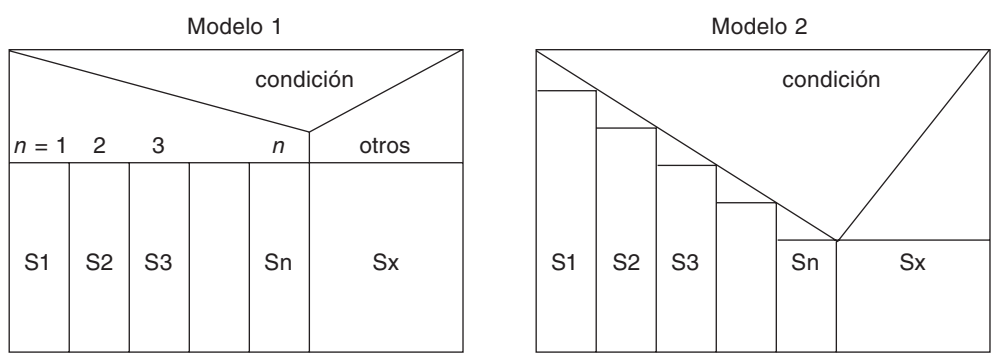


Diagrama N-S



Pseudocódigo

En inglés la estructura de decisión múltiple se representa:

```
case expresión of
  [e1]: acción S1
```

```
case expresión of
  [e1]: acción S1
```

```
    [e2]: acción S2                [e2]: acción S2
    .
    .
    [en]: acción Sn                [en]: acción Sn
    otherwise                      else
        acción Sx                      acción Sx
end_case                       end_case
```

Como se ha visto, la estructura de decisión múltiple en pseudocódigo se puede representar de diversas formas, pudiendo ser las acciones `S1`, `S2`, etc., *simples* como en el caso anterior o *compuestas* y su funcionalidad varía algo de unos lenguajes a otros.

Notas

1. Obsérvese que para cada valor de la expresión (e) se pueden ejecutar una o varias acciones. Algunos lenguajes como Pascal a estas instrucciones les denominan *compuestas* y las delimitan con las palabras reservadas **begin-end** (**inicio-fin**); es decir, en pseudocódigo.

```
según_sea E hacer
    e1: acción S1
    e2: acción S2
    .
    .
    en: acción Sn
    otros: acción Sx
fin_según
```

   o bien en el caso de instrucciones compuestas

```
según_sea E hacer
  e1: inicio
      acción S11
      acción S12
      .
      .
      acción S1a
      fin
  e2: inicio
      acción S21
      .
      .
      .
      fin
  en: inicio
      .
      .
      .
      fin
  si-no
      acción Sx
fin_según
```

2. Los valores que toman las expresiones (`E`) no tienen por qué ser consecutivos ni únicos; se pueden considerar rangos de constantes numéricas o de caracteres como valores de la expresión `E`.

```
caso_de E hacer
  2, 4, 6, 8, 10: escribir ('números pares')
  1, 3, 5, 7, 9:  escribir ('números impares')
fin_caso
```

¿Cuál de los modelos expuestos se puede considerar representativo? En realidad, como el pseudocódigo es un lenguaje algorítmico universal, cualquiera de los modelos se podría ajustar a su presentación; sin embargo, nosotros consideramos como más estándar los modelos 1, 2 y 4. En esta obra seguiremos normalmente el modelo 1, aunque en ocasiones, y para familiarizar al lector en su uso, podremos utilizar los modelos citados 2 y 4.

Los lenguajes como **C** y sus derivados **C++**, **Java** o **C#** utilizan como sentencia selectiva múltiple la sentencia `switch`, cuyo formato es muy parecido al modelo 6.

EJEMPLO 4.9

Se desea diseñar un algoritmo que escriba los nombres de los días de la semana en función del valor de una variable `DIA` *introducida por teclado.*

Los días de la semana son `7`; por consiguiente, el rango de valores de `DIA` será `1 .. 7`, y caso de que `DIA` tome un valor fuera de este rango se deberá producir un mensaje de error advirtiendo la situación anómala.

```
algoritmo DiasSemana
var
  entero: DIA
inicio
  leer(DIA)
  según_sea DIA hacer
  1: escribir('LUNES')
  2: escribir('MARTES')
  3: escribir('MIERCOLES')
  4: escribir('JUEVES')
  5: escribir('VIERNES')
  6: escribir('SABADO')
  7: escribir('DOMINGO')
  sí-no
      escribir('ERROR')
  fin_según
fin
```

EJEMPLO 4.10

Se desea convertir las calificaciones alfabéticas A, B, C, D, E y F a calificaciones numéricas 4, 5, 6, 7, 8 y 9 respectivamente.

Los valores de `A`, `B`, `C`, `D`, `E` y `F` se representarán por la variable `LETRA`, el algoritmo de resolución del problema es:

```
algoritmo Calificaciones
var
  carácter: LETRA
  entero: calificación
inicio
  leer(LETRA)
  según_sea LETRA hacer
    'A': calificación ← 4
    'B': calificación ← 5
    'C': calificación ← 6
    'D': calificación ← 7
    'E': calificación ← 8
    'F': calificación ← 9
```

```
    otros:
       escribir ('ERROR')
  fin_según
fin
```

Como se ve en el pseudocódigo, no se contemplan otras posibles calificaciones —por ejemplo, 0, resto notas numéricas—; si así fuese, habría que modificarlo en el siguiente sentido:

```
según_sea LETRA hacer
  'A': calificación ← 4
  'B': calificación ← 5
  'C': calificación ← 6
  'D': calificación ← 7
  'E': calificación ← 8
  'F': calificación ← 9

  otros: calificación ← 0
fin_según
```

EJEMPLO 4.11

Se desea leer por teclado un número comprendido entre 1 y 10 (inclusive) y se desea visualizar si el número es par o impar.

En primer lugar, se deberá detectar si el número está comprendido en el rango válido (1 a 10) y a continuación si el número es 1, 3, 5, 7, 9, escribir un mensaje de *"impar"*; si es 2, 4, 6, 8, 10, escribir un mensaje de *"par"*.

```
algoritmo PAR_IMPAR
var entero: numero
inicio
  leer(numero)
  si numero >= 1 y numero <= 10 entonces
    según_sea numero hacer
      1, 3, 5, 7, 9: escribir ('impar')
      2, 4, 6, 8, 10: escribir ('par')
    fin_según
  fin_si
fin
```

EJEMPLO 4.12

Leída una fecha, decir el día de la semana, suponiendo que el día 1 de dicho mes fue lunes.

```
algoritmo Día_semana
var
  entero : dia
inicio
  escribir('Diga el día ')
  leer(dia)
  según_sea dia MOD 7 hacer
     1:
       escribir('Lunes')
     2:
       escribir('Martes')
```

```
      3:
        escribir('Miercoles')
      4:
        escribir('Jueves')
      5:
        escribir('Viernes')
      6:
        escribir('Sabado')
      0:
        escribir('Domingo')
  fin_según
fin
```

EJEMPLO 4.13

Preguntar qué día de la semana fue el día 1 del mes actual y calcular que día de la semana es hoy.

```
algoritmo Dia_semana_modificado
var
    entero   : dia,d1
    carácter : dia1

inicio
  escribir('El dia 1 fue (L,M,X,J,V,S,D) ')
  leer( dia1)
  según_sea dia1 hacer
      'L':
        d1← 0
      'M':
        d1← 1
      'X':
        d1← 2
      'J':
        d1← 3
      'V':
        d1← 4
      'S':
        d1← 5
      'D':
        d1← 6
      si_no
        d1← -40
  fin_según

  escribir('Diga el dia ')
  leer( dia)
  dia ← dia + d1

  según_sea dia MOD 7 hacer
    1:
      escribir('Lunes')
    2:
      escribir('Martes')
    3:
      escribir('Miercoles')
```

```
    4:
      escribir('Jueves')
    5:
      escribir('Viernes')
    6:
      escribir('Sabado')
    0:
      escribir('Domingo')
  fin_según
fin
```

EJEMPLO 4.14

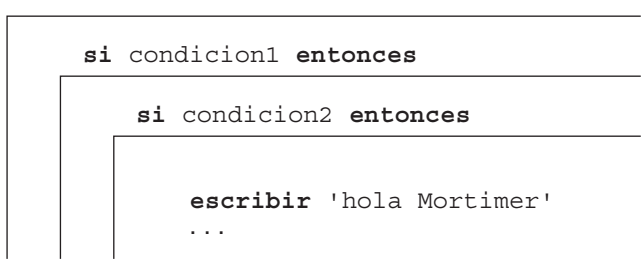
Algoritmo que nos indique si un número entero, leído de teclado, tiene 1, 2, 3 o más de 3 dígitos. Considerar los negativos.

Se puede observar que la estructura **según_sea** *<expresion>* **hacer** son varios **si** *<expr.logica>* **entonces ...** anidados en la rama **si_no**. Si se cumple el primero ya no pasa por los demás.

```
algoritmo Digitos
var
  entero : n
inicio
  leer(n)
  según_sea n hacer
    -9 .. 9:
      escribir('Tiene 1 digito')
    -99 .. 99:
      escribir('Tiene 2')
    -999 .. 999:
      escribir('Tiene tres')
    si_no
      escribir('Tiene mas de tres')
  fin_según
fin
```

4.6. ESTRUCTURAS DE DECISIÓN ANIDADAS (EN ESCALERA)

Las estructuras de selección **si-entonces** y **si-entonces-si_no** implican la selección de una de dos alternativas. Es posible también utilizar la instrucción **si** para diseñar estructuras de selección que contengan más de dos alternativas. Por ejemplo, una estructura **si-entonces** puede contener otra estructura **si-entonces**, y esta estructura **si-entonces** puede contener otra, y así sucesivamente cualquier número de veces; a su vez, dentro de cada estructura pueden existir diferentes acciones.



Las estructuras **si** interiores a otras estructuras **si** se denominan *anidadas* o *encajadas*:

```
si <condicion1> entonces
  si <condicion2> entonces
    .
    .
    .
      <acciones>
  fin_si
fin_si
```

Una estructura de selección de *n* alternativas o de decisión múltiple puede ser construida utilizando una estructura **si** con este formato:

```
si <condicion1> entonces
  <acciones>
si_no
  si <condicion2> entonces
    <acciones>
  si_no
    si <condicion3> entonces
      <acciones>
    si_no
      .
      .
      .
    fin_si
  fin_si
fin_si
```

Una estructura selectiva múltiple constará de una serie de estructuras **si**, unas interiores a otras. Como las estructuras **si** pueden volverse bastante complejas para que el algoritmo sea claro, será preciso utilizar *indentación* (sangría o sangrado), de modo que exista una correspondencia entre las palabras reservadas **si** y **fin_si**, por un lado, y **entonces** y **si_no**, por otro.

La escritura de las estructuras puede variar de unos lenguajes a otros, por ejemplo, una estructura **si** admite también los siguientes formatos:

```
si <expresion booleana1> entonces
  <acciones>
si_no
  si <expresion booleana2> entonces
    <acciones>
  si_no
    si <expresion booleana3> entonces
      <acciones>
    si_no
      <acciones>
    fin_si
  fin_si
fin_si
```

o bien

```
si <expresion booleana1> entonces
  <acciones>
```

```
si_no si <expresion booleana2> entonces
  <acciones>
  fin_si
.
.
.
fin_si
```

EJEMPLO 4.15

Diseñar un algoritmo que lea tres números A, B, C *y visualice en pantalla el valor del más grande. Se supone que los tres valores son diferentes.*

Los tres números son A, B y C; para calcular el más grande se realizarán comparaciones sucesivas por parejas.

```
algoritmo Mayor
var
  real: A, B, C, Mayor
inicio
  leer(A, B, C)
  si A > B entonces
    si A > C entonces
      Mayor ← A         //A > B, A > C
    si_no
      Mayor ← C         //C >= A > B
    fin_si
  si_no
    si B > C entonces
      Mayor ← B         //B >= A, B > C
    si_no
      Mayor ← C         //C >= B >= A
    fin_si
  fin_si
  escribir('Mayor:', Mayor)
fin
```

EJEMPLO 4.16

El siguiente algoritmo lee tres números diferentes, A, B, C, *e imprime los valores máximo y mínimo. El procedimiento consistirá en comparaciones sucesivas de parejas de números.*

```
algoritmo Ordenar
var
  real : a,b,c
inicio
  escribir('Deme 3 numeros')
  leer(a, b, c)
  si a > b entonces          // consideramos los dos primeros (a, b)
                             // y los ordenamos
     si b > c entonces       // tomo el 3° (c) y lo comparo con el menor
                             // (a o b)
       escribir(a, b, c)
      si-no                  // si el 3° es mayor que el menor averiguo si
      si c > a entonces      // va delante o detras del mayor
        escribir(c, a, b)
```

```
      si_no
        escribir(a, c, b)
      fin_si
    fin_si
  si_no
    si a > c entonces
      escribir(b, a, c)
    si_no
      si c > b entonces
        escribir(c, b, a)
      si_no
        escribir(b, c, a)
      fin_si
    fin_si
  fin_si
fin
```

EJEMPLO 4.17

Pseudocódigo que nos permita calcular las soluciones de una ecuación de segundo grado, incluyendo los valores imaginarios.

```
algoritmo Soluciones_ecuacion
var
  real : a,b,c,d,x1,x2,r,i
inicio
  escribir('Deme los coeficientes')
  leer(a, b, c)
  si a = 0 entonces
    escribir('No es ecuacion de segundo grado')
  si_no
    d ← b * b - 4 * a * c
    si d = 0 entonces
      x1 ← -b / (2 * a)
      x2 ← x1
      escribir(x1, x2)
    si_no
      si d > 0 entonces
        x1 ← (-b + raiz2(d)) / (2 * a)
        x2 ← (-b - raiz2(d)) / (2 * a)
        escribir(x1, x2)
      si_no
        r ← (-b) / (2 * a)
        i ← raiz2(abs(d)) / (2 * a)
        escribir(r, '+', i, 'i')
        escribir(r, '-', i, 'i')
      fin_si
    fin_si
  fin_si
fin
```

EJEMPLO 4.18

Algoritmo al que le damos la hora HH, MM, SS *y nos calcule la hora dentro de un segundo. Leeremos las horas minutos y segundos como números enteros.*

```
algoritmo Hora_segundo_siguiente
var
  entero : hh, mm, ss
inicio
  escribir('Deme hh,mm,ss')
  leer(hh, mm, ss)
  si (hh < 24) y (mm < 60) y (ss < 60) entonces
    ss ← ss + 1
    si ss = 60 entonces
      ss ← 0
      mm ← mm + 1
      si mm = 60 entonces
        mm ← 0
        hh ← hh + 1
        si hh = 24 entonces
           hh ← 0
        fin_si
      fin_si
    fin_si
      escribir(hh, ':', mm, ':', ss)
  fin_si
fin
```

4.7. LA SENTENCIA ir-a (goto)

El flujo de control de un algoritmo es siempre secuencial, excepto cuando las estructuras de control estudiadas anteriormente realizan transferencias de control no secuenciales.

La programación estructurada permite realizar programas fáciles y legibles utilizando las tres estructuras ya conocidas: *secuenciales, selectivas* y *repetitivas*. Sin embargo, en ocasiones es necesario realizar bifurcaciones incondicionales; para ello se recurre a la instrucción **ir_a** (**goto**). Esta instrucción siempre ha sido problemática y prestigiosos informáticos, como Dijkstra, han tachado la instrucción **goto** como nefasta y perjudicial para los programadores y recomiendan no utilizarla en sus algoritmos y programas. Por ello, la mayoría de los lenguajes de programación, desde el mítico Pascal —padre de la programación estructurada— pasando por los lenguajes más utilizados en los últimos años y en la actualidad como **C**, **C++**, **Java** o **C#**, *huyen* de esta instrucción y prácticamente no la utilizan nunca, aunque eso sí, mantienen en su juego de sentencias esta "dañina" sentencia por si en situaciones excepcionales es necesario recurrir a ella.

La sentencia **ir_a** (**goto**) es la forma de control más primitiva en los programas de computadoras y corresponde a una bifurcación incondicional en código máquina. Aunque lenguajes modernos como **VB .NET** (**Visual Basic .NET**) y **C#** están en su juego de instrucciones, prácticamente no se utiliza. Otros lenguajes modernos como **Java** no contienen la sentencia goto, aunque sí es una palabra reservada.

Aunque la instrucción **ir_a** (**goto**) la tienen todos los lenguajes de programación en su juego de instrucciones, existen algunos que dependen más de ellas que otros, como BASIC y FORTRAN. En general, no existe ninguna necesidad de utilizar instrucciones **ir_a**. Cualquier algoritmo o programa que se escriba con instrucciones **ir_a** se puede reescribir para hacer lo mismo y no incluir ninguna instrucción **ir_a**. Un programa que utiliza muchas instrucciones **ir_a** es más difícil de leer que un programa bien escrito que utiliza pocas o ninguna instrucción **ir_a**.

En muy pocas situaciones las instrucciones **ir_a** son útiles; tal vez, las únicas razonables son diferentes tipos de situaciones de salida de bucles. Cuando un error u otra condición de terminación se encuentra, una instrucción **ir_a** puede ser utilizada para saltar directamente al final de un bucle, subprograma o un procedimiento completo.

Las bifurcaciones o *saltos* producidos por una instrucción **ir_a** deben realizarse a instrucciones que estén numeradas o posean una etiqueta que sirva de punto de referencia para el salto. Por ejemplo, un programa puede ser diseñado para terminar con una detección de un error.

```
algoritmo error
   .
   .
   .
   si <condicion error> entonces
     ir_a(100)
   fin_si
100: fin
```

La sentencia **ir-a** (**goto**) o sentencia de invocación directa transfiere el control del programa a una posición especificada por el programador. En consecuencia, interfiere con la ejecución secuencial de un programa. La sentencia **ir-a** tiene una historia muy controvertida y a la que se ha hecho merecedora por las malas prácticas de enseñanza que ha producido. Uno de los primeros lenguajes que incluyó esta construcción del lenguaje en sus primeras versiones fue FORTRAN. Sin embargo, en la década de los sesenta y setenta, y posteriormente con la aparición de unos lenguajes más sencillos y populares por aquella época, BASIC, la historia negra siguió corriendo, aunque llegaron a existir teorías a favor y en contra de su uso y fue tema de debate en foros científicos, de investigación y profesionales. La historia ha demostrado que no se debe utilizar, ya que produce un código no claro y produce muchos errores de programación que a su vez produce programas poco legibles y muy difíciles de mantener.

Sin embargo, la historia continúa y uno de los lenguajes más jovenes, de propósito general, como C# creado por Microsoft en el año 2000 incluye esta sentencia entre su diccionario de sentencias y palabras reservadas. Como regla general es un elemento superfluo del lenguaje y sólo en muy contadas ocasiones, precisamente con la sentencia **switch** en algunas aplicaciones muy concretas podría tener alguna utilidad práctica.

Como regla general, es interesante que sepa cómo funciona esta sentencia, pero no la utilice nunca a menos que le sirva en un momento determinado para resolver una situación no prevista y que un salto prefijado le ayude en esa resolución. La sintaxis de la sentencia **ir_a** tiene tres variantes:

ir_a *etiqueta*	(**goto** *etiqueta)*
ir_a ***caso***	(**goto case,** en la sentencia **switch)**
ir_a ***otros***	(**goto default,** en la sentencia **switch**)

La construcción **ir_a** etiqueta consta de una sentencia **ir_a** y una sentencia asociada con una etiqueta. Cuando se ejecuta una sentencia **ir_a**, se transfiere el control del programa a la etiqueta asociada, como se ilustra en el siguiente recuadro.

```
…
inicio
    …
   ir_a etiqueta1
      …
fin
    …
etiqueta1:
…           // el flujo del programa salta a la sentencia siguiente
            // a la rotulada por etiqueta1
```

Normalmente, en el caso de soportar la sentencia **ir_a** como es el caso del lenguaje **C#**, la sentencia **ir_a** (**goto**) transfiere el control fuera de un ámbito anidado, no dentro de un ámbito anidado. Por consiguiente, la sentencia siguiente no es válida.

```
inicio
            ir_a etiquetaC
     …
     inicio
          ….                                  No válido: transferencia de control dentro
          etiquetaC                           de un ámbito anidado
          …
     fin
     …
fin
```

Sin embargo, sí se suele aceptar por el compilador *(en concreto C#)* el siguiente código:

```
inicio
     …
     inicio
          ….
          ir_a etiquetaC
          …
     fin
     etiquetaC
     …
fin
```

La sentencia **ir_a** pertenece a un grupo de sentencias conocidas como **sentencias de salto** (*jump*). Las sentencias de salto hacen que el flujo de control salte a otra parte del programa. Otras sentencias de salto o bifurcación que se encuentran en los lenguajes de programación, tanto tradicionales como nuevos (**Pascal**, **C**, **C++**, **C#**, **Java**...) son **interrumpir** (**break**), **continuar** (**continue**), **volver** (**return**) y **lanzar** (**throw**). Las tres primeras se suelen utilizar con sentencias de control y como retorno de ejecución de funciones o métodos. La sentencia **throw** se suele utilizar en los lenguajes de programación que poseen mecanismos de manipulación de excepciones, como suelen ser los casos de los lenguajes orientados a objetos tales como **C++**, **Java** y **C#**.

ACTIVIDADES DE PROGRAMACIÓN RESUELTAS

4.1. *Leer dos números y deducir si están en orden creciente.*

Solución

Dos números *a* y *b* están en orden creciente si *a* <= *b*.

```
algoritmo comparacion1
var
  real : a, b
inicio
  escribir('dar dos numeros')
  leer(a, b)
  si a <= b entonces
    escribir('orden creciente')
  si_no
    escribir('orden decreciente')
  fin_si
fin
```

4.2. *Determinar el precio del billete de ida y vuelta en avión, conociendo la distancia a recorrer y sabiendo que si el número de días de estancia es superior a 7 y la distancia superior a 800 km el billete tiene una reducción del 30 por 100. El precio por km es de 2,5 euros.*

Solución

Análisis

Las operaciones secuenciales a realizar son:

```
1. Leer distancia, duración de la estancia y precio del kilómetro.
2. Comprobar si distancia > 800 km. y duración > 7 días.
3. Cálculo del precio total del billete:
     precio total = distancia * 2.5
   • si distancia > 800 km. y duración > 7 días
     precio total = (distancia*2.5) - 30/100 * (precio total).
```

Pseudocódigo

```
algoritmo billete
var
  entero : E
  real : D, PT
inicio
  leer(E)
  PT ← 2.5*D
  si (D > 800) y (E > 7) entonces
     PT ← PT - PT * 30/100
  fin_si
  escribir('Precio del billete', PT)
fin
```

4.3. *Los empleados de una fábrica trabajan en dos turnos: diurno y nocturno. Se desea calcular el jornal diario de acuerdo con los siguientes puntos:*

1. *la tarifa de las horas diurnas es de 5 euros,*
2. *la tarifa de las horas nocturnas es de 8 euros,*
3. *caso de ser domingo, la tarifa se incrementará en 2 euros el turno diurno y 3 euros el turno nocturno.*

Solución

Análisis

El procedimiento a seguir es:

1. Leer nombre del turno, horas trabajadas (HT) y día de la semana.
2. Si el turno es nocturno, aplicar la fórmula JORNAL = 8*HT.
3. Si el turno es diurno, aplicar la fórmula JORNAL = 5*HT.
4. Si el día es domingo:
   - *turno diurno* JORNAL = (5 + 2)* ht,
   - *turno nocturno* JORNAL = (8 + 3)* HT.

Pseudocódigo

```
algoritmo jornal
var
  cadena : Dia, Turno
  real : HT, Jornal
inicio
  leer(HT, Dia, Turno)
  si Dia < > 'Domingo' entonces
    si Turno = 'diurno' entonces
      Jornal ← 5 * HT
    si_no
      Jornal ← 8 * HT
    fin_si
  si_no
    si Turno = 'diurno' entonces
      Jornal ← 7 * HT
    si_no
      Jornal ← 11 * HT
    fin_si
  fin_si
  escribir(Jornal)
fin
```

4.4. *Construir un algoritmo que escriba los nombres de los días de la semana, en función de la entrada correspondiente a la variable DIA.*

Solución

Análisis

El método a seguir consistirá en clasificar cada día de la semana con un número de orden:

```
1. LUNES
2. MARTES
3. MIERCOLES
4. JUEVES
5. VIERNES
6. SABADO
7. DOMINGO
```

si Dia > 7 y < 1 **error de entrada. rango (1 a 7)**.

si el lenguaje de programación soporta sólo la estructura **si-entonces-si_no (if-then-else)**, se codifica con el método 1; caso de soportar la estructura **según_sea (case)**, la codificación será el método 2.

Pseudocódigo

Método 1

```
algoritmo Dias_semana1
var
  entero : Dia
inicio
  leer(Dia)
  si Dia = 1 entonces
    escribir('LUNES')
  si_no
    si Dia = 2 entonces
      escribir('MARTES')
    si_no
      si Dia = 3 entonces
        escribir('MIERCOLES')
      si_no
        si Dia = 4 entonces
          escribir('JUEVES')
        si_no
          si Dia = 5 entonces
            escribir('VIERNES')
          si_no
            si Dia = 6 entonces
              escribir('SABADO')
            si_no
              si Dia = 7 entonces
                escribir('DOMINGO')
              si_no
                escribir('error')
                escribir('rango 1-7')
              fin_si
            fin_si
          fin_si
        fin_si
      fin_si
    fin_si
  fin_si
fin
```

Método 2

```
algoritmo Dias_semana2
var
  entero : Dia
inicio
  leer(Dia)
  segun_sea Dia hacer
    1: escribir('LUNES')
    2: escribir('MARTES')
    3: escribir('MIERCOLES')
    4: escribir('JUEVES')
    5: escribir('VIERNES')
    6: escribir('SABADO')
    7: escribir('DOMINGO')
    en_otro_caso escribir('error de entrada, rango 1-7')
  fin_según
fin
```

CONCEPTOS CLAVE

- Ámbito.
- Claúsula **else**.
- Condición.
- Condición falsa.
- Condición verdadera.
- Expresión ***booleana***.
- Expresión lógica.
- Operador de comparación.
- Operador de relación.
- Operador lógico.
- Sentencia compuesta.
- Sentencia **if, switch**.
- Sentencia **según-sea**.
- Sentencia **si-entonces**.
- Sentencia **si-entonces-sino**.
- **Si** anidada.
- **Si** en escalera.

RESUMEN

Las estructuras de selección **si** y **según_sea** son sentencias de bifurcación que se ejecutan en función de sus elementos relacionados en las expresiones o condiciones correspondientes que se forman con operadores lógicos y de comparación. Estas sentencias permiten escribir algoritmos que realizan tomas de decisiones y reaccionan de modos diferentes a datos diferentes.

1. Una sentencia de bifurcación es una construcción del lenguaje que utiliza una condición dada (expresión booleana) para decidir entre dos o más direcciones alternativas (ramas o bifurcaciones) a seguir en un algoritmo.
2. Un programa sin ninguna sentencia de bifurcación o iteración se ejecuta secuencialmente, en el orden en que están escritas las sentencias en el código fuente o algoritmo. Estas sentencias se denominan secuenciales.
3. La sentencia **si** es la sentencia de decisión o selectiva fundamental. Contiene una expresión booleana que controla si se ejecuta una sentencia (simple o compuesta).
4. Combinando una sentencia **si** con una cláusula **sino**, el algoritmo puede elegir entre la ejecución de una o dos acciones alternativas (simple o compuesta).
5. Las expresiones relacionales, también denominadas *condiciones simples,* se utilizan para comparar operandos. Si una expresión relacional es verdadera, el valor de la expresión se considera en los lenguajes de programación el entero 1. Si la expresión relacional es falsa, entonces toma el valor entero de 0.
6. Se pueden construir condiciones complejas utilizando expresiones relacionales mediante los operadores lógicos, **Y**, **O**, **NO**.
7. Una sentencia **si-entonces** se utiliza para seleccionar entre dos sentencias alternativas basadas en el valor de una expresión. Aunque las expresiones relacionales se utilizan normalmente para la expresión a comprobar, se puede utilizar cualquier expresión válida. Si la expresión (condición) es verdadera se ejecuta la sentencia1 y en caso contrario se ejecuta la sentencia2

```
si (expresión) entonces
   sentencia1
sino
   sentencia2
fin_si
```

8. Una sentencia compuesta consta de cualquier número de sentencias individuales encerradas dentro de las palabras reservadas **inicio** y **fin** (en el caso de lenguajes de programación como C y C++, entre una pareja de llaves "{ y }"). Las sentencias compuestas se tratan como si fuesen una única unidad y se pueden utilizar en cualquier parte en que se utilice una sentencia simple.
9. Anidando sentencias **si**, unas dentro de otras, se pueden diseñar construcciones que pueden elegir entre ejecutar cualquier número de acciones (sentencias) diferentes (simples o compuestas).
10. La sentencia **según_sea** es una sentencia de selección múltiple. El formato general de una sentencia **según_sea** (switch, en inglés) es

```
según_sea E hacer
  e1: inicio
    acción S11
    acción S12
    .
    .
    acción S1a
    fin
  e2: inicio
    acción S21
    .
    .
    .
    fin
  en: inicio
    .
    .
    .
    fin
  otros: acción Sx
fin_según
```

El valor de la expresión entera se compara con cada una de las constantes enteras (también pueden ser carácter o expresiones constantes). La ejecución del programa se transfiere a la primera sentencia compuesta cuya etiqueta precedente (valor `e1`, `e2`,`--`) coincida con el valor de esa expresión y continúa su ejecución hasta la última sentencia de ese bloque, y a continuación termina la sentencia **`según_sea`**. En caso de que el valor de la expresión no coincida con ningún valor de la lista, entonces se realizan las sentencias que vienen a continuación de la cláusula otros.

11. La sentencia **`ir_a`** (**`goto`**) transfiere el control (salta) a otra parte del programa y, por consiguiente, pertenece al grupo de sentencias denominadas de salto o bifurcación. Es una sentencia muy controvertida y propensa a errores, por lo que su uso es muy reducido, por no decir nunca, y sólo se recomienda en una sentencia **`según_sea`** para salir del correspondiente bloque de sentencias.
12. La sentencia **`según_sea`** (**`switch`**) es una sentencia construida a medida de los requisitos del programador para seleccionar múltiples sentencias (simples o compuestas) y es similar a múltiples sentencias **`si-entonces`** anidadas pero con un rango de aplicaciones más restringido. Normalmente, es más recomendable usar sentencias **`según_sea`** que sentencias **`si-entonces`** anidadas porque ofrecen un código más simple, más claro y más eficiente.

EJERCICIOS

4.1. Escribir las sentencias **`si`** apropiadas para cada una de las siguientes condiciones:

a) Si un ángulo es igual a 90 grados, imprimir el mensaje `"El ángulo es un ángulo recto"` sino imprimir el mensaje `"El ángulo no es un ángulo recto"`.
b) Si la temperatura es superior a 100 grados, visualizar el mensaje "por encima del punto de ebullición del agua" sino visualizar el mensaje "por debajo del punto de ebullición del agua".
c) Si el número es positivo, sumar el número a total de positivos, sino sumar al total de negativos.
d) Si `x` es mayor que `y`, y `z` es menor que 20, leer un valor para `p`.
e) Si `distancia` es mayor que 20 y menos que 35, leer un valor para `tiempo`.

4.2. Escribir un programa que solicite al usuario introducir dos números. Si el primer número introducido es mayor que el segundo número, el programa debe imprimir el mensaje `El primer número es el mayor`, en caso contrario el programa debe imprimir el mensaje `El primer número es el más pequeño`. Considerar el caso de que ambos números sean iguales e imprimir el correspondiente mensaje.

4.3. Dados tres números deducir cuál es el central.

4.4. Calcular la raíz cuadrada de un número y escribir su resultado. Considerando el caso en que el número sea negativo.

4.5. Escribir los diferentes métodos para deducir si una variable o expresión numérica es par.

4.6. Diseñar un programa en el que a partir de una fecha introducida por teclado con el formato `DIA`, `MES`, `AÑO` se obtenga la fecha del día siguiente.

4.7. Se desea realizar una estadística de los pesos de los alumnos de un colegio de acuerdo a la siguiente tabla:

Alumnos de menos de 40 kg.
Alumnos entre 40 y 50 kg.
Alumnos de más de 50 kg y menos de 60 kg.
Alumnos de más o igual a 60 kg.

4.8. Realizar un algoritmo que averigüe si dados dos números introducidos por teclado uno es divisor del otro.

4.9. Un ángulo se considera agudo si es menor de 90 grados, obtuso si es mayor de 90 grados y recto si es igual a 90 grados. Utilizando esta información, escribir un algoritmo que acepte un ángulo en grados y visualice el tipo de ángulo correspondiente a los grados introducidos.

4.10. El sistema de calificación americano (de Estados Unidos) se suele calcular de acuerdo al siguiente cuadro:

Grado numérico	**Grado en letra**
Grado mayor o igual a 90	A
Menor de 90 pero mayor o igual a 80	B
Menor de 80 pero mayor o igual a 70	C
Menor de 70 pero mayor o igual a 69	D
Menor de 69	F

Utilizando esta información, escribir un algoritmo que acepte una calificación numérica del estudiante (0-100), convierta esta calificación a su equivalente en letra y visualice la calificación correspondiente en letra.

4.11. Escribir un programa que seleccione la operación aritmética a ejecutar entre dos números dependiendo del valor de una variable denominada `seleccionOp`.

4.12. Escribir un programa que acepte dos números reales de un usuario y un código de selección. Si el código introducido de selección es 1, entonces el programa suma los dos números introducidos previamente y se visualiza el resultado; si el código de selección es 2, los números deben ser multiplicados y visualizado el resultado; y si el código seleccionado es 3, el primer número se debe dividir por el segundo número y visualizarse el resultado.

4.13. Escribir un algoritmo que visualice el siguiente doble mensaje

```
Introduzca un mes (1 para Enero, 2 para
Febrero,…)
Introduzca un día del mes
```

El algoritmo acepta y almacena un número en la variable `mes` en respuesta a la primera pregunta y acepta y almacena un número en la variable `dia` en respuesta a la segunda pregunta. Si el mes introducido no está entre 1 y 12 inclusive, se debe visualizar un mensaje de información al usuario advirtiéndole de que el número introducido no es válido como mes; de igual forma se procede con el número que representa el día del mes si no está en el rango entre 1 y 31.

Modifique el algoritmo para prever que el usuario introduzca números con decimales.

Nota: como los años bisiestos, febrero tiene 29 días, modifique el programa de modo que advierta al usuario si introduce un día de mes que no existe (por ejemplo, 30 o 31). Considere también el hecho de que hay meses de 30 días y otros meses de 31 días, de modo que nunca se produzca error de introducción de datos o que en su defecto se visualice un mensaje al usuario advirtiéndole del error cometido.

4.14. Escriba un programa que simule el funcionamiento normal de un ascensor (elevador) moderno con 25 pisos (niveles) y que posee dos botones de *SUBIR* y *BAJAR*, excepto en el piso (nivel) inferior, que sólo existe botón de llamada para *SUBIR* y en el último piso (nivel) que sólo existe botón de *BAJAR*.