

## Índice

- Introducción a la Informática
- Introducción a la Programación
- Introducción a la Ingeniería del Software

## Introducción a la Programación

- Conceptos básicos
- Algorítmica básica
- Subprogramas
- Constructores de tipos
- Metodología de diseño:
  - algoritmos iterativos
  - algoritmos recursivos

## Conceptos Básicos

- Computadores y programación
- Diseño e implementación de programas
- Introducción intuitiva a la programación

## Computadores y programación

- Computador: máquina electrónica capaz de realizar cálculos y tratar grandes cantidades de información de forma automática, siguiendo un conjunto de instrucciones.
- Hardware: componentes físicos
- Software: conjunto de programas ejecutables
  
- Primer computador: máquina analítica (C. Babbage 1792-1871)
- Primer programador: Lady Ada Augusta Byron (1815-1852)

## Diseño e implementación de programas

- Algoritmo:
- 1) Conjunto de instrucciones o sentencias que pueden ser ejecutadas ordenadamente en el tiempo, para resolver un problema concreto, obteniendo unos resultados a partir de unos datos iniciales
- 2) Sucesión de estados definidos por los valores que van tomando las variables del programa
- <Estado0> instrucción1 <Estado1> ... instrucciónN <EstadoN>
- Un algoritmo describe un método general
- Un mismo algoritmo se puede aplicar a distintos conjuntos de datos

## Diseño e implementación de programas

- Datos (entrada): 3.1 2.4 1.0

- Algoritmo:

leeReal(a)

leeReal(b)

leeReal(x)

$y := a * x + y$

escribeReal(y)

- Resultados (salida): 5.5

## Diseño e implementación de programas

- Un **algoritmo** (del matemático persa al-Jwarizmi) es un conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.
- Un algoritmo debe ser correcto para cualquier conjunto de datos.
- Los algoritmos pueden ser expresados de muchas maneras, incluyendo:
  - lenguaje natural: tienden a ser ambiguas y extensas
  - pseudocódigo: evita muchas ambigüedades del LN
  - diagramas de flujo
  - lenguajes de programación

## Conceptos básicos: Diseño e implementación de programas

- Ciclo de vida: Conjunto de etapas que permiten pasar de un problema a un programa ejecutable que lo resuelve.
- Especificación: QUÉ debemos resolver: pre/post
- Diseño: CÓMO lo resolvemos: algoritmo
- Implementación: programa
- Uso y mantenimiento:

## Introducción intuitiva

- **Objetivos:**
  - Familiarizarse con la notación algorítmica:
    - Sintaxis: CÓMO se expresa
    - Semántica: QUÉ se expresa
  - Aprender a interpretar un programa escrito en notación algorítmica:
    - Dado un programa, unos datos de entrada, saber cuáles deberían ser los resultados.

## Conceptos básicos: Introducción intuitiva

```
algoritmo suma  
  var d1, d2, r: entero fvar  
  leerEntero(d1)  
  leerEntero(d2)  
  r := d1 + d2  
  escribir(r)  
falgoritmo
```

- Palabras clave: algoritmo, falgoritmo, var, fvar
- El bloque **algoritmo**, **falgoritmo** define el algoritmo suma
- Las variables d1, d2, r son contenedores que permiten representar números enteros
- Las variables se declaran en el bloque formado por **var** y **fvar**
- Los datos de entrada son dos enteros, y el resultado otro entero, suma de los anteriores.

## Introducción intuitiva

```
algoritmo suma
  var d1, d2, r: entero fvar
  leerEntero(d1)
  leerEntero(d2)
  r := d1 + d2
  escribir(r)
falgoritmo
```

- La operación de asignación indicada como “:=” se usa para modificar el contenido de una variable
- La acción leerEntero y escribir entero, leen y escriben datos
- El algoritmo se tabula (indenta) para resaltar los bloques y facilitar su legibilidad
- Las cuatro líneas entre fvar y falgoritmo son las instrucciones o sentencias ejecutables

## Introducción intuitiva

```
algoritmo suma
  var d1, d2, r: entero fvar
    Estado0
  leerEntero(d1)
    Estado1
  leerEntero(d2)
    Estado2
  r := d1 + d2
    Estado3
  escribir(r)
    Estado4
falgoritmo
```

Estado	Entrada	d1	d2	r	Salida
0	$\wedge 4 5$	-	-	-	-
1	4 $\wedge$ 5	4	-	-	-
2	4 5 $\wedge$	4	5	-	-
3	4 5 $\wedge$	4	5	9	-
4	4 5 $\wedge$	4	5	9	9

- El símbolo  $\wedge$  precede al dato que se leerá a continuación

## Tipos de Lenguajes de programación

- De alto y bajo nivel
- Interpretados y compilados
- Estructurados
- Fuerte o débilmente tipados
- Orientados a Objetos
- Imperativos
- Funcionales
- Lógicos
- ...

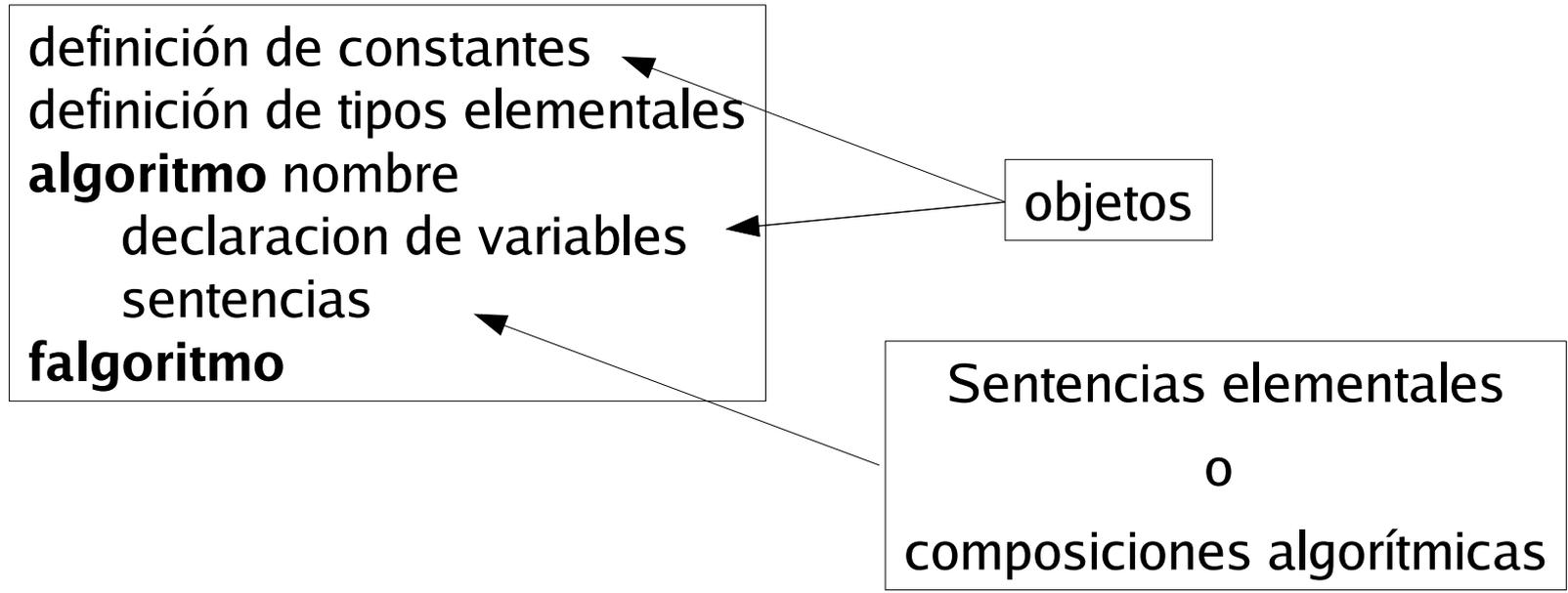
## Algorítmica básica

- Objetivos
- Estructura general de un algoritmo
- Objetos
- Tipos elementales
- Expresiones
- Sentencias
- Composiciones algorítmicas

## Objetivos

- Estudiar la parte fundamental de la notación algorítmica (siguiendo un paradigma imperativo).
- Consolidar el concepto de sintaxis formal
- Estudiar los tipos de datos
- Estudiar las sentencias fundamentales de la notación y como se combinan

## Estructura general de un algoritmo



## Objetos

- Objeto: unidad de información que interviene en la resolución de un problema.
  - Nombre:
    - secuencia de uno o más caracteres alfanuméricos (letras y dígitos) y el carácter '\_'.
    - El primer carácter tiene que ser una letra.
    - No pueden ser palabras clave o reservadas
  - Tipo: rango de valores y operaciones de los objetos
  - Valor:
    - Constantes: valor no modificable (nombre en mayúsculas)
    - Variables: valor modificable (nombre en minúsculas)

## Objetos

- Declaración y definición de constantes

```
const nombre_constante : nombre_tipo = valor  
fconst
```

```
const  
  MAX: entero = 8  
  PI: real = 3.1415  
  LETRA_A: caracter = 'a'  
  FIN: booleano = falso  
fconst
```

## Objetos

- Declaración y definición de variables

```
var nombre_variable : nombre_tipo  
fvar
```

```
var  
  a: entero  
  p1, p2, p3: real  
  LETRA: caracter  
  FIN: booleano  
fconst
```

## Tipos elementales

- Los tipos determinan cuáles son los valores que pueden contener y qué operaciones se permiten hacer con estos valores
- Los tipos elementales vienen predefinidos en los lenguajes de programación
- Nosotros consideraremos los siguientes:
  - Entero
  - Real
  - Carácter
  - Booleano

## Tipos elementales

- La **signatura** de una operación indica qué operandos hay y de qué tipo son, así como la naturaleza del resultado

+ : entero, entero -> entero      4 + 2 -> 6

\* : real, real -> real              2.0 \* 4.0 -> 8.0

- Un símbolo de operación está **sobrecargado** cuando puede significar operaciones distintas

\* : entero, entero -> entero      2 \* 4 -> 8

- La **aridad** de una operación indica el número de argumentos de la operación

+ : entero, entero -> entero      (binaria)

**no** : booleano -> booleano      (unaria)

## Tipo entero

- Subconjunto de números enteros entre MIN y MAX
- Los valores se representan en base 2
  - 2 bytes:  $[-2^{15}, 2^{15}-1]$
  - 4 bytes:  $[-2^{31}, 2^{31}-1]$
  - Ejemplo: 152 : 00000000 10011000
  - Limitado en magnitud, pero no en precisión
- Las constantes de tipo entero: [0-9,-]
- Operaciones aritméticas:
  - Binarias: +, -, \*, **div**, **mod** (cociente y residuo de la división entera)
  - Unarias: -
- Operaciones de relación:
  - =, != (<>), <, >, <=, >=

## Tipo real

- Subconjunto de los números reales: número = mantisa \*  $2^{\text{exponente}}$
- 
- s      exponente      mantisa
- 4 bytes :  $[-0.29 * 10^{-38}, 1.7 * 10^{38}]$
- Limitaciones en magnitud (exponente) y precisión (mantisa)
- Constantes reales: [0-9, ., -, E]
- Ejemplos: 4.0    -3.8765    1009.001    0.21E4
- Operaciones aritméticas:
  - Binarias: +, -, \*, /
  - Unarias: -
- Operaciones de relación:
  - =, != (<>), <, >, <=, >=
-

## Tipo real

- **función** sin (**ent** x: real) **devuelve** real
- **función** cos (**ent** x: real) **devuelve** real
- **función** tan (**ent** x: real) **devuelve** real
- **función** exp (**ent** x: real) **devuelve** real
- **función** log (**ent** x: real) **devuelve** real
- **función** log10 (**ent** x: real) **devuelve** real
- **función** pow (**ent** x: real, **ent** x: real) **devuelve** real
- **función** sqrt (**ent** x: real) **devuelve** real
- **función** fabs (**ent** x: real, **ent** x: real) **devuelve** real
- Conversión (coerción) entre entero y real:
  - entero(2.0) -> 2      entero(-1.01) -> -2
  - real(2) -> 2.0

## Tipo caracter

- Valores: un conjunto de símbolos (alfabeto) mediante distintos tipos de codificación: ASCII, Unicode, etc.
- Operaciones de relación:
  - =, != (<>), <, >, <=, >=
  - Por ejemplo: 'A' < 'B' -> cierto
- código: caracter -> entero
  - Por ejemplo: codigo('A')=65

## Tipo booleano

- Valores: {cierto, falso}
- Operaciones:
  - Binarias: **y**, **o**
  - Unarias: **no**
  - Operaciones de relación:
    - =, != (<>), <, >, <=, >=
    - Por convenio: falso < cierto
- Leyes de morgan
  - no (a y b) = no a o no b**
  - no (a o b) = no a y no b**

p	q	no p	p y q	p o q
C	C	F	C	C
C	F	F	F	C
F	C	C	F	C
F	F	C	F	F

## Expresiones

- Una **expresión** es un conjunto de variables, constantes y operadores que denotan un valor
- Una expresión debe ser correcta sintáctica y semánticamente
  - Sintáctica: los símbolos que la componen deben formar una estructura válida
  - Semántica: el valor se puede calcular

## Expresiones: corrección sintáctica

- Sea  $op1$  una operación unaria,  $op2$  una binaria y  $E1$  y  $E2$  expresiones, una expresión puede ser:

- Una constante 3.1415
- Una variable  $r$
- Una llamada a una función  $\sin(x)$
- $op1 E1$  - 4.0
- $E1 op2 E2$  - 4.0 \*  $r$
- $( E1 )$  **no**  $( a > 50 \text{ o } b )$

## Expresiones: corrección sintáctica

- Para determinar si una expresión es sintácticamente correcta, se debe realizar un análisis sintáctico
  - Se etiquetan las constantes y variables como expresiones
  - Se aplican reglas sintácticas para formar subexpresiones
  - El proceso termina cuando no se pueden agrupar más elementos
  - Si el análisis es completo, la expresión es correcta
  - En caso contrario, no es correcta sintácticamente



## Expresiones: ambigüedades

- Para evitar las ambigüedades hacen falta dos reglas más:
  - Regla de la prioridad: establece un orden de prioridad entre los operadores

()

máxima prioridad

Funciones

- , **no**

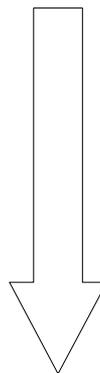
\* , / , **div** , **mod**

+ , -

= , != , < , > , <= , >=

**y** , **o**

mínima prioridad

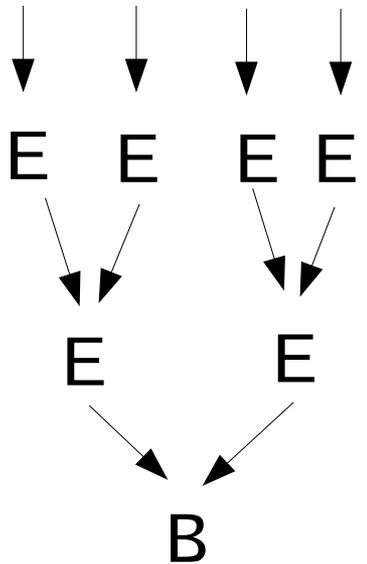


- Regla de la asociatividad: análisis de izquierda a derecha para los operadores con la misma prioridad (excepto booleanos, donde la prioridad se marca con paréntesis)

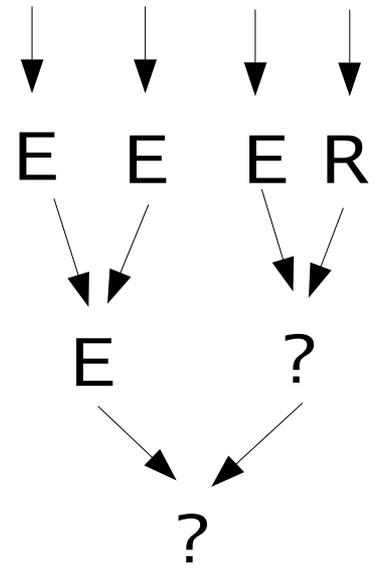
## Expresiones: corrección semántica

- La corrección semántica determina si la expresión es correcta en cuanto a combinación de tipos de datos
- Se realiza sobre el árbol sintáctico partiendo de las hojas y etiquetando todas las subexpresiones con su tipo

5 + 3 > 5 \* 7

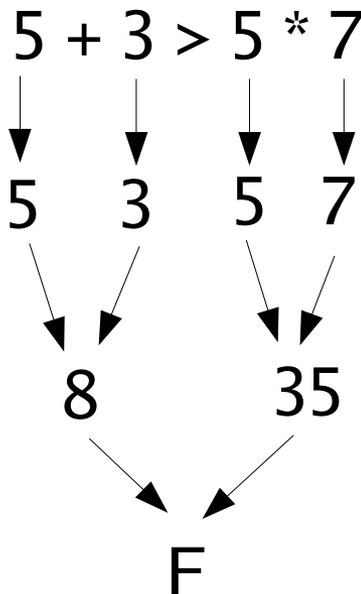


5 + 3 > 5 \* 7.2



## Expresiones: evaluación

- Sólo se puede evaluar una expresión sintáctica y semánticamente correcta
- Evaluar es obtener (calcular) el valor que denota una expresión
- Se realiza sobre el árbol de análisis partiendo de las hojas y etiquetando todas las subexpresiones con su valor resultante



## Expresiones booleanas

- Siempre debemos poner paréntesis

~~a y b o c~~      ->      (a y b) o c

- La comparación de una expresión booleana con una constante booleana es redundante (mejor consultar por el valor de la variable)

~~b = cierto~~      ->      b

~~b = falso~~      ->      no b

## Acciones simples: sentencias

- La asignación permite dar valor a una variable  
variable := expresión



- El símbolo de asignación “:=” se lee como “toma por valor”
- En el lado izquierdo siempre una variable
- En el lado derecho siempre una expresión
- Primero se evalúa la expresión y a continuación se asigna el valor resultante a la variable
- El tipo de la variable y la expresión deben coincidir

## Acciones simples: sentencias

- Acciones de lectura y escritura
- CEE: Canal estándar de entrada (teclado, fichero ...)
- CES: Canal estándar de salida (pantalla, fichero ...)
- Lectura: se transfiere información del CEE a una variable  
**var x: entero fvar**  
leerEntero(x)                      x := CEE
- Escritura: se transfiere información de una variable al CES  
**var x: entero fvar**  
escribirEntero(x)                      CES := x

## Acciones simples: sentencias

- Toda variable, antes de ser consultada debe tener aún valor
- Las variables toman valores cuando están en el lado izquierdo de una asignación o cuando són leídos
- Una variable se inicializa cuando toma valor por primera vez
- Una variable se consulta cuando está en el dado derecho de una asignación o cuando se escribe

## Composición de acciones

- Una herramienta fundamental de los LP es la composición de acciones. Se define de forma recurrente:
- Una composición de acciones es:
  - Una sentencia (acción simple)
  - Una sentencia y una composición de acciones
- Componer significa combinar acciones para describir cálculos más complejos
- Hay tres formas clásicas de componer acciones:
  - La composición secuencial
  - La composición alternativa
  - La composición iterativa

## Composición alternativa o condicional

- La evaluación de una (o varias) expresiones booleanas provoca la ejecución de un conjunto u otro de sentencias.

**si** expresiónBooleana **entonces**

composición de sentencias

**fsi**

**si** expresiónBooleana **entonces**

composicion de sentencias1

**sino**

composicion de sentencias2

**fsi**

## Composición alternativa o condicional

### **opción**

**caso** expresiónBooleana1 **entonces** composición de sentencias1

**caso** expresiónBooleana2 **entonces** composición de sentencias2

**otro caso** composición de sentencias3

### **fopción**

- La composición está bien construida si se cumple una y sólo una condición expresiónBooleana y las opciones constituyen una partición del conjunto de acciones

## Composición alternativa o condicional

```
algoritmo ejemplo
  var a, b, min: entero fvar
  leerEntero(a);
  leerEntero(b);
  si a < b entonces
    min := a
  sino
    min := b
  fsi
  escribirEntero(min)
falgoritmo
```

## Composición iterativa

**mientras** expresiónBooleana **hacer**

composición de sentencias

**fmientras**

- Mientras la evaluación de la expresiónBooleana sea cierta, se ejecuta la composición de sentencias
- Sólomente cuando la expresiónBooleana es falsa, se sale de la iteración
- Puede que la composición de sentencias no se ejecute nunca si la expresiónBooleana es falsa de entrada

## Composición iterativa

**algoritmo** ejemplo

**var** n, i, suma: **entero** **fvar**

leerEntero(n);

i := 1;

**mientras** i <= n **hacer**

    suma := suma + i;

    i := i + 1

**fmientras**

escribirEntero(suma)

**falgoritmo**

- Este algoritmo hace n iteraciones
- Calcula la suma de los n primeros naturales
- Al final se cumple que  $i > n$  ( $i=n+1$ )

## Composición iterativa

```
algoritmo ejemplo
  var n, i, b: entero fvar
  leerEntero(n);
  i := n;
  b := 1;
  mientras i >= 0 hacer
    b := b * 3;
    i := i - 1
  fmientras
  escribirEntero(b)
falgoritmo
```

- Este algoritmo hace  $n+1$  iteraciones
- Calcula  $3^{n+1}$
- Al final se cumple que  $i < n$  ( $i=-1$ )

## Composición iterativa

**algoritmo** ejemplo

**var** n, i, suma: **entero** **fvar**

leerEntero(n);

i := 1;

**mientras** i < n **hacer**

    suma := suma + i;

    i := i + 1

**fmientras**

escribirEntero(suma)

**falgoritmo**

- Error: el algoritmo hace n-1 iteraciones: no calcula la suma de los n primeros naturales

## Composición iterativa

**algoritmo** ejemplo

**var** n, i, suma: **entero** **fvar**

leerEntero(n);

i := 1;

**mientras** i <= n **hacer**

    suma := suma + i;

**fmientras**

    escribirEntero(suma)

**falgoritmo**

- Error: el algoritmo no termina. i siempre vale 1.

## Subprogramas

- Objetivos
- Introducción
- Parámetros
- Funciones
- Acciones
- Diseño de un subprograma

## Objetivos

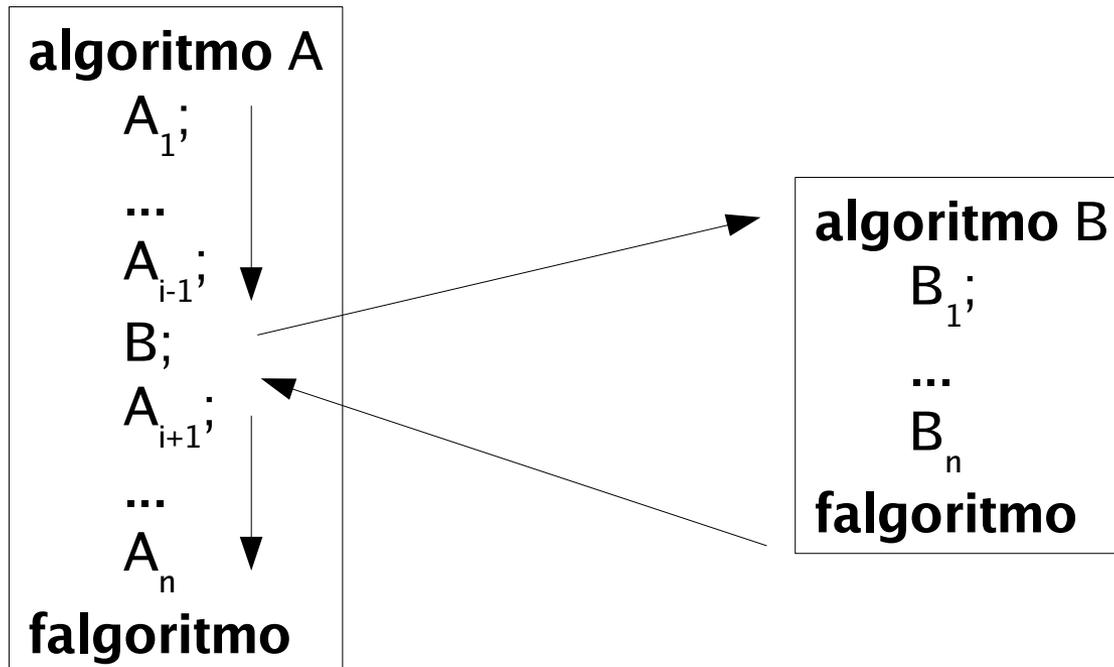
- Entender las ventajas de estructurar programas en subprogramas
- Entender los conceptos y notación asociados a los subprogramas

## Introducción

- Los subprogramas son un mecanismo de los lenguajes estructurados que permiten agrupar sentencias
- Hay dos tipos de subprogramas:
  - Acciones
  - Funciones
- Los subprogramas constan de una cabecera y un cuerpo
- La llamada a un subprograma pasa el control al código del subprograma

## Introducción: control de la ejecución

- Cuando un subprograma A llama a un subprograma B:
  - A pasa el control a B
  - Se establece una comunicación entre A y B: paso de parámetros
  - Cuando B acaba, devuelve el control a A



## Introducción: ventajas de los subprogramas

- El programa es estructura como un conjunto de subprogramas cortos, en lugar de un programa muy largo (de miles de líneas de código) que resuelve todos los problemas (spaguetti code)
- El programa gana en claridad estructurándose de forma jerárquica. El programa principal se convierte en una guía-resumen del programa completo.
- Evita repeticiones factorizando el código
- Facilita la depuración: se verifican primero los subprogramas, y luego el programa principal

## Introducción: ejemplo

- Dados dos enteros,  $n$  y  $k$ , diseñad un algoritmo que calcule el número combinatorio:

$$C_n^k = \binom{n}{k} = \frac{n!}{(n-k)!k!}$$

- Este algoritmo calcula el factorial de 3 valores distintos, por tanto, se diseñará un subprograma (una función) que calcule el factorial, de forma que el código correspondiente sólo se escribe una vez

## Introducción: ejemplo

**algoritmo** Comb

var n, k, c: **entero** fvar

leerEntero(n);

leerEntero(k);

c := Factorial(n) **div** (Factorial(n-k) \* Factorial(k))

escribirEntero(c)

**falgoritmo**

**función** Factorial (ent n: **entero**) devuelve entero

var i, f: **entero** fvar

i := 1; f := 1

**mientras** i <= n **hacer**

    f := f \* i; i := i + 1

**fmientras**

**devuelve** f

**ffunción**

← cabecera

## Parámetros

- Los parámetros son datos que se pasan los subprogramas
- Vienen definidos por:
  - Tipo
  - Mecanismo de paso: forma de transmisión de los datos entre el algoritmo A y el subprograma B
    - Entrada (**ent**): B sólo podrá consultar el parámetro
    - Salida (**sal**): B genera el parámetro
    - Entrada/Salida (**ent/sal**): B puede consultar y modificar el parámetro

## Parámetros actuales y formales

- Los parámetros **actuales** son los que aparecen en la sentencia de la llamada al subprograma
- Los parámetros **formales** son los que aparecen en la cabecera de la definición del subprograma
- Los parámetros actuales y formales deben coincidir en:
  - Número
  - Tipo
  - Orden
  - Significado
- El nombre no tiene por qué coincidir

## Parámetros: ejemplo

```
algoritmo Comb  
  var n, k, c: entero fvar  
  leerEntero(n);  
  leerEntero(k);  
  c := Factorial(n) div (Factorial(n-k) * Factorial(k))  
  escribirEntero(c)  
falgoritmo
```

Parámetros  
actuales

```
función Factorial (ent n: entero) devuelve entero  
  var i, f: entero fvar  
  i := 1; f := 1  
  mientras i <= n hacer  
    f := f * i; i := i + 1  
  fmientras  
  devuelve f  
ffunción
```

Parámetro  
formal

## Funciones

- La llamada a una función se realiza dentro de una expresión  
  
nombre\_función ( lista de parámetros actuales);
- Los distintos parámetros actuales pueden ser:
  - Nombre de variables del algoritmo que llama a la función
  - Expresión (si el parámetro es de entrada)

## Funciones

**función** nombre\_función (lista parámetros formales) **devuelve** tipo  
declaración de variables y constantes locales  
composición de sentencias  
**devuelve** expresión  
**ffunción**

The diagram illustrates the components of a function definition. A rectangular box contains the text: **función** nombre\_función (lista parámetros formales) **devuelve** tipo, declaración de variables y constantes locales, composición de sentencias, **devuelve** expresión, and **ffunción**. An arrow labeled 'cabecera' points to the first line. A bracket on the right side of the box groups the last three lines, with an arrow labeled 'cuerpo' pointing to it.

- Notación de los parámetros formales:  
mecanismo\_de\_paso nombre\_parámetro : tipo\_parámetro
- Las funciones sólo tienen parámetros de entrada
- Toda función devuelve el valor de la expresión del tipo indicado en la cabecera

## Funciones: ejemplo

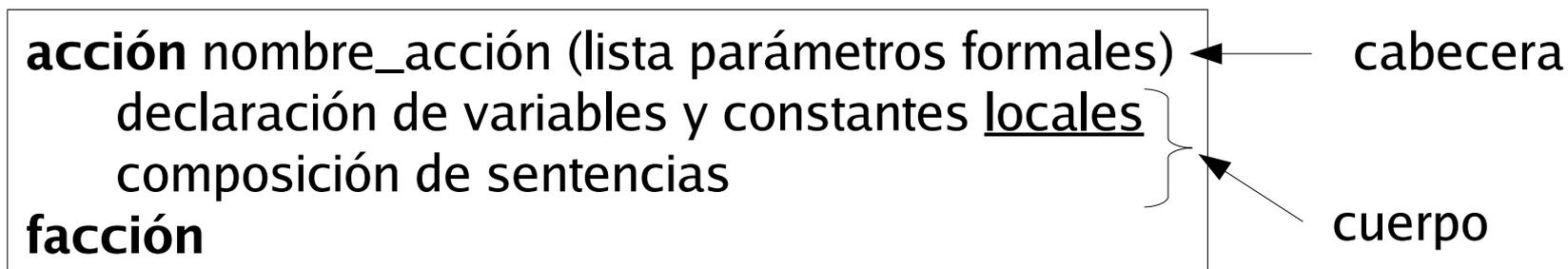
```
función Factorial (ent n: entero) devuelve entero  
  var i, f: entero fvar  
  i := 1; f := 1  
  mientras i <= n hacer  
    f := f * i; i := i + 1  
  fmientras  
  devuelve f  
ffunción
```

- En una función todos los parámetros son de entrada
- Toda función devuelve el valor de una expresión indicada en la cabecera
- Las variables i, f son locales

## Acciones

- La llamada a una acción es una sentencia ejecutable  
  
nombre\_acción ( lista de parámetros actuales);
  
- Los distintos parámetros actuales pueden ser:
  - Nombre de variables del algoritmo que llama a la acción
  - Expresión (si el parámetro es de entrada)

## Acciones



- Notación de los parámetros formales:  
mecanismo\_de\_paso nombre\_parámetro : tipo\_parámetro
- Los parámetros de las acciones pueden ser de entrada, salida o entrada/salida
- Las acciones no devuelven ningún valor

## Acciones: Ejemplo

- Diseñar un algoritmo que dados tres enteros, los ordene de pequeño a grande

## Acciones: ejemplo

**algoritmo** ordenar

**var** a, b, c: **entero** **fvar**

leerEntero(a); leerEntero(b); leerEntero(c);

**si** (a > b) **entonces** intercambiar(a,b) **fsi**

**si** (b > c) **entonces** intercambiar(b,c) **fsi**

**si** (a > b) **entonces** intercambiar(a,b) **fsi**

escribirEntero(a); escribirEntero(b); escribirEntero(c)

**falgoritmo**

**acción** intercambiar (**ent/sal** x, y: **entero**)

**var** temp: **entero** **fvar**

temp := x; x := y; y := temp

**facción**

## Acciones: ejemplo

**algoritmo** ordenar

**var** a, b, c: **entero** **fvar**

leerEntero(a); leerEntero(b); leerEntero(c);

**si** (a > b) **entonces** intercambiar(a,b) **fsi**

**si** (b > c) **entonces** intercambiar(b,c) **fsi**

**si** (a > b) **entonces** intercambiar(a,b) **fsi**

escribirEntero(a); escribirEntero(b); escribirEntero(c)

**falgoritmo**

- La acción intercambiar se llama tres veces con parámetros actuales distintos
- Aunque en la última llamada los parámetros actuales son los mismos, sus valores pueden haber cambiado

## Acciones: ejemplo

```
acción intercambiar (ent/sal x, y: entero)  
  var temp: entero fvar  
  temp := x; x := y; y := temp  
facción
```

- Los dos parámetros son de entrada/salida
- Los parámetros formales coinciden con los actuales en:
  - Número (2)
  - Tipo (entero)
  - Significado (dos variable que deben intercambiar su valor)
  - El nombre no coincide
  - La variable temp es local

## Diseño de un subprograma

- El diseño de un subprograma consta de dos etapas:
  
- Diseño de la cabecera:
  - Determinación de los parámetros
  - Tipo de los parámetros
  - Mecanismo de paso
  - Acción o función
  
- Diseño del cuerpo:
  - Diseño del correspondiente algoritmo
  
- A este proceso se le llama “análisis descendente”

## Constructores de tipos

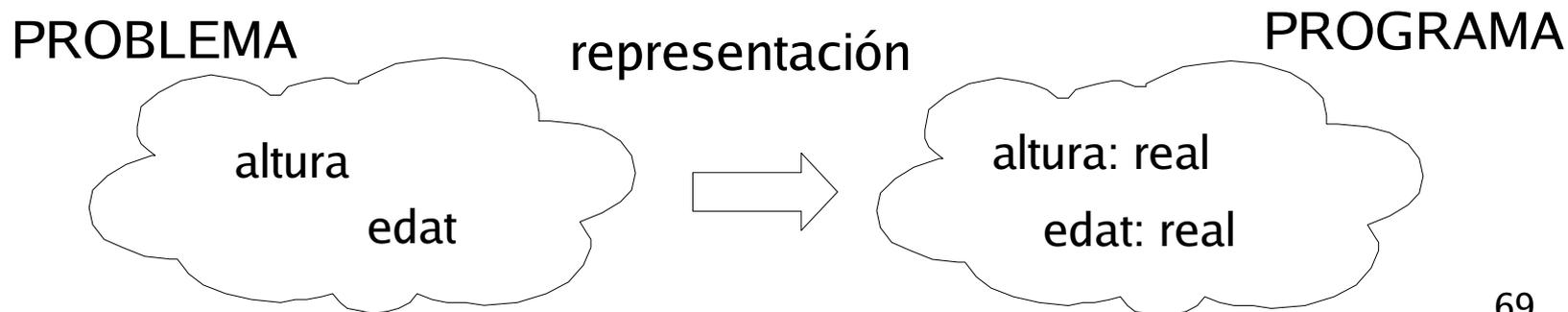
- Objetivos
- Constructores de tipos
- Constructor enumeración
- Constructor tupla (registro)
- Constructor tabla (vector, matriz, array)
- Tablas multidimensionales
- Construcciones con tuplas y tablas
- Operatividad

## Objetivos

- Entender el concepto de representación
- Comprender que los tipos elementales no pueden representar muchas entidades
- Aprender cómo se soluciona el problema de la representación con los constructores de tipos
- Entender cómo se definen los constructores enumeración, tupla y tabla
- Aprender cómo se usan las variables de los nuevos tipos definidos
- Aprender a combinar adecuadamente todos los constructores de tipos

## Constructores de tipos

- Los programas explican cómo solucionar un problema real
- En un problema pueden aparecer entidades del mundo real como velocidades, edades, fuerzas, etc.
- Las entidades se representan mediante las variables
- Las variables denotan las entidades del mundo real
- Cada entidad debe representarse sobre variables del tipo adecuado
- Escoger la representación más adecuada para una entidad es una de las tareas fundamentales de la programación





## Constructores de tipos

- Los constructores de tipos son mecanismos del lenguaje de programación que permiten definir nuevos tipos
- El número de tipos nuevos que pueden definirse es ilimitado
- Una vez definido un nuevo tipo, se pueden declarar variables del nuevo tipo
- Por convención, los nombres de los tipos nuevos comienzan con una t minúscula

**tipo**

definición nuevo tipo

**ftipo**

**tipo** tPunto3D = ... **ftipo**

...

**var** posicion : tPunto3D **fvar**

## Constructor de tipos enumeración

- Permite representar objetos que pueden tomar un conjunto arbitrario de valores
- El nuevo tipo se define enumerando todos los valores posibles
- Operaciones de relación:
  - =, != (<>), <, >, <=, >=

**tipo**

nombre nuevo tipo = **enumeración** {val<sub>1</sub>, ... val<sub>N</sub>}

**f tipo**

## Constructor de tipos enumeración: ejemplo

### **tipo**

tRGB = **enumeración** {RED, GREEN, BLUE}

tDiasSemana = **enumeración** { LUNES, MARTES,  
MIERCOLES, JUEVES, VIERNES, SABADO,  
DOMINGO}

### **ftipo**

#### **var**

hoy : tDiaSemana

color : tRGB

#### **fvar**

hoy := LUNES

color := BLUE

## Constructor de tipos tupla

- Permite representar objetos que son una composición de elementos de tipos distintos
- El nuevo tipo se define enumerando todos los elementos que lo constituyen (campos) indicando el tipo de cada uno
- El acceso a uno de esos campos de una variable se realiza con el operador **punto**: nombre\_variable.nombre\_campo

```
tipo nombre_nuevo_tipo =  
    tupla  
        nombre_campo_1: nombre_tipo_1  
        ...  
        nombre_campo_N : nombre_tipo_N  
    ftupla  
ftipo
```

## Constructor de tipos tupla: ejemplo

**tipo**

tComplejo = **tupla** r, i : real **ftupla**

tTiempo = **tupla**

temperatura : real

llueve : booleano

**ftupla**

**ftipo**

**var** c1, c2, c3 : tComplejo; tiempo : tTiempo **fvar**

{c1, c2 inicializados}

c3.r := c1.r + c2.r;

C3.i := c1.i + c2.i;

tiempo.temperatura := 15.7; tiempo.llueve := cierto;

## Constructor de tipos tabla

- Permite representar objetos que son una composición de elementos del mismo tipo y cada elemento se identifica con un índice
- El nuevo tipo se define indicando el rango de valores que puede tomar el índice  $i$  el tipo de componentes de la tabla
- El acceso a uno de los componentes de la tabla se realiza con el operador  $[ ]$ : nombre\_variable.[índice\_componente]

**tipo** nombre\_nuevo\_tipo = **tabla** [rango] **de** nombre\_tipo **f**tipo

[rango] = [rango\_mínimo ... rango\_máximo] donde

rango\_mínimo y rango\_máximo son constantes de tipo entero

## Constructor de tipos tabla: ejemplo

```
tipo tNIF = tabla [1..9] de caracter ftipo  
var nif : tNIF fvar  
nif[1] := '3';  
nif[2] := '5';  
...  
nif[9] := 'C';
```

## Constructor de tipos tabla: ejemplo

- Iteración para recorrer todos los elementos de una tabla

Inicializaciones

$i := \text{rango\_mínimo}$

**mientras**  $i \leq \text{rango\_máximo}$  **hacer**

    tratar\_elemento\_i;

$i := i + 1$

**fmientras**

tratamiento\_final

## Constructor de tipos tabla: ejemplo

- Iteración y tablas

```
const MIN = 0; MAX = 9 fconst  
tipo t10Enteros = tabla [MIN..MAX] de entero ftipo  
var u, v, w : t10Enteros; i : entero fvar  
{u, v ya están inicializados}  
i := 0;  
mientras (i<MAX+1) hacer  
    w[i] := u[i] + v[i];  
    i := i + 1  
fientras
```

## Tablas multidimensionales

**tipo**

nombre\_nuevo\_tipo = **tabla** [rango1, rango2] **de** nombre\_tipo

**ftipo**

- Acceso a un componente de una variable de tipo tabla multidimensional

nombre\_variable[índice1, ... , índiceN]

## Constructor de tipos tabla multidimensional: ejemplo

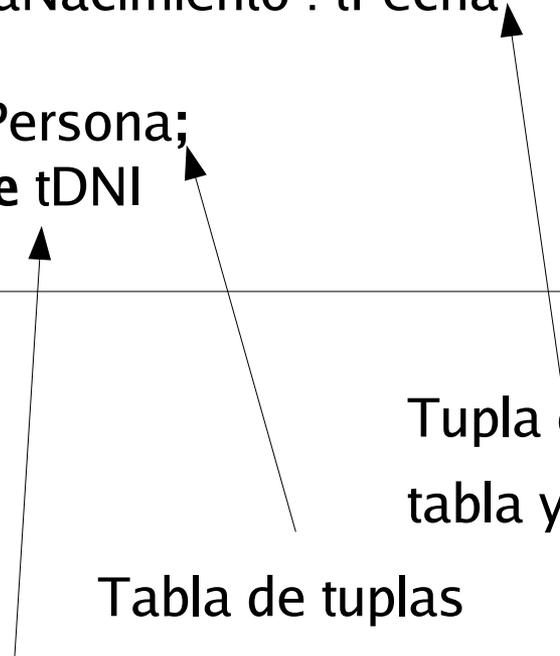
```
const MIN : entero = 0; MAX : entero = 9 fconst  
tipo  
    tMatriz = tabla [MIN .. MAX, MIN .. MAX] de entero;  
    tVector = tabla [MIN .. MAX] de real  
ftipo  
var m : tMatriz; v : tVector; i : entero fvar  
{m ya está inicializada}  
i := 0; j := 5;  
mientras (i<MAX+1) hacer  
    v[i] := mat[i, j];  
    i := i + 1  
fmientras
```

## Constructor anidadas con tuplas y tablas

- Una tupla puede tener campos de tuplas
- Una tupla puede tener campos de tablas
- Una tabla puede tener componentes de tipo tupla
- Una tabla puede tener componentes de tipo tabla
- Así, podemos construir tipos para representar objetos de cualquier grado de complejidad.

## Constructor anidadas con tuplas y tablas: ejemplo

```
const NDNI : entero = 8; MAX : entero = 100 fconst
tipo
  tDNI = tabla [0 .. NDNI-1] de caracter;
  tFecha = tupla dia, mes, año : entero ftupla
  tPersona = tupla dni : tDNI; fechaNacimiento : tFecha
ftupla
  tGrupo = tabla [0 .. MAX-1] de tPersona;
  tTablaDNI = tabla [0 .. MAX-1] de tDNI
ftipo
```



Tupla con una  
tabla y una tupla

Tabla de tuplas

Tabla de tablas

## Constructor anidadas con tuplas y tablas: ejemplo

```
const NDNI : entero = 8; MAX : entero = 100 fconst
```

```
tipo
```

```
  tDNI = tabla [0 .. NDNI-1] de caracter;
```

```
  tTablaDNI = tabla [0 .. MAX-1] de tDNI;
```

```
  tM = tabla [0 .. MAX-1, 0 .. MAX-1] de caracter
```

```
ftipo
```

```
var a : tTablaDNI; b : tM fvar
```

```
a[0,4] := 'a';
```

```
b[0][4] := 'a'
```

Tabla multidimensional

Tabla de tablas

## Constructor anidadas con tuplas y tablas: ejemplo

- Tabla de enteros de longitud variable (indicada por el campo sl) i de longitud máxima MAX = 100

```
const MAX : entero = 100 fconst  
tipo  
  tVector = tabla [0 .. MAX-1] de entero;  
  tConj = tupla v : tVector; sl : entero ftupla  
ftipo  
var a : tConj fvar
```



0

sl

MAX

## Operatividad

- Los campos de las tuplas y los componentes de las tablas se operan siguiendo las reglas de los tipos a los que pertenecen
- En principio, las variables de tipo tabla y tupla no se pueden sumar, restar, ... NI comparar, NI asignar

```
tipo tComplejo = tupla r, i : real ftupla ftipo  
var c1, c2 : tComplejo fvar  
{c1, c2 inicializados}  
si c1 = c2 entonces ... fsi  
mientras c1 <> c2 hacer ... fmientras
```

- Se deben comparar todos los campos y componentes de la estructura: iguales?(c1, c2)

## Metodología de diseño: algoritmos iterativos

- Objetivos
- Introducción
- Secuencias
- Esquema de recorrido
- Esquema de búsqueda

## Objetivos

- Reconocer la secuencia que hay en todo algoritmo iterativo
- Estudiar los esquemas de aplicación
- Saber aplicar el esquema de recorrido
- Saber aplicar el esquema de búsqueda

## Introducción

- La construcción de programas eficaces, eficientes y robustos es una tarea compleja
- La depuración por “prueba y error” es ineficiente
- Hace falta una metodologíia que permita diseñar algoritmos correctos
- Esta metodología se basa en el uso de esquemas (solución patrón)
- Esta metodología permite sistematizar la solución de problemas

## Introducción

- Un esquema de programación es un patrón de solución que puede aplicarse para resolver un amplio espectro de problemas concretos
- Para aplicar un esquema de programación hace falta:
  - Conocer los esquemas que pueden usarse
  - Saber a qué tipo de problemas son aplicables los esquemas
  - Saber adaptar el esquema al problema concreto

## Introducción

- En el diseño de una composición iterativa nos debemos centrar en:
  - Las acciones que hay antes del mientras
  - Las acciones que hay dentro del mientras
  - La condición que regula el fin del mientras

acciones que se ejecutan antes del mientras;

**mientras** condición que regula el mientras **hacer**

acciones que se ejecutan dentro del mientras

**fmientras**

## Secuencias

- Toda composición iterativa tiene una secuencia asociada que debemos *identificar* y *caracterizar*
- Identificar una secuencia:
  - determinar el tipo de elementos que la componen
- Caracterizar una secuencia:
  - Determinar el primer elemento
  - Determinar la relación de sucesión (regla que nos permite obtener el siguiente elemento)
  - Determinar el final de la secuencia
    - Último elemento de la secuencia (se tiene que tratar)
    - Centinela: propiedad que caracteriza el primer elemento después del último elemento (no se tiene que tratar)
    - Número de elementos de la secuencia

## Secuencias: ejemplo

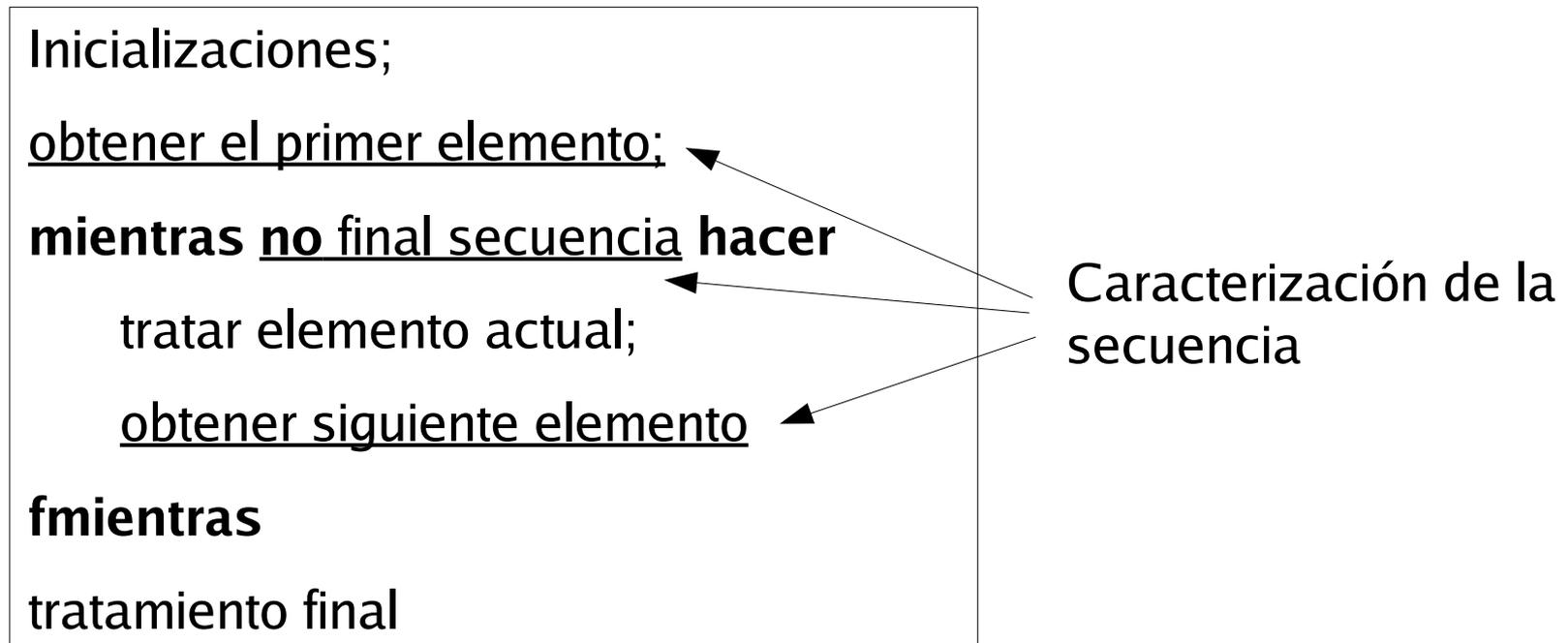
- Identificar y caracterizar la secuencia [-2, 5]
- Identificación: es una secuencia de enteros
- Caracterización:
  - El primer elemento es el entero -2
  - La regla que nos permite calcular un elemento dado el anterior es:  $e$

$$e_{i+1} := e_i + 1$$

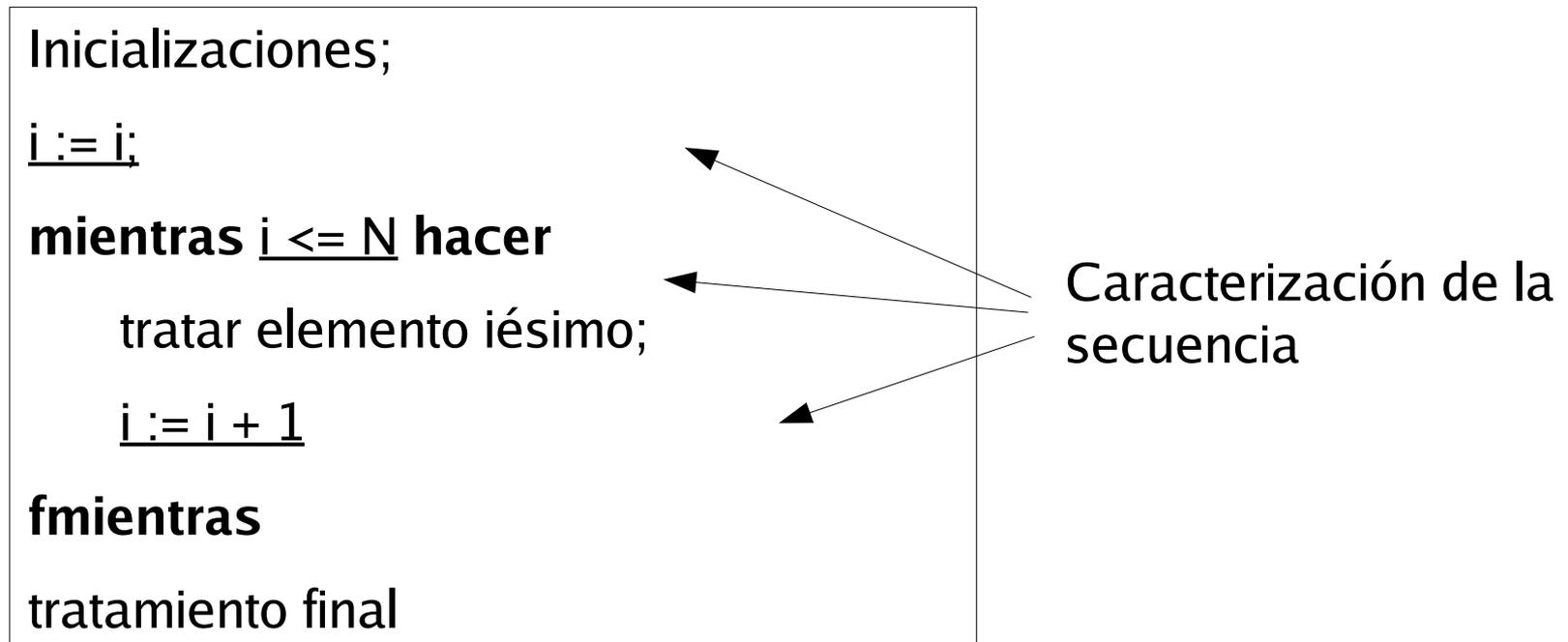
- Podemos determinar el final de la secuencia de tres formas:
  - Último elemento:  $e = 5$
  - Centinela:  $e > 5$
  - Número de elementos: 8

## Esquema de recorrido

- La composición iterativa debe recorrer TODOS los elementos de la secuencia



## Esquema de recorrido en tablas



## Esquema de búsqueda

- La composición iterativa NO SIEMPRE debe recorrer todos los elementos de la secuencia

```
Inicializaciones;  
encontrado := falso;  
obtener el primer elemento;  
mientras no final secuencia y no encontrado hacer  
    si propiedad(elemento actual) entonces  
        encontrado := cierto;  
    sino  
        obtener siguiente elemento  
fsi  
fmientras  
tratamiento final
```