



Tema 3

Subprogramas

Grado de Ingeniería Informática
Introducción a la Programación



Subprogramas

- 3.1. Estructura en subprogramas
 - 3.2. Subprogramas con parámetros
 - 3.3. Vigencia y ámbito
 - 3.4. Aspectos metodológicos
 - 3.5. Aspectos formales
-



Objetivos

- Exponer las principales ventajas de la descomposición de programas en subprogramas
 - Presentar la correcta construcción de programas y de subprogramas
-



3.1 Estructura en subprogramas

- **Problemas al escribir programas:**
 - ❑ Código fuente repetido
 - ❑ Falta de estructuración del código fuente

 - **Solución: subprogramas**
 - ❑ Pueden ser utilizados desde distintos puntos de un programa (evitan la repetición de código)
 - ❑ Un subprograma soluciona una parte del problema inicial (facilita la estructuración)
-



3.1 Estructura de subprogramas

- **Problemas complejos:**

- Se resuelven usando diseño descendente y la estrategia de divide y vencerás

- ***Diseño descendente y divide y vencerás:***

- Dividir el problema en subproblemas cada vez más simples, que se resuelven con **subprogramas**



3.1 Estructura de subprogramas

- Hay dos tipos de subprogramas:
 - **Procedimientos:** **Instrucciones** definidas por el programador
 - **Funciones:** **Expresiones** definidas por el programador



Ejemplo de procedimiento

Dibujar el patrón:

--
|
|
--
|
|
--



Solución NO modular

PROGRAM EseNOModular;

{ Programa que permite dibujar una ESE,
sin utilizar subprogramas

1.- Se dibuja la horizontal

2.- " " " vertical al margen
izquierdo

3.- Se dibuja la horizontal

4.- " " " vertical al margen
derecho

5.- Se dibuja la horizontal de abajo

NOTA: No necesita entradas }

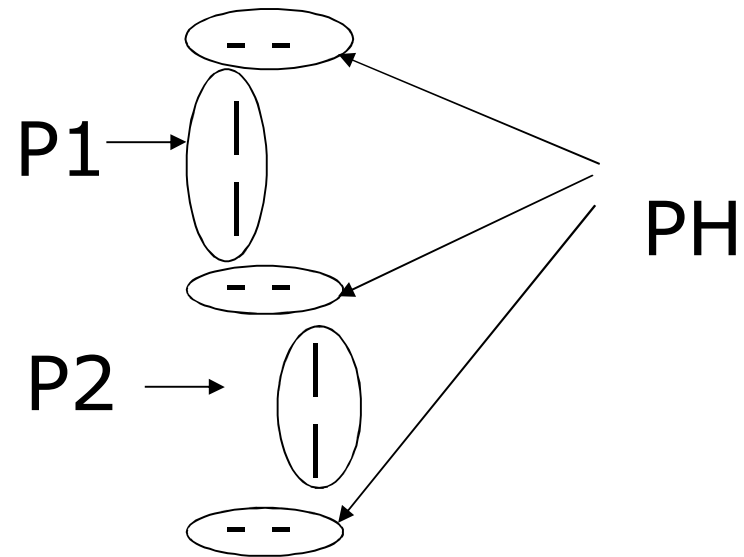


Solución NO modular

```
BEGIN      { Programa Principal}
  {1.- Se dibuja la horizontal y la vertical
    al margen izquierdo}
    writeln('  --');
    writeln('|');
    writeln('|');
  {2.- Se dibuja la horizontal y la vertical
    al margen derecho}
    writeln('  --');
    writeln('    |');
    writeln('    |');
  {3.- Se dibuja la horizontal de abajo}
    writeln('  --');
END.      { Programa Principal}
```

Solución modular

Identificar patrones:





Solución modular

PROGRAM ESeModular;

{Programa que permite dibujar una ESE, utilizando
subprogramas para cada parte de la ESE}

- 1.- DibujaPH;** Dibuja el patrón horizontal
- 2.- DibujaP1;** Dibuja el patrón vertical P1
- 3.- DibujaPH;** Dibuja el patrón horizontal
- 4.- DibujaP2;** Dibuja el patrón vertical P2
- 5.- DibujaPH;** Dibuja el patrón de la horizontal

NOTA: No necesita entradas}



Análisis del ejemplo

- Doble aspecto de los subprogramas
 - **Declaración** de subprogramas
 - Declaraciones y sentencias del subprograma
 - **Uso** de subprogramas (llamadas)
 - Llamada a ejecución del subprograma desde:
 - Otro subprograma
 - El programa principal



Análisis del ejemplo

- **Procedimientos:**

- Las **llamadas** se tratan como **instrucciones** (de igual manera que si estuvieran predefinidas en Pascal)

- **Writeln('hola');**

- (procedimiento predefinido en Pascal)

- **DibujaP1;**

- (procedimiento definido por el programador)



Ejemplo de función (1)

PROGRAM CalculaTangenteGrados;

```
{ Este programa calcula la tangente de un ángulo,  
  dado en grados, y la muestra por la pantalla con  
  una precisión de dos decimales. Los pasos:  
      1.- Lee los grados  
      2.- Calcula la tangente  
      3.- La expresa por pantalla con la  
          precisión que se pide }  
{ variables globales }
```

```
VAR    angulo,  
        tangente:    real;
```



Ejemplo de función (2)

```
FUNCTION TanDeGrados(valor:real) : real;  
  {Pre: un valor que representa el ángulo en grados  
   Post: la tangente del ángulo }  
VAR  
    angRad: real; {ángulo en radianes}  
BEGIN  {Comienza la función TanDeGrados}  
        {Conversión de grados a radianes}  
    angRad:=valor * PI / 180;  
        {Cálculo de la tangente y asignación a la  
         función del valor}  
    TanDeGrados:= sin(angRad) / cos(angRad)  
END; {Salida de la función y devolución del valor  
      calculado}
```



Ejemplo de función (y 3)

```
BEGIN {Programa principal calculaTangenteGrados}
  Write ('Escribe el ángulo en grados:
' );
  Readln (angulo) ;

      {Llamada a la función que calcula la
tangente          y la devuelve}
  tangente:=TanDeGrados (angulo) ;
      {Escritura del resultado formateado}
  Writeln('El valor es: ', tangente:14:2)
END.   {Fin del programa CalculaTangenteGrados}
```



Análisis del ejemplo

■ **Funciones:**

- ❑ **Devuelven** exactamente **un valor**. Dicho valor se determina, asignándolo al “nombre” de la función
- ❑ Las **llamadas** se tratan como **expresiones** (como si estuvieran predefinidas en Pascal)
 - **t := TanDeGrados (a) ;**
(función definida por el programador)
 - **n := abs (a) ;**
(función predefinida en Pascal)



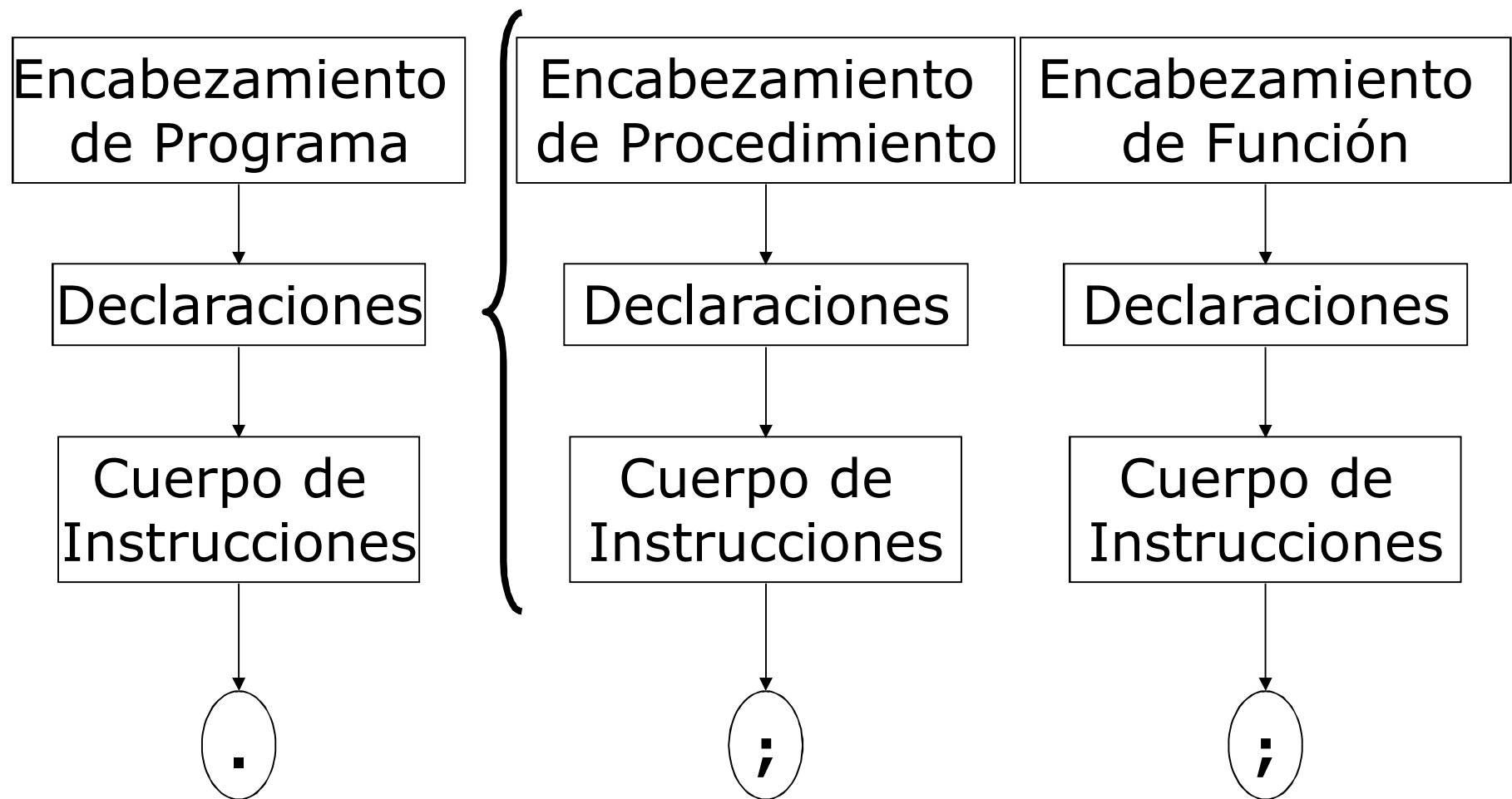
Análisis del ejemplo

■ **Parámetros:**

- Permiten **pasar información** al subprograma
- El **tipo** y el número de argumentos en la **llamada** han de **coincidir** con el **tipo** y el número de los parámetros de la **declaración** del subprograma



Estructura sintáctica de los subprogramas

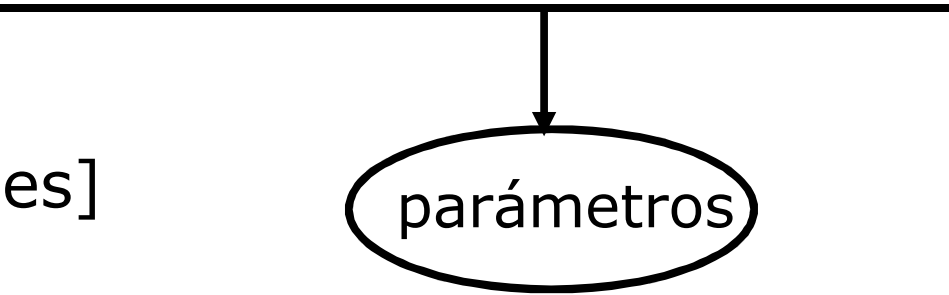




Estructura sintáctica

■ Procedimientos:

PROCEDURE <idproc> [([VAR] [idenpara:tipo,...]; ...)];
[declaraciones]
BEGIN
[Cuerpo de instrucciones]
END;



parámetros



Estructura sintáctica

■ Funciones:

FUNCTION <idenfunc> [([VAR] [idenpara:tipo,...];...)]:
tipo;

[declaraciones]

BEGIN

[Cuerpo de instrucciones]

<idenfunc> := expresion;

END;

tipo de la
función

parámetros



Estructura sintáctica: Declaraciones

■ Declaraciones y definiciones

- Idénticas que las del programa:

constantes

tipos

variables

procedimientos

funciones

Propios del subprograma.
Solo desde él se accede a
ellos

- Estas definiciones y los parámetros constituyen los **elementos locales** del subprograma



Estructura sintáctica: Cuerpo

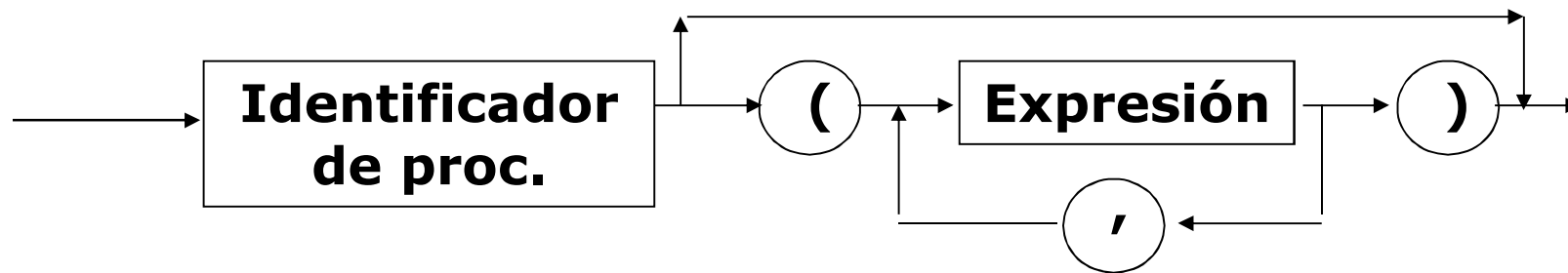
■ **Cuerpo de instrucciones**

- ❑ Secuencia de instrucciones
- ❑ Implementa el algoritmo que resuelve el problema para el que se definió el subprograma
- ❑ Las instrucciones del cuerpo se ejecutan únicamente con cada llamada al subprograma

Estructura sintáctica: Llamadas

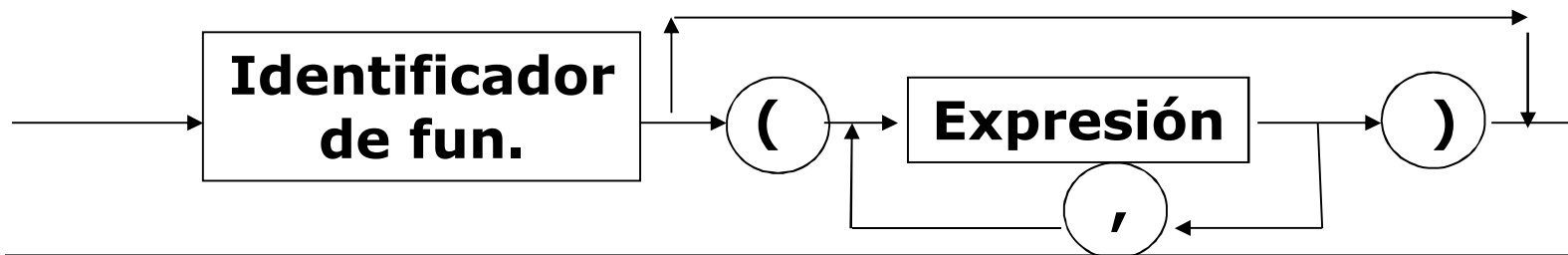
■ Llamadas a procedimientos

- Igual que las **instrucciones predefinidas**



■ Llamadas a funciones

- Igual que las **funciones predefinidas**





Semántica: Llamada a procedimiento

PROGRAM ...

declaraciones

PROCEDURE P (parámetros) ;

declaraciones

BEGIN

instrucciones

END ;

BEGIN

...

P () ;

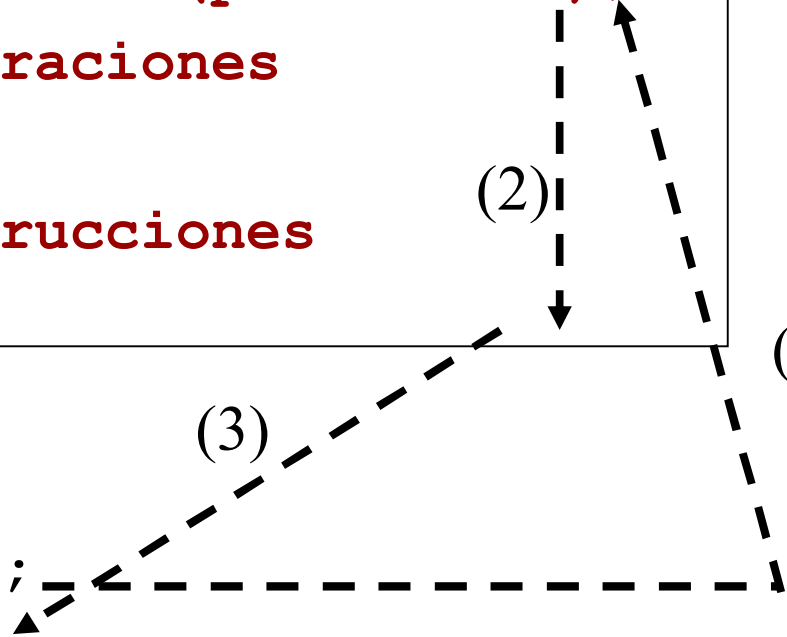
...

END.

(2)

(1)

(3)





Semántica: Llamada a función

PROGRAM ...

declaraciones

```
FUNCTION F (parámetros) : tipoF;  
  declaraciones  
  BEGIN  
    instrucciones  
    F := expresión;  
  END;
```

BEGIN

...

iden := F () ;

...

END .

(2)

(1)

(3)

Iden ha de ser de un tipo compatible con **tipoF**



3.2 Subprogramas con parámetros

- **Parámetros formales** (“ficticios”)

- Se utilizan en la **definición**, en el encabezamiento del subprograma
- Son datos **genéricos**.

```
function TanDeGrados(valor:real) :  
    real;
```



3.2 Subprogramas con parámetros

- **Parámetros reales** (“actuales”)
 - Se utilizan en la **llamada** al subprograma
 - Son variables (o expresiones) que determinan los valores concretos con los que se ejecuta el subprograma

```
t := TanDeGrados(a) ;
```



3.2 Subprogramas con parámetros

- Formas de **intercambiar información** entre el programa principal y el subprograma
 - Parámetros de **entrada** (al subprograma)
 - Parámetros de **salida** (al subprograma)
 - Parámetros de **entrada/salida** (al subprograma)
-



3.2 Subprogramas con parámetros

■ **Entrada:**

Parámetros **POR VALOR:**

- ❑ Utilización: Parámetros de entrada
- ❑ Sintaxis: Igual que una declaración de variable
- ❑ Ejemplo:

```
function TanDeGrados (valor:real) :  
    real;
```



Mecanismo para el paso de parámetros por valor

■ Semántica:

- ❑ Se calcula el valor de los parámetros reales en la llamada (evaluando las expresiones correspondientes)
- ❑ Una copia de dicho valor se asigna a los parámetros formales del subprograma
- ❑ El subprograma opera sobre esta copia
- ❑ Al finalizar el subprograma se pierde su estado de cómputo local y cualquier cambio hecho en el parámetro formal **NO** quedará reflejado en el parámetro real



Mecanismo para el paso de parámetros por valor

■ **Restricciones:**

- ❑ Los parámetros reales pueden ser expresiones, variables o constantes.
- ❑ Los parámetros formales y reales han de ser de tipo **asignación-compatibles**



Mecanismo para el paso de parámetros por valor. Ejemplo

```
PROGRAM ejemploPasoPorValor(input,output);  
VAR w: integer;  
  
PROCEDURE escribirSiguiente1(v:integer);  
BEGIN  
    v:=v+1;  
    writeln(v)  
END; {escSiguiente1}  
  
BEGIN {programa principal}  
    w:=5;  
    writeln(w);  
    escribirSiguiente1(w);  
    writeln(w)  
END. {programa principal}
```

Resultado en
pantalla

5
6
5



3.2 Subprogramas con parámetros

■ **Salida (Entrada / Salida):**

Parámetros **POR REFERENCIA**:

- ❑ Utilización: Parámetros de entrada/salida
- ❑ Sintaxis: Se antepone "VAR" a los parámetros formales en el encabezamiento.
- ❑ Ejemplo:

```
procedure LeePunto (VAR x,y:real) ;
```



Mecanismo para el paso de parámetros por referencia

■ Semántica:

- ❑ Los parámetros reales sustituyen directamente a los parámetros formales (es decir, los parámetros formales son sinónimos de los parámetros reales)
- ❑ El subprograma va modificando dichos parámetros
- ❑ Aunque al finalizar el subprograma se pierde su estado de cómputo local, cualquier cambio hecho en el parámetro formal **SÍ** quedará reflejado en el parámetro real



Mecanismo para el paso de parámetros por referencia

■ **Restricciones:**

- ❑ Solo se permiten **variables** como parámetros reales
- ❑ Los parámetros formales y reales han de ser de tipos **idénticos**



Mecanismo para el paso de parámetros por referencia. Ejemplo

PROGRAM

```
ejemploPasoPorReferencia(input,output);
```

```
VAR w: integer;
```

```
PROCEDURE escribirSiguiente2(VAR v:integer);
```

```
BEGIN
```

```
    v:=v+1;
```

```
    writeln(v)
```

```
END; {escSiguiente2}
```

```
BEGIN {programa principal}
```

```
    w:=5;
```

```
    writeln(w);
```

```
    escribirSiguiente2(w);
```

```
    writeln(w)
```

```
END. {programa principal}
```

Resultado en pantalla

5

6

6



Restricciones de los parámetros reales

- Paso de parámetros por valor
 - Los parámetros reales pueden ser expresiones
 - Parámetros formales y reales han de ser del mismo tipo
(formal: de tipos *asignación-compatibles*)
 - `escribirSiguiente1 (sqr (w*3))`
 - `escribirSiguiente1 (10)`
 - `escribir (sqrt (10))`

ERROR



Restricciones de los parámetros reales

- Paso de parámetros por referencia
 - Los parámetros reales han de ser variables
 - Parámetros formales y reales han de ser de tipos *idénticos*
 - `escribirSiguiente2 (sqr (w*3))`
ERROR
 - `escribirSiguiente2 (10)`
ERROR



Restricciones de los parámetros reales

- Errores de tipo:
 - ❑ ¿Cuáles se detectan en tiempo de compilación?
 - Tipos idénticos y compatibles
 - ❑ ¿Cuáles se detectan en tiempo de ejecución?
 - Tipos asignación-compatibles
 - ❑ ¿Qué reglas hay para los mecanismos de paso de parámetros ?
-



Tipos idénticos

- Los tipos que son usados en dos o más lugares de un programa son ***idénticos***, si:
 - Se definen explícitamente como equivalentes.
 - Ejemplo:
 - Idénticos: T1 con T2
 - No idénticos: T3 con los demás

```
TYPE  T1  = 1..100;  
      T2  = T1;  
      T3  = 1..100;
```



Tipos compatibles

- Son ***compatibles***, si al menos una de las siguientes declaraciones es verdadera:
 - ❑ Ellos son idénticos
 - ❑ Uno es subrango de otro
 - ❑ Ambos son subrangos del mismo tipo
 - ❑ Ejemplo:
 - Compatibles: T5,T6,T7

```
TYPE T5 = (A, B, C, D, E)
      T6 = A..C;
      T7 = D..E;
```



Tipos asignación-compatible

- Una expresión e del tipo t_2 es **asignación-compatible** con el tipo t_1 si:
 - t_1 y t_2 son idénticos
 - t_1 es real y t_2 es entera o un subrango de entero ("conversión automática")
 - t_1 y t_2 son tipos compatibles ordinales, y e es un valor permitido del tipo t_1 ("evitar desbordamientos")
 - Ejemplo:

```
VAR  X: T5;  X:= Y;  {asig.-compatible}  
      Y: T6;  Y:= X;  {a veces asig.-compatible}  
      Z: T7;  Y:= Z;  {nunca asig.-compatible}
```



Compatibilidad de tipos

- **Chequeo de tipos:**

- ❑ **Compilación:** tipos que han de ser idénticos y/o compatibles.
- ❑ **Ejecución:** tipos/valores que han de ser asignación-compatibles.



Compatibilidad de tipos

- **Tipos idénticos:**

- Requeridos con paso de parámetros por referencia

- **Tipos asignación-compatibles:**

- Suficientes en asignaciones normales y con paso de parámetros por valor



Tipos anónimos

■ Tipos anónimos:

- ❑ En Pascal es posibles definir variables de un nuevo tipo, sin que éste tenga nombre
- ❑ Ejemplo:

```
VAR    x: 1..10;  
        y: (A, B, C) ;  
        z: (A, B, C) ;
```

Nótese que z e y **no**
son de tipos idénticos

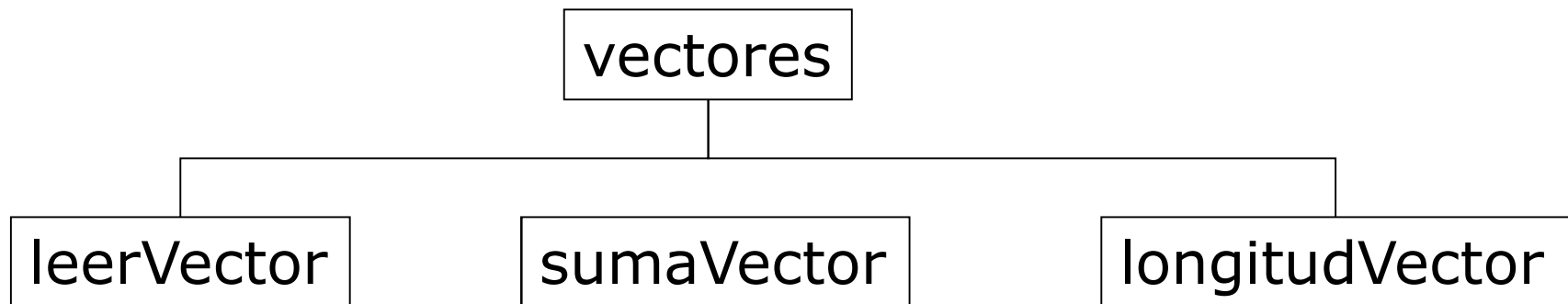
■ Recomendación:

- ❑ Evitar el uso de tipos anónimos



Mecanismos de paso de parámetros. Ejemplo

- Problema: Leer dos vectores del R2 y mostrar su suma y longitud en pantalla
- Idea: Modelar un vector mediante dos variables de tipo real.
- Diagrama estructural:



0 entradas/ 2 salidas 4 entradas / 2 salidas 2 entradas/ 1 salida



Mecanismos de paso de parámetros.

Ejemplo

```
PROGRAM vectores(input,output);  
{Propósito: Leer dos vectores y mostrar su suma y  
             la longitud de la suma en pantalla}  
{Pre:       input = [ $\alpha_1$   $\alpha_2$   $\beta_1$   $\beta_2$  ]}  
{Post:      output = Suma:  $(\alpha_1+\beta_1, \alpha_2+\beta_2)$ .  
                      Longitud:  $\sqrt{(\alpha_1+\beta_1)^2 + (\alpha_2+\beta_2)^2}$ }  
  
VAR x1,x2,y1,y2,z1,z2: real; {Variables globales}  
PROCEDURE leerVector(VAR q1,q2: real);  
{Pre:  true}  
{Post: q1,q2 son números reales}  
BEGIN  
    write('Introduzca el vector: ');  
    readln(q1,q2)  
END; {leerVector}
```




Mecanismos de paso de parámetros. Ejemplo

```
PROCEDURE sumaVector(a1,a2,b1,b2: real;  
                      VAR c1,c2: real);
```

```
{Pre: a1=A1 y a2=A2 y b1=B1 y b2=B2}
```

```
{Post: c1=A1+B1 y c2=A2+B2}
```

```
BEGIN
```

```
    c1 := a1+b1;
```

```
    c2 := a2+b2
```

```
END; {sumaVector}
```

```
FUNCTION longitudVector(a1,a2:real): real;
```

```
{Pre: a1=A1 y a2=A2}
```

```
{Post: longitudVector =  $\sqrt{A1^2+A2^2}$ }
```

```
BEGIN
```

```
    longitudVector := sqrt(sqr(a1)+sqr(a2))
```

```
END; {longitudVector}
```



Mecanismos de paso de parámetros. Ejemplo

```
BEGIN {programa principal}
    leerVector(x1,x2);
    leerVector(y1,y2);
    sumaVector(x1,x2,y1,y2,z1,z2);
    writeln('Suma: (' ,z1:6:2,',',z2:6:2,')');
    writeln('Longitud:' ,
longitudVector(z1,z2):6:2)
END. {programa principal}
```

3.3 Vigencia y ámbito

■ Aspectos a considerar:

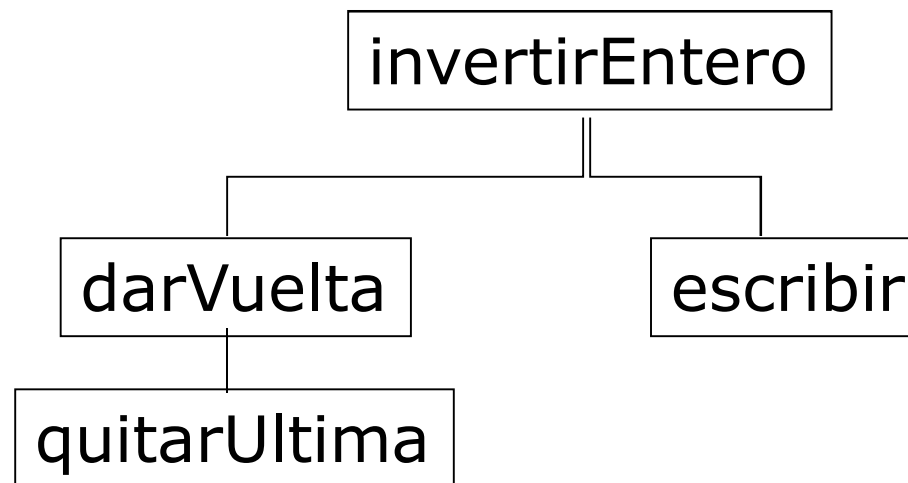
- ❑ Estructura de bloques
 - ❑ Vigencia de objetos
 - ❑ Ámbito de identificadores
 - ❑ Efectos laterales
-

3.3.1 Estructura de bloques

■ Ejemplo:

- Escribir un programa que lea un entero de 4 cifras, invierta el orden de las cifras, y escriba el número inicial y el número invertido en pantalla

■ Diagrama estructural:





Estructura de bloques. Ejemplo

```
PROGRAM invertirEntero;  
VAR numero, numInvertido: integer;  
FUNCTION darVuelta(numero: integer): integer;  
    VAR aux, cifraUnidades: integer;  
    PROCEDURE quitarUltima(VAR n, ultima: integer);  
        BEGIN {quitarUltima}  
            . . .  
        END; {quitarUltima}  
    BEGIN {darVuelta}  
        . . .  
    END; {darVuelta}  
PROCEDURE escribir(x,y: integer);  
    BEGIN {escribir}  
        . . .  
    END; {escribir}  
BEGIN {programa principal}  
    . . .  
END. {programa principal}
```

— **Bloque 1**



Estructura de bloques. Ejemplo

```
PROGRAM invertirEntero;  
VAR numero, numInvertido: integer;  
FUNCTION darVuelta(numero: integer): integer;  
    VAR aux, cifraUnidades: integer;  
    PROCEDURE quitarUltima(VAR n, ultima: integer);  
        BEGIN {quitarUltima}  
            . . .  
        END; {quitarUltima}  
    BEGIN {darVuelta}  
        . . .  
    END; {darVuelta}  
PROCEDURE escribir(x,y: integer);  
    BEGIN {escribir}  
        . . .  
    END; {escribir}  
BEGIN {programa principal}  
    . . .  
END. {programa principal}
```

— **Bloque 1**

— **Bloque 1.1**

— **Bloque
1.2**



Estructura de bloques. Ejemplo

```
PROGRAM invertirEntero;
```

```
VAR numero, numInvertido: integer;
```

```
FUNCTION darVuelta(numero: integer): integer;
```

```
VAR aux, cifraUnidades: integer;
```

```
PROCEDURE quitarUltima(VAR n, ultima: integer);
```

```
BEGIN {quitarUltima}
```

```
    . . .
```

```
END; {quitarUltima}
```

```
BEGIN {darVuelta}
```

```
    . . .
```

```
END; {darVuelta}
```

```
PROCEDURE escribir(x,y: integer);
```

```
BEGIN {escribir}
```

```
    . . .
```

```
END; {escribir}
```

```
BEGIN {programa principal}
```

```
    . . .
```

```
END. {programa principal}
```

Bloque 1

Bloque 1.1

Bloque 1.1.1

Bloque 1.2



Estructura de bloques

■ Ejemplo:

- Escribir un programa que lea un entero de 4 cifras, invierta el orden de las cifras, y escriba el número inicial y el número invertido en pantalla

```
PROGRAM invertirEntero (input, output);  
    { Propósito: invierte el orden de las  
    cifras de un entero }  
    { Pre:   input = [N] y  $1000 \leq N \leq 9999$  }  
    { Post: output= [M] y  $M = \text{inv. Cifras de } N$  }  
  
VAR numero, numInvertido: integer;
```



Estructura de bloques

```
FUNCTION darVuelta(numero: integer): integer;  
  { Pre: 1000<= numero <=9999 }  
  { Post: 1000<= darVuelta <=9999 y darVuelta = inverso  
    cifras de numero}
```

```
VAR aux, cifraUnidades: integer; {Var. locales de  
  darVuelta}
```

```
PROCEDURE quitarUltima(VAR n, ultima: integer);  
  { Pre: 0<= n <=9999 }  
  { Post: n es sin_ultima-cifra(n) y ultima es la ultima  
    cifra de n }  
  { NOTA: n es parametro de entrada/salida}
```

```
  BEGIN {quitarUltima}  
    ultima := n mod 10;  
    n := n div 10;  
  END; {quitarUltima}
```



Estructura de bloques

```
BEGIN {darVuelta}
    quitarUltima(numero,cifraUnidades);
    aux := cifraUnidades * 1000;
    quitarUltima(numero,cifraUnidades);
    aux := aux + cifraUnidades * 100;
    quitarUltima(numero,cifraUnidades);
    aux := aux + cifraUnidades * 10;
    quitarUltima(numero,cifraUnidades);
    darVuelta := aux + cifraUnidades;
END; {darVuelta}
```



Estructura de bloques

```
PROCEDURE escDebajo(x,y: integer);  
  {Pre:  x,y son enteros }  
  {Post: output = [x ↵ y] }
```

```
  BEGIN {escDebajo}  
    writeln('Número 1:',x:5);  
    writeln('Número 2:',y:5)  
  END; {escDebajo}
```

```
BEGIN {programa principal}  
  writeln('Introducir número: ');  
  readln(numero);  
  numInvertido := darVuelta(numero);  
  escDebajo(numero,numInvertido);  
END. {programa principal}
```



Alcance de las variables

- **VARIABLES GLOBALES:**
 - Se declaran en el bloque del **programa principal**
Ejemplo: `numero`, `numInvertido`
 - **VARIABLES LOCALES A UN SUBPROGRAMA:**
 - Se declaran en el bloque del mismo **subprograma**
Ejemplo: `aux` es variable local de `darVuelta`
 - **VARIABLES NO-LOCALES A UN SUBPROGRAMA:**
 - Se declaran en un bloque **exterior al subprograma**
Ejemplo: `aux` es una variable no local de `quitarUltima`
-



3.3.2 Vigencia

- **Vigencia** o vida de un objeto:
 - Los bloques del programa en los que el objeto “**existe**” (i.e. tiene espacio de memoria asignado)

 - Regla en Pascal
 - Un objeto es **vigente** en el **bloque** en el que está definido **y** en todos los **bloques interiores** a él
-



3.3.3 Ámbito

- **Ámbito** o visibilidad de un identificador:
 - Los bloques en los que se puede **acceder** a un objeto
 - Regla en Pascal
 - El ámbito de un identificador es el **bloque** del subprograma en el que está definido, incluyendo todos los **bloques interiores** a él
 - **EXCEPTO** si en los bloques interiores hay un **identificador idéntico** que lo **oculte**
-



Ámbito. Ejemplo

```
PROGRAM invertirEntero;
```

```
VAR numero, numInvertido: integer;
```

```
FUNCTION darVuelta (numero: integer): integer;
```

```
VAR aux, cifraUnidades: integer;
```

```
  PROCEDURE quitarUltima (VAR n, ultima: integer);
```

```
    BEGIN {quitarUltima}
```

```
      . . .
```

```
    END; {quitarUltima}
```

```
  BEGIN {darVuelta}
```

```
    . . .
```

```
  END; {darVuelta}
```

```
  PROCEDURE escribir(x,y: integer);
```

```
    BEGIN {escribir}
```

```
      . . .
```

```
    END; {escribir}
```

```
BEGIN {programa principal}
```

```
  . . .
```

```
END. {programa principal}
```

Bloque 1

Bloque 1.1

Bloque 1.1.1

Bloque 1.2



Ámbito. Ejemplo

- ❑ El ámbito del identificador `numero` del bloque 1 son todos los bloques excepto en el bloque 1.1
- ❑ En el bloque 1.1 está oculto por el parámetro formal con el mismo nombre
- ❑ Por lo tanto, al utilizar el identificador **`numero`**, en el bloque 1.1 nos referimos al parámetro formal del subprograma correspondiente. En los demás bloques, nos referimos a la variable global



3.3.4 Efectos laterales

■ Efectos laterales:

- Cuando un subprograma **influye** directamente en el estado de cómputo de **otros**, es decir, **sin** que estos efectos sean producidos por el **paso de parámetros**
- Ejemplo:

```
PROCEDURE incrementarContador;  
  {Modifica la variable no local "contador"}  
BEGIN  
    contador := contador+1  
END {incrementarContador}
```

3.3.4 Efectos laterales

■ Efectos laterales:

- ❑ Los efectos laterales se producen al **asignar** valores a variables **globales** o **no-locales** a un subprograma
- ❑ Aunque dichas asignaciones son sintácticamente correctas, **quedan absolutamente PROHIBIDAS** porque:
 - **Disminuyen** la **reusabilidad** del subprograma
 - **Dificultan** la **depuración** y **verificación** del programa

3.4 Aspectos metodológicos

- **Desarrollo o diseño descendente**

- Proceso de **descomposición** y **refinamiento progresivo** de un problema en subproblemas más pequeños hasta llegar a un problema fácilmente resoluble por la mente humana.

- **Refinamiento sucesivo**

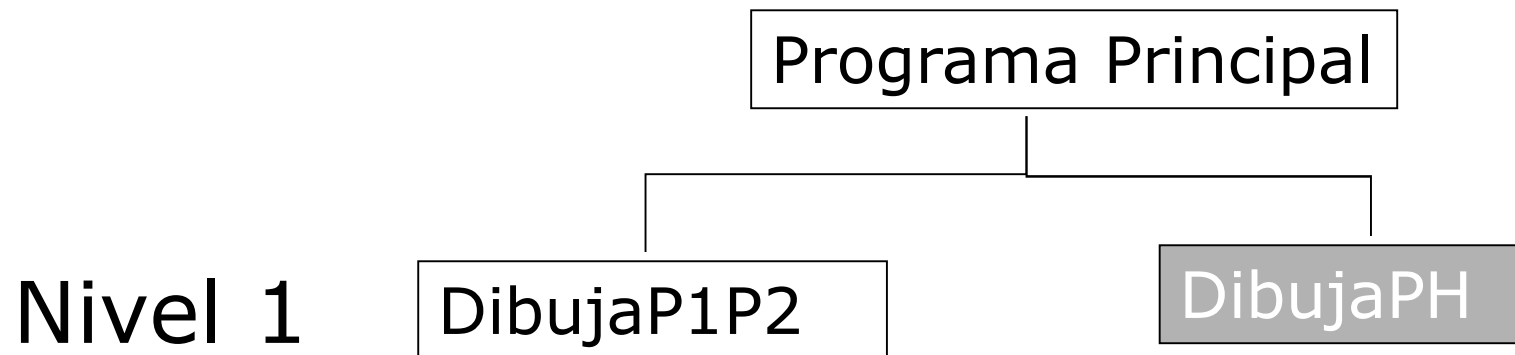
- **Descomposición por niveles** cada vez más específicos (soluciones más detalladas).
 - Programa principal
 - Subprogramas del siguiente nivel
 - Subprogramas de niveles sucesivos

3.4.1 Desarrollo descendente

- **Desarrollo descendente y subprogramas**
 - ❑ Se suele **definir** un **subprograma** para cada **subproblema**
 - ❑ La precondition y la postcondición del subprograma especifican el (sub-)problema que resuelve
 - ❑ El cuerpo del subprograma implementa el algoritmo que resuelve el subproblema
 - ❑ Los diagramas estructurales ilustran la descomposición del problema en subproblemas/subprogramas
-

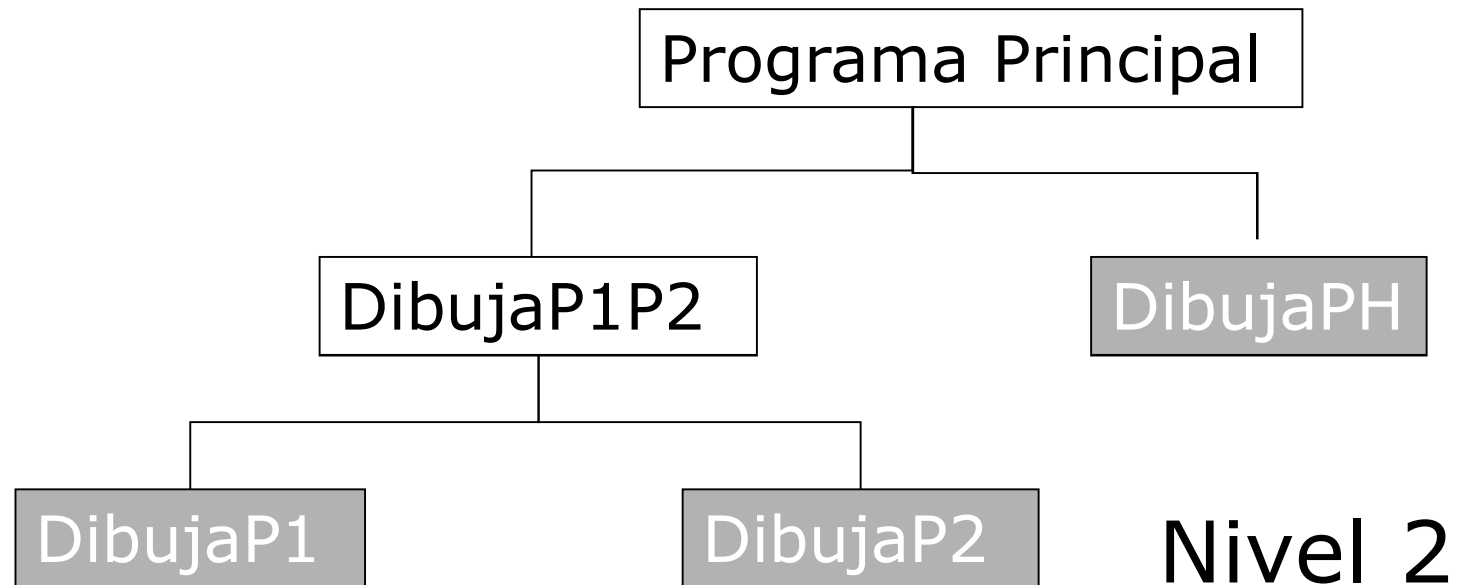


Ejemplo: Dibujar una Ese





Ejemplo: Dibujar una Ese





Recomendaciones técnicas

- Utilizar funciones y procedimientos en el marco del **desarrollo descendente**

 - Aspectos relevantes:
 - ❑ **Encapsulación**
 - ❑ **Variables globales**
 - ❑ **Procedimiento o función**
 - ❑ **Parámetros por valor o por referencia**
-



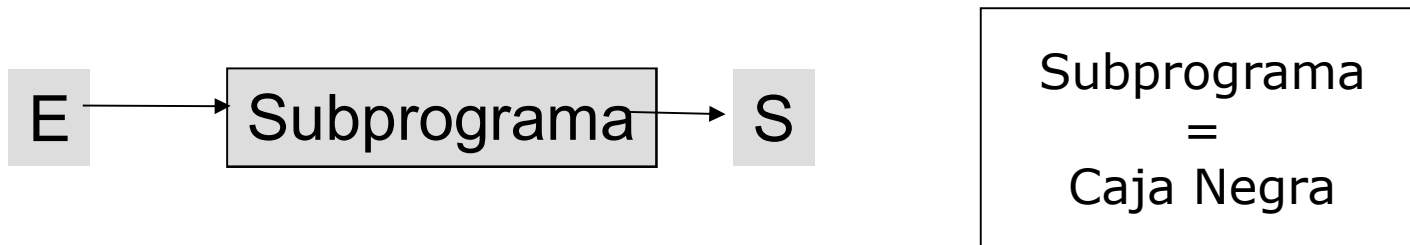
Encapsulación

- **Principio de máxima localidad**

- ❑ Los objetos particulares y necesarios para un subprograma, especialmente las variables, deben ser locales al mismo

- **Principio de autonomía de los subprogramas**

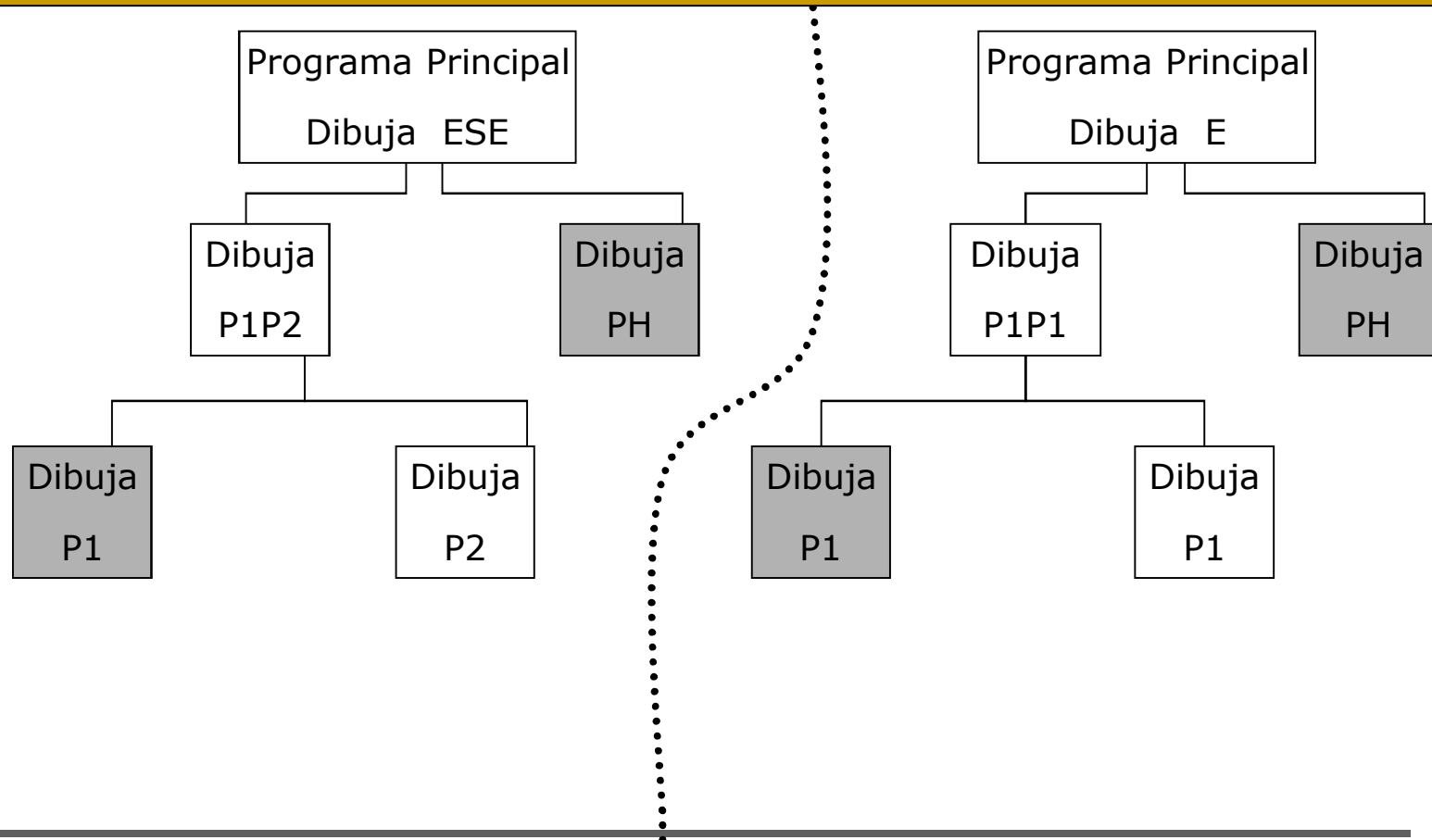
- ❑ La comunicación con el exterior debe realizarse exclusivamente mediante parámetros, evitándose dentro de los subprogramas toda referencia a objetos globales





Reutilización mediante encapsulación

P1, PH: Subprogramas usados tanto por DibujaESE como por DibujaE





Variables globales

- Un **subprograma** debe **usar** sus **variables locales**
- Un **subprograma NO** debe usar **variables globales**
- **Lectura y modificación de variables globales** en un subprograma: pasándolas como **parámetros por referencia**
- Para **usar** el valor de variables globales en subprogramas: pasarlas como **parámetros por valor**



Variables globales

- **Escritura** de variables globales en un subprograma:
 - Para **cambiar** el valor de variables globales debemos pasarlas como **parámetros por referencia**
- Queda **ABSOLUTAMENTE PROHIBIDO** modificar el valor de las variables globales directamente en un subprograma (efectos laterales)



¿Procedimiento o función?

■ **Procedimiento**

- ❑ Cuando no se devuelve ningún valor
- ❑ Cuando se devuelve más de un valor

■ **Función**

- ❑ Cuando se quiere representar un cálculo que devuelve un sólo valor



¿Parámetros por valor o por referencia?

■ **Parámetros por referencia**

- ❑ Usar para la **salida** de valores (excepcionalmente como parámetros de entrada/salida)
- ❑ Evitar parámetros por referencia en funciones

■ **Parámetros por valor**

- ❑ Usar como **entrada** de valores
- ❑ EXCEPCIÓN: estructuras de datos muy grandes.
- ❑ No deben modificarse en el subprograma



Ejemplo MUY MAL

```
PROGRAM muyMal (input, output);  
    {Muy mal porque utiliza una variable global  
    para la salida del subprograma, causando un  
    efecto lateral}  
VAR entrada, resultado: integer; {Variables  
    globales}  
PROCEDURE siguiente (x: integer);  
BEGIN  
    resultado := x+1  
END; {siguiente}  
BEGIN {programa principal}  
    write('Introduzca un número: ');  
    readln(entrada);  
    siguiente(entrada);  
    writeln('El siguiente número es:  
    ', resultado)  
END. {programa principal}
```



Ejemplo BASTANTE MAL

```
PROGRAM bastanteMal (input, output);  
    {Bastante mal porque utiliza una variable  
    global para la entrada al subprograma,  
    causando un efecto lateral}  
  
VAR entrada, resultado: integer; {Variables  
    globales}  
  
PROCEDURE siguiente (VAR y: integer);  
BEGIN  
    y := entrada +1  
END; {siguiente}  
  
BEGIN {programa principal}  
    write('Introduzca un número: ');  
    readln(entrada);  
    siguiente(resultado);  
    writeln('El siguiente número es:  
    ', resultado)  
END. {programa principal}
```



Ejemplo BASTANTE BIEN

```
PROGRAM bastanteBien (input, output);  
    {Bastante bien porque no utiliza variables  
    globales en el subprograma pero usa un  
    parámetro de entrada/salida}  
VAR entres: integer; {Variable global}  
PROCEDURE siguiente (VAR x: integer);  
BEGIN  
    x := x+1  
END; {siguiente}  
BEGIN {programa principal}  
    write('Introduzca un número: ');  
    readln(entres);  
    siguiente(entres);  
    writeln('El siguiente número es:  
    ', entres)  
END. {programa principal}
```




Ejemplo MUY BIEN

```
PROGRAM muyBien(input,output);  
    {Muy bien porque no utiliza variables globales  
    en el subprograma y utiliza un parámetro de  
    entrada y otro de salida}  
VAR entrada, resultado: integer; {Variables  
    globales}  
PROCEDURE siguiente (x: integer; VAR y: integer);  
BEGIN  
    y := x+1  
END; {siguiente}  
BEGIN {programa principal}  
    write('Introduzca un número: ');  
    readln(entrada);  
    siguiente(entrada, resultado);  
    writeln('El siguiente número es:  
, resultado)  
END. {programa principal}
```

3.5 Aspectos formales

- Aspectos formales de subprogramas:
 - Análisis de complejidad
 - **Análisis de corrección**
 - Corrección sintáctica
 - Corrección semántica: depuración
 - Corrección semántica: verificación formal



3.5.1 Análisis de corrección

■ **Corrección sintáctica:**

- Del subprograma
 - Diagramas sintácticos (i.e. similar al programa principal)
- De la llamada
 - Las llamadas a funciones se usan como expresiones, las llamadas a procedimientos se usan como instrucciones
 - Los parámetros formales y los reales tienen que coincidir en número y tipo
 - Además, para poder aplicar el mecanismo de paso de parámetros por referencia, los parámetros formales y reales han de ser de tipos idénticos



Depuración

■ Depuración de la llamada

- ❑ Depurar la llamada como una sola instrucción / expresión (definida por el programador)
- ❑ Se ejecuta el cuerpo completo del subprograma y se examina el estado de cómputo resultante
- ❑ En el depurador de TurboPascal (para DOS):
 - Menú "Run"
 - Opción "Step Over"



Depuración

■ **Depuración del subprograma**

- ❑ Depurar las instrucciones del subprograma una por una
 - ❑ Se examinan sucesivamente los estados de cómputo locales del subprograma
 - ❑ En el depurador de TurboPascal (para DOS):
 - Menú "Run",
 - Opción "Trace Into"
-



Verificación

- Verificación de la corrección parcial de la llamada
 - **Recomendaciones técnicas:**
 - En el cuerpo del subprograma S no aparecen variables globales
 - En la precondition p y en la postcondition q de la especificación del subprograma no aparecen variables locales
 - En el cuerpo del subprograma S no hay asignaciones a parámetros por valor
 - En las funciones no hay parámetros por referencia
 - Se consideran solo subprogramas acordes con las recomendaciones técnicas



Proceso de corrección y depuración

■ **Proceso:**

- ❑ Los subprogramas se deben poder depurar, verificar y probar independientemente
- ❑ La construcción, depuración, verificación se hará de forma incremental

- **Construcción:** Desde los niveles de mayor abstracción a los de menor abstracción (diseño descendente)

- **Depuración y verificación:** Se suelen aplicar primero a los subprogramas de menor abstracción, para luego ir "ascendiendo"



Recomendaciones

- Para utilizar de manera correcta la subprogramación debe seguir los siguientes pasos:

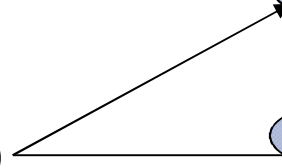
1. Analizar el problema

2. Diseñar un algoritmo

- Diseñar el programa principal, utilizando subprogramas (instrucciones o funciones nuevas)
- Diseñar los subprogramas

Datos

Resultados





Recomendaciones

3. Implementar el programa
 - ▣ Escribir el programa y las cabeceras de todos los subprogramas
 - ▣ Escribir un subprograma y probarlo
 - ▣ Si quedan subprogramas, volver al apartado 3.2 de nuevo
 4. Probar el programa completo
-



ascension.lovillo@urjc.es
