UNIVERSIDAD DE MALAGA
DPTO. DE LENGUAJES Y C. DE LA COMPUTACION
E.T.S. DE INGENIERIA INFORMATICA
INGENIERIA INFORMATICA

ELEMENTOS DE PROGRAMACIÓN

(CURSO 2006-2007)

TEMA VI

SUBPROGRAMAS

VI.1. Subprogramas. Procedimientos y funciones.
VI.1.1. Subprogramas como abstracción de operaciones.
VI.1.2. Declaración y llamada a subprogramas.
VI.1.3. Parámetros formales y reales.
VI.1.4. Interfaz.
VI.1.5. Paso de parámetros por valor y referencia.
VI.1.6. Procedimientos y funciones.
VI.1.7. Criterios de modularización.
VI.2. Anidamientos y ámbitos.
VI.2.1. Subprogramas anidados. Reglas de ámbito.
VI.2.2. Declaraciones locales y globales.

Bibliografía: [DALE89a][JOYA03]

VI.1.- SUBPROGRAMAS. PROCEDIMIENTOS Y FUNCIONES.

VI.1.1.- Subprogramas como abstracción de operaciones.

- Diseño descendente:

  Un problema se divide en subproblemas (a varios niveles) para conseguir una resolución más sencilla del mismo.

- Ejemplo: Algoritmo para calcular un número combinatorio.

$$\binom{m}{n} = \frac{m!}{n!\ (m-n)!}$$

Primer nivel de refinamiento:

```
Algoritmo calcular_numero_combinatorio
Inicio
    • leer los valores de m y n (m ≥n)
    • calcular el combinatorio
    • escribir el resultado
Fin
```

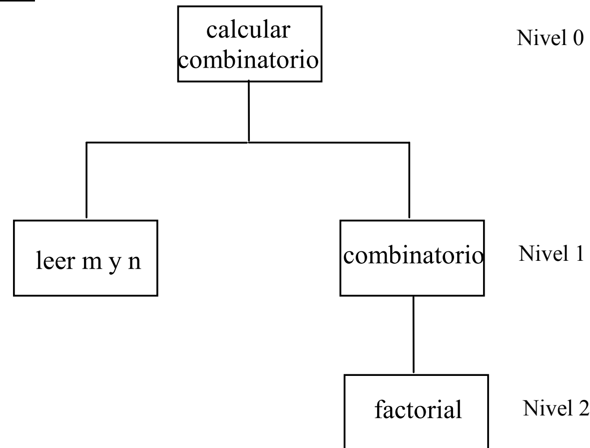
Segundo nivel de refinamiento:

```
PROC combinatorio(↓m,n:N;↑comb:N)
Inicio
    • calcular m!
    • calcular n!
    • calcular (m-n)!
    • aplicar
          comb = m! / (n! (m-n)!)
Fin
```

```
PROC leer_datos(↑m,n:N)
Inicio
    • leer m y n hasta m ≥ n
Fin
```

Tercer nivel de refinamiento:

```
PROC factorial(↓x:N;↑fact:N)
Inicio
    • calcular el producto acumulado
        fact = 1* 2 * ... * x
Fin
```



Los lenguajes de programación (y nuestro pseudolenguaje también) nos van a permitir escribir un programa de una forma modular, respetando la descomposición obtenida en el análisis descendente.

A cada módulo ----------> un Algoritmo

Un módulo es un *algoritmo autocontenido* fruto de la descomposición de un problema al aplicar el diseño descendente. Dicho módulo puede ser diseñado "independientemente" del ámbito en el que va a ser usado.

Al código correspondiente al módulo raiz o principal le denominaremos **algoritmo principal** o simplemente programa. Al código correspondiente a cada uno de los restantes módulos le llamaremos subprograma o **subalgoritmo**.

En el ejemplo anterior, nuestro algoritmo principal sería:

```
Algoritmo calcular_numero_combinatorio
Variables
    m,n,comb: N
Inicio
    leer_datos(m,n)
    combinatorio(m,n,comb)
    escribir("el resultado es: ",comb)
Fin
```

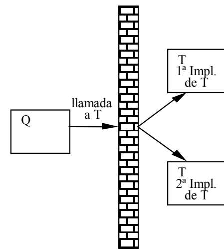
y los subalgoritmos:

```
PROC leer_datos(↑m,n:N)
Inicio
    REPETIR
        escribir("Introduzca dos numeros:")
        leer(m,n)
    HASTA QUE (m ≥ n))
Fin

PROC factorial(↓x:N;↑fact:N)
Variables
    i: N
Inicio
    fact ← 1
    PARA i ← 1 HASTA x HACER
        fact ← fact * i
    FINPARA
Fin

PROC combinatorio(↓m,n:N;↑comb:N)
Variables
    m_fact, n_fact, mn_fact: N
Inicio
    factorial(m, m_fact)
    factorial(n, n_fact)
    factorial(m-n, mn_fact)
    comb ← (m_fact DIV (n_fact * mn_fact))
Fin
```

Este enfoque proporciona indudables **ventajas**:

- Simplificación del diseño.
- Posible programación aislada.
- Posibilidad de reutilización del módulo en otro contexto.
- No sólo simplifica el diseño de algoritmos sino también su comprensión.

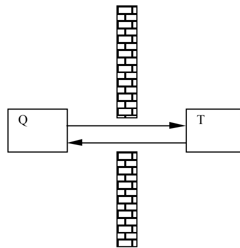
La modularidad aisla (**encapsula**) las diferentes tareas que componen un programa. Como beneficio, si el método para solucionar una tarea T cambia, el aislamiento evita que dicho cambio influya en cualquier otra tarea Q.



Ejemplo: T es nuestro procedimiento "combinatorio" y Q nuestro algoritmo principal "calcular_numero_combinatorio". T puede modificar el método para calcular un número combinatorio. Esto no afecta a Q.

Sin embargo, el aislamiento de los módulos no puede ser total; aunque a un módulo Q le baste saber qué resuelve un módulo T y no cómo, hay ciertas condiciones que debe cumplir Q para que T realice su tarea satisfactoriamente.

Ejemplo. El procedimiento "combinatorio" debe recibir dos números naturales "m" y "n" (y no dos reales, por ejemplo) tales que $\mathbf{m \geq n}$, y tiene que devolver otro número natural como resultado.



VI.1.2.- Declaración y llamada a subprogramas.

- Cuando para procesar un algoritmo A necesitemos emplear otro algoritmo B lo expresaremos diciendo que "el algoritmo A *llama* al algoritmo B".

- Necesitaremos una instrucción de llamada en el primero, que provocará la ejecución del segundo sobre los valores que le indiquemos, y puede que nos devuelva unos resultados en la forma adecuada.

- Para las instrucciones que sigan a la llamada podemos suponer que se dispone de los resultados correctamente.

- La llamada a un subprograma se realiza escribiendo el nombre del mismo seguido de las expresiones sobre las que queremos que trabaje.

- Creamos una via de comunicación entre la instrucción de llamada y el subprograma llamado, que facilita el intercambio de información entre ambos.

- En nuestro pseudolenguaje:

    - Declaración de un subprograma:

```
PROC  nombre(parámetros)
   declaraciones
Inicio
   sentencias
Fin
```

    - LLamada a un subprograma:

```
nombre(parámetros)
```

Cuando se produce una llamada a un subprograma, se establecen las vias de comunicación entre los algoritmos llamante y llamado. Se crean

las variables especificadas en la zona de declaraciones, que inicialmente tendrán un valor indefinido. Posteriormente el flujo de control pasa a la primera instrucción del cuerpo del subprograma llamado, ejecutándose éste. Cuando finaliza la ejecución, las variables previamente creadas se destruyen y el flujo de control continúa por la siguiente instrucción a la llamada realizada.

VI.1.3.- Parámetros formales y reales.

Parámetros formales:

Son los que aparecen e**n la cabecera** de la declaración de un subalgoritmo. Son nombres de variables separados por comas (o por puntos y comas), junto con su tipo y una especificación del sentido de la transferencia de información (↓,↑,↕)

PROC nombre(parámetros formales)

Parámetros reales o actuales:

Son los que aparecen **en la llamada** al subalgoritmo. Son variables o expresiones separadas por comas.

nombre(parámetros reales)

Los **parámetros** han de someterse a unas reglas:

1) Número de parámetros formales = Número de parámetros reales.
2) El i-ésimo parámetro formal se corresponde con el i-ésimo parámetro real
3) El tipo del i-ésimo parámetro formal debe ser igual que el tipo del i-ésimo parámetro real.
4) Los parámetros de un subalgoritmo pueden ser de cualquier tipo, al igual que cualquier variable.
5) Los nombres de un parámetro formal y su correspondiente real pueden o no ser diferentes.
6) Un parámetro formal de entrada permite constantes y expresiones como parámetro real. En cambio, un parámetro formal de salida o de entrada/salida requiere una variable como parámetro real.

- Ejercicio sobre paso de parámetros:

Supongamos que en un programa principal se declaran las siguientes variables:

```
x,y: R
m: Z
c: C
```

y tenemos un subalgoritmo con la siguiente cabecera:

```
PROC prueba(↓ a,b : Z; ↕ c,d : R; ↓e : C)
```

averigua cuáles de las siguientes llamadas a prueba desde el programa principal son incorrectas y cuál es la razón:

```
prueba(m+3,10,x,y,c)  prueba(m,m*m,y,x,c)       prueba(m,3.5,x,y,c)
prueba(30,10,m,x,c)   prueba(35,m*10,x,c,y)     prueba(30,10,x,x+y,c)
prueba(m,19,x,y)      prueba(m,10,35.0,y,'E')   prueba(30,10,c,d,e)
```

VI.1.4. Interfaz.

- Término genérico aplicable a diferentes ramas: define la interaccion entre dos entidades.
- En el contexto de subprogramas se denomina interfaz a la forma en que se comunican y cooperan dos subprogramas.
- Los errores en el uso de subprograma se presentan fundamentalmente debido a una interfaz incorrecta entre el algoritmo llamante y el llamado.
- Para diseñar el interfaz debemos considerar:
    - ¿Qué información del programa llamante necesita conocer el subprograma para poder trabajar correctamente?
    - ¿Qué información producirá el algoritmo llamado que después sea necesitada en el algoritmo llamante?
    - ¿Bajo qué condiciones se realiza este intercambio de información
- Atendiendo a la solución dada a las preguntas anteriores se definen parámetros capaces de comunicarla

VI.1.5. Paso de parámetros por valor y por referencia.

- En la práctica, los lenguajes de programación normalmente implementan el flujo de información establecido en el interfaz (entrada, salida o entraqda/salida) mediante los mecanismos conocidos como paso por valor y paso por referencia.
- El paso por valor consiste en almacenar en el parámetro formal una copia del valor del parámetro real. Como consecuencia, cualquier modificación sobre el parámetro formal no afecta al parámetro real. Los dos parámetros estan totalmente aislados.
- El paso por referencia consiste en almacenar en el parámetro formal una referencia a la variable situada como parámetro real. Como consecuencia, la variable especificada como parámetro formal pasa a ser la misma que la variable especificada en el parámetro real, por lo que cualquier modificación del parámetro formal implica en la práctica una modificación del parámetro real.

Características de ambas formas de comunicación.

- Paso por valor:
    - Sólo permite comunicación de entrada (↓).
    - Aisla.
    - Permite constantes y expresiones como parámetro real.
    - Duplica memoria y realiza copia
    - Utiliza más memoria en el caso de tipos estructurados.
- Paso por referencia:
    - Permite comunicación de entrada (↓) si no hay modificación del parámetro formal, salida (↑) si la primera utilización del parámetro formal es para darle un valor y entrada/salida (↕).
    - No aisla.
    - Sólo permite la variable como parámetro real.
    - No duplica memoria y no realiza copia
    - Utiliza menos memoria en el caso de tipos estructurados

VI.1.6.- Procedimientos y funciones.

- Todos los ejemplos de subalgoritmos vistos hasta ahora son procedimientos, con lo que la declaración de un procedimiento se hará de la forma:

```
PROC nombre(parámetros formales)
    declaraciones
Inicio
    acciones
Fin
```

y la llamada a un procedimiento será de la forma:

nombre(parámetros reales)

llamada que **por sí sola constituye una acción** o sentencia en el cuerpo del algoritmo "llamante" (sea el algoritmo principal u otro subalgoritmo).

- Supongamos que necesitamos conocer el más pequeño de 2 números, y para ello construimos el siguiente procedimiento:

```
PROC pequeño(↓ x,y : Z; ↑ resultado : Z)
Inicio
    SI x < y ENTONCES
        resultado ← x
    SINO
        resultado ← y
    FINSI
Fin
```

- Hay muchos casos como éste. El subalgoritmo realiza una serie de acciones y devuelve un resultado (y sólo uno) al algoritmo llamante mediante alguno de sus parámetros. Es mejor utilizar un tipo de subalgoritmo llamado función.

- Una función es un subalgoritmo declarado de la forma:

```
FUNC nombre(parámetros formales): TipoResultado
    declaraciones
Inicio
    acciones
Fin
```

- En toda función existe una variable predefinida llamada **RESULTADO** (del tipo que la función devuelve). El valor que la función desee devolver será el que dicha variable tenga al final de la función.

- Aunque no hay ninguna restricicción al respecto, es recomendable que todos los parámetros formales de la función sean de entrada. Por tanto, al diseñar un subprograma, si pretendemos que devuelva más de un valor, lo más adecuado es definirlo como un procedimiento con varios parámetros de salida.

- **La llamada a una función no puede constituir por sí sola una sentencia del algoritmo llamante**, sino que debe aparecer dentro de alguna sentencia del algoritmo llamante en la que el valor devuelto por la función (al terminar su ejecución), sea "utilizado" de alguna forma, por ejemplo a la derecha de una sentencia de asignación. En nuestro ejemplo:

```
FUNC pequeño_f(↓ x,y : Z): Z
Inicio
    SI x < y ENTONCES
        RESULTADO ← x
    SINO
        RESULTADO ← y
    FINSI
Fin
```

En el algoritmo que la llama se podría poner:

```
menor ← pequeño_f(a,b)            ó bien
escribir(pequeño_f(a,b) +1)
```

siendo menor, a, b variables del algoritmo llamante.

Las funciones que devuelven un valor booleano son particularmente útiles, convirtiéndose en las expresiones booleanas de decisión para la sentencia SI y en las condiciones de control para los bucles. Por ejemplo, un algoritmo que pretenda hacer una u otra cosa dependiendo de que un número sea o no primo, se puede escribir

```
...
SI Es_primo(numero) ENTONCES
    acciones
SINO
    acciones
FINSI
...
```

```
FUNC Es_primo(↓ Num:N):B
Variables
    pos_divisor: N
Inicio
    pos_divisor ← 2
    MIENTRAS (pos_divisor < Num) ∧
             (Num MOD pos_divisor ≠ 0) HACER
        pos_divisor ← pos_divisor + 1
    FINMIENTRAS
    RESULTADO ← pos_divisor ≥ Num
Fin
```

- Ejemplo: algoritmo para calcular números combinatorios.

$$\binom{m}{n} = \frac{m!}{n!\ (m-n)!}$$

```
FUNC factorial(↓ x:N):N
Variables
    i: N
Inicio
    RESULTADO ← 1
    PARA i ← 1 HASTA x HACER
        RESULTADO← RESULTADO * i
    FINPARA
Fin

FUNC Combinatorio(↓ m,n:N):N
Inicio
    RESULTADO ← (factorial(m) DIV
                 (factorial(n) * factorial(m-n)))
Fin
```


VI.1.7.- Criterios de modularización.

No existen algoritmos formales para determinar cómo descomponer un problema en módulos, es una **labor subjetiva.** De cualquier forma se siguen algunos criterios que pueden guiarnos al modularizar.

1.- Acoplamiento.

Una de las ventajas frente a la programación convencional (sin ninguna técnica rigurosa de diseño) es que con el diseño modular obtenemos software fácilmente modificable. La idea es que una modificación del sistema sólo requiera cambios en pocos módulos, con objeto de que podamos restringir nuestra atención a dicha porción del mismo.

Esta idea tendrá éxito si se puede suponer que cambios en determinados módulos no afectarán inadvertidamente a otros. Por tanto, un objetivo del diseño original deberá ser **maximizar la independencia entre módulos**.

Es obvio que en contra de dicho objetivo está el hecho antes mencionado de que debe haber alguna conexión entre módulos para formar un sistema coherente. Dicha conexión se conoce como acoplamiento.

Maximizar la independencia será **minimizar el acoplamiento**.

Hay diversas formas de acoplamiento entre los que destaca el **acoplamiento de datos**. Éste se refiere al intercambio de información entre diferentes módulos.

2.- Cohesión

Tan importante como minimizar el acoplamiento entre diferentes módulos es **maximizar las ligaduras internas** dentro de cada módulo individual.

Se usa el término cohesión para hacer referencia al grado de relación entre las diferentes partes internas a un módulo.

Si la cohesión es muy débil, la diversidad entre las distintas tareas realizadas dentro de un módulo es tal que posteriores modificaciones podrían resultar complicadas. Además dificulta la reutilización.

El diseñador de sotfware debe buscar un bajo acoplamiento entre los módulos y una alta cohesión dentro de cada uno.

VI.2.- ANIDAMIENTOS Y AMBITOS.

VI.2.1.- Subprogramas anidados. Reglas de ámbito.

- Desde el cuerpo del programa principal, o desde el cuerpo de cualquier procedimiento o función, se pueden realizar llamadas o invocaciones a otros procedimientos y funciones.

- Al igual que ocurre con las variables y las constantes, para poder utilizar (invocar a) un subprograma es necesario que se haya declarado con anterioridad.

- Hasta ahora no hemos reparado en ello, pero es muy importante el lugar en el que se deben declarar los subprogramas. Existen dos posibilidades no excluyentes:

a) Declarados antes que el algoritmo que realiza la llamada.

b) Declarados dentro de él, justo en su zona de declaraciones => anidamiento.

- Veamos un ejemplo:


```
Algoritmo Principal
Variables
    x,y,z:N
Subalgoritmos
    Proc Sub1(↓m: Z;↑ n:Z)
    Variables
        x,z:N
    Inicio
        x ← y + z
        .............
    Fin (* Sub1 *)
    Proc Sub2(↓z:N)
    Subalgoritmos
        Proc Sub3
        Variables
            x:Z
        Inicio
            Sub1(3,x)
            .............
        Fin (* Sub3 *)
    Variables
        i,j:N
    Inicio
        .............
    Fin (* Sub2 *)
Variables
    i,j:N
Inicio
    .............
Fin(* Principal *)
```

Si observamos la sentencia `x ← y + z` del ejemplo anterior, notaremos que existe una cierta ambigüedad al no poder determinar de antemano sobre qué variables actúa dicha acción.

Para saber dónde se puede utilizar un idenficador y para evitar situaciones de ambigüedad como la anterior, necesitamos ciertas reglas denominadas de ámbito.

Reglas de ámbito.

- Dentro de un algoritmo, un identificador es visible (accesible) en la zona comprendida entre el lugar en que se declara y el final del cuerpo del mismo. A esta zona se le denomina ámbito o zona de visibilidad.
- Regla de ocultación: si dentro del ámbito de un determinado identificador, éste aparece declarado en un nivel de anidamiento más interno, entonces dentro del ámbito de esta segunda declaración, la declaración más externa quedará ocultada (y por lo tanto no accesible).

VI.2.2.- Declaraciones locales y globales.

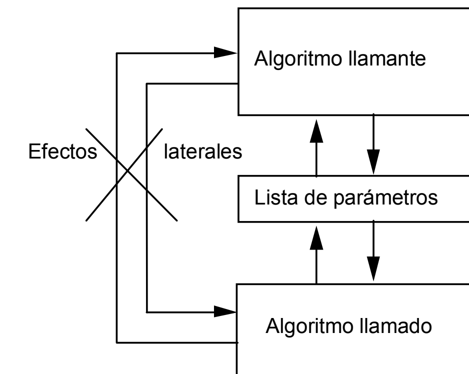
Variables locales: son aquellas que sólo son accesibles en el *cuerpo* del algoritmo en el que han sido declaradas.

Variables globales: son aquellas que pueden ser accesibles tanto en el *cuerpo* del algoritmo en el que han sido declaradas como en sus algoritmos anidados.

Nosotros no utilizaremos variables globales (únicamente los parámetros de los subprogramas podrán tener un ámbito "global", pero no se utilizaran nada más que en el cuerpo del algoritmo en cuya cabecera aparecen).

Efectos laterales.

- Es cualquier intercambio de información producida entre dos algoritmos realizado a través de variables globales.



- La posibilidad de usar variables globales puede hacer pensar que no es necesario el uso de parámetros, ya que el subalgoritmo llamado puede acceder globalmente a las variables del algoritmo que lo llama. No obstante este uso en programación **está totalmente desaconsejado** (aumenta el acoplamiento, reduce la posibilidad de reutilización).

NOTA FINAL DEL TEMA

En el **tema anterior**, nos aparecieron dos aspectos **muy importantes** que el alumno debe tener muy en cuenta:

- **Sólo** están **permitidas** las estructuras de control descritas en clase y que aparecen en los apuntes del tema: **sentencia de asignación**, sentencias de selección **SI** y **CASO**, y sentencias de iteración **MIENTRAS**, **REPETIR** y **PARA**.
- **Dentro** del bloque de sentencias de un bucle **PARA no se puede modificar el valor de la variable de control** del mismo.

En **este tema** hay que resaltar un aspecto **muy importante** que el alumno debe tener en cuenta:

- **No** se puede hacer **uso** de **variables globales**.

Estos tres **aspectos** son **fundamentales**, no sólo en la asignatura de este cuatrimestre (EP) sino también en las dos del segundo cuatrimestre (MP y LP).

Un alumno suspenderá una asignatura si incumple alguno de estos aspectos durante la realización de su examen.

Anexo. Teoría de la calculabilidad.

El problema de la detención expuesto en el tema 2, se podría plantear en nuestro pseudolenguaje como :

```
Func Detencion (↓ P,D: generico):B
Inicio
     /* acaba es una función que devuelve un valor booleano que
      * indica si el programa P se detiene al ejecutarse sobre
      * los datos D
     */
     SI acaba(P, D) ENTONCES
          RESULTADO ← TRUE
     SINO
          RESULTADO ← FALSE
     FINSI
Fin


FUNC Nueva_Detencion(↓ P:generico):B
Inicio
     RESULTADO ← Detencion(P,P)
Fin

FUNC Simpático(↓P:generico):B
Inicio
     SI Nueva_Detencion(P) ENTONCES
          MIENTRAS TRUE HACER
               // Este bucle nunca acaba
          FINMIENTRAS
     SINO
          RESULTADO ← TRUE
     FINSI
Fin
```

¿Simpatico(Simpatico) acaba o no?