UCA
Universidad de Cádiz

Introducción a la Programación
Grado en Ingeniería Informática

Teoría - Curso 2015-2016

Contenido 4 – Abstracción operacional

Tema 4.- Abstracción Operacional

4.1.- Descomposición de problemas. Modularidad

- El diseño de un algoritmo implica la descomposición del problema en módulos.
- La **modularidad** consiste en estructurar la solución del problema en bloques, cada uno de los cuales realiza una tarea independiente pero que juntos resuelven el problema original.
- Generalmente un programa posee un algoritmo principal que posee el control y lo transfiere a los subalgoritmos cuando se requiera que cada uno de ellos realice la tarea para la cuál fue diseñado.

4.1.- Descomposición de problemas. Modularidad

- Ventajas del diseño modular:
  - Facilita la resolución del problema.
  - Aumenta la claridad.
  - Facilita la identificación y corrección de errores.
  - Facilita la actualización y el mantenimiento.
  - Varias personas pueden trabajar en módulos independientes reduciendo el tiempo de diseño y codificación.

4.1.1- Concepto de abstracción operacional

- La abstracción operacional permite a los programadores, realizar sus propias abstracciones en forma de subalgoritmos: **procedimientos y funciones**.
- En todo procedimiento o función se pueden distinguir dos partes fundamentales:
  1. La especificación ¿***qué*** hace el algoritmo?
  2. La implementación. ¿***cómo*** lo hace?
- La abstracción operacional incluye dos aspectos:
  - Abstracción por parametrización
  - Abstracción por especificación

4.1.1- Concepto de abstracción operacional

- Cláusulas de una especificación no formal:
  - **Cabecera**: información sintáctica que indica que hay que hacer para proceder a la ejecución del subalgoritmo.
  - **Precondición**: Condiciones que deben cumplir los valores de entrada.
  - **Postcondición**: Efecto producido por el subalgoritmo. Condiciones que se cumplen al finalizar el algoritmo.

4.1.1- Concepto de abstracción operacional

- La especificación debe ser independiente de la implementación
- Una misma especificación puede tener implementaciones diferentes.

Ejemplo:

```
//Cabecera: entero maximo(E entero: a, E entero:b)
//Precondición: a y b están inicializados
//Postcondición: devuelve el máximo entre a y b
```

```
entero funcion maximo(E entero:a, E
entero :b)
var entero : max
inicio
   si a>b entonces
  max← a
   si_no
        max ← b
   devolver(max)
fin_funcion
```

```
entero funcion maximo(E entero: a, E
entero :b)
var entero: max
inicio
   max← a
   si  b>max entonces
      max←b
   fin_si
   devolver max
fin_funcion
```

4.2.1.- Funciones:

- Una función matemática es una operación que recibe uno o más valores llamados argumentos y devuelve un valor denominado resultado.
- Todas las funciones devuelven como resultado un único valor
- Tipos de funciones:
  - Internas: incorporadas en los lenguajes
  - Externas: definidas por el programador

4.2.1.- Funciones

Declaración de funciones en pseudocódigo:

<especificación de la función>
*<tipo del resultado>***funcion***<nombre función>*(*<lista de parámetros formales>*)
[declaraciones locales]
inicio
    <instrucciones> //cuerpo
    devolver *(<expresión resultado>)*
fin_funcion

<lista_de_parámetros >:
    Será la lista de parámetros formales de la siguiente forma:
    ({E|S|E/S} tipoA: parametro1, {E|S|E/S} tipoB: parametro2,...
    Siendo, E: Entrada, S: Salida, E/S: Entrada y Salida.
<nombre_funcion>: Identificador válido para la función
<instrucciones>:
    instrucciones que constituyen el cuerpo de la función, deberán contener una única expresión: **devolver** (*<expresión>*)
<tipo_de_resultado>:
    tipo de dato correspondiente al resultado de la función.

4.2.1.- Funciones

Ejemplo: $f(x) = x^2$

```
//Cabecera: real cuadrado(E real: x)
//Precondición: recibe x real
//Postcondición: devuelve el cuadrado de x (x2)
real función cuadrado(E real: x)
inicio
        devolver (x*x)
fin_funcion
```

- **Llamada a una función:** ***nombre_funcion (lista de parámetros actuales)***

```
Algoritmo elevar_cuadrado
Principal
  var
        real : numero, res
  inicio
        escribir (“introduzca un número”)
        leer (numero)
        res ← cuadrado(numero)
        escribir(“el cuadrado de”, numero, ”es”, res)
  fin_principal
fin_algoritmo
```

4.2.2.- Procedimientos

Los procedimientos no devuelven expresamente ningún valor

<especificación del procedimiento>
procedimiento*<nombre_procedimento >*(*<lista de parámetros formales>*)
[declaraciones locales]
inicio
    <instrucciones> //cuerpo
fin_procedimiento

<lista_de_parámetros >:
    Será la lista de parámetros formales de la siguiente forma:
    ({E|S|E/S} tipoA: parametro1,{E|S|E/S} tipoB: parametro2)
    Siendo, E: Entrada, S: Salida, E/S: Entrada y Salida.
<nombre_procedimento>: Identificador válido para el procedimiento
<instrucciones>:
    instrucciones que constituyen el cuerpo del procedimiento (observamos que no existe una sentencia *devolver* como sucedía en las funciones)

4.2.2.- Procedimientos

Ejemplo: Procedimiento que realiza la división de dos números enteros, calculando el cociente y el resto e imprimiendo los valores por pantalla.

```
//Cabecera: division(E entero: dividendo,E entero: divisor)
//Precondición: dividendo > = divisor y ambos mayores que 0
//Postcondición: escribe en pantalla el cociente y el resto

procedimiento división (E entero: dividendo, E entero : divisor)
var
        entero: cociente, resto
inicio
        cociente ← dividendo / divisor
        resto ← dividendo mod divisor
        escribir (cociente, resto)
fin_procedimiento
```

4.2.2.- Procedimientos

Llamada a un procedimiento:

nombre_procedimiento (lista de parámetros actuales)

Ejemplo:

```
Algoritmo divide
 Principal
  var
        entero: divdo, divi
  inicio
     repetir
       escribir("introduce los valores del dividendo y divisor")
       leer(divdo, divi)
     hasta_que (divdo >= divi y divi>0 y divdo>0)
       division(divdo, divi)
  fin_principal
fin_algoritmo
```

4.3.- Ámbito y persistencia de las variables

- **Ámbito:** fragmento de código en el cual una variable puede ser referenciada.

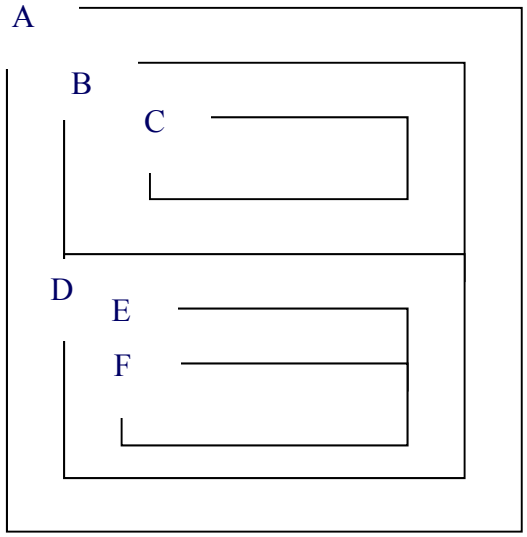
  - **Ámbito local:** son las declaradas dentro del Principal o dentro de los subalgoritmos (procedimientos y funciones), bien en el lugar correspondiente a la definición de variables (**var**) o bien como parámetro formal, y pueden ser referenciadas sólo dentro de ese subalgoritmo. Si una variable local se define con el mismo nombre que una variable global, entenderemos que se trata de una variable distinta cuyo ámbito será el del subalgoritmo en el que ha sido definida. El uso de las variables locales preserva la independencia de los subalgoritmos y hace un uso más eficiente de la memoria.

  - **Ámbito global:** declaradas fuera del principal y de cualquier subalgoritmo. Una variable tiene ámbito global cuando puede ser referenciada desde cualquier punto del cuerpo del Principal, así como desde cualquier punto de cualquiera de los subalgoritmos.

4.3.- Ámbito y persistencia de las variables

- **Ejemplo:**

Supongamos que el siguiente diagrama de cajas representa la estructura en niveles de un determinado algoritmo.



Variables definidas en	Accesibles desde
A	Todos
B	B,C
C	C
D	D,E,F
E	E
F	F

4.3.- Ámbito y persistencia de las variables

Supondremos una estructura modular con un algoritmo principal que controlará el flujo de la ejecución y diferentes subalgoritmos definidos todos a un nivel inferior que serán llamados cuando se requiera su ejecución.

```
Algoritmo ejemplo
var
   entero: a, b
Principal
 inicio
   a←2
   b←3
   proc1(a)
 Fin_principal
fin_algoritmo
```

```
procedimiento proc1(E entero: x)
var
            entero: b
inicio
            b←x+a
            escribir(b)
fin_procedimiento
```

En este ejemplo *a* y *b* definidas fuera de Principal son variables de ámbito **global**, a su vez *a* es parámetro real o actual en la llamada al procedimiento *proc1* concretamente con el valor real 2. La variable *x* es un parámetro formal y a su vez variable local de *proc1*, *b* es otra variable local a *proc1*, distinta de la *b* global. En la instrucción *b*← *x*+*a* estamos asignando a la variable local *b* el resultado de sumar el valor de la variable local *x* con el valor de la variable global *a*. **Debemos tener en cuenta que tanto Principal como el procedimiento *proc1* podrían modificar el valor de la variable global *a*.**

4.3.- Ámbito y persistencia de las variables

- **Persistencia:** duración de la variable en memoria
  - **Persistencia indefinida:** permanece en memoria durante toda la ejecución de un programa
  - **Persistencia dinámica:** cuando la variable existe sólo durante la ejecución de un fragmento de código. Por ejemplo, persiste en memoria durante la ejecución de un procedimiento o función y posteriormente desaparece.
- Generalmente suele asociarse una persistencia indefinida al ámbito global y una persistencia dinámica al ámbito local.
- En algunos lenguajes es posible modificar el ámbito y persistencia de las variables a través de determinadas instrucciones siendo posible que existan las cuatro combinaciones: ámbito global y persistencia indefinida, ámbito global y persistencia dinámica, ámbito local y persistencia indefinida y ámbito local y persistencia dinámica.

4.4.- Correspondencia entre argumento y parámetro formal

- El paso de parámetros es el método mediante el cual los distintos subalgoritmos de un algoritmo pueden comunicarse, compartir valores y variables sin perder su independencia.
- Existe una correspondencia automática entre los parámetros formales y los parámetros actuales que puede ser de dos tipos:
  - **Posicional**: la correspondencia se establece de izquierda a derecha entre parámetros formales y actuales
  - **Por nombre**: se indica de forma explícita en la llamada al subalgoritmo cómo ha de realizarse esta correspondencia.

    Ejemplo: proc(a=>x, b=>3)
- Tipos de paso de parámetros:
  - Por valor
  - Por referencia

4.4.1.- Paso por valor y por referencia

- Paso por valor:
  - se produce una copia del valor de los parámetros actuales en los parámetros formales
  - Los parámetros actuales pueden ser constantes, variables o expresiones. Los parámetros formales serán variables locales al procedimiento que serán inicializadas con el valor que el parámetro actual les pasa en la llamada.
  - Los parámetros pasados por valor serán siempre parámetros de entrada y llevarán en su definición la letra E. A través de estos parámetros no se devuelve ningún tipo de información al algoritmo desde el cual se realizó la llamada.
  - Cuando el subalgoritmo devuelve el control al algoritmo que realiza la llamada los parámetros actuales conservarán exactamente el valor que tenían antes de realizar la llamada.

4.4.1.- Paso por valor y por referencia

- Paso por referencia:
  - El parámetro formal recibe una referencia (dirección de memoria) del parámetro actual
  - Los parámetros actuales deben ser siempre variables
  - Parámetros formales y actuales comparten la misma dirección de memoria por tanto todo cambio en el parámetro formal tendrá efecto en el parámetro actual.
  - Los parámetros pasados por valor serán siempre parámetros de salida ó entrada/salida y llevarán en la definición de los parámetros formales las letras S o E/S.
  - Este paso de parámetros se puede usar para que un subalgoritmo devuelva valores al algoritmo que realiza la llamada.

4.4.1.- Paso por valor y por referencia

```
Algoritmo ejemplo
 Principal
 var
     entero: x, y, z
 inicio
    x← 2
    y← 3
    z← fun (x,y)
    escribir (x,y,z)
 fin_principal
fin_algoritmo
```

```
entero funcion fun (<modo1> entero: a, <modo2>
entero: b)
var
     entero: c
inicio
     c← 5
     a← c + 1
     b ← a + c
     devolver (b)
fin_funcion
```

4.4.1.- Paso por valor y por referencia

```
Algoritmo ejemplo
 Principal
 var
    entero: x, y, z
 inicio
   x← 2
   y← 3
   z← fun (x,y)
   escribir (x,y,z)
 fin_principal
fin_algoritmo
```

```
entero funcion fun (<modo1> entero: a, <modo2>
entero: b)
var
    entero: c
inicio
    c← 5
    a← c + 1
    b ← a + c
    devolver (b)
fin_funcion
```

modo1 (E)	modo2 (E)	2 3 11
modo1 (E)	modo2 (S)	2 11 11
modo1 (S)	modo2 (E)	6 3 11
modo1 (S)	modo2 (S)	6 11 11

4.4.1.- Paso por valor y por referencia

```
Algoritmo divide
 Principal
 var
            entero: dividendo, divisor, cociente, resto
 inicio
      escribir("introduce los valores del dividendo y divisor")
      repetir
        leer (dividendo, divisor)
      hasta que (dividendo >= divisor)
      division (dividendo, divisor, cociente, resto)
      escribir (cociente, resto)
 fin_principal
fin_algoritmo
```

4.4.1.- Paso por valor y por referencia

```
//Cabecera: division(E entero: divi,E entero: d, S entero: c, S entero: r)
//Precondición: divi ≥ divi y ambos mayores que 0
//Postcondición: devuelve a través de los parámetros de salida c y r, el cociente y el
// resto de dividir divi entre d
procedimiento división (E entero: divi, E entero: d, S entero: c, S entero: r)
Inicio
    c ←divi div d
    r ← divi mod d
fin_procedimiento
```

4.4.2.- Efectos laterales

•Los *efectos laterales* son los cambios o modificaciones que se producen en las variables de un módulo como consecuencia de una instrucción de otro módulo.
•La comunicación de un módulo con el resto del algoritmo se debe realizar siempre a través del paso de parámetros. Cualquier otra comunicación puede provocar efectos laterales. Una función o procedimiento puede cambiar el contenido de una variable global y provocará un efecto lateral.
•Si se necesita una variable temporal en un procedimiento o función se debe utilizar una variable local, nunca global.
•Si se desea que el programa modifique el valor de una variable global es mejor utilizar un parámetro formal pasado por referencia (E/S o S) en la declaración del procedimiento y utilizar la variable global como parámetro actual en la llamada al procedimiento.
•Los efectos laterales están considerados como una mala técnica de programación, pues disminuyen la legibilidad del código, resultando más complicada la depuración y detección de errores, así como el mantenimiento de los programas

4.5.- Funciones y procedimientos como parámetros

Algunos lenguajes de programación admiten la posibilidad de pasar como parámetros de funciones o procedimientos, otros nombres de funciones o procedimientos, aumentando así el nivel de abstracción.

Ejemplo: Algoritmo que lee 50 veces dos números y realiza alternativamente una suma y un producto.

```
Algoritmo principal
tipo     entero funcion (E entero: x, E entero: y): func
 Principal
 var      entero: a, b, cont
 inicio
    cont ← 0
    mientras cont<> 50 hacer
              leer(a, b)
              si (cont mod 2=0) entonces
                        operación (suma,a,b)
              si_no
                        operación (prod, a, b)
              fin_si
              cont ← cont + 1
     fin_mientras
 fin_principal
fin_algoritmo
```

4.5.- Funciones y procedimientos como parámetros

```
//Cabecera: operacion(E func: f, E entero: a, E entero: b)
//Precondición: f es el nombre de una función y a y b sus argumentos
//Postcondición: escribe el resultado de llamar a la función f con a y b
procedimiento operacion (E func: f, E entero: a, E entero: b)
inicio
     escribe (“El resultado es”, f(a,b))
fin_procedimiento

//Cabecera: entero suma(E entero: x,E entero: y)
//Precondición: x e y argumentos inicializados
//Postcondición: devuelve la suma de a y b (a+b)
entero funcion suma (E entero: x, E entero: y)
var
    entero: suma
inicio
    suma← x+ y
    devolver(suma)
fin_funcion
```

4.5.- Funciones y procedimientos como parámetros

```
//Cabecera: entero producto(E entero: x, E entero: y)
//Precondición: x e y argumentos inicializados
//Postcondición: devuelve a*b
entero funcion producto (E entero: x, E entero: y)
var
    entero: prod
inicio
    prod← x* y
    devolver(prod)
fin_funcion
```