Departamento de Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

Apuntes para la asignatura

Informática

Facultad de Ciencias (Matemáticas)
http://www.lcc.uma.es/personal/pepeg/mates

Tema 6. Procedimientos y funciones

Bibliografía

- *Programación 1*. José A. Cerrada y Manuel Collado. Universidad Nacional de Educación a Distancia.
- *Programming with TopSpeed Modula-2*. Barry Cornelius. Addison-Wesley.

Introducción

Cualquier programa de ordenador que realice una labor útil suele tener una extensión considerable. El diseño de un algoritmo complejo de una sola vez es una labor complicada. Vimos como la metodología de diseño descendente permite distinguir diferentes partes en la descripción de un algoritmo, de modo que la agrupación de éstas constituya el algoritmo completo. Estudiamos en este tema los mecanismos del lenguaje de programación *Modula-2* que permiten dividir un programa complejo en partes más manejables que puedan ser posteriormente combinadas.

6.1 Concepto de subprograma

Un *subprograma* es una parte de un programa. Desde el punto de vista de la programación, un subprograma es un trozo de programa que se escribe por separado y que puede ser utilizado invocándolo mediante su nombre.

Esto hace que distingamos dentro de un programa el *programa principal* (el algoritmo correspondiente al módulo raíz) y *subprogramas* (los demás módulos). Cada subprograma puede a su vez estar dividido en otros subprogramas.

Supongamos que queremos escribir un programa que calcule el perímetro de un triángulo con vértices A, B y C:

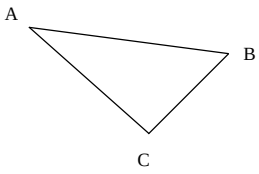

Figura 1. Triángulo de vértices A, B y C

El diagrama de bloques para el algoritmo puede ser:

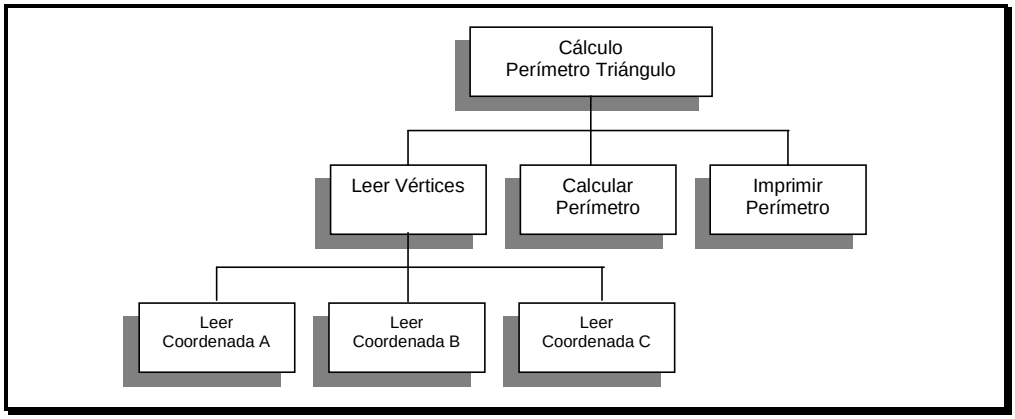

Figura 2. Diagrama de bloques para cálculo del perímetro del triángulo

El problema de calcular el perímetro de un triángulo puede dividirse en tres subproblemas: leer las coordenadas, calcular el perímetro e imprimir el perímetro. A su vez, el primero de estos subproblemas puede dividirse en tres subproblemas: leer la coordenada A, leer la coordenada B y leer la coordenada C.

El aspecto del programa resultante es el siguiente:

```
MODULE Perimetro;
  ...
  PROCEDURE LeerVertices;
    ...
    PROCEDURE LeerUnaCoordenada...;
    BEGIN
    ...
    END LeerUnaCoordenada;
  ...
  BEGIN
    (* Leer las tres coordenadas *)
    LeerUnaCoordenada...;
    LeerUnaCoordenada...;
    LeerUnaCoordenada...;
  END LeerVertices;

  PROCEDURE CalcularPerimetro...;
  BEGIN
    ...
  END CalcularPerimetro;


  PROCEDURE ImprimirPerimetro...;
  BEGIN
    ...
  END ImprimirPerimetro;
BEGIN
  LeerVertices...
  CalcularPerimetro...
  ImprimirPerimetro...
END Perimetro.
```

Vemos como cada subprograma aparece descrito en la parte declarativa del programa (*declaración de subprograma*) y luego se usa escribiendo su nombre en la parte ejecutiva (*llamada a subprograma*).

Es deseable que en el cuerpo del programa principal se evite la aparición excesiva de estructuras de control (selección y repetición) utilizando subprogramas. De esta forma, el cuerpo del programa principal estará constituido fundamentalmente por llamadas a subprogramas.

Se suelen emplear subprogramas en los siguientes casos:

- *En programas complejos:* si un programa complejo se escribe sin subprogramas resulta difícil de entender. Dividiéndolo en subprogramas podemos centrarnos en cada momento en un problema más pequeño que el problema original.
- *Cuando se repite dentro de un algoritmo algún tipo de tratamiento:* en este caso podemos escribir un subprograma que describe cómo se hace el tratamiento una sola vez y realizar una llamada a este subprograma cada vez que queramos usar el tratamiento.

6.2 Definición y llamadas a subprogramas

Cuando para describir un algoritmo A se necesita emplear otro algoritmo B, de dice que el algoritmo A *llama* al algoritmo B. En el ejemplo anterior el algoritmo principal llama a los algoritmos `LeerVertices`, `CalcularPerimetro` e `ImprimirPerimetro`. El algoritmo `LeerVertices` llama al algoritmo `LeerUnaCoordenada`.

A veces, un subprograma depende del valor de una o más variables. Por ejemplo, nos puede interesar escribir un subprograma `EscribeSumatorio` que dado un número N calcule el valor de la suma 1 + 2 + ... + (N-1) + N y escriba el resultado por pantalla. Para distintos valores de N el subprograma obtiene distintos resultados. En este caso decimos que el subprograma `EscribeSumatorio` está *parametrizado* por N, o que N es un *parámetro* del subprograma `EscribeSumatorio`. Un subprograma puede depender de cero, uno o más parámetros.

Lo primero que debemos hacer para utilizar un subprograma es *declararlo*. La primera parte de la declaración de un subprograma es su *cabecera*. En la cabecera de un subprograma se indica el nombre de éste y los parámetros o argumentos que toma (especificando para cada uno su tipo). La sintaxis que sigue *Modula-2* para esto es:

PROCEDURE Nombre (argumento1 : Tipo1; argumento2 : Tipo2; ...);

En la cabecera del subprograma aparece la palabra reservada seguida del nombre del subprograma. Este nombre debe ser un identificador válido. A continuación, y entre paréntesis, se escribe la lista de argumentos separados por puntos y comas. El nombre de cada argumento debe ser también un identificador. Detrás de cada argumento se especificará su tipo precedido de un signo dos puntos. Si hay varios argumentos seguidos con el mismo tipo, podemos abreviar y escribirlos separados por comas, indicando una sola vez el tipo de todos ellos.

La cabecera del subprograma `EscribeSumatorio` comentado previamente es:

```
PROCEDURE EscribeSumatorio ( N : CARDINAL );
```

Después de la cabecera del subprograma viene su cuerpo. La declaración finaliza con la palabra END seguida del nombre del subprograma. En el cuerpo, aparecen las declaraciones de los elementos necesarios para el subprograma y tras la palabra reservada **`BEGIN`** el código necesario para resolver el subproblema. La declaración completa para el ejemplo es:

```
PROCEDURE EscribeSumatorio ( N : CARDINAL );
VAR Suma, i : CARDINAL;
BEGIN
  Suma := 0;
  FOR i := 1 TO N DO
    Suma := Suma + i
  END;
  WrCard (Suma, 0)
END EscribeSumatorio;
```

Podemos observar que las variables `Suma` e `i` se declaran dentro del subprograma y no en el programa principal. Esto se hace así porque sólo son necesarias para el cómputo que hace el subprograma. A estas variables se les llama *variables locales* del subprograma. Las variables locales de un subprograma se crean automáticamente cada vez que se llama al subprograma y se destruyen cada vez que se sale del subprograma. Por esto NO CONSERVAN EL VALOR que tenían de una llamada a otra. Tampoco toman valores iniciales por defecto; es el programador el encargado de proporcionárselos.

Una vez que tenemos un subprograma declarado, podemos utilizarlo dentro del cuerpo de otro subprograma (en condiciones que especificaremos un poco más adelante) o del programa principal. Para utilizar el subprograma hay que llamarlo. La *llamada* a un subprograma se realiza utilizando su nombre seguido de los valores (entre paréntesis y separados por comas) con los que queremos que trabaje el subprograma. Una llamada a un subprograma con *N* parámetros se escribe:

NombreSubprograma (parámetro1, parámetro2, ... ,parámetroN)

Si queremos llamar al subprograma `EscribeSumatorio` para que calcule la suma de los primeros 50 naturales escribiremos:

```
EscribeSumatorio (50)
```

6.3 Parámetros formales y reales

Supongamos que queremos escribir un programa que escriba por pantalla todos los números primos menor a uno dado (Max). Para ello, escribiremos primero un subprograma que dado un número Num escriba por pantalla si es primo y en otro caso no escriba nada:

```
PROCEDURE EscribirSiPrimo (Num : CARDINAL);
VAR
  Divisor : CARDINAL;
  EsPrimo : BOOLEAN;
BEGIN
  EsPrimo := TRUE;
  Divisor := 2;
  WHILE EsPrimo AND (Divisor <= Num DIV 2) DO
    EsPrimo := NumMOD Divisor <> 0;
    Divisor := Divisor +1
  END;
  IF EsPrimo THEN
    WrCard (Num, 0); WrLn
  END
END EscribirSiPrimo;
```

El programa principal leerá el número Max del teclado y llamará a este subprograma para todos los valores en el intervalo [1,Max]:

```
MODULE Primos;
FROM IO IMPORT WrCard, RdCard, RdLn, WrStr, WrLn;
VAR
  Max, i : CARDINAL;

  PROCEDURE EscribirSiPrimo (Num : CARDINAL);
  VAR
    Divisor : CARDINAL;
    EsPrimo : BOOLEAN;
  BEGIN
    EsPrimo := TRUE;
    Divisor := 2;
    WHILE EsPrimo AND (Divisor <= Num DIV 2) DO
      EsPrimo := NumMOD Divisor <> 0;
      Divisor := Divisor + 1
    END;
    IF EsPrimo THEN
      WrCard (Num, 0); WrLn
    END
  END EscribirSiPrimo;

BEGIN
  WrStr (“Dame el máximo: “);
  Max := RdCard(); RdLn;
  WrStr (“Los primos menores son:”); WrLn;
  FOR i := 1 TO Max DO
    EscribirSiPrimo(i)
  END
END Primos.
```

Se llaman *parámetros formales* a la lista de parámetros que aparece en la declaración de un subprograma parametrizado. En el ejemplo, Num es un parámetro formal del subprograma EscribirSiPrimo.

Se llaman *parámetros reales* a la lista de parámetros que aparece en la llamada al subprograma. En el ejemplo, `i` es un parámetro real en la llamada a `EscribirSiPrimo`. Los parámetros reales pueden ser, además de variables, expresiones y por lo tanto valores (a veces, sólo pueden ser variables). En una llamada como `EscribirSiPrimo(143)`, el parámetro real es `143`.

Los parámetros formales de un subprograma son opcionales, esto es, se pueden escribir subprogramas no parametrizados, pero si aparecen deben seguir una serie de normas:

- El número de parámetros reales en una llamada a un subprograma parametrizado debe ser igual al número de parámetros formales en la definición de dicho subprograma.
- La correspondencia entre parámetros formales y reales es por posición. El *i*-ésimo parámetro real se corresponde con el *i*-ésimo parámetro formal.
- El tipo del *i*-ésimo parámetro real debe ser compatible (o igual si es por referencia) con el declarado para el *i*-ésimo formal.
- Los nombres de un parámetro formal y su correspondiente real pueden ser distintos.

Así, con el siguiente programa:

```
MODULE Ejemplo;
FROM IO IMPORT WrReal;
VAR  x, y : REAL;
  PROCEDURE EscribirDivCuadrados (x, y : REAL);
  CONST
    PRECISION = 5;
    ANCHO     = 0;
  VAR Cociente : REAL;
  BEGIN
    Cociente := (x * x) / (y * y);
    WrReal (Cociente, PRECISION, ANCHO)
  END EscribirDivCuadrados;
BEGIN
  x := 1.0;
  y := 2.0;
  EscribirDivCuadrados (y, x)
END Ejemplo.
```

El resultado que obtenemos es 4.0/1.0 = 4.0, ya que la correspondencia entre parámetros formales y reales está dada por el orden y no por los nombres.

El efecto producido por la llamada a un subprograma puede resumirse como:

1) Se evalúan las expresiones que aparezcan en los argumentos (parámetros reales).
2) Se crean las variables correspondientes a los parámetros formales.
3) Se asignan los valores calculados en 1) a los correspondientes parámetros formales.
4) Se crean las variables locales al subprograma.
5) Se ejecuta el código correspondiente al subprograma.
6) Se destruyen las variables locales y las correspondientes a los parámetros formales.
7) Se continúa el programa por la instrucción siguiente a la llamada al subprograma.

6.4 Paso de parámetros por valor y por referencia

Consideremos el siguiente programa, en el que declaramos un subprograma `EscribirModulo` que escribe por pantalla el módulo de un complejo con parte real `Real` y parte imaginaria `Imag`:

```
MODULE Ejemplo;
FROM IO        IMPORT WrLngReal, WrStr, WrLn;
FROM MATHLIB   IMPORT Sqrt;
VAR x, y : LONGREAL;

  PROCEDURE EscribirModulo (Real, Imag : LONGREAL);
  VAR Modulo : LONGREAL;
  BEGIN
    WrStr ("El módulo del complejo ");
    WrLngReal (Real, 5, 0);
    WrStr (" ");
    WrLngReal (Imag, 5, 0);
    WrStr (" i es: ");
    Real := Real * Real;
    Imag := Imag * Imag;
    Modulo := Sqrt (Real + Imag);
    WrLngReal (Modulo, 5, 0);
    WrLn
  END EscribirModulo;

BEGIN
  x := 5.0;
  y := 6.0;
  EscribirModulo (x, y);
  WrLngReal (x, 5, 0); WrLn;
  WrLngReal (y, 5, 0); WrLn
END Ejemplo.
```

La salida producida por pantalla al ejecutar este programa es:

```
El módulo del complejo  5.0000E+0  6.0000E+0 i es: 7.8102E+0
5.0000E+0
6.0000E+0
```

Al realizar la llamada al subprograma los valores de `x` (5.0) e `y` (6.0) se copian en los parámetros formales `Real` y `Imag`. A continuación, se pasa a ejecutar el subprograma: se escribe la primera línea por pantalla, `Real` pasa a valer 25.0, `Imag` pasa a valer 36.0, `Módulo` pasa a valer `Sqrt (61.0)`, se escribe este valor por pantalla (“7.8102E+0”) y se salta a la siguiente línea de pantalla. Después de esto, se vuelve al programa principal y continuamos por donde se dejó: se escriben por pantalla los valores de `x` e `y` en distintas líneas (5.0000E+0 y 6.0000E+0). Lo importante de este ejemplo es que al cambiar el valor de los parámetros formales (`Real` e `Imag`) dentro del subprograma, el valor de los parámetros reales (`x` e `y`) se ha conservado.

A este modo de pasar parámetros se le llama *paso de parámetros por valor*. Los parámetros reales y formales representan variables distintas, aunque al llamar al subprograma se copian los valores de las reales sobre las correspondientes formales. Los argumentos reales en la llamada al subprograma pueden darse en forma de expresiones, cuyos tipos deben ser COMPATIBLES EN ASIGNACIÓN con los tipos de los argumentos formales.

Existe otra forma de pasar parámetros conocida como *paso de parámetros por referencia*. En este caso, a la hora de realizar la llamada al subprograma, no se copian los valores de los parámetros reales en los formales, sino que se identifican parámetros reales y formales, de modo que cada vez que cambiemos el valor de un parámetro formal en el subprograma estaremos cambiando el valor del parámetro real original. Para indicar que queremos que un parámetro pase por referencia utilizaremos la palabra reservada VAR delante de dicho argumento en la cabecera del subprograma. En nuestro ejemplo:

```
PROCEDURE EscribirModulo (VAR Real, Imag : LONGREAL);
```

En este caso estamos pasando ambos parámetros por referencia y el resultado que obtenemos por pantalla es:

```
El módulo del complejo  5.0000E+0  6.0000E+0 i es: 7.8102E+0
2.5000E+0
3.6000E+0
```

Cada parámetro puede independientemente estar declarado por valor o referencia:

```
PROCEDURE EscribirModulo (VAR Real : LONGREAL; Imag : LONGREAL);
```

Por referencia sólo el primero

```
PROCEDURE EscribirModulo (Real : LONGREAL; VAR Imag : LONGREAL);
```

Por referencia sólo el segundo

Si un argumento está declarado por referencia no podremos pasar como parámetros reales expresiones, sino SOLO VARIABLES. Para argumentos declarados por referencia, el tipo de la variable pasada como parámetro debe ser EXACTAMENTE EL MISMO que el del parámetro formal correspondiente.

Tanto el paso por valor como el paso por referencia tienen sus ventajas e inconvenientes:

- *Paso por valor:*
  - **Ventajas**: Aísla el efecto del subprograma a su propio ámbito. Esto hace más fácil de seguir los programas. Si se usa paso por valor y se sigue el programa principal sin entrar en los subprogramas, se sabe que las variables sólo cambian si cambian en el principal.
  - **Desventajas**: Utiliza más memoria. El parámetro real y el formal ocupan cada uno su propio trozo de memoria en el ordenador.
- *Paso por referencia:*
  - **Ventajas**: Utiliza menos memoria. El parámetro formal y real son los mismos y ocupan el mismo espacio de memoria.
  - **Desventajas**: Sólo permite variables como parámetros reales y el seguimiento del programa resulta más complejo.

6.5 Procedimientos y Funciones

6.5.1 Diferencias entre procedimientos y funciones

Todos los subprogramas vistos hasta ahora son procedimientos. Un *procedimiento* es un subprograma que toma opcionalmente una serie de parámetros y hace algo con ellos. Ejemplos de procedimientos son las instrucciones de salida de datos del módulo de biblioteca IO.

Sin embargo, cuando se diseña un algoritmo aparecen con frecuencia operaciones que calculan un valor simple a partir de ciertos parámetros o argumentos:

- Calcular el volumen de un cubo de lado *l*: se calcula el valor l^3 a partir del parámetro *l*.
- Calcular la potencia x^n : se calcula el valor *x* elevado a *n* a partir de dos argumentos *x* y *n*.
- Calcular el máximo de dos números *a* y *b*: se devuelve el mayor de *a* y *b*.

Todas estas operaciones se pueden considerar subprogramas del tipo función. Una *función* es un subprograma que calcula como resultado un valor simple a partir de otros valores dados como argumentos. Una función se asemeja bastante al concepto de función matemática con argumentos:

$$\begin{aligned} VolumenCubo(l) &= l^3 \\ Potencia(x,n) &= x^n \\ Maximo(a,b) &= \begin{cases} a, si\ a > b \\ b, en\ otro\ caso \end{cases} \end{aligned}$$

La definición de una función es similar a la de un procedimiento. Sin embargo, la cabecera de una función incluye al final el tipo de valor que devuelve precedido de dos puntos. Para los ejemplos anteriores las cabeceras en *Modula-2* son:

```
PROCEDURE VolumenCubo (l : REAL) : REAL;
PROCEDURE Potencia (x : REAL; n : CARDINAL) : REAL;
PROCEDURE Máximo (a, b : REAL) : REAL;
```

En el cuerpo de la función se debe indicar el valor devuelto. Esto se hace mediante la sentencia RETURN. La declaración completa de las funciones anteriores es:

```
PROCEDURE VolumenCubo (l : REAL) : REAL;
BEGIN
  RETURN (l*l*l)
END VolumenCubo;

PROCEDURE Potencia (x : REAL; n : CARDINAL) : REAL;
VAR  i   : CARDINAL;
     Pot : REAL;
BEGIN
  Pot := 1.0;
  FOR i := 1 TO n DO
    Pot := Pot * x
  END;
  RETURN Pot
END Potencia;

PROCEDURE Maximo (a, b : REAL) : REAL;
VAR Max : REAL;
BEGIN
  IF (a>b) THEN
    Max := a
  ELSE
    Max := b
  END;
  RETURN Max
END Maximo;
```

El tipo de la expresión que sigue a la sentencia RETURN debe ser COMPATIBLE EN ASIGNACIÓN con el tipo declarado en la cabecera. Al ejecutarse la sentencia RETURN la función termina y se vuelve al programa llamante (aunque hubiese más instrucciones en la función). Dentro del cuerpo de una función puede aparecer más de una sentencia RETURN. Sin embargo, esto es una mala práctica de programación y debe ser evitado. Como buena norma de programación, EXIGIREMOS que dentro del cuerpo de una función sólo haya una sentencia RETURN y que ésta sea la última sentencia en el cuerpo de la función (salvo en el caso de funciones recursivas de las que hablaremos en un tema posterior).

La llamada a un procedimiento constituye una sentencia del programa por sí sola. Sin embargo, la llamada a una función representa un valor y no es una sentencia por sí sola. Para construir una sentencia con una llamada a función debemos hacer algo con el valor que devuelve

(asignarlo a una variable compatible, escribirlo por pantalla, hacer que forme parte de una expresión, etc. ...). Una llamada a función sólo puede aparecer en donde pueda aparecer una expresión del tipo de la función. Así, son llamadas válidas a funciones:

```
x := VolumenCubo(3.0);                              (* Si x es REAL *)
WrReal (Maximo(1.0, 2.0), 5, 0);
x := Maximo (Potencia(10.0, 5), Volumen(8.0)); (* Si x es REAL *)
```

Pero no lo es:

```
x := x + 1.0;
VolumenCubo(3.0);
...
```

Como ejemplo, las operaciones de entrada del módulo de biblioteca IO son funciones.

6.5.2 Funciones predefinidas

Modula-2 dispone de una serie de funciones predefinidas, que pueden usarse el cualquier programa sin ser importadas. La lista de estas funciones es:

```
ABS(X)      CAP(C)    CHR(X)    FLOAT(X)    HIGH(A)
MAX(T)      MIN(T)    ODD(X)    ORD(C)      SIZE(T)
TRUNC(R)    VAL(T,X)
```

En los argumentos `C` representa un valor de tipo carácter, `X` de tipo número, `T` el nombre de un tipo y `A` un array (ver tema sobre tipos de datos estructurados). Hemos descrito todas en los temas anteriores excepto **`HIGH`**`(A)` (que veremos en el tema sobre tipos de datos estructurados) y **`VAL`**`(T, X)` que devuelve el valor de `X` convertido al tipo `T`. Algunas de estas funciones utilizan tipos como argumentos, cosa que el programador no puede hacer con los subprogramas definidos por él.

6.5.3 Procedimientos predefinidos

También existen procedimientos predefinidos en *Modula-2*. Estos son:

`DEC``(X)`: Decrementa en 1 el valor de la variable `X`. `X` debe ser de tipo ordinal.

`DEC``(X,N)`: Decrementa en `N` el valor de la variable `X` que debe ser de tipo ordinal. `N` puede ser CARDINAL o INTEGER pero siempre positiva

`EXCL``(S,X)`: Excluye el elemento `X` del conjunto `S` (ver tema sobre tipos de datos estructurados).

`HALT`: Hace que finalice la ejecución del programa. Esta instrucción sólo debe utilizarse en caso de que el programa alcance un error y éste impida que sea posible continuar con la ejecución del programa. La instrucción **`HALT`** debe precederse de una instrucción de salida que indique el error cometido. Nunca debe utilizarse esta sentencia para otra cosa que no sea un error.

`INC``(X)`: Incrementa en 1 el valor de la variable `X` que debe ser de tipo ordinal.

`INC``(X,N)`: Incrementa en `N` el valor de la variable `X` que debe ser de tipo ordinal.

`INCL``(S,X)`: Incluye el elemento `X` en el conjunto `S` (ver tema sobre tipos de datos estructurados)

6.6 Uso de parámetros

Los parámetros de un subprograma suelen utilizarse fundamentalmente de tres modos:

- Como parámetros de entrada
- Como parámetros de salida
- Como parámetros de entrada y salida

6.6.1 Parámetros de entrada

Un *parámetro* es *de entrada* cuando se usa para pasar un valor desde el programa llamante al subprograma llamado. Este tipo de parámetros debe declararse por valor.

❏ Ejemplo: El siguiente subprograma escribe por pantalla la tabla de multiplicar correspondiente al argumento que toma:

```
PROCEDURE EscribirTablaDe (Num : CARDINAL);
VAR i : CARDINAL;
BEGIN
  FOR i := 1 TO 10 DO
    WrCard(Num, 0);WrStr(" * ");      WrCard(i,0);
    WrStr(" = ");   WrCard(Num*i, 0);  WrLn
  END
END EscribirTablaDe;
```

El modo de usar el suprograma es el siguiente:

```
...
WrStr("La tabla del 7 es"); WrLn;
EscribirTablaDe (7);
...
```

❏

6.6.2 Parámetros de salida

Un *parámetro* es *de salida* cuando se usa para pasar un valor desde el subprograma llamado al programa llamante. Este tipo de parámetros debe declararse por referencia.

Para comunicar un valor desde un subprograma al programa llamante debe utilizarse una función. Sin embargo, una función sólo puede devolver un resultado. Si el subprograma ha de devolver más de un valor, es necesario utilizar parámetros de salida.

❏ Ejemplo: El siguiente subprograma convierte un punto expresado en coordenadas polares a los valores correspondientes en coordenada cartesianas. Los dos primeros parámetros son de entrada y los dos últimos de salida:

```
PROCEDURE PolaresACartesianas ( Modulo, Angulo :LONGREAL;
                                VAR EjeX, EjeY :LONGREAL );
BEGIN
  EjeX := Modulo * Cos(Angulo);
  EjeY := Modulo * Sin(Angulo)
END PolaresACartesianas;
```

El modo de usar el suprograma es el siguiente:

```
...
PolaresACartesianas (1.0, PI/2.0, x, y);
WrStr("La coordenada X es "); WrLngReal(x, PRECISION, ANCHO);
WrLn;
```

```
WrStr("La coordenada Y es "); WrLngReal(y, PRECISION, ANCHO);
...
```

❑

6.6.3 Parámetros de entrada y salida

Un *parámetro* es *de entrada y salida* cuando establece una comunicación en ambos sentidos entre el programa llamante y el subprograma llamado. El parámetro se usa para pasar un valor desde el programa llamante al subprograma, pero este valor puede también ser modificado por el subprograma y el programa llamante debe, también, ver el cambio. Este tipo de parámetros deben declararse por referencia.

❑ Ejemplo: El siguiente subprograma toma un argumento real y lo redondea:

```
PROCEDURE Redondear (VAR Num : REAL);
BEGIN
  Num := REAL (TRUNC (Num + 0.5))
END Redondear;
```

El modo de usar el suprograma es el siguiente:

```
...
x := 10.7;
Redondear (x);
WrReal(x, 5, 0);
...
```

❑

❑ Ejemplo: El siguiente subprograma intercambia los valores almacenados en dos variables de tipo INTEGER, utilizando para ello una tercera variable temporal:

```
PROCEDURE Intercambiar (VAR Var1, Var2 : INTEGER);
VAR Temp : INTEGER;
BEGIN
  Temp := Var1;
  Var1 := Var2;
  Var2 := Temp
END Intercambiar;
```

El modo de usar el suprograma es el siguiente:

```
...
x := 10;
y := 20;
Intercambiar(x,y);
WrStr("El valor de x es "); WrInt(x,0);
WrLn;
WrStr("El valor de y es "); WrInt(y,0);
...
```

❑

6.7 Anidamiento de subprogramas. Ámbitos

6.7.1 Estructura de bloques

Un subprograma se define de forma similar a un programa completo. En la parte de declaraciones de un subprograma pueden aparecer constantes, variables y otros subprogramas.

Si un subprograma sólo se utiliza dentro de otro subprograma, debe aparecer declarado dentro de éste. Esto da lugar a lo que se llama *anidamiento* de subprogramas: subprogramas declarados dentro de otro subprograma.

Todo lo que aparece declarado dentro de un subprograma se dice que es *local* a éste. Esto significa que las variables, constantes y procedimientos definidos dentro de un subprograma sólo pueden ser usadas por el propio subprograma y por los subprogramas definidos dentro de él, pero no por el programa principal o por los subprogramas externos. Así surge una estructura de bloques, de modo que los bloques más internos pueden acceder a lo declarado en los bloques que les anidan pero no al revés:

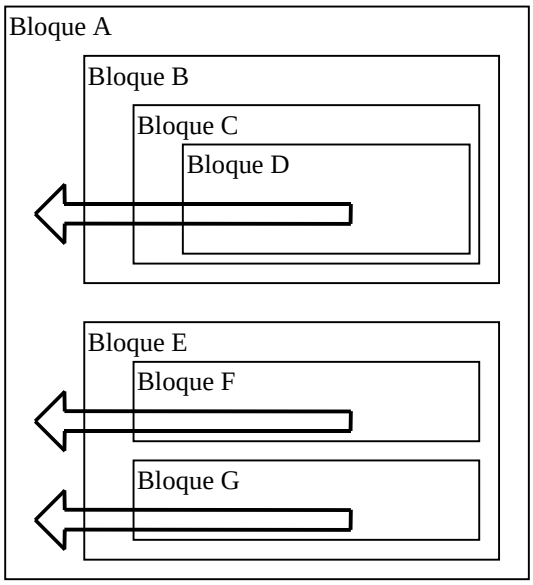

Figura 3. Visibilidad entre bloques

La siguiente tabla muestra qué declaraciones ve cada bloque:

Bloque	Bloques a cuyas declaraciones accede
A	A
B	B, A
C	C, B, A
D	D, C, B, A
E	E, A
F	F, E, A
G	G, E, A

Además, hay que tener en cuenta que un subprograma puede llamar a otro que esté previamente declarado, por lo que E y F pueden llamar además al subprograma B, y G a F y B. En general, si se utiliza un identificador en la declaración de otro identificador, el primero debe estar declarado antes. Por ejemplo, si se desea utilizar una variable no local en un subprograma la declaración de esta variable debe ser anterior a la declaración del subprograma.

Modulo-2 permite además que un subprograma se llame a sí mismo, que es lo que se denomina subprograma recursivo. Hablaremos de esto con más detalle en un tema posterior.

6.7.2 Redefinición de elementos

Dentro de cada bloque se deben declarar los elementos necesarios dándoles los nombres más adecuados en cada caso. Puede ocurrir que dentro de un bloque se defina un elemento con el mismo nombre que un elemento definido en otro bloque y que también sea accesible. En dicho caso se pierde la accesibilidad al elemento más externo.

Sea el siguiente programa:

```
MODULE Ejemplo;
VAR a, b, c : INTEGER;

   PROCEDURE Anidado1 (a : REAL);
   VAR b : REAL;

      PROCEDURE Anidado2;
      VAR c, d : CHAR;
      BEGIN
         Cuerpo de Anidado2
      END Anidado2;

   BEGIN
      Cuerpo de Anidado1
   END Anidado1;

BEGIN
   Cuerpo del principal
END Ejemplo.
```

El procedimiento `Anidado1` tiene visibilidad, en principio, sobre las variables `a`, `b` y `c` del programa principal por estar anidado dentro de él. Sin embargo, uno de sus parámetros se llama `a` y una variable local `b`, por lo que pierde la accesibilidad sobre las variables `a` y `b` del programa principal. Si accedemos a las variables `a` y `b` dentro del cuerpo de `Anidado1`, estaremos accediendo a las suyas locales. Aún así, podremos acceder a la variable `c` del principal, ya que no hay ninguna `c` local. En el cuerpo del procedimiento `Anidado2`, si nombramos la variable `c` estaremos accediendo a la de tipo **`CHAR`** y si accedemos a `a` y `b` lo haremos a las de `Anidado1`. La variable `d` sólo es visible desde el cuerpo de `Anidado2`.

Hay que señalar además que, si la declaración de las variables del programa principal, se realizan después de las declaraciones de los subprogramas `Anidado1` y `Anidado2`, éstos pierden automáticamente el acceso a dichas variables: RECUERDE: SOLO SE PUEDE ACCEDER A ELEMENTOS YA DEFINIDOS.

6.7.3 Efectos laterales

Hemos visto que un subprograma puede acceder a variables no locales a él, siempre que no haya colisión de nombres. Cuando un subprograma accede a una variable que no es ni local a él ni es un parámetro formal suyo decimos que se está produciendo un *efecto lateral*.

Este tipo de situaciones son válidas en *Modula-2* pero nosotros las PROHIBIREMOS totalmente. Cuando se declara un subprograma con una serie de parámetros se está diciendo que el subprograma depende únicamente de ellos. Un efecto lateral traiciona este convenio y hace que los programas sean menos claros y más difíciles de entender.

Siempre es posible que un subprograma no efectúe efectos laterales: si un subprograma necesita una serie de valores deben aparecer todos en la cabecera de su declaración y si necesita variables para realizar cálculos locales a él, se deben declarar como variables locales.

6.7.4 Doble referencia

La *doble referencia* se produce cuando un mismo elemento se referencia con dos nombres distintos. Esto suele ocurrir en dos situaciones concretas:

1) Cuando un subprograma utiliza una variable externa que también se le pasa como argumento
2) Cuando en la llamada a un subprograma se le pasa la misma variable en más de un argumento.

En ambos casos se pueden producir resultados difíciles de entender. Veamos un ejemplo de la primera situación:

```
...
VAR Global : INTEGER;
...
  PROCEDURE Cuadrado (VAR Dato : INTEGER);
  BEGIN
    Global := 5;
    Dato := Dato * Dato;
  END Cuadrado;
...
Global := 3;
Cuadrado (Global);
...
```

Después de la llamada a `Cuadrado(Global)` la variable `Global` vale 25 cuando esperaríamos que valiese 9. Este tipo de situaciones se puede evitar si no usamos efectos laterales.

Un ejemplo del segundo tipo de situación es:

```
PROCEDURE CuadradoCubo (VAR x1, x2, x3 : INTEGER);
BEGIN
  x2 := x1*x1;
  x3 := x1*x1*x1
END CuadradoCubo;
```

que devuelve el cuadrado y el cubo del primer argumento en los argumentos segundo y tercero, y la siguiente llamada:

```
...
A := 4;
CuadradoCubo (A, A, B);
...
```

Tras la ejecución de este fragmento de programa, los valores de las variables son A = 16 y B = 4096, en vez de 64. Este problema se podría haber resuelto declarando el primer argumento como parámetro por valor.

6.8 Sintaxis BNF para subprogramas

Veamos la sintaxis BNF para la declaración de subprogramas en *Modula-2*:

Declaración_Subprograma	**:=**	Cabecera_Subprograma ; Bloque Identificador ;
Cabecera_Subprograma	**::=**	Cabecera_de_Procedimiento **|** Cabecera_de_Función
Cabecera_de_Procedimiento	**::=**	PROCEDURE Identificador **[** Parámetros_Formales **]**

Cabecera_de_Función	::=	PROCEDURE Identificador Parámetros_Formales : Identificador_de_Tipo
Parámetros_Formales	::=	([Grupo_de_Parámetros { ; Grupo_de_Parámetros }])
Grupo_de_Parámetros	::=	[VAR] Lista_de_Identificadores : Identificador_de_Tipo
Bloque	::=	Parte_Declarativa Parte_Ejecutiva END
Parte_Declarativa	::=	{ Declaración }
Declaración	::=	Declaración_de_Constantes | Declaración_de_Variables | Declaración_Subprograma

La sintaxis para la sentencia RETURN es:

Sentencia_RETURN	::=	RETURN [Expresión]

La llamada a un procedimiento es una sentencia:

Sentencia_de_Llamada_a_Procedimiento		
	::=	Identificador [Parámetros_de_Llamada]
Parámetros_de_Llamada	::=	([Lista_de_Expresiones])
Lista_de_Expresiones	::=	Expresión { , Expresión }

Por último la llamada a una función es un factor (ver definición de Expresión):

Factor	::=	Variable | Identificador_de_Constante | ... ... | Llamada_a_Función
Llamada_a_Función	::=	Identificador Parámetros_de_Llamada

Obsérvese que tanto en la llamada como en la definición de procedimientos, si no hay argumentos, los paréntesis son opcionales, mientras que en las funciones son siempre obligatorios.

Relación de Problemas 6

1. Supongamos que tenemos un programa principal con las siguientes variables:

```
VAR

  x, y : REAL;
  m    : INTEGER;
  c    : CHAR;
  n    : CARDINAL;
```

y la siguiente declaración de subprograma:

```
PROCEDURE Prueba (a, b : INTEGER; VAR c, d : REAL;
                  VAR e : CHAR);
```

Averigua cuáles de las siguientes llamadas son válidas:

1) `Prueba (m+3,10,x,y,c);`
2) `Prueba (n+3,10,x,y,c);`
3) `Prueba (m, 19, x, y);`
4) `Prueba (m, m*m, y, x, c);`
5) `Prueba (m, 10, 35.0, y, 'E');`
6) `Prueba (30, 10, x, x+y, c);`

2. Escribe una función que calcule las combinaciones de *m* elementos tomados de *n* en *n*. Para ello escribir primero otra función que calcule el factorial de un número.

NOTA: Recuérdese que

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

Escribe otra función más eficiente que no necesite calcular el factorial.

3. Escribe una función que tome tres parámetros, dos de tipo REAL y uno de tipo CHAR. La función debe sumar, restar, multiplicar o dividir los valores de los dos primeros parámetros dependiendo del código indicado en el tercero (+,-,*,/) y devolver el resultado. Al diseñar esta función, supón que el tercer argumento es necesariamente uno de los cuatro (+,-,*,/).

Escribe un programa que ofrezca al usuario la posibilidad de realizar una de esas cuatro operaciones o salir del programa hasta que el usuario decida abandonar el programa, lea los operandos necesarios para realizar la operación, utilice la función anterior para hacer los cálculos y muestre el resultado en pantalla. Asegúrate de que el programa utiliza la función anterior de forma correcta.

4. Dos números *a* y *b* se dice que son amigos si la suma de los divisores de *a* (salvo el mismo) coincide con *b* y viceversa. Diseña un programa que tenga como entrada dos número naturales *n* y *m*, y muestre por pantalla todas las parejas de números amigos que existan en el intervalo determinado por *n* y *m*.

5. Considera el siguiente procedimiento:

```
PROCEDURE Escr (C : CHAR; Long : INTEGER);
BEGIN
  WHILE Long > 0 DO
    WrChar (C);
    Long := Long – 1
  END
END Escr;
```

a) Si `Ch` tiene el valor `'X'` y `Numero` el valor 5, ¿cuál sería el efecto de ejecutar cada una de las siguientes llamadas al procedimiento?:

```
Escr(Ch,4*Numero-12)    Escr(Ch,6)   Escr(5,Numero)
Escr('/',Numero)       Escr('.',6)  Escr('p',-10)
```

b) Escribe llamadas al procedimiento `Escr` para que cuando se ejecuten produzcan las siguientes salidas:

- 35 guiones sucesivos
- 6 veces tantos espacios en blanco como el valor de `Numero`
- el valor actual de `Ch` 14 veces

6. Dadas las siguientes declaraciones en un determinado algoritmo:

```
VAR  a,b,c   : CARDINAL;
     si      : BOOLEAN;
...
PROCEDURE Uno(x, y : CARDINAL) : BOOLEAN;
...
PROCEDURE Dos(VAR x : CARDINAL; y : CARDINAL);
...
PROCEDURE Tres(x : CARDINAL) : CARDINAL;
```

¿ Cuáles de las siguientes llamadas a subprograma son válidas?

a) `IF Uno(a,b) THEN...`

b) `Dos(a,b+3)`

c) `si := Uno(c,5)`

d) `si := Dos(c,5)`

e) `Dos(a,Tres(a))`

f) `Dos(Tres(b),c)`

g) `IF Tres(a) THEN ...`

h) `b := Tres(Dos(a,5))`

i) `Dos(4,c)`

7. Escribe un algoritmo que tome como entrada desde teclado dos números naturales `N` e `i`, e imprima en pantalla el dígito que ocupa la posición *i*-ésima del número `N`. Si `i` es mayor que el número de dígitos de `N`, se escribirá en pantalla el valor -1. Por ejemplo, para `N` = 25064 e `i` = 2, el resultado es el dígito 6, y para `i` = 7, el resultado es -1.

8. ¿ Qué salida produce por pantalla la ejecución del siguiente algoritmo?

```
MODULE Anidado;
FROM IO IMPORT WrStr, WrCard, WrLn;
VAR a, b, c, x, y : CARDINAL;
  PROCEDURE Primero;
  BEGIN
```

```
    a := 3 * a;
    c := c + 4;
    WrStr ("Primero");
    WrCard (a, 0); WrCard (b, 0); WrCard (c, 0);
    WrLn
  END Primero;
  PROCEDURE Segundo;
  VAR b : CARDINAL;
  BEGIN
    b := 8;
    c := a + c + b DIV 3;
    WrStr ("Segundo");
    WrCard (a, 0); WrCard (b, 0); WrCard (c, 0);
    WrLn
  END Segundo;
  PROCEDURE Tercero (VAR x : CARDINAL; y : CARDINAL);
  BEGIN
    x := x + 4;
    y := y + 1;
    WrStr ("Tercero");
    WrCard (a, 0); WrCard (b, 0);
    WrCard (c, 0); WrCard (x, 0);
    WrLn
  END Tercero;
BEGIN
  a := 3;b := 2;
  c := 1;x := 11;
  y := 22;
  Primero; Segundo;
  Tercero (a,b);
  WrStr ("Anidado");
  WrCard (a, 0); WrCard (b, 0);
  WrCard (c, 0); WrCard (x, 0);
  WrCard (y, 0)
END Anidado.
```

9. Realiza un procedimiento que intercambie el valor de dos variables de tipo CHAR.

10. Escribe un procedimiento que, dadas las coordenadas polares de un número complejo (r, θ), obtenga las correspondientes cartesianas (x, y).

NOTA: Recordar que

$$x = r \cdot \cos\theta$$

$$y = r \cdot \operatorname{sen}\theta$$

11. Escribir un programa que lea un número positivo del teclado y escriba por pantalla los números perfectos menores a él. Para ello definir primero una función `EsPerfecto` que tome como parámetro un número y devuelva `TRUE` o `FALSE` dependiendo de que sea perfecto o no.

12. Escribe un programa que lea un número positivo menor que 100 y escriba la traducción de dicha cantidad a inglés. Por ejemplo para la entrada 42 se obtendría la salida "forty two".

Relación complementaria 6

1. Escribe un algoritmo que acepte como entrada desde teclado un número entero positivo. El algoritmo debe producir como salida el resultado de sumar dos a dos los dígitos que aparecen en posiciones simétricas respecto al dígito central dentro del número dado como entrada. Por ejemplo, para el número 23548 la salida es: 2+8 = 10, 3 + 4 = 7, 5 para el número 6582 la salida es: 6 + 2 = 8, 5 + 8 = 13

2. Escribe un subprograma para calcular el máximo común divisor de cuatro números utilizando para ello otro subprograma que calcule el máximo común divisor de dos números (algoritmo de Euclides).

3. Escribe un programa que lea dos enteros positivos correspondientes a un año y un mes, y escriba el correspondiente calendario de la siguiente forma:

```
 D  L  M  X  J  V  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
```

Este problema puede dividirse en las siguientes partes:

1) Leer los valores del mes y año
2) Encontrar el día de la semana en que empieza el mes (ver ejercicio sobre congruencia de Zeller).
3) Averiguar cuántos días tiene el mes.
4) Escribir el correspondiente calendario.

Usa un subprograma separado para cada parte.

4. Escribe un programa que lea un número positivo del teclado y escriba por pantalla los números primos menores a él. Para ello define primero una función `EsPrimo` que tome como parámetro un número y devuelva `TRUE` o `FALSE` dependiendo de que sea primo o no.

5. Escribe un programa que calcule el máximo de tres números enteros, definiendo previamente una función que calcule el máximo de dos.

El ejercicio 3 se realizará como práctica en el laboratorio.