5 PROCEDIMIENTOS

5.1 Diseño descendente

- En programación es muy importante elegir el diseño adecuado a cada problema. Para esta tarea, se utiliza un diseño descendente, *top-down*, que consiste en dividir el problema en subproblemas más pequeños, que se pueden tratar de forma separada.
- Hasta ahora, no había manera de compilar, testear y depurar cada uno de esos subproblemas separadamente hasta construir el programa final combinando adecuadamente las diferentes secciones de código Fortran generadas.
- Sin embargo, existe la posibilidad de *tratar* cada subproblema de un problema más grande de forma independiente. Consiste en codificar cada subproblema en una *unidad de programa*[5] separada llamada *procedimiento externo*. Cada procedimiento externo puede ser compilado, testado y depurado independientemente de los otros procedimientos del programa antes de combinarlos entre sí para dar lugar al programa final.
- En Fortran, hay dos tipos de procedimientos externos: son los *subprogramas funciones* o simplemente *funciones* y las *subrutinas*. Su ejecución se controla desde alguna otra unidad de programa (que puede ser el programa principal u otro procedimiento externo). Ambos tipos de procedimientos externos se estudian en este capítulo.
- Los beneficios del diseño descendente en los programas son:
    - Es mucho más fácil encontrar errores en el código, sobre todo en programas largos.
    - Permite usar procedimientos construidos por otros programadores.
    - Evita cambios indeseables en las variables del programa. Sólo algunas de ellas se transfieren entre las diferentes unidades de programa, aquellas variables que son necesarias para realizar los cálculos previstos. Las demás variables sólo son accesibles en la unidad de programa donde se declaran,

[5] Unidad de programa es una porción de un programa Fortran compilada separadamente. Son unidades de programa los programas principales, las subrutinas y los subprogramas función.

quedando por lo tanto a salvo de cambios imprevistos para el resto de las unidades de programa.

5.2 Funciones

- Hay dos tipos de funciones:
    - *Intrínsecas*: todas aquellas funciones que suministra el propio lenguaje (ya comentadas en el capítulo 1).
    - *definidas por el propio programador* o *subprogramas función*: procedimientos que permiten responder a necesidades particulares del usuario, no proporcionadas por las funciones intrínsecas.
- Las funciones definidas por el programador se usan igual que las funciones intrínsecas, pueden formar parte de expresiones, y por lo tanto, pueden aparecer en todos aquellos lugares donde se puede usar una expresión. Su resultado es un valor numérico, lógico, cadena de caracteres o array.
- La estructura general de un procedimiento función es:

Cabecera de función

Sección de especificaciones

Sección de ejecución

Terminación de la función

A continuación, se explica en detalle la estructura anterior.

- La sintaxis general de la cabecera de función es la siguiente:

[TIPO] FUNCTION nombre_funcion ([*Lista de argumentos formales*])

    - Es la primera sentencia no comentada del procedimiento, e identifica esa unidad de programa como procedimiento función.
    - TIPO es cualquier tipo Fortran válido relativo a nombre_funcion. Si no aparece TIPO en la cabecera de la función, se debe especificar en la sección de especificaciones.
    - nombre_funcion es cualquier identificador Fortran válido.
    - *Lista de argumentos formales* es una lista (puede ser vacía) de constantes, variables, arrays o expresiones, separados por comas. Se emplean para pasar información al cuerpo de la

función. Se les llama formales porque no conllevan una reserva de espacio en memoria.

- La sección de especificaciones debe declarar el TIPO de nombre_funcion, si no se ha declarado en la cabecera. Además, debe declarar el tipo de los argumentos formales y las variables *locales*[6] a la función, si las hay.
- La sección de ejecución debe incluir al menos una sentencia de asignación en la que se cargue al nombre de la función el resultado de una expresión del mismo tipo (de ahí la necesidad de declarar el tipo del nombre de la función):

nombre_funcion = expresion

- Finalmente, la terminación de la función tiene la sintaxis general:

END FUNCTION [nombre_funcion]

Una función se invoca escribiendo:

nombre_función ([lista de argumentos actuales])

- formando parte de una *expresión* o en cualquier lugar donde puede aparecer una expresión. Como resultado de la evaluación de la función en sus argumentos actuales se devuelve un valor que es usado para evaluar, a su vez, la expresión de la que forme parte.
- Debe haber una concordancia en el número, tipo y orden de los argumentos actuales que aparecen en la llamada a la función y los argumentos formales que aparecen en la cabecera de la misma. Asimismo, el tipo del nombre de la función debe ser el mismo en la(s) unidad(es) de programa que invoca(n) a la función y el declarado en el propio procedimiento función.
- La ejecución de la llamada ocurre de la siguiente manera:
    - Se evalúan los argumentos actuales que son expresiones.
    - Se *asocian* los argumentos actuales con sus correspondientes argumentos formales.
    - Se ejecuta el cuerpo de la función especificada.
    - Se devuelve el control a la unidad de programa que hizo la llamada, en concreto, a la sentencia donde se invocó a la función, sustituyendo su nombre por el resultado de la ejecución de la función.

[6] Variables necesarias para llevar a cabo los cálculos previstos dentro de la función. Son inaccesibles fuera de la misma.

- Una función bien diseñada debe producir un resultado (transferido por el nombre de la misma) a partir de uno o más valores de entrada (transferidos por la lista de argumentos).
- En Fortran, la transferencia de argumentos entre dos unidades de programa cualesquiera se realiza *por dirección*. Así, la asociación entre argumentos actuales y formales significa pasar las direcciones de memoria que ocupan los argumentos actuales al procedimiento llamado, de modo que éste puede leer y escribir en esas direcciones de memoria. Por ejemplo, para escribir en un argumento formal basta ponerle a la izquierda de una sentencia de asignación en el interior de un procedimiento llamado.
- La sintaxis general de los argumentos formales es:

TIPO, [INTENT(*intencion_paso*)] :: arg_formal1[,arg_formal2]...

- donde *intencion_paso* se sustituye por:
  - IN: si el argumento formal es un valor de entrada.
  - OUT: si el argumento formal es un valor de salida.
  - IN OUT: si el argumento formal es un valor de entrada y salida.
- El atributo INTENT ayuda al compilador a encontrar errores por el uso indebido de los argumentos formales de un procedimiento.
- En el caso de un procedimiento función bien diseñado, el atributo de todos los argumentos formales debe ser IN.
- Ejemplo. Sea la función que convierte una temperatura en grados Fahrenheit a grados Celsius:

```
FUNCTION cent (temperatura)
REAL:: cent !declaracion del nombre de la funcion
REAL, INTENT(IN):: temperatura !declaracion del argumento formal
REAL, PARAMETER:: CTE=5./9. !parametro local a la funcion
cent = CTE*(temperatura – 32.)
END FUNCTION cent
```

- Llamadas válidas desde un programa principal son:

```
PROGRAM conversion
REAL:: cent, x,var1,var2
…
var1 = cent (23.)
x=35.75
var2 = -2.*cent (x)+4
WRITE(*,*) var1,var2
…
```

```
END PROGRAM conversion
```

- Se observa que la declaración de CENT es REAL en la unidad de programa principal en concordancia con su tipo en la función y que el argumento actual en ambas llamadas es REAL, en concordancia con su tipo como argumento formal en la función.
- Los nombres de los argumentos actuales y sus correspondientes formales no tienen porqué ser iguales.
- Ejemplo. Sea la función factorial:

```
INTEGER FUNCTION factorial (n)
INTEGER, INTENT(IN) :: n ! declaración del argumento formal
INTEGER :: i ! variable local a la funcion
factorial = 1
DO i = 2, n
factorial = factorial*i
END DO
END FUNCTION factorial
```

- Un programa principal puede ser:

```
PROGRAM ejemplo
INTEGER:: i, n, factorial
! argumento actual y variable local a la unidad de programa principal
WRITE (*,*) 'Teclee un numero entero:'
READ (*,*) i
n = factorial (i)
..
END PROGRAM ejemplo
```

- Las variables *n, i* del programa principal no tienen nada que ver con las variables *n, i* de la función. Puesto que las variables locales a una unidad de programa son visibles únicamente en el interior de la misma, se pueden usar los mismos nombres para variables locales en unidades de programa diferentes, sin problemas de conflictos.

5.3 Subrutinas

- Son procedimientos más generales que las funciones, aunque comparten casi todas sus características. Pueden retornar más de un valor, o no retornar nada en absoluto. Reciben los valores de entrada y devuelven los valores de salida a través de su lista de argumentos.
- La estructura general de una subrutina es idéntica a la de una función:

Cabecera de subrutina

Sección de especificaciones

Sección ejecutable

Terminación de subrutina

- donde la sintaxis general de la cabecera de subrutina es:

SUBROUTINE nombre_subrutina ([Lista de argumentos formales])

- y la sintaxis general de la terminación de subrutina es:

END SUBROUTINE [nombre_subrutina]

- La sección de especificaciones incluye la declaración de los tipos de los argumentos formales con su atributo INTENT correspondiente a su intención de uso y la de las variables locales a la subrutina, si las hay.
- La sintaxis general de llamada a una subrutina desde cualquier unidad de programa es:

CALL nombre_subrutina ([lista de argumentos actuales])

- La ejecución de la llamada ocurre de la siguiente manera:
    - Se evalúan los argumentos actuales que son expresiones.
    - Se *asocian* los argumentos actuales con sus correspondientes argumentos formales. El paso de los argumentos se realiza por dirección.
    - Se ejecuta el cuerpo de la subrutina especificada.
    - Se devuelve el control a la unidad de programa que hizo la llamada, en concreto, a la sentencia siguiente a la sentencia CALL.
    - La subrutina NO devuelve ningún valor a través de su nombre, sino que son los argumentos los encargados de realizar las transferencias de resultados.
- Debe haber concordancia en el número, tipo y orden de los argumentos actuales y sus correspondientes argumentos formales, al igual que en las funciones.
- Ejemplo. Sea la subrutina para convertir grados, minutos y segundos a grados decimales:

```
SUBROUTINE convierte (grados, minutos, segundos, grads)
INTEGER, INTENT(IN) :: grados, minutos, segundos
REAL, INTENT(OUT)  :: grads
grads= REAL (grados) + REAL (minutos)/60. + REAL (segundos) /3600
END SUBROUTINE convierte
```

- Llamadas válidas desde un programa principal son:

```
PROGRAM principal
INTEGER :: g,m,s
REAL:: n,gd
…
CALL convierte (30, 45, 2, n) ! llamada 1
WRITE (*,*) ’30 grados, 45 minutos, 2 segundos equivalen a‘
WRITE (*,*) n,’grados decimales’
CALL convierte (g, m, s, gd) ! llamada 2
WRITE (*,*) g,’grados’, m,’minutos’, s,’segundos equivalen a‘
WRITE (*,*) gd,’grados decimales’
…
END PROGRAM principal
```

5.4 Transferencia de arrays a procedimientos

- Cuando la llamada a un procedimiento incluye el nombre de un array en un argumento actual, se transfiere la dirección de memoria del primer elemento del mismo. De esta manera, el procedimiento es capaz de acceder al array, pues todos sus elementos ocupan direcciones de memoria consecutivas.
- Además, el procedimiento debe conocer el tamaño del array, en concreto, los límites de los índices de cada dimensión para que las operaciones efectuadas en el cuerpo del procedimiento se realicen sobre elementos permitidos. Hay dos formas de hacerlo:
    - array formal con perfil explícito. Consiste en pasar la extensión de cada dimensión del array en la lista de argumentos y usarlas en la declaración del array formal en el procedimiento. Esta forma permite operaciones con arrays completos y subconjuntos de arrays dentro del procedimiento.
    - Ejemplo:

```
…
CALL proced (matriz, d1, d2, resul) !llamada a subrutina
…
SUBROUTINE proced (matriz, d1, d2, resul)
INTEGER, INTENT(IN):: d1,d2
INTEGER, INTENT(IN), DIMENSION(d1,d2)::matriz ! perfil explícito
INTEGER, INTENT(OUT):: resul
```

    - array formal con perfil asumido. La declaración de un array formal de este tipo usa dos puntos : para cada índice del

mismo. Permite operaciones con arrays completos y subconjuntos de arrays dentro del procedimiento. El procedimiento debe tener interfaz explícita, concepto que se estudiará más adelante.

- Ejemplo:

```
MODULE mod1
CONTAINS
SUBROUTINE sub1 (matriz)
INTEGER, INTENT(INOUT), DIMENSION(:,:)::matriz ! perfil asumido
…
END SUBROUTINE sub1
END MODULE mod1
```

5.5 Compartir datos con módulos

- Hasta ahora, un programa Fortran intercambia datos entre sus distintas unidades de programa (principal, función, subrutina) a través de las listas de argumentos.
- Además de las listas de argumentos, un programa Fortran puede intercambiar datos a través de *módulos*.
- Un módulo es una unidad de programa compilada por separado del resto que contiene, al menos, las declaraciones e inicializaciones necesarias de los datos que se quieren compartir entre las unidades de programa.
- La sintaxis general de un módulo es:

```
MODULE nombre_modulo
[SAVE]
Declaración e inicialización datos compartidos   Cuerpo del módulo
END MODULE nombre_modulo
```

- La sentencia SAVE es útil para preservar los valores de los datos del módulo cuando éste se comparte entre varias unidades de programa.
- Para poder usar los datos de módulos en una unidad de programa, escribir la sentencia:

```
USE nombre_modulo1[,nombre_modulo2],…
```

- como primera sentencia no comentada inmediatamente después del cabecero de la unidad de programa que quiere usarlos.
- Los módulos de datos son útiles cuando se necesita compartir grandes cantidades de datos entre muchas unidades de programa, pero manteniéndolos invisibles para las demás.

- Ejemplo. Escribir un módulo de datos que comparta dos vectores con valores iniciales v1(1 1 1 1 1) y v2(10 11 12 13 14) y una matriz *m* entre el programa principal y una subrutina *sub*. El programa principal debe calcular el vector suma de v1 y v2 y la subrutina debe volcar el vector suma en la primera columna de la matriz y el vector v2 en la segunda columna.

```
MODULE comparte_datos
IMPLICIT NONE
SAVE
INTEGER, PARAMETER:: TM=5
INTEGER:: i
INTEGER, DIMENSION(TM) :: v1=1,v2=(/ (i, i=10,14) /)
INTEGER, DIMENSION(TM,2) :: m
END MODULE comparte_datos

PROGRAM principal
USE comparte_datos
IMPLICIT NONE
INTEGER :: j
WRITE(*,*) 'v1',v1
WRITE(*,*) 'v2',v2
v1=v1+v2
WRITE(*,*) 'v1',v1
CALL sub
WRITE(*,*) 'm'
DO i=1,TM
WRITE(*,*) (m(i,j),j=1,2)
END DO
END PROGRAM principal

SUBROUTINE sub
USE comparte_datos
IMPLICIT NONE
m( : , 1 ) = v1
m( : , 2 ) = v2
END SUBROUTINE sub
```

5.6 Procedimientos módulo

- Además de datos, un modulo puede contener procedimientos (subrutinas y/o funciones), que se denominan entonces *procedimientos módulo*.
- La sintaxis general de un procedimiento módulo es:

```
MODULE nombre_modulo
[SAVE]
Declaración e inicialización datos compartidos
CONTAINS
Estructura general procedimiento1
[Estructura general procedimiento2]
...
END MODULE nombre_modulo
```

- Como ocurre con los módulos de datos, para hacer accesibles procedimientos módulos a una unidad de programa, escribir:

```
USE nombre_modulo1[,nombre_modulo2]...
```

- como la primera sentencia no comentada, inmediatamente después del cabecero de la unidad de programa que quiere usarlo.
- Un procedimiento contenido en un módulo se dice que tiene una *interfaz explícita*, pues el compilador conoce todos los detalles de su lista de argumentos. Como consecuencia, cuando se usa el módulo en cualquier unidad de programa, el compilador chequea la concordancia de número, tipo y orden entre las listas de argumentos actuales y sus correspondientes formales, así como usos indebidos de los últimos según el valor del atributo INTENT.
- Por contraposición, un procedimiento externo fuera de un módulo se dice que tiene una *interfaz implícita*. El compilador desconoce los detalles de las listas de argumentos y, por tanto, no puede chequear errores de concordancias en las mismas. Es responsabilidad del programador encargarse de chequearlo.
- Ejemplo. Sumar dos números enteros usando un procedimiento módulo.

```
MODULE mod1
CONTAINS
SUBROUTINE sub1 (a, b, sumar)
IMPLICIT NONE
INTEGER, INTENT(IN):: a,b
INTEGER, INTENT(OUT):: sumar
sumar=a+b
```

```
END SUBROUTINE sub1
END MODULE mod1

PROGRAM principal
USE mod1
IMPLICIT NONE
INTEGER::x,y,resul
WRITE(*,*) ‘dame dos numeros’
READ(*,*) x,y
CALL sub1(x,y,resul)
WRITE(*,*) ‘la suma es’,resul
END PROGRAM principal
```

- Comprobar que si se declaran los argumentos actuales del tipo REAL, el compilador encuentra el error de concordancia de tipos. Sin embargo, si se repite el ejercicio con la subrutina sub1 como procedimiento externo, el compilador no encuentra errores.

5.7 Procedimientos como argumentos

- Los argumentos actuales de un procedimiento pueden ser nombres de subrutinas o funciones definidas por el programador. Como el paso de argumentos se realiza por dirección, en este caso, se pasa la dirección de memoria de comienzo del procedimiento.
    - Si el argumento actual es una función, necesita el atributo EXTERNAL en su sentencia de declaración de tipo, tanto en el procedimiento de llamada como en el procedimiento llamado. La sintaxis general es:

```
TIPO, EXTERNAL:: nombre_funcion
```

- Ejemplo:

```
PROGRAM principal
IMPLICIT NONE
INTEGER:: a=5, b=7
REAL, EXTERNAL:: fun
REAL:: x
CALL sub (fun, a, b, x)
WRITE(*,*) ‘resultado’,x
END PROGRAM principal

SUBROUTINE sub (f, a,b,res)
```

```
IMPLICIT NONE
REAL, EXTERNAL:: f
INTEGER, INTENT(IN):: a,b
REAL, INTENT(OUT):: res
res=f(a)+b**2
END SUBROUTINE sub

REAL FUNCTION fun(a)
INTEGER, INTENT(IN):: a
fun=(2.*a-5.)/7.
END FUNCTION fun
```

- Si el argumento actual es una subrutina, es necesario escribir una sentencia EXTERNAL, tanto en el procedimiento de llamada como en el procedimiento llamado. La sintaxis general es:

```
EXTERNAL:: nombre_subrutina
```

5.8 Atributo y sentencia SAVE

- Cada vez que se sale de un procedimiento, los valores de sus variables locales se pierden, a menos que se guarden poniendo el atributo SAVE en las sentencias de declaración de tipo de aquellas variables que se quieren guardar. La sintaxis general es:

```
TIPO, SAVE:: variable_local1[, variable_local2]...
```

- Para guardar todas las variables locales a un procedimiento escribir simplemente SAVE en una sentencia ubicada en la sección de especificaciones del procedimiento.
- Automáticamente, toda variable local inicializada en su sentencia de declaración se guarda.
- Ejemplo:

```
INTEGER FUNCTION fun(N)
INTEGER INTENT(IN):: N
INTEGER, SAVE:: cuenta
cuenta=0
cuenta = cuenta + 1! Cuenta las veces que se llama la función
...
END FUNCTION fun
```

5.9 Procedimientos internos

- Hasta ahora, se han estudiado dos tipos de procedimientos: los procedimientos externos y los procedimientos módulo. Además, existe un tercer tipo de procedimientos, los llamados procedimientos internos.
- Un procedimiento interno es un procedimiento completamente contenido dentro de otra unidad de programa, llamada *anfitrión* o *host*. El procedimiento interno se compila junto con su anfitrión y sólo es accesible desde él. Debe escribirse a continuación de la última sentencia ejecutable del anfitrión, precedido por una sentencia CONTAINS.
- La estructura general de una unidad de programa que contiene un procedimiento interno es:

Cabecero de unidad de programa

Sección de especificaciones

Sección ejecutable

CONTAINS

Procedimiento interno

Fin de unidad de programa

- Un procedimiento interno tiene acceso a todos los datos definidos por su anfitrión, salvo aquellos datos que tengan el mismo nombre en ambos. Los procedimientos internos se usan para realizar manipulaciones de bajo nivel repetidamente como parte de una solución.

5.10 Procedimientos recursivos

- Un procedimiento es recursivo cuando puede llamarse a sí mismo directa o indirectamente las veces que se desee.
- Para declarar un procedimiento como recursivo, añadir la palabra clave RECURSIVE a la sentencia cabecero del procedimiento.
    - La sintaxis general para el caso de un procedimiento subrutina es:

RECURSIVE SUBROUTINE nombre_subrutina [(Lista arg formales)]

    - Además de esto, en el caso de una función recursiva, Fortran permite especificar dos nombres distintos para invocar a la función y para devolver su resultado, con el fin de evitar confusión entre los dos usos del nombre de la función. En particular, el nombre de la función se usa para invocar a la función, mientras que el resultado de la misma se devuelve a través de un argumento formal especial especificado entre paréntesis a la derecha de una cláusula RESULT en la propia

sentencia cabecero de la función. La sintaxis general de una función recursiva es:

RECURSIVE FUNCTION nom_fun ([List arg form]) RESULT (resultado)

- Con esta forma, la sección de especificaciones no incluirá declaración de tipo del nombre de la función; en su lugar, debe declararse el tipo de *resultado*.

- Ejemplo. Calcular el factorial de un número usando una subrutina recursiva.

```
RECURSIVE SUBROUTINE factorial (n, resultado)
INTEGER, INTENT(IN):: n
INTEGER, INTENT(OUT):: resultado
INTEGER :: temp
IF (n>=1) THEN
CALL factorial (n-1, temp)
resultado=n*temp
ELSE
resultado=1
ENDIF
END SUBROUTINE factorial
```

- Ejemplo. Lo mismo pero usando una función recursiva.

```
RECURSIVE FUNCTION factorial (n) RESULT (resultado)
INTEGER, INTENT(IN):: n
INTEGER:: resultado
IF (n>=1) THEN
resultado=n*factorial(n-1)
ELSE
resultado=1
ENDIF
END FUNCTION factorial
```

5.11 Argumentos opcionales y cambios de orden

- Hasta ahora se ha dicho que la lista de argumentos actuales debe coincidir en orden, tipo y número con su lista de argumentos formales.
- Sin embargo, es posible cambiar el orden de los argumentos actuales y/o especificar sólo algunos de ellos pero no todos según

interese en cada llamada al procedimiento, siempre que tal procedimiento tenga interfaz explícita.

- Para cambiar el orden de los argumentos actuales en la llamada a un procedimiento, cada argumento actual se debe especificar en la llamada con la sintaxis general:

nombre_argumento_formal= argumento_actual

- Ejemplo. Sea la función con interfaz explícita:

```
MODULE mod
CONTAINS
FUNCTION calcula (primero, segundo, tercero)
…
END FUNCTION calcula
END MODULE mod
```

Llamadas idénticas que producen los mismos resultados son:

```
WRITE(*,*) calcula (5, 3, 7)
WRITE(*,*) calcula (primero=5, segundo=3, tercero=7)
WRITE(*,*) calcula (segundo=3, primero=5, tercero=7)
WRITE(*,*) calcula (5, tercero=7, segundo=3)
```

- El cambio de orden de los argumentos actuales constituye una complicación por sí solo sin interés. Su utilidad radica en combinar el cambio de orden con el hecho de que algunos argumentos sean opcionales.
- Un argumento formal es opcional cuando no necesita siempre estar presente cuando se llama al procedimiento que lo incluye. Para definir un argumento formal como opcional hay que añadir en su declaración de tipo el atributo OPTIONAL.
- Ejemplo:

```
MODULE mod
CONTAINS
SUBROUTINE sub (arg1, arg2, arg3)
INTEGER, INTENT(IN), OPTIONAL:: arg1
INTEGER, INTENT(IN):: arg2
INTEGER, INTENT(OUT):: arg3
…
END SUBROUTINE sub
END MODULE mod
```

- Los argumentos formales opcionales sólo pueden declararse en procedimientos con interfaz explícita. Cuando están presentes en la

llamada al procedimiento, éste los usa, sino están presentes, el procedimiento funciona sin ellos. La forma de testar si el argumento opcional debe usarse en el procedimiento o no, es con la función intrínseca lógica PRESENT. Para ello, en el ejemplo anterior, el cuerpo de la subrutina *sub* debe incluir la estructura condicional:

IF (PRESENT(arg1)) THEN

Acciones a realizar cuando arg1 está presente

ELSE

Acciones a realizar cuando arg1 está ausente

ENDIF

- Al llamar a un procedimiento con argumentos opcionales, pueden ocurrir varias situaciones:
    - Si están presentes los argumentos opcionales en el orden adecuado, la llamada tiene la forma habitual: CALL sub (2, 9, 0)
    - Si están ausentes los argumentos opcionales y se respeta el orden de los mismos, la llamada es: CALL sub (9, 0)
    - Si están ausentes los argumentos opcionales y hay cambios de orden en los mismos, la llamada es: CALL sub (arg3=0,arg2=9)

EJERCICIOS RESUELTOS

Objetivos:

Aprender a dividir un programa Fortran en subprogramas FUNCTION o SUBROUTINE.

Se muestran los aspectos básicos de los procedimientos, relativos a las formas de llamada a un procedimiento y de transferencias de datos entre distintas unidades de programa. También se ven algunos aspectos avanzados que aportan mayor flexibilidad y control sobre los procedimientos.

En la práctica, todos los ejercicios que se muestran en este capítulo se pueden realizar de dos formas en un entorno de programación FORTRAN: escribiendo todas las unidades de programa en el mismo archivo Fortran, unas a continuación de otras, o bien, creando un proyecto y añadiendo un archivo por cada unidad de programa, que se compilará por separado. En este caso, es recomendable usar procedimientos módulo, según se ha estudiado en la teoría.

1. Sumar dos números enteros usando una función.

```
PROGRAM cap5_1
IMPLICIT NONE
INTEGER :: a,b,suma
WRITE(*,*) 'DAME 2 NUMEROS'
READ(*,*) a,b
WRITE(*,*) 'LA SUMA ES',suma(a,b)
END PROGRAM cap5_1

INTEGER FUNCTION suma(x,y)
IMPLICIT NONE
INTEGER, INTENT(IN) :: x,y
suma=x+y
END FUNCTION suma
```

– La llamada a la función SUMA se realiza en una sentencia WRITE en el programa principal.

– La ejecución de la llamada ocurre de la siguiente manera:

– 1. Se asocian los argumentos actuales con sus correspondientes argumentos formales. Es decir, a (variable entera) se asocia con x (del mismo tipo que a) y b (variable entera) con y (del mismo tipo que b).

– 2. Se ejecuta el cuerpo de la función SUMA, lo cual requiere cargar en SUMA el resultado de la adición de x e y.

– 3. Se devuelve el control a la sentencia WRITE del programa principal.

2. Calcular el número combinatorio $\binom{m}{n}$ sabiendo que m debe ser mayor o igual que n.

```
PROGRAM cap5_2
IMPLICIT NONE
INTEGER :: n,m
INTEGER :: fact
REAL :: resul
WRITE(*,*) 'DAME 2 NUMEROS'
READ(*,*) m,n
```

```
IF (m<n) THEN
  WRITE(*,*) 'NO SE PUEDE'
ELSE
  resul=fact(m)/(fact(n)*fact(m-n))
  WRITE(*,*) 'RESULTADO',resul
END IF
END PROGRAM cap5_2

FUNCTION fact(x)
IMPLICIT NONE
INTEGER, INTENT(IN) :: x
INTEGER :: i,fact
fact=1
DO i=1,x
  fact=fact*i
END DO
END FUNCTION fact
```

– En este ejercicio se realizan tres llamadas a la función FACT, que calcula el factorial de un número. Estas llamadas forman parte de una expresión aritmética. En el caso de FACT(m-n), en primer lugar se evalúa la resta en el argumento actual, a continuación se asocia con el argumento formal x definido en la función y se ejecuta la función.

– Obviamente, cuanto mayor es el número de llamadas a una función, mayor es la motivación de codificarla por separado del programa principal.

3. Calcular $\sum_{i=1}^{n} \frac{1}{i!}$ siendo n leído por teclado. Usar una función para calcular el factorial (i!). El programa se ejecuta tantas veces como el usuario quiera, por ejemplo, mientras se teclee la letra 'S'.

```
PROGRAM cap5_3
IMPLICIT NONE
CHARACTER (LEN=1) :: seguir
INTEGER :: fact,i,n
REAL :: sumator
DO
  WRITE(*,*) 'NUMERO DE ELEMENTOS DEL SUMATORIO?'
```

```
  READ(*,*) n
  sumator=0
  DO i=1,n
    sumator=sumator+1./fact(i)
  END DO
  WRITE(*,*) 'EL RESULTADO ES:',sumator
  WRITE(*,*) 'DESEA CONTINUAR (S/N)?'
  READ(*,*) seguir
  IF (seguir /= 'S') EXIT
END DO
END PROGRAM cap5_3

FUNCTION fact(x)
IMPLICIT NONE
INTEGER, INTENT(IN) :: x
INTEGER :: fact, i
fact=1
DO i=1,x
  fact=fact*i
END DO
END FUNCTION fact
```

– La llamada a la función FACT se realiza, en este caso, desde el interior de un bucle, y forma parte de una expresión aritmética. La ejecución de la función se realiza tantas veces como valores toma el índice del bucle.

4. Calcular la media de cinco números (leídos por teclado) utilizando para ello una función.

```
PROGRAM cap5_4
IMPLICIT NONE
REAL :: media
REAL, DIMENSION(5):: vector
INTEGER :: i
DO i=1,5
  WRITE(*,*) 'DAME COMPONENTE',i,'DEL VECTOR'
  READ(*,*) vector(i)
END DO
WRITE(*,*) 'LA MEDIA ES:',media(vector,5)
```

```
END PROGRAM cap5_4

REAL FUNCTION media(xx,d1)
IMPLICIT NONE
INTEGER, INTENT(IN) :: d1
REAL,DIMENSION(d1),INTENT(IN):: xx!perfil explicito
REAL :: suma
INTEGER :: i
suma=0
DO i=1,d1
  suma=suma+xx(i)
END DO
media=suma/d1
END FUNCTION media
```

– Notar que para transferir un array (vector o matriz) a un procedimiento como argumento basta escribir el nombre del mismo. El array formal XX se ha declarado usando la dimensión transferida en el argumento d1. ¿Qué cambios hay que hacer en el programa para usar perfil asumido en el array formal xx?

– El resultado de la media se devuelve al programa principal a través del nombre de la función.

5. Lo mismo que en el ejercicio CAP5_1 pero usando una subrutina.

```
PROGRAM cap5_5
IMPLICIT NONE
INTEGER :: a,b,s
WRITE(*,*) 'DAME 2 NUMEROS'
READ(*,*) a,b
CALL suma(a,b,s)
WRITE(*,*) 'LA SUMA ES',s
END PROGRAM cap5_5

SUBROUTINE suma(x,y,z)
IMPLICIT NONE
INTEGER, INTENT(IN) :: x
INTEGER, INTENT(IN) :: y
INTEGER, INTENT(OUT) :: z
```

```
z=x+y
END SUBROUTINE suma
```

– Notar la forma de llamar a la subrutina usando la sentencia CALL.

– La ejecución de la llamada ocurre de la misma manera que en el caso de una función.

– El resultado de la suma de las dos variables se transfiere al programa principal a través del argumento Z.

– ¿En qué tipo de problemas usarías subprogramas función y en cuáles subprogramas subrutina? ¿Es indiferente?

6. Intercambiar los valores de dos variables enteras. Usar una subrutina para realizar el intercambio.

```
PROGRAM cap5_6
IMPLICIT NONE
INTEGER :: a=5,b=10
WRITE(*,*) 'ANTES DEL CAMBIO'
WRITE(*,*)' A= ',a,' B= ',b
CALL cambia(a,b)
WRITE(*,*) 'DESPUES DEL CAMBIO EN PRINCIPAL'
WRITE(*,*) ' A= ',a,' B= ',b
END PROGRAM cap5_6

SUBROUTINE cambia(x,y)
IMPLICIT NONE
INTEGER, INTENT(IN OUT) :: x
INTEGER, INTENT(IN OUT) :: y
INTEGER :: aux
aux=x
x=y
y=aux
WRITE(*,*) 'DESPUES DEL CAMBIO EN SUBRUTINA'
WRITE(*,*) ' X= ',x,' Y= ',y
END SUBROUTINE cambia
```

– En este programa vemos que el cambio de valores de los argumentos formales x e y se refleja también en los argumentos actuales a y b.

7. Lo mismo que en el ejercicio CAP5_4 pero usando una subrutina.

```
PROGRAM cap5_7
IMPLICIT NONE
REAL :: resul
REAL, DIMENSION(5)::vector
INTEGER :: i
WRITE(*,*) 'DAME VECTOR'
READ(*,*) (vector(i),i=1,5)
CALL media(vector,resul,5)
WRITE(*,*) 'LA MEDIA ES:',resul
END PROGRAM cap5_7

SUBROUTINE media(num,solu,d1)
IMPLICIT NONE
INTEGER, INTENT(IN):: d1
REAL, DIMENSION(d1),INTENT(IN) :: num !perfil explicito
REAL, INTENT(OUT) :: solu
REAL :: suma
INTEGER :: i
suma=0
DO i=1,d1
  suma=suma+num(i)
END DO
solu=suma/d1
END SUBROUTINE media
```

– En la llamada a la subrutina, la dirección de memoria del primer elemento de vector se pasa al argumento formal num y la solución calculada, media de los cinco números de ese array, se pasa a su vez al programa principal a través del argumento formal solu.

8. Calcular la cantidad de números positivos, negativos y ceros que hay en una matriz, sabiendo que el nº de filas y columnas es como máximo 10. Usar una subrutina para leer el número de filas y columnas de la matriz, así como sus elementos y otra subrutina para calcular el número de positivos, negativos y ceros que tiene la matriz.

```
PROGRAM cap5_8
```

```
IMPLICIT NONE
INTEGER, PARAMETER:: TM=10
INTEGER, DIMENSION(TM,TM) :: mat
INTEGER:: fila,columna,pos,neg,ceros
CALL leer(mat,fila,columna,TM)
CALL cuenta(mat,fila,columna,pos,neg,ceros)
WRITE(*,*) 'EL NUMERO DE POSITIVOS ES:',pos
WRITE(*,*) 'EL NUMERO DE NEGATIVOS ES:',neg
WRITE(*,*) 'EL NUMERO DE CEROS ES:',ceros
END PROGRAM cap5_8

SUBROUTINE leer(mat,fil,col,TM)
IMPLICIT NONE
INTEGER, INTENT(IN):: TM
INTEGER, DIMENSION(TM,TM),INTENT(OUT):: mat
INTEGER, INTENT(OUT)   :: fil,col
INTEGER :: k,j

DO
 WRITE(*,*) '¿Nº DE FILAS?'
 READ(*,*) fil
 WRITE(*,*)'¿Nº DE COLUMNAS?'
 READ(*,*) col
 IF (fil<=10 .AND. col<=10) EXIT
END DO
DO  k=1,fil
 DO  j=1,col
  WRITE(*,*) 'ELEMENTO',k,j,'DE LA MATRIZ'
  READ(*,*) mat(k,j)
 END DO
END DO
END SUBROUTINE leer

SUBROUTINE cuenta(mat,fil,col,pos,neg,cer)
IMPLICIT NONE
INTEGER, INTENT(IN):: fil,col
INTEGER, DIMENSION(10,10), INTENT(IN):: mat
INTEGER, INTENT(OUT):: pos,neg,cer
INTEGER :: k,j
pos=0;neg=0;cer=0;
```

```
DO  k=1,fil
 DO  j=1,col
  IF (mat(k,j) < 0) THEN
   neg=neg+1
  ELSE IF (mat(k,j) == 0) THEN
     cer=cer+1
  ELSE
     pos=pos+1
  END IF
 END DO
END DO
END SUBROUTINE cuenta
```

– Un bucle controla que se introduce el tamaño permitido para la matriz, es decir, como máximo 10X10. Notar que el tamaño de la matriz debe ser en todas las unidades de programa igual 10X10, puesto que la transferencia del mismo se realiza por columnas.

– Se usan tres contadores para calcular las cantidades pedidas.

9. Generar aleatoriamente un número de 1 a 100. Se trata de adivinar qué número es, con sucesivos intentos.

```
PROGRAM cap5_9
IMPLICIT NONE
INTEGER :: suerte,n2,n,intento
n=suerte()
WRITE(*,*) "SE HA GENERADO UN NUMERO ENTRE 1 Y 100"
WRITE(*,*) "INTENTA ADIVINARLO"
 intento=0
 DO
  WRITE(*,*) 'DAME UN NUMERO'
  READ(*,*) n2
  intento=intento+1
  IF (n2==n) THEN
   WRITE(*,*)'ACERTASTES!'
   WRITE(*,*) 'HAS NECESITADO',intento,'    INTENTOS!'
   EXIT
  END IF
  CALL pista(n,n2)
```

```
  END DO
END PROGRAM cap5_9

INTEGER FUNCTION suerte()
IMPLICIT NONE
REAL :: n
CALL random_seed()
CALL random_number(n)
n=n*100
suerte =n+1
END FUNCTION suerte

SUBROUTINE pista(n,n2)
INTEGER, INTENT(IN) :: n,n2
IF (n<n2) THEN
  WRITE(*,*) "LO QUE BUSCAS ES MENOR"
ELSE
  WRITE(*,*) "LO QUE BUSCAS ES MAYOR"
END IF
END SUBROUTINE pista
```

– RANDOM_SEED() y RANDOM_NUMBER(arg) son dos subrutinas intrínsecas FORTRAN. La primera inicializa el procedimiento aleatorio y la segunda genera un número aleatorio real arg tal que 0<=arg<1.

10. Leer por teclado el coeficiente de convección (h), la diferencia de temperatura (dT), el radio y la altura de un cilindro y calcular la pérdida de calor, según la fórmula q=hAdT, donde A es el área del cilindro.

```
PROGRAM cap5_10
IMPLICIT NONE
REAL :: h,dt,r,al,area
WRITE(*,*) 'COEFICIENTE DE CONVECCION'
READ(*,*) h
WRITE(*,*) 'DIFERENCIA DE TEMPERATURA'
READ(*,*) dt
WRITE(*,*) 'RADIO Y ALTURA'
READ(*,*) r,al
```

```
WRITE(*,*) 'PERDIDA DE CALOR: ',h*area(r,al)*dt
END PROGRAM cap5_10

REAL FUNCTION area(r,al)
IMPLICIT NONE
REAL, INTENT(IN) :: r,al
REAL :: pi,circulo
pi=2*ASIN(1.)
area=(2*pi*r*al)+2*circulo(r,pi)
END FUNCTION area

REAL FUNCTION circulo(rad,pi)
IMPLICIT NONE
REAL, INTENT(IN) :: rad,pi
circulo=pi*rad**2
END FUNCTION circulo
```

– En este ejercicio se ve cómo se realiza la llamada a una función dentro de otra función. En concreto, la función *area* realiza la llamada a la función *circulo* para completar el cálculo del área del cilindro que se requiere.

11. Cargar por teclado la temperatura T de 25 puntos del espacio con coordenadas (X, Y) en un instante de tiempo dado. Se pide:

– Visualizar la temperatura T de un punto del espacio (X, Y) solicitado por el usuario por teclado.

– Visualizar los puntos del espacio (X, Y), si los hay, que tienen el mismo valor de temperatura T, dada por el usuario por teclado.

– Calcular la mayor (en valor absoluto) de las diferencias de las temperaturas respecto de la temperatura media.

– Usar programación modular en la elaboración del programa. Unidades: (X, Y, T) = (m, m, C).

```
PROGRAM cap5_11
IMPLICIT NONE
INTEGER, PARAMETER:: d1=25,d2=3
REAL :: x,y,t,buscat,difm
REAL, DIMENSION(d1,d2):: mat
REAL, EXTERNAL:: media
```

```
CALL lectura(mat,d1,d2)
WRITE(*,*) 'DAME UN PUNTO'
READ(*,*) x,y
WRITE(*,*) 'TEMPERATURA EN ESE PUNTO ES',buscat(x,y,mat,d1,d2)
WRITE(*,*) 'DAME UNA TEMPERATURA'
READ(*,*) t
CALL puntos(mat,t,d1,d2)
WRITE(*,*)'DESV MAX RESPECTO TEMP. MEDIA', difm(mat,media,d1,d2)
! LA FUNCIÓN MEDIA ES UN ARGUMENTO ACTUAL
END PROGRAM cap5_11

SUBROUTINE lectura(mat,d1,d2)
IMPLICIT NONE
INTEGER, INTENT(IN)::d1,d2
REAL, DIMENSION(d1,d2),INTENT(OUT):: mat
INTEGER:: i,j
DO  i=1,d1
 DO  j=1,d2
  WRITE(*,*) 'DAME X,Y Y T DEL PUNTO',i
  READ(*,*) mat(i,j)
 END DO
END DO
END SUBROUTINE lectura

REAL FUNCTION buscat(x,y,mat,d1,d2)
IMPLICIT NONE
INTEGER, INTENT(IN)::d1,d2
REAL, INTENT(IN) :: x,y
REAL, DIMENSION(d1,d2),INTENT(IN) :: mat
INTEGER:: i
DO i=1,d1
 IF (mat(i,1)==x .AND. mat(i,2)==y) THEN
  buscat=mat(i,d2)
 END IF
END DO
END FUNCTION buscat

SUBROUTINE puntos(mat,t,d1,d2)
IMPLICIT NONE
INTEGER, INTENT(IN)::d1,d2
```

```
REAL, DIMENSION(d1,d2),INTENT(IN):: mat
REAL, INTENT(IN) :: t
INTEGER :: sw,i
sw=0
DO i=1,d1
 IF (mat(i,d2) == t) THEN
  sw=1
  WRITE(*,*) 'COORDENADAS DE ESA TEMPER.',mat(i,1),mat(i,2)
 END IF
END DO
IF(sw == 0) WRITE(*,*) 'NO REGISTRADA ESA TEMPER.EN EL ARRAY'
END SUBROUTINE puntos

REAL FUNCTION difm(x,f,d1,d2)
IMPLICIT NONE
INTEGER, INTENT(IN)::d1,d2
REAL, DIMENSION(d1,d2), INTENT(INOUT):: x!ENTRA AQUI PERO SALE
A F
REAL, EXTERNAL:: f
REAL :: y,dif
INTEGER:: i
y=f(x,d1,d2)
difm=ABS(x(1,d2)-y)
DO i=2,d1
 dif=ABS(x(i,d2)-y)
 IF (dif > difm) THEN
  difm=dif
 END IF
END DO
END FUNCTION difm

REAL FUNCTION media(mat,d1,d2)
IMPLICIT NONE
INTEGER, INTENT(IN)::d1,d2
REAL, DIMENSION(d1,d2),INTENT(IN) :: mat
INTEGER:: i
media=0
DO i=1,d1
 media=media+mat(i,d2)
```

```
END DO
media=media/d1
END FUNCTION media
```

– La función *media* es un argumento actual de la función *difm*. Para especificar este hecho en Fortran se usa el atributo EXTERNAL. Repasa la sección 5.7.

– La utilidad de la función *media* aumentaría si la función *difm* calculara las medias de varios conjuntos de temperaturas.

EJERCICIOS PROPUESTOS

1) Programa que pida dos números naturales y use una función lógica para saber si ambos son *cuadrones pares* o no. A saber: dos números son cuadrones pares si al sumarlos y restarlos se obtienen cuadrados perfectos. Ejemplo: 10 y 26 son cuadrones pares pues: 10+26 =36 (cuadrado perfecto) y 26-10 = 16 (cuadrado perfecto).

2) Modifica el ejercicio anterior para obtener todos los números cuadrones pares hasta 1000.

3) Programa que lee por teclado una matriz 3x3 y calcula su determinante. Utilizar la función siguiente para calcular adjuntos:

```
INTEGER FUNCTION adjto (a,b,c,d)
INTEGER, INTENT(IN):: a,b,c,d
adjto=a*d-b*c
END FUNCTION adjto
```

4) Programa que lee por teclado una matriz 3x3 y calcula su determinante. Utilizar la subrutina siguiente para calcular adjuntos:

```
SUBROUTINE adjto (a,b,c,d,det2)
INTEGER, INTENT(IN):: a,b,c,d
INTEGER, INTENT(OUT):: det2
det2=a*d-b*c
END SUBROUTINE adjto
```

5) Programa que pida 5 números por teclado y averigüe si son primos o no utilizando el algoritmo de Wilson. A saber: un número K es primo si (K-1)!+1 es divisible entre K. Utilizar una función que devuelva a través de su nombre los valores .TRUE: si el número dado es primo y .FALSE. si no lo es.

6) Programa que pida por teclado una matriz cuadrada de 4X4 y calcule su traza y la suma de los elementos por encima y por debajo de la diagonal principal. Usar una subrutina para la lectura de la matriz, una función para calcular la traza y una subrutina para las dos sumas pedidas.

7) Programa que desplace los valores de las componentes del vector A(5,10,15,20,25,30,35) una posición hacia la derecha de modo que el valor de la última componente pase a la primera, es decir, después del desplazamiento, el vector resultante es A(35,5,10,15,20,25,30). Usar una subrutina para realizar el desplazamiento a la derecha.

8) Lo mismo que en el ejercicio anterior pero desplazando los valores de las componentes del vector A(5,10,15,20,25,30,35) una posición hacia la izquierda de modo que el valor de la primera componente pase a la última, es decir, después del desplazamiento, el vector resultante es A(10,15,20,25,30,35,5). Usar una subrutina para realizar el desplazamiento a la izquierda.

9) Programa que calcule el producto de 2 matrices de 3X2 y 2X3, respectivamente. Usar una subrutina para la lectura de las dos matrices a multiplicar y otra subrutina para calcular la matriz producto. (Guíate por el programa CAP4_14).

10) Programa que pida al usuario por teclado el número de filas y columnas de dos Matrices A y B (iguales para ambas) y sus componentes. Dimensiona dinámicamente las matrices. A continuación, el programa presentará estas opciones:

– 1. Mostrar por monitor la Matriz A.

– 2. Mostrar por monitor la Matriz B.

– 3. Mostrar por monitor la traspuesta de la Matriz A.

– 4. Mostrar por monitor la traspuesta de la Matriz B.

– 5. Mostrar por monitor Matriz A + Matriz B.

– 6. Mostrar por monitor Matriz A - Matriz B.

– 7. Salir.

– Usa el mismo subprograma para responder a las opciones 1 y 2 del menú anterior, otro subprograma para responder a las opciones 3 y 4 y otro para responder a las opciones 5 y 6.

– Antes de acabar el programa libera el espacio reservado en memoria previamente para las matrices A y B.

11) Repite cualquier ejercicio resuelto de este capítulo que use arrays pero dimensionándolos dinámicamente, en tiempo de ejecución.