# Capítulo 3

# Subprogramas

Con lo explicado hasta aquí se pueden escribir programas sencillos y no demasiado largos. Pero varias razones justifican la necesidad de disponer de otro tipo de recursos.

Por una parte, puede haber, en la resolución de un problema, partes que se repitan. Por otra parte, es conveniente partir la resolución de un problema "largo" en una serie de etapas más cortas que se concatenan para resolver el problema global. Los programas demasiado largos son difíciles de revisar y de corregir.

En muchas ocasiones, determinadas etapas de la resolución de un problema son comunes a otros problemas. Por ejemplo, la resolución de un sistema lineal de ecuaciones de matriz simétrica definida positiva forma parte de la resolución de un problema de contorno para una ecuación en derivadas parciales (edp) por el método de los elementos finitos. Es obvio que no se va a escribir el programa para resolverlo cada vez que se escribe un programa para resolver una edp. Bastará escribir, de una vez por todas, un "programa" que resuelva un sistema lineal y que se pueda concatenar con otros "programas" para resolver un problema más complejo. Se podrá así, además, utilizar códigos escritos por otras personas como parte de la resolución de un problema concreto.

Esta función la cumplen los **subprogramas**. Un subprograma es una clase de programa que no se puede ejecutar por sí sólo: su ejecución tiene que ser ordenada por otro programa. A los programas que se pueden ejecutar directamente se les suele llamar **programas principales**. Un proyecto complejo estará en general formado por un programa principal que gestiona el control general del cálculo y una serie de subprogramas que llevan a cabo tareas secundarias.

En FORTRAN existen dos clases de subprogramas: **subrutinas** y **funciones**. Las funciones son un tipo de subprogramas que permiten escribir un código que produce un valor que puede ser utilizado en una expresión: como las funciones elementales explicadas en el Capítulo 1, pero escritas por nosotros. Las subrutinas son una clase de subprogramas que se utilizan para realizar cualquier secuencia de operaciones y cuya ejecución puede ser ordenada por cualquier otro programa o subprograma, mediante una orden específica.

## 3.1 Subprogramas `FUNCTION`

Este tipo de subprograma sirve, con carácter general, para definir funciones de usuario: tiene uno o varios argumentos y devuelve un valor.

La primera linea de un subprograma `FUNCTION` debe ser de la forma

```
TIPO FUNCTION nombre(lista-de-argumentos)
```

donde `TIPO` es una cláusula de declaración de tipo (`INTEGER, REAL, ...`), `nombre` es el nombre que se quiera dar a la función y `lista-de-argumentos` es una lista de variables separadas por comas. La declaración `TIPO` es opcional: puede usarse la declaración de tipo implícito por defecto.

La última línea de un subprograma `FUNCTION` debe ser

```
END FUNCTION nombre
```

Entre ambas líneas se sitúa el código-fuente de la función, utilizando las instrucciones ya explicadas.

El objetivo de una `FUNCTION` es devolver un valor. Para ello, en el " cuerpo" del subprograma se debe **asignar un valor a la variable** `nombre`, cuyo tipo es el de la función.

La forma de utilizar una función es similar a la ya vista para las funciones intrínsecas: mediante el `nombre` de la función, seguido de los valores de los argumentos entre paréntesis y separados por comas.

```
real*4 function fun(x,n)
real*4  :: x
integer :: n
if (x.le.(1./n)) then
  fun=1.-n*x
else
  fun=0.
end if
end function fun
```

Esté subprograma calcula el valor de la función:

$$f_n(x) = \begin{cases} 1 - nx & \text{si } x \leq \frac{1}{n} \\ 0 & \text{si no} \end{cases}$$

La forma de usar este subprograma en otro sería escribiendo, por ejemplo:

```
....
m=7
y=z/2.
val=sqrt(fun(y,m))
....
```

## 3.2 Subprogramas SUBROUTINE

Este tipo de subprograma, como se ha indicado antes, sirve para realizar cualquier secuencia de operaciones. Se diferencia de las funciones en que no devuelve un valor: las subrutinas no se pueden usar en una expresión. Simplemente, su ejecución es ordenada por otro programa o subprograma.

La primera linea de un subprograma `SUBROUTINE` debe ser de la forma

```
SUBROUTINE nombre(lista-de-argumentos)
```

donde `nombre` es el nombre que se quiera dar a la subrutina y `lista-de-argumentos` es una lista de variables separadas por comas. Las subrutinas no devuelven ningún valor, por eso no tienen atributo de tipo: no hay que decir de qué tipo es `nombre`. La última línea de un subprograma `SUBROUTINE` debe ser

```
END SUBROUTINE nombre
```

Entre ambas líneas se sitúa el código-fuente que lleve a cabo las acciones deseadas.

Para llamar a una subrutina desde otro programa se usa la instrucción:

```
CALL nombre(lista-de-argumentos)
```

en donde `lista-de-argumentos` son los valores concretos para los cuales queremos ejecutar la subrutina.

```
EJEMPLO:
      subroutine escribe(mensaje,a)
      character*32  :: mensaje
      real*4        :: a
      print*,"<---------------------------->"
      print*,mensaje
      print*,a
      print*,"<---------------------------->"
      end subroutine escribe
```

Así, en otro programa podemos poner:

```
EJEMPLO:
      ...
      a=1.2345
      call escribe("El ultimo valor calculado es: ",a)
      ...
```

## 3.3 La instrucción RETURN

Cuando un subprograma termina de ejecutarse, el control regresa al programa que lo llamó, cuya ejecución se reanuda exactamente en el punto en que se dejó. De igual forma que en un programa principal, la ejecución de un subprograma se termina cuando se terminan sus instrucciones. Pero también se puede terminar en cualquier otro punto, utilizando la instrucción

```
RETURN
```

Naturalmente, un subprograma puede tener varios puntos de retorno, es decir, varias instrucciones `RETURN`.

En un subprograma puede, también, aparecer la instrucción `STOP`. Su aparición provoca la inmediata detención de la ejecución del **programa global**, es decir del programa principal que provocó la cadena de llamadas que termina en el subprograma.

Esto tiene interés, por ejemplo, cuando se detecta un error grave que hace imposible o carente de sentido continuar ejecutando el programa.

## 3.4 Argumentos ficticios y argumentos actuales

A los nombres de variables que se usan como argumentos en la definición de un subprograma se les suele llamar **argumentos ficticios**. También hay quien los llama **formales**, **virtuales** o **mudos**. Por el contrario, a los argumentos que se usan cuando se llama al subprograma, se les llama **argumentos actuales**. Cuando se llama al subprograma los argumentos ficticios son "sustituidos" por los argumentos actuales, en el orden en que aparecen. Los argumentos actuales pueden ser constantes, variables o expresiones. Si son expresiones se evalúan antes de llamar a la función.

En el ejemplo de la sección **3.1**, `x` y `n` en la definición de la función `fun` son los argumentos ficticios que se usan para escribir el código que permite calcular el valor de la función en un punto, mientras que `y` y `m` son los argumentos actuales, es decir, las variables que contienen los valores para los cuales se quiere calcular, *de facto*, el valor de la función.

Los argumentos actuales tienen que coincidir, en número, tipo y orden, con los argumentos ficticios a los que van a sustituir. Por ejemplo, no se podría llamar a la función del ejemplo anterior con `fun(7,3.)`, ya que la función espera que el primer argumento actual sea un real y recibe un entero, y que el segundo sea un entero y recibe un real.

La asignación de argumentos actuales en FORTRAN es **por referencia** (también se dice por direcciones). Esto significa que, cuando un programa llama a otro, si uno de los argumentos actuales es una variable, lo que se transmite al subprograma llamado es la dirección de memoria de dicha variable, de modo que si el subprograma llamado cambia el valor de ese argumento, al volver al programa llamante, el valor de la variable estará cambiado.

Aunque las subrutinas no devuelven un valor como las funciones, sí pueden transmitir valores al programa llamante modificando los valores de los argumentos, gracias a lo comentado en el párrafo anterior. Lógicamente, es algo que debe hacerse con cuidado, ya que puede ocasionar errores difíciles de detectar.

Por ejemplo, la siguiente subrutina recibe dos números reales en entrada y los devuelve ordenados en orden creciente.

```
EJEMPLO:
      subroutine ordena(a,b)
      real*4 :: a,b,aux
      if(b.lt.a) then
        aux=a
        a=b
        b=aux
      end if
      end subroutine ordena
```

Si usamos esta subrutina en un programa en la forma siguiente

```
EJEMPLO:
      ...
      a=5.
      b=2.
      call ordena(a,b)
      ...
```

al volver al programa, una vez ejecutada la subrutina `ordena`, las variables `a` y `b` habrán cambiado de valor!

Aunque no se ha mencionado antes, también es posible (aunque nada recomendable) modificar, dentro de una función, los valores de los argumentos. Es algo que debe evitarse a toda costa, como mala técnica de programación, ya que no es el objetivo de las funciones.

### 3.4.1 El atributo `INTENT`

En una buena práctica de programación, al diseñar un subprograma, se debe prestar especial atención a determinar qué argumentos son **de entrada**, es decir, están encargados de transmitir valores desde el programa llamante al subprograma, y qué argumentos son **de salida**, es decir, están encargados de transmitir valores desde el subprograma al programa llamante. Si es necesario, puede haber argumentos **de entrada y salida**, es decir, argumentos de entrada pero cuyo valor será modificado dentro del subprograma.

Se puede, si se desea, utilizar el atributo `INTENT`, que sirve para declarar en un subprograma si un argumento es de entrada, de salida o de entrada y salida. **Esto no es obligatorio**; sólo es una ayuda al programador. Por un lado puede ayudar a comprender un programa y por otro puede hacer que el compilador detecte posibles errores.

Este atributo se puede usar junto con una declaración de tipo, o en una declaración aparte:

```
TIPO, INTENT(uso) :: argumentos
INTENT(uso) :: argumentos
```

donde `uso` debe ser una de las tres cláusulas siguientes: `IN` para declarar que los argumentos son de entrada, `OUT` para declarar que son de salida o `INOUT` para declarar que son de entrada y salida.

## 3.5 Subprogramas internos y externos

Los subprogramas en FORTRAN90 pueden también clasificarse en **internos** y **externos**.

Los primeros están 'contenidos" dentro de otro programa o subprograma, que es el único que puede utilizarlos. Su ubicación precisa es al final de todas las instrucciones del programa, justo antes de la instruccion `END PROGRAM` o `END FUNCTION` o `END SUBROUTINE` y precedidos de la instrucción `CONTAINS`. Por ejemplo, la forma de escribir subprogramas internos a un programa principal sería:

```
PROGRAM nombre
  ....
  ....
CONTAINS
SUBROUTINE primera(argumentos-de-primera)
  ...
  ...
END SUBROUTINE primera
FUNCTION segunda(argumentos-de-segunda)
  ...
  ...
END FUNCTION segunda
END PROGRAM nombre
```

Un subprograma interno no puede, a su vez, tener otros subprogramas internos.

Los subprogramas externos, en cambio, pueden ser utilizados por cualquier otro programa o subprograma. El código de un subprograma externo se puede ubicar:

- En un fichero, aparte de cualquier otro programa o subprograma.
- En el mismo fichero que otro programa o subprograma, pero "fuera" de él, es decir, después de la instrucción `END PROGRAM` o `END FUNCTION` o `END SUBROUTINE`.
- En un **módulo** de subprogramas (se verán posteriormente).

## 3.6 Ámbito de las variables

Como norma general, los nombres de las variables que aparecen en un programa o subprograma son **locales**, es decir, sólo son *visibles* "dentro" de ese programa o subprograma.

Así, si un (sub)programa `prog1` usa una variable de nombre `var` y llama a otro subprograma, `prog2`, que también usa una variable de nombre `var`, ambas variables son distintas (excepto que se hagan coincidir en la identificación argumento actual → argumento ficticio).

Esta norma general tiene dos excepciones: algunos casos en subprogramas internos y el caso de variables incluidas en un bloque `COMMON` (que se verán más adelante).

La situación entre un subprograma interno y su programa anfitrión es especial: las variables del anfitrión son compartidas con su subprograma interno, excepto las explícitamente declaradas en éste. Las declaraciones de tipo del programa anfitrión, incluso las implícitas,

también afectan al subprograma interno, a menos que sean "tapadas" con declaraciones propias. Para ilustrar esta situación, incluimos el ejemplo siguiente:

```
EJEMPLO:
      program principal
      implicit real*8 (x-z)
      real*4 :: a,b
      b=1.
      call interno(a)
      ...
!--------------------------
      contains
      subroutine interno(t)
      implicit real*4 :: (x)
      real*4 :: a,t
      t=b
      a=4.
      end subroutine interno
!--------------------------
      end program principal
```

Por un lado, en el programa principal, todas las variables que empiecen por `x`, `y` o `z` son reales doble precisión. En el subprograma lo son las que empiecen por `y` o `z`, pero las que empiecen por `x` son reales simple precisión, ya que prevalece la declaración del subprograma.

La variable `b` de `principal` y la de `interno` son la misma, es decir, el nombre `b` hace referencia a la misma dirección de memoria en ambos programas, ya que es una variable del programa principal y no está declarada en el subprograma.

Por el contrario, la variable `a` del programa principal y la del subprograma son distintas, ya que al ser `a` expresamente declarada en la subrutina `interno`, se convierte en variable local.

En la llamada a la subrutina, el argumento ficticio `t` es sustituido por el argumento actual `a`, de manera que la instrucción `t=b` hará que se almacene el valor `1.` en la variable `a` del programa principal.

La instrucción `a=4.` de la subrutina no afectará al valor de la variable `a` del programa principal.

**Recuérdese que estas circunstancias afectan sólo a los subprogramas internos.**