Notas (breves) de FORTRAN 90

Funciones, subrutinas y módulos

En lo referido a las funciones y subrutinas, siguen valiendo las mismas consideraciones y reglas que en Fortran 77. Un agregado que tiene Fortran 90 es que se puede definir si las variables son de entrada, de salida, de entrada y salida, o internas del subprograma. Obviamente, una `FUNCTION` no tendrá explícitamente declaradas variables de salida. A continuación veamos un ejemplo de función:

```
REAL FUNCTION F(X,Y)
  IMPLICIT NONE
  REAL, INTENT(IN) :: X, Y
  REAL :: Z
  Z = X / Y
  F = X + Y + Z
  RETURN
END FUNCTION F
```

Aquí, `X` e `Y` son las variables de entrada, `Z` es una variable interna auxiliar, y `F` es el valor que tomará la función y no es necesario declarar que es de salida.
Como ejemplo de subrutina, supongamos la siguiente (donde sólo mostramos la declaración de variables):

```
SUBROUTINE SUB1(N,A,B,C)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: N
  REAL, INTENT(IN) :: A
  REAL, INTENT(INOUT) :: B
  REAL, INTENT(OUT) :: C
  INTEGER :: I, K
  REAL :: V, W
  ...
END SUBROUTINE SUB1
```

Aquí, las variables `N` y `A` son de entrada. Si dentro de la subrutina se les asigna un valor, al compilar aparecerá como error. La variable `B` es de entrada y salida, con lo cual trae consigo un valor desde otro subprograma y se le puede asignar un nuevo valor dentro de la subrutina. Por otro lado, la variable `C` es sólo de salida. En este último caso, si en la subrutina no se le asigna un valor, también tendremos error al momento de compilar. Finalmente, las variables `I`, `K`, `V` y `W` son internas de la subrutina.
Fortran 90 no controla si el número de argumentos y su tipo son el mismo entre el subprograma y su invocación, salvo que se lo indique expresamente, utilizando lo que se llama interfaz. Como veremos más adelante, en el caso de usar módulos, esta interfaz no es necesaria. La interfaz se

escribe al comienzo del programa, después de la declaración de variables, y en ella se declaran todas las funciones y subrutinas que usa dicho programa, y la declaración de sus argumentos. Por ejemplo:

```
PROGRAM PPAL
  IMPLICIT NONE
  INTEGER :: G, M
  REAL :: S, R, PI

  INTERFACE
     FUNCTION ARADIANES(G,M,S,PI)
       IMPLICIT NONE
       REAL :: ARADIANES
       INTEGER, INTENT(IN) :: G, M
       REAL, INTENT(IN) :: S, PI
     END FUNCTION ARADIANES

     SUBROUTINE AGRADOS(R,G,M,S,PI)
       IMPLICIT NONE
       REAL, INTENT(IN) :: R, PI
       INTEGER, INTENT(OUT) :: G, M
       REAL, INTENT(OUT) :: S
     END SUBROUTINE AGRADOS
  END INTERFACE
  ...
```

En este ejemplo, tenemos declaradas en la interfaz una función y una subrutina. Véase que sólo aparecen en la interfaz la declaración de los argumentos, y no la de las variables internas que estos subprogramas pudieran tener.

Llamamos módulo a un conjunto de subprogramas relacionados, agrupados, y que nos permite construir una biblioteca de rutinas que podrán ser reutilizadas en otros programas. La forma general de un módulo es:

```
MODULE NOMBRE_MODULO
  IMPLICIT NONE
CONTAINS
  SUBPROGRAMA 1
  SUBPROGRAMA 2
  ...
  SUBPROGRAMA N
END MODULE NOMBRE_MODULO
```

Cada subprograma es una función o subrutina completa. El módulo puede guardarse en un archivo y puede compilarse por separado usando la opción -c.

```
> gfortran -Wall -c Nombre_Modulo.f90
```

Si compila correctamente, se generará un archivo llamado *NombreModulo.o*. Para que un programa haga uso de todos o algunos de los subprogramas contenidos en el módulo, debemos agregar en el programa principal (o en el caso de un módulo que haga uso de otro módulo) inmediatamente después de la identificación del programa (`PROGRAM`, `MODULE`), antes del `IMPLICIT NONE`, la siguiente sentencia:

```
USE NOMBRE_MODULO, ONLY: Lista de subprogramas
```

En la lista de subprogramas irán, separados por comas, los nombres de las subrutinas y funciones a utilizar. Si se omite `ONLY: Lista de Subprogramas`, todos los subprogramas del módulo estarán disponibles. Para compilar, usamos:

```
> gfortran -Wall -o Ejecutable Nombre_Modulo.o Ppal.f90
```

si el módulo ya está compilado, o

```
> gfortran -Wall -o Ejecutable Nombre_Modulo.f90 Ppal.f90
```

si tenemos el código fuente del módulo. En la compilación, los módulos siempre deben colocarse antes que el programa principal.
Algo muy útil que podemos tener en cuenta al emplear módulos es que si varias subrutinas o funciones de un módulo, usan una misma variable, por ejemplo π, en vez de definirla en cada subprograma, lo podemos hacer una única vez antes de la sentencia `CONTAINS`.
Si queremos pasar funciones como argumento, lo más fácil es construir un módulo con las funciones. De lo contrario debe usarse la interfaz (No se usan las sentencias `EXTERNAL` e `INTRINSIC`).
Si usamos interfaces, en la subrutina que recibe una función como argumento, debemos escribir una interfaz de la siguiente manera:

```
...
INTERFACE
   REAL FUNCTION F(X)
     REAL, INTENT(IN) :: X
   END FUNCTION F
END INTERFACE
...
```

donde `F(X)` es la función general que usa la subrutina. Las funciones que serán pasadas como argumento desde un programa principal, las agregamos al final del mismo antes del `END PROGRAM` de la siguiente manera:

```
  ...
CONTAINS
  REAL FUNCTION G(X)
    REAL, INTENT(IN) :: X
    G = X**2
  END FUNCTION G

  REAL FUNCTION H(X)
    REAL, INTENT(IN) :: X
    H = LOG10(X)
  END FUNCTION H
  ...
END PROGRAM NOMBRE_PROGRAMA
```

En Fortran 90, podemos construir nuestros programas para una precisión de trabajo general, y de acuerdo a nuestras necesidades, ejecutarlos con una u otra precisión sin tener que modificar todo el programa.
Por ejemplo, en Fortran 77 si teníamos un programa en simple precisión y queríamos que calcule en doble precisión, debíamos redeclarar todas las variables involucradas. En Fortran 90, agregamos en el comienzo de los módulos o de los subprogramas (principal, funciones y subrutinas que no estén en módulos), la siguiente línea:

```
  USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: WP => REAL32
```

si queremos trabajar en simple precisión, o

```
  USE, INTRINSIC :: ISO_FORTRAN_ENV, ONLY: WP => REAL64
```

si queremos trabajar en doble precisión.
Al momento de declarar las variables debemos hacerlo de la siguiente manera:

```
  REAL(WP) :: A
  COMPLEX(WP) :: Z
```

Todas las variables que sean declaradas sin `(WP)`, serán consideradas en simple precisión. Para indicarle a las variables literales que están en la precisión de trabajo, debemos agregarles `_WP`, por ejemplo:

```
  PAR = 0.0_WP
  A = 5.23569E-13_WP
  B = 2.5_WP * A
```

Entonces, si estábamos usando nuestro programa en simple precisión y lo queremos pasar a doble precisión, sólo tenemos que cambiar `REAL32` por `REAL64` en nuestros programas fuente. En el caso de que los compiladores lo permitan, podríamos cambiar a cuádruple u óctuple precisión de esta manera.