Índice

1. Programación 3
1.1. Primer programa en Fortran (prog01.f90) . . . 3
1.2. Segundo programa (más útil) . . . 4
1.3. Conjunto de caracteres permitidos (85 símbolos) . . . 4
1.4. Tipos de datos . . . 5
1.4.1. Ejemplos . . . 5
1.4.2. Algo más sobre datos de tipo `CHARACTER` . . . 6
1.5. Constantes con nombre (atributo `PARAMETER`) . . . 6
1.6. Los nombres en Fortran . . . 6
1.7. Sentencia `IMPLICIT NONE` . . . 7
1.8. Instrucciones de asignación . . . 7
1.9. Expresiones aritméticas . . . 7
1.10. Orden de precedencia para expresiones aritméticas . . . 8
1.11. Expresiones en modo simple y modo mixto . . . 9
1.12. Asignando el resultado de expresiones aritméticas . . . 9
1.13. Operadores relacionales . . . 10
1.14. Operadores lógicos . . . 11
1.14.1. Tabla de verdad del operador `.AND.` . . . 11
1.14.2. Tabla de verdad del operador `.OR.` . . . 12
1.14.3. Tabla de verdad del operador `.NOT.` . . . 12
1.14.4. Tabla de verdad del operador `.EQV.` . . . 12
1.14.5. Tabla de verdad del operador `.NEQV.` . . . 12
1.15. Orden de precedencia de operadores . . . 13
1.16. Operadores y datos de tipo `CHARACTER` . . . 13
1.16.1. Operadores relacionales aplicados a datos de tipo `CHARACTER` . . . 13
1.16.2. Operador concatenación . . . 14
1.16.3. Subcadenas . . . 14
1.17. Estructuras de decisión . . . 15
1.17.1. `IF` lógico . . . 15
1.17.2. Bloque `IF` . . . 15
1.17.3. Bloque `IF - ELSE` . . . 16
1.17.4. Bloque `IF - ELSEIF - ELSE` (`IF` generalizado) . . . 16
1.17.5. Estructura de bifurcación `SELECT CASE` . . . 19
1.18. Estructuras de repetición . . . 20
1.18.1. Bucle `DO` finito . . . 20
1.18.2. Bucle `DO` infinito . . . 21
1.18.3. Bucle `DO WHILE` . . . 22
1.18.4. Comandos `EXIT` y `CYCLE` . . . 22
1.19. Lectura/escritura de datos desde/hacia archivos . . . 23
1.19.1. Sentencia `OPEN` . . . 24
1.19.2. Sentencia `CLOSE` . . . 25
1.19.3. Sentencia `READ` (sin formato) . . . 25
1.19.4. Sentencia `WRITE` (sin formato) . . . 26

1.19.5. Sentencia `WRITE` (con formato) . . . 27
1.19.6. Descriptores de edición utilizados para SALIDA . . . 27
1.19.7. Descriptores de edición utilizados para ENTRADA . . . 28
1.20. Otros comandos para lectura/escritura de archivos . . . 28
1.20.1. Comando `BACKSPACE` . . . 28
1.20.2. Comando `REWIND` . . . 29
1.21. Arreglos . . . 29
1.21.1. Introducción . . . 29
1.21.2. Algunas definiciones . . . 30
1.21.3. Inicialización de arreglos . . . 30
1.21.4. Lazos `DO` implícitos . . . 31
1.21.5. Un error común: salirse de rango . . . 31
1.21.6. Declarando arreglos usando constantes con nombre . . . 32
1.21.7. Operaciones con arreglos como un todo . . . 32
1.21.8. Algo más sobre arreglos bidimensionales (matrices) . . . 33
1.21.9. Funciones intrínsecas para arreglos . . . 34
1.21.10.Asignación dinámica de memoria . . . 35
1.21.11.Sentencia `WHERE` . . . 36
1.21.12.Bloque `WHERE` . . . 37
1.21.13.Otras funciones intrínsecas para arreglos . . . 37
1.21.14.Bloque `FORALL` . . . 38
1.22. Procedimientos . . . 38
1.22.1. Introducción . . . 38
1.22.2. Subrutinas . . . 39
1.22.3. Pasando variables: el esquema de pasaje por referencia . . . 40
1.22.4. Funciones . . . 41
1.22.5. Pasando arreglos como argumento . . . 42
1.23. Creando interfases explícitas . . . 43
1.23.1. Módulos . . . 43
1.23.2. Bloques de interfaz . . . 45
1.24. Nota sobre la representación de números reales . . . 45

Capítulo 1

Programación

1.1. Primer programa en Fortran (prog01.f90)

```
PROGRAM saludo
CHARACTER(10)::nombre

WRITE(*,*) 'Como␣te␣llamas?'
WRITE(*,*) '(Escribe␣tu␣nombre␣y␣presiona␣Enter)'
READ(*,*) nombre
WRITE(*,*) 'Hola,␣',nombre
END PROGRAM saludo
```

Observaciones:

- hay palabras clave
- es un "programa fuente", que todavía no se puede ejecutar (no lo entiende la computadora); para que lo entienda hay que "compilarlo":

  `gfortran -o prog01 prog01.f90`

- Ejecución:

```
$> prog

Como te llamas?
(Escribe tu nombre y presiona Enter)
Juan <Enter>
Hola, Juan

$>
```

- notar cómo se da el flujo del programa (se respeta siempre, para todo programa)
- cada línea de texto: hasta 132 caracteres (los blancos cuentan), hasta 40 líneas (Fortran 90); hasta 256 líneas (Fortran 2003)

1.2. Segundo programa (más útil)

Programa para calcular el radio terrestre r a una latitud φ dada por el usuario. Supondremos para la forma de la Tierra la de un elipsoide de radio

$$r = a(1 - \alpha \sin^2 \varphi) \tag{1.1}$$

donde $a = 6378{,}39$ km es el radio ecuatorial y $\alpha = 1/297 = 0{,}003367$ el factor de achatamiento $(a - c)/a$.

```
PROGRAM elipsoide
! Calcula el radio terrestre usando la formula del elipsoide,
! para una latitud ingresada por el usuario

REAL, PARAMETER :: A=6378.38,ALPHA=0.003367
REAL :: R,FI,PI,FIR
!
! Calculo PI
!
PI = 4.*ATAN(1.)
WRITE(*,*) 'INGRESE␣LA␣LATITUD,␣EN␣GRADOS:'
READ(*,*) FI
!
! Convierto de grados a radianes
!
FIR = FI*PI/180.
!
! Calculo el radio
!
R = A*(1.-ALPHA*SIN(FIR)**2)
!
! Mostramos el resultado en pantalla
!
WRITE(*,*)'El␣radio␣a␣una␣latitud␣de␣',FI,'␣grados␣es␣de␣',R,'␣km'
END PROGRAM elipsoide
```

Se repiten algunos elementos del primer programa:

- palabras clave (READ, WRITE, PROGRAM, END PROGRAM)
- cartel pidiendo datos + asignación del valor ingresado

Aparecen elementos nuevos:

- comentarios
- declaración de variables (REAL)
- asignación de valor al momento de la declaración
- atributo PARAMETER
- funciones intrínsecas (SIN, ATAN)
- operadores aritméticos
- notar el punto que aparece en algunas constantes (1., 4.)

Ya estamos en condiciones de sistematizar el estudio del lenguaje. Comenzaremos por definir el conjunto de caracteres reconocidos por el Fortran 90.

1.3. Conjunto de caracteres permitidos (85 símbolos)

Al escribir un programa en Fortran no podemos usar cualquier carácter del teclado porque el lenguaje sólo reconoce algunos.

- a - z (26, no existen la ñ ni las letras acentuadas)

- A - Z (26)
- 0, 1,..., 9 (10)
- +, −, *, /, ** (5)
- _ (guión bajo, 1)
- . (punto), ',' (coma), ; (punto y coma), : (dos puntos), ! (exclamación), ? (interrogación) (6)
- comilla simple ('), comillas dobles (") (2)
- =, <, > (3)
- '(' (abre paréntesis), ')' (cierra) (2)
- $, &, %, ƀ(4)

Fortran no distingue entre MAYÚSCULAS y MINÚSCULAS.

Con estos caracteres se pueden armar sentencias (o instrucciones) que pueden ser de dos tipos:

$$\text{sentencias} \begin{cases} \text{ejecutables} \\ \text{no ejecutables} \end{cases} \tag{1.2}$$

Sentencias ejecutables: determinan acciones a realizar (operaciones aritméticas, lectura de datos, salida de resultado, etc.)
Sentencias no ejecutables: proveen información para que el programa se ejecute adecuadamente (por ejemplo, el formato de salida de un resultado)
Las sentencias deben respetar un orden en cada unidad de programa del "fuente":

- 1ra sección: DECLARACIÓN (conjunto de instrucciones no ejecutables al comienzo del programa que definen, por ejemplo, el nombre del programa, el nombre y tipo de las variables, etc.)
- 2da sección: EJECUCIÓN (una o más sentencias ejecutables)
- 3ra sección: FINALIZACIÓN (instrucciones que finalizan el programa, por ejemplo, STOP y END PROGRAM)

1.4. Tipos de datos

Existen cinco tipos de datos:

$$\begin{cases} \texttt{INTEGER} \\ \texttt{REAL} \\ \texttt{COMPLEX} \\ \texttt{LOGICAL} \\ \texttt{CHARACTER} \end{cases} \tag{1.3}$$

1.4.1. Ejemplos

$$\texttt{INTEGER} \begin{cases} \texttt{0} \\ \texttt{999} \\ \texttt{-10} \end{cases} \qquad \texttt{REAL} \begin{cases} \texttt{1.} \\ \texttt{2.1E-14} \\ \texttt{-5E10} \\ \texttt{1.23} \end{cases} \qquad \texttt{COMPLEX} \begin{cases} \texttt{(1.,2.3)} \\ \texttt{(-1.5,3.12E9)} \end{cases} \tag{1.4}$$

$$\texttt{LOGICAL} \begin{cases} \texttt{.TRUE.} \\ \texttt{.FALSE.} \end{cases} \qquad \texttt{CHARACTER} \begin{cases} \texttt{'~'} \\ \texttt{'Jorge'} \\ \texttt{'3.1416'} \\ \texttt{'\$50.-'} \end{cases} \tag{1.5}$$

Los datos pueden corresponder a CONSTANTES o a VARIABLES.

1.4.2. Algo más sobre datos de tipo CHARACTER

Por sus características particulares, conviene detenerse un instante más en estudiar estas variables. La forma de declararlas es la siguiente:

```
PROGRAM vars_character
CHARACTER (8) :: nombre,apellido
...
```

El número entero 8 indica la longitud máxima de la cadena de caracteres que puede guardarse en ella. Si se asigna una cadena con diferente longitud pueden pasar 3 cosas:

- sobra espacio

  `nombre = 'Jorge'` → `'Jorge␢␢␢'`

- el espacio alcanza justo

  `nombre = 'Jeronimo'` → `'Jeronimo'`

- el espacio no es suficiente para guardar toda la cadena

  `nombre = 'Juan Manuel'` → `'Juan Man'`

1.5. Constantes con nombre (atributo PARAMETER)

Las CONSTANTES pueden tener nombre, o no. Las VARIABLES siempre lo tienen.

La situación habitual para la declaración de una constante con nombre se da cuando uno necesita utilizar una constante muchas veces pero no desea tipearla una y otra vez. En estos casos, se declara utilizando el atributo PARAMETER y se la invoca por su nombre, como en el siguiente ejemplo:

```
PROGRAM cte_con_nombre
IMPLICIT NONE
INTEGER :: i
REAL :: c,d
REAL,PARAMETER :: PI=3.1415926
...
```

Noten que el lugar para hacerlo es la sección de declaración. De esta manera, cuando el compilador encuentra el nombre `PI` lo reemplazará automáticamente por el valor asignado. Si por error, uno intentara modificar a PI en la sección ejecutable del programa fuente, al intentar compilar, el compilador abortaría señalando el error.

1.6. Los nombres en Fortran

En Fortran los nombres tienen que cumplir ciertos requisitos:

- pueden contener caracteres
  - ALFABÉTICOS (a,b,...,z,A,B...,Z),
  - NUMÉRICOS (0,1,...,9) o
  - GUIÓN BAJO,
- deben comenzar con un carácter ALFABÉTICO,
- pueden tener un máximo de 31 caracteres.

Ejemplos de nombres (¿correcto o incorrecto?)

1. NOMBRE
2. DOMICILIO_PARTICULAR
3. GÉNERO
4. _MATERIA
5. EQUIPO1
6. $PRECIO
7. DIA-HORA
8. VARIABLE 1

1.7. Sentencia IMPLICIT NONE

Para utilizar una variable debemos **declararla** primero con uno de los tipos de datos existentes en Fortran. Esta operación se hace en la primera sección del programa.
Ejemplo:

```
PROGRAM ejemplo
REAL :: A,B
INTEGER :: D1,DIA,MES
COMPLEX :: W,Z
CHARACTER :: C1,MIS_DATOS
...
```

Si una variable es utilizada sin haber sido declarada, Fortran supone que si el nombre empieza con I, J, K, L, M o N, la variable es entera (INTEGER), si no, real. Observen que esta convención puede llevar a errores involuntarios. Para evitarlos, vamos a considerar OBLIGATORIO (aunque en la práctica no lo sea) incluir la instrucción no ejecutable

```
PROGRAM ejemplo
IMPLICIT NONE      <----------------
```

inmediatamente abajo de la instrucción `PROGRAM`. De esta forma, si alguna constante (o variable) que no ha sido declarada es utilizada, el compilador señalará el error.

1.8. Instrucciones de asignación

Su forma general es:

```
NOMBRE_DE_VARIABLE = EXPRESION
```

Mediante este formato, que no es una igualdad en el sentido usual, el valor calculado en `EXPRESION` es **asignado** a la variable `NOMBRE_DE_VARIABLE`. Ejemplo:

```
PROGRAM asignaciones
IMPLICIT NONE
INTEGER :: i

i = 1
i = i + 1
...
```

Como resultado de esta asignación, la variable `i` valdrá 2 luego de ejecutadas ambas sentencias. La segunda expresión se denomina "expresión aritmética", y vamos a estudiarlas a continuación.

1.9. Expresiones aritméticas

Una expresión aritmética es aquella que combina operadores aritméticos (+,-,*,/,**) con operandos numéricos (`INTEGER`, `REAL` o `COMPLEX`). Ejemplo:

```
a = c*b + d**2 + e*f
```

Acá el resultado de la expresión aritmética es asignado a la variable `a`.

En principio, esta expresión no tiene nada de extraordinario, salvo que...

> en el campo numérico, el orden en el que se realizan las operaciones, en general, afecta al resultado.

Por eso, necesitamos aprender cómo evalúa Fortran las expresiones aritméticas (qué hace primero y qué después).

1.10. Orden de precedencia para expresiones aritméticas

Los operadores aritméticos se ordenan de la siguiente manera, según su precedencia:

1. ** (de derecha a izquierda)
2. * / (de izquierda a derecha)
3. + - (de izquierda a derecha)

la aclaración entre paréntesis corresponde a "reglas de asociatividad" que se siguen para evaluar una expresión en caso de tener operadores de igual precedencia.

NOTA 1: la importancia de estas reglas se puede ver con el siguiente ejemplo. Consideremos la expresión `2**2**3`. Notar que no es lo mismo hacer

```
(2**2)**3 = 4**3 = 64
```

que

```
2**(2**3) = 2**8 = 256.
```

Para Fortran, por las reglas de asociatividad, el resultado de la expresión sin paréntesis será 256.

NOTA 2: el paréntesis tiene la máxima precedencia así que, ante la duda, usarlo (sin abusar).

> Orden que sigue el Fortran:
>
> 1. las expresiones se evalúan de IZQUIERDA a DERECHA,
> 2. si se encuentra un OPERADOR se lo compara con el siguiente
>   *a*) si el siguiente tiene MENOR precedencia → evalúa el operador previo (`3*5-4`),
>   *b*) si tiene IGUAL precedencia → aplica reglas de asociatividad (`3*8*6`),
>   *c*) si tiene MAYOR precedencia → busca el siguiente operador (`4+5*7**3*4`).

Ejemplo: identificar el orden de evaluación de la expresión `Z-(A+B/2.)+W*Y`

$$\underbrace{\underbrace{\mathtt{Z}-(\underbrace{\mathtt{A}+\underbrace{\mathtt{B/2.}}_{1}}_{2})}_{3}+\underbrace{\mathtt{W}*\mathtt{Y}}_{4}}_{5} \tag{1.6}$$

1.11. Expresiones en modo simple y modo mixto

Cuando todos los operandos son del mismo tipo (INTEGER, REAL o COMPLEX) decimos que la expresión está en modo SIMPLE.

Ejemplos:

`4+25` `(2.,5.)-(7.,5.)` `15./4E-5` (1.7)

Cuando los tipos de dato son distintos, decimos que está en modo MIXTO.

Ejemplos:

`4+25.` `(2.,5.)+21` `23/4E-5` (1.8)

Cuando Fortan encuentra una expresión en modo MIXTO, antes de realizar la operación debe transformar los operandos al mismo tipo (promoviendo al más "débil").

Ejemplo de modo mixto:

```
1 + 2.5  →  3.5
  1/2.0  →  0.5
  2.0/8  →  0.25
```

La excepción a la regla de promover al tipo de dato más débil es el caso de la exponenciación:

`A**N`, (1.9)

si `A` es `REAL` y `N` es `INTEGER` no se da la conversión de `N` a `REAL`, sino que la expresión se interpreta de la manera usual como `A` multiplicado por sí mismo `N` veces. Así, la expresión `3.5**3` es interpretada por Fortran como `3.5*3.5*3.5`.

No olvidar las reglas de precedencia

`-4.0**2` → `-16.0` (1.10)

Notas:

- Los operadores + y - pueden actuar sobre 2 operandos (operador binario) o sobre uno (operador "unario"). Ejemplos de operador actuando sobre un sólo operando: `-18`, `+38.`, `-1.34E-5`.
- Nunca deben colocarse 2 operadores juntos (++,+-,***,*/, etc.). Separarlos con un espacio en blanco no subsana el error.
- En caso de producto, no olvidarse de colocar el *. Fortran no entiende el producto implícito cuando se prescinde del operador.

1.12. Asignando el resultado de expresiones aritméticas

La asignación del resultado de una expresión aritmética tiene esta forma

`variable_con_nombre = expresión_aritmética` (1.11)

Fortran realiza primero los cálculos correspondientes a la expresión aritmética y luego le asigna el resultado a la variable. Si el tipo de dato que da por resultado la expresión no es igual al de la variable, Fortran lo convertirá al tipo de la variable. Ojo con esto, si la variable es `INTEGER` y la expresión da por resultado un `REAL`, habrá un truncamiento. Si la situación es la inversa, Fortran promoverá a `REAL` el resultado de la expresión para luego asignarlo a la variable.

Cualquier valor previo de la variable es sobreescrito (y no podrá recuperarse).

> En Fortran, el signo igual (=) es el operador asignación y no tiene el mismo significado que el que se utiliza habitualmente en matemática.

Para fijar esto, lo mejor es verlo con ejemplos. Supongamos que tenemos una variable `A` a la que le vamos a asignar el resultado de una expresión aritmética en modo simple

$$\texttt{A = 2/3 + 1} \tag{1.12}$$

Lo primero es calcular cuánto da la expresión aritmética. La división (`2/3`) da cero, por lo que el resultado de la expresión es el número entero `1`. Así, en `A` se guardará:

$$\texttt{A} = \begin{cases} \texttt{1} & \text{si } \texttt{A} \text{ es } \texttt{INTEGER} \\ \texttt{1.0} & \text{si } \texttt{A} \text{ es } \texttt{REAL} \end{cases} \tag{1.13}$$

Si la idea es no perder los decimales, la expresión debería modificarse a

$$\texttt{A = 2./3. + 1.} \tag{1.14}$$

Nuevamente habrá que tener en cuenta el tipo de dato de `A`, porque de eso dependerá si finalmente se retienen o no los decimales:

$$\texttt{A} = \begin{cases} \texttt{1} & \text{si } \texttt{A} \text{ es } \texttt{INTEGER} \\ \texttt{1.6666667} & \text{si } \texttt{A} \text{ es } \texttt{REAL} \end{cases} \tag{1.15}$$

Como vimos antes, la expresión aritmética puede estar en modo mixto

$$\texttt{A = 3./2 + 1} \tag{1.16}$$

lo que hará que se promuevan los `INTEGER` a `REAL` antes de hacer la cuenta. Naturalmente el resultado de la expresión (`2.5`) será del tipo de dato más “fuerte” (`REAL`, en nuestro caso). Dependiendo de si `A` es `INTEGER` o `REAL` se podrán (o no) guardar los decimales del resultado.

Los operadores binarios descriptos también pueden usarse en forma natural entre variables complejas:

$$\texttt{Z1 = (1.,3.)} \tag{1.17}$$

$$\texttt{Z2 = (4.,5.)} \tag{1.18}$$

$$\texttt{Z3 = Z1 + Z2} \longrightarrow \texttt{(1.+4.,3.+5.)} \longrightarrow \texttt{(5.,8.)} \tag{1.19}$$

1.13. Operadores relacionales

Son operadores binarios que actúan sobre números reales y enteros. Como estos números siguen un orden, tiene sentido probar si uno es mayor, menor o igual a otro. Los operadores son:

$$== \tag{1.20}$$

$$/= \tag{1.21}$$

$$< \tag{1.22}$$

$$<= \tag{1.23}$$

$$> \tag{1.24}$$

$$>= \tag{1.25}$$

Con estos operadores se pueden formar EXPRESIONES LÓGICAS como las siguientes:

- `A == B`
- `A <= B`
- `A + B**2 < B`

donde `A` y `B` deben haber sido declaradas antes como `INTEGER` o `REAL` y, además, contener algún valor (haber sido inicializadas). Como resultado de una expresión lógica sólo se pueden obtener `.TRUE.` o `.FALSE.`. Este resultado puede ser asignado, pero sólo a una variable lógica:

```
PROGRAM op_relacionales
IMPLICIT NONE
LOGICAL :: C
INTEGER :: A,B

A = 1
B = 2
C = A+B**2 < B
...
```

1.14. Operadores lógicos

Vimos que una variable lógica sólo puede tomar uno de dos valores: `.TRUE.` o `.FALSE.`. ¿Es posible relacionarlas mediante operadores? La respuesta es sí, mediante operadores lógicos. En Fortran existen 5 operadores lógicos:

.AND.
.OR.
.NOT.
.EQV.
.NEQV.

Veamos un ejemplo. Supongamos que nos proponemos detectar, en un listado de notas, aquéllas que están entre 4 y 7 (inclusive):

```
PROGRAM notas
IMPLICIT NONE
INTEGER :: nota
LOGICAL :: cuatro_a_siete

(lectura de NOTA de una base de datos)

cuatro_a_siete = nota >= 4.AND.nota <= 7
...
END PROGRAM notas
```

1.14.1. Tabla de verdad del operador .AND.

Si `A` y `B` son constantes o variables lógicas, tenemos:

A	B	A.AND.B
.FALSE.	.FALSE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.
.TRUE.	.TRUE.	.TRUE.

Ejemplo. Sean dos variables `REAL`, `X = 1.` y `Y = 3.`, determine el resultado de la siguiente expresión:

```
X + Y <Y .AND. Y**2 >X/Y
```

Resultado: `.FALSE.`

1.14.2. Tabla de verdad del operador `.OR.`

A	B	A.OR.B
.FALSE.	.FALSE.	.FALSE.
.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.TRUE.
.TRUE.	.TRUE.	.TRUE.

Ejemplo. Con los mismos `X`, `Y` de antes, ¿qué da la siguiente expresión?

```
X + Y <Y .OR. Y**2 >X/Y
```

Resultado: `.TRUE.`

1.14.3. Tabla de verdad del operador `.NOT.`

A	.NOT.A
.FALSE.	.TRUE.
.TRUE.	.FALSE.

Ejemplo. `I=7`, `J=11`, `X=21.`, `Y=6.` (`I,J` son `INTEGER` y `X,Y REAL`). Evalúen la siguiente expresión:

```
.NOT.(I+J == 2*I+4. AND. NINT(X - Y)/= 2*I+1)
```

Resultado: `.TRUE.`

Los operadores `.EQV.` y `.NEQV.` son análogos a los operadores `==` y `/=` pero los primeros se utilizan para comparar expresiones lógicas y variables lógicas, mientras que los otros dos operan entre variables reales y enteras.

1.14.4. Tabla de verdad del operador `.EQV.`

A	B	A.EQV.B
.FALSE.	.FALSE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.
.TRUE.	.TRUE.	.TRUE.

Ejemplo. Si `X=1.`, `Y=2.`, `I=3`, `J=5`, ¿qué da por resultado la siguiente expresión?

```
X >Y .EQV. J <I
```

Respuesta: `.TRUE.`

1.14.5. Tabla de verdad del operador `.NEQV.`

A	B	A.EQV.B
.FALSE.	.FALSE.	.FALSE.
.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.TRUE.
.TRUE.	.TRUE.	.FALSE.

Ejemplo. Si `X=1.`, `Y=2.`, `I=3`, `J=5`, ¿qué da por resultado la siguiente expresión?

```
X+2. >Y .NEQV. J <I
```

Respuesta: `.TRUE.`

1.15. Orden de precedencia de operadores

De mayor a menor precedencia:

```
         **
        *, /
        +, -
==, /=, <, <=, >, >=
       .NOT.
       .AND.
        .OR.
   .EQV., .NEQV.
```

Ejemplos. Suponiendo las variables reales `X = 3.`, `Y = 4.`, `Z = 2.` y a `FLAG = .FALSE.` del tipo `LOGICAL`, corrobore los siguientes resultados (OJO, uno está mal).

`.NOT. FLAG` → `.TRUE.`

`.NOT. FLAG .OR. X+Z >= X-Z` → `.FALSE.`

`X>Z .AND. Y>Z` → `.TRUE.`

`X + Y/Z <= 3.5` → `.FALSE.`

Si tienen dudas, siempre se pueden poner paréntesis para asegurarse el orden de precedencia.

1.16. Operadores y datos de tipo `CHARACTER`

1.16.1. Operadores relacionales aplicados a datos de tipo `CHARACTER`

Los datos de tipo `CHARACTER` se pueden relacionar mediante operadores relacionales. Así, podemos comparar dos cadenas, por ejemplo, `A='PATO'` y `B='JIRAFA'` mediante:

`A >B` → `.TRUE.`

A pesar de que la cadena `'PATO'` es más corta que `'JIRAFA'`, Fortran compara caracteres de izquierda a derecha para establecer el orden de prelación, usando las siguientes reglas:

1. el carácter blanco precede a todos los caracteres correspondientes a números y letras,
2. las letras siguen su orden alfabético,
3. los números, interpretados como caracteres, siguen la secuencia numérica, es decir, el '0' precede al '1', el '1' al '2', etc.,
4. no se pueden secuenciar de manera mixta números y letras.

Ejemplos.

`'AIRE' <'AIRES'` → `.TRUE.`

`'PEREZ, ADRIAN' <'PEREZ, JULIETA'` → `.TRUE.`

`'9AQ' <'7BZ'` → `.FALSE.`

NOTAS IMPORTANTES

- No existe una regla para ordenar los símbolos ('+', '-',...).
- Tampoco hay una regla que establezca si los caracteres correspondientes a números van antes o después de las letras.
- Como no hay reglas establecidas, este ordenamiento puede variar con el compilador que elijamos, lo que puede provocar resultados diferentes en diferentes máquinas.

> MORALEJA: No comparar nunca caracteres que provengan de distintos conjuntos, por ejemplo, letras con números, símbolos con letras, etc.

1.16.2. Operador concatenación

Fortran solo tiene un operador exclusivo para cadenas de caracteres, el operador concatenación (//). Es un operador binario que funciona de la siguiente forma.

Dadas 2 cadenas, `C1` y `C2`, de longitud m y n, respectivamente, la concatenacion de `C1` y `C2`, escrita `C1 // C2` , da por resultado una cadena que contiene todos los caracteres de la cadena `C1` seguida por todos los caracteres de `C2`. Así, la longitud de la cadena resultante será $m+n$.
Ejemplo:

```fortran
PROGRAM concatenacion
CHARACTER (4) :: J='JUAN',L='LEO'
CHARACTER (6) :: V='VEGA',R='ROLDAN'
CHARACTER (10) :: CAD1,CAD2,CAD3,CAD4

CAD1 = J//V
CAD2 = L//R
CAD3 = R//L
CAD4 = V//L
...
END PROGRAM concatenacion
```

Cadenas resultantes:

$$\texttt{CAD1} \rightarrow \texttt{'JUANVEGA␢␢'}$$

$$\texttt{CAD2} \rightarrow \texttt{'LEO␢ROLDAN'}$$

$$\texttt{CAD3} \rightarrow \texttt{'ROLDANLEO␢'}$$

$$\texttt{CAD4} \rightarrow \texttt{'VEGA␢␢LEO␢'}$$

1.16.3. Subcadenas

Una subcadena esta formada por caracteres contiguos de una cadena. Si `A` es una variable de tipo CHARACTER de longitud `LONG`,

`A(L1:L2)`

es una cadena de caracteres que comienza en la `L1`-ésima posición (inclusive) de `A` y termina en la `L2`-ésima posición (inclusive). Debe cumplirse que

$$1 \leq \texttt{L1} \leq \texttt{L2} \leq \texttt{LONG}. \tag{1.26}$$

Si se omite `L1`, se lo considera igual a 1. Si se omite `L2`, se lo considera igual a `LONG`. La longitud de una subcadena es

$$\texttt{L1 - L2 + 1.} \tag{1.27}$$

Ejemplo.

```fortran
PROGRAM subcadenas
CHARACTER (11) :: A
CHARACTER (6) :: B

A = 'CONTRAGOLPE'
B = A(1:6)          ! B = 'CONTRA'
A = A(7:)           ! A = 'GOLPE      '
...
END PROGRAM subcadenas
```

1.17. Estructuras de decisión

La línea de flujo de un programa Fortran es, hasta ahora, unidireccional. Programar de esta manera deja afuera a múltiples problemas en donde deben tomarse decisiones y evaluar distintas opciones de acuerdo a las mismas. La primera estructura que vamos a ver es el `IF` lógico.

1.17.1. `IF` lógico

Ejemplo:

```
IF (nota >= 7) WRITE(*,*) 'Aprobado'
```

Forma de la sentencia:

```
IF (condicion logica) sentencia_ejecutable_unica
```

Funcionamiento:

1. La condición lógica es evaluada.
2. Si es verdadera, ejecuta la sentencia. Si es falsa, sigue adelante sin ejecutarla.
3. La condición lógica puede ser también una variable lógica.

Si se quiere ejecutar más de una proposición ejecutable, hay que usar el bloque IF.

1.17.2. Bloque `IF`

Ejemplo.

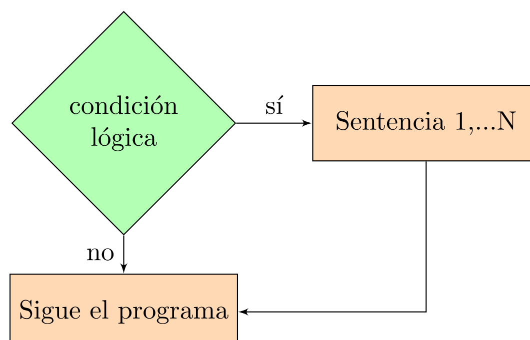
```
...
IF (discriminante >= 0.) THEN
   d = SQRT(discriminante)
   x1 = (-b+d)/(2.*a)
   x2 = (-b-d)/(2.*a)
ENDIF
...
```

(Completar el programa, y agregar que imprima en pantalla el resultado `x1` y `x2`.)

Forma de la sentencia:

```
[nombre:] IF (condicion logica) THEN
   sentencia 1
   sentencia 2
   ...
   sentencia N
ENDIF [nombre]
```

El funcionamiento es idéntico al del IF lógico, sólo que ahora, si la condición lógica es cierta, se ejecutarán las sentencias 1, 2,..., N (en orden). Una vez ejecutada la última sentencia, el flujo del programa seguirá con las instrucciones que estén por debajo del bloque. Si la condición lógica da por resultado `.FALSE.`, el flujo del programa saltará el bloque sin ejecutar ninguna de las sentencias del mismo.

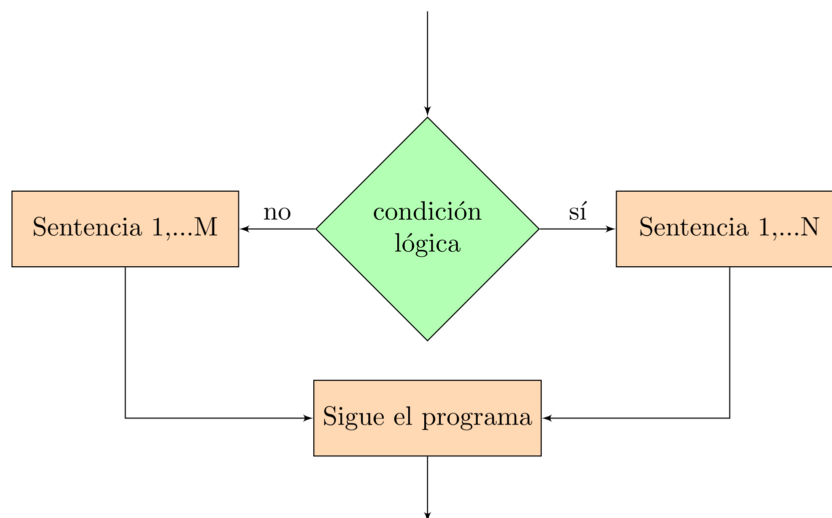


1.17.3. Bloque `IF - ELSE`

En general, no alcanza con sólo tener la posibilidad de ejecutar ciertas tareas si se cumple una condición. Muchas veces es necesario realizar otras si la condición no se cumple. Para esto se utiliza el bloque `IF - ELSE`, cuya forma general es la siguiente:

```
[nombre:] IF (condicion logica) THEN
   sentencia 1
   sentencia 2
   ...
   sentencia N
ELSE
   sentencia 1
   sentencia 2
   ...
   sentencia M
ENDIF [nombre]
```

Diagrama de flujo:



Ejemplo

```
...
IF (nota >= 4) THEN
   WRITE(*,*)'Aprobado'
ELSE
   WRITE(*,*)'Desaprobado'
ENDIF
...
```

1.17.4. Bloque `IF - ELSEIF - ELSE` (`IF` generalizado)

La versión más general del bloque `IF` que existe es la siguiente:

```
[nombre:] IF (condicion logica 1) THEN
   sentencia 1
   ...
   sentencia N
ELSEIF (condicion logica 2) THEN
   sentencia 1
   ...
```

```
    sentencia J
ELSEIF (condicion logica 3) THEN
    sentencia 1
    ...
    sentencia K
ELSE
    sentencia 1
    ...
    sentencia M
ENDIF [nombre]
```

El funcionamiento del bloque es el siguiente. Primero, se evalúa la condición 1. Si es cierta (`.TRUE.`), se ejecutan las sentencias que están a continuación (grupo de sentencias 1,...,N) y sale del bloque. Si no es cierta, analiza la siguiente condición lógica y actúa análogamente a como lo hizo antes. Si ninguna condición es cierta, ejecuta las sentencias que siguen a `ELSE`. Si no se desea ejecutar ninguna tarea si ninguna condición se verifica, entonces se puede omitir la sentencia `ELSE`.

Es posible que las opciones no sean mutuamente excluyentes. En este caso el orden en el que están escritas es importante, ya que sólo las tareas asociadas a la primera de las condiciones lógicas verdaderas son ejecutadas. A continuación mostramos un ejemplo de este tipo de programación.

```
...
IF (nota == 10) THEN
    WRITE(*,*)'Sobresaliente'
ELSEIF (nota >= 8) THEN
    WRITE(*,*)'Distinguido'
ELSEIF (nota >= 6) THEN
    WRITE(*,*)'Bueno'
ELSEIF (nota >= 4) THEN
    WRITE(*,*)'Suficiente'
ELSE
    WRITE(*,*)'Insuficiente'
ENDIF
...
```

Problema. Escriba el bloque `IF` generalizado que permita calcular cuánto vale $y(x)$, si la función está definida a trozos:

$$y = \begin{cases} x-2 & x < -5 \\ x+3 & -5 \le x \le 5 \\ 4x+1 & 5 \le x \le 10 \\ x^2 & x \ge 10 \end{cases} \tag{1.28}$$

Considere aplicar condiciones que no sean mutuamente excluyentes.

Los bloques `IF` pueden "anidarse" (colocar un bloque adentro de otro):

```
...
IF (discriminante >= 0.) THEN
    IF(discriminante > 0.) THEN
        d = SQRT(discriminante)
        x1 = (-b+d)/(2.*a)
        x2 = (-b-d)/(2.*a)
    ELSE
        x1 = -b/(2.*a)
        x2 = x1
    ENDIF
ELSE
    WRITE(*,*)'El discriminante es negativo'
    STOP
```

```
ENDIF
...
```

Recordar siempre que el bloque más interno deberá cerrarse adecuadamente dentro de la opción elegida.

Cuando uno tiene varios bloques `IF`, conviene utilizar la opción de ponerles un nombre a cada uno:

```
...
if1: IF ( x <= 1.) THEN
     ...
     ...
     if2: IF( a+1 == 3. ) THEN
          ...
          ...
     ENDIF if2
     ...
     ...
ENDIF if1
...
```

1.17.5. Estructura de bifurcación SELECT CASE

Esta estructura permite seleccionar un bloque de código en base al valor de una única expresión de tipo INTEGER, CHARACTER o LOGICAL. La forma general es la siguiente:

```
...
[nombre:] SELECT CASE ( expresion con valor definido)
CASE (selector 1) [nombre]
   instruccion 1
   ...
   instruccion N
CASE (selector 2) [nombre]
   instruccion 1
   ...
   instruccion M
...
CASE DEFAULT  [nombre]
   instruccion 1
   ...
   instruccion R
END SELECT [nombre]
...
```

Cada selector puede hacer referencia a:

- un valor,
- varios valores (separados por comas),
- rangos de valores (por ej., (:v2),(v1:),(v1:v2)),
- valores y rangos (separados por comas).

El conjunto de instrucciones ejecutadas será aquel donde el valor de la expresión caiga dentro del rango de valores establecidos por el selector. Si esto no sucede en ningún caso, se ejecutarán las instrucciones dentro del bloque CASE DEFAULT, si este ha sido incluido en la estructura (es opcional). Una vez que un bloque sea ejecutado, el control del programa saltará a la instrucción siguiente a la estructura SELECT CASE de modo que sólo uno o ninguno de los bloques será ejecutado.

Los valores de los selectores deben ser mutuamente excluyentes.

Ejemplo

```
PROGRAM nota
IMPLICIT NONE
INTEGER :: nota
...
nota = 8
...
SELECT CASE ( nota)
CASE (0:3)
   WRITE(*,*)'Insuficiente'
CASE (4,5)
   WRITE(*,*)'Suficiente'
CASE (6,7)
   WRITE(*,*)'Bueno'
CASE (8,9)
   WRITE(*,*)'Distinguido'
CASE (10)
   WRITE(*,*)'Sobresaliente'
END SELECT
...
END PROGRAM nota
```

Notar que no existe una parte con `CASE DEFAULT`. Esto implica que si, por error, una nota estuviera fuera del rango natural, el programa no lo detectaría. Para evitar que esto suceda, conviene agregar la opción `CASE DEFAULT` para manejar dicha eventualidad. Es de buena práctica de programación incluir en el programa instrucciones que permitan detectar este tipo de errores.

1.18. Estructuras de repetición

1.18.1. Bucle `DO` finito

Supongamos que queremos calcular el valor de la serie

$$\sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + ... \tag{1.29}$$

(¿cuál es el resultado teórico?)

Con lo que sabemos de Fortran hasta ahora, el programa tendría la forma:

```
PROGRAM serie
IMPLICIT NONE
REAL :: suma

suma =  0.5
suma = suma + 0.25
suma = suma + 0.125
... (infinitas lineas)
END PROGRAM serie
```

Claramente este tipo de tarea podría beneficiarse de una estructura de repetición, aunque sería imposible hacerlo un número infinito de veces. En Fortran existen varias estructuras de repetición, una de las cuales es el lazo `DO`:

```
PROGRAM serie
IMPLICIT NONE
INTEGER i
REAL :: suma=0.

DO i=1,20          ! notar que solo sumamos 20 terminos (por que?)
   suma = suma + 1./2.**i
ENDDO
...
END PROGRAM serie
```

Así evitamos tener que repetir la sentencia de la suma 20 veces. El incremento en la variable `i` toma el valor por omisión, que es 1. Si queremos que el incremento sea distinto de uno, tenemos que usar la forma más general:

```
suma_pares = 0
DO i=2,20,2
   suma_pares = suma_pares + 1./2.**i
ENDDO
```

donde se agregó un parámetro al lazo. Esta estructura permite escribir de manera compacta una parte del programa que hay que repetir *un número finito de veces*.

En resumen, la forma más general de un lazo `DO` es

```
[nombre:] DO v = vi,vf,incremento
   ...
   ...
   (conjunto de sentencias ejecutables)
   ...
   ...
ENDDO [nombre]
```

Cuando el programa llega a una sentencia `DO` le asigna a la variable del lazo `v` el valor inicial `vi` y controla si `v`> `vf` (si el incremento es positivo) o `v`< `vf` (si el incremento es negativo). Si ocurre alguna de estas dos situaciones, el lazo no se ejecuta. En caso contrario, el ciclo se repite hasta que `v` verifique alguna de las dos condiciones. En cada repetición, `v` se modificará pasando a ser `v + incremento`. El incremento puede ser negativo.

Observaciones:

- al finalizar el lazo, la variable `v` puede o no retener el último valor (depende del compilador),
- `vi`, `vf` y el incremento pueden ser *expresiones aritméticas* que son evaluadas *sólo una vez*, cuando entra a la estructura `DO`,
- dentro del lazo se podrá modificar los valores que se usaron para calcular `vi`, `vf` y el incremento, sin que esto implique que se modifiquen estos parámetros,
- está prohibido modificar la variable `v` adentro del lazo,
- está prohibido que el incremento sea nulo,
- aunque es legal utilizar expresiones de tipo real para los parámetros del lazo, su uso se tornó obsoleto en Fortran 90, y prohibido en Fortran 95;
- es perfectamente posible que un lazo `DO` no se ejecute nunca,
- el número de iteraciones que realizará un bucle será

$$\frac{\texttt{vf} - \texttt{vi} + \texttt{incremento}}{\texttt{incremento}}. \tag{1.30}$$

1.18.2. Bucle `DO` infinito

A veces se necesita que la estructura de repetición se ejecute infinitamente hasta que se cumpla cierta condición (lógica). En estos casos se puede usar el lazo `DO` infinito:

```
[nombre:] DO
   sentencia 1
   ...
   sentencia N
   IF (condicion logica) EXIT [nombre]
   sentencia N+1
   ...
   sentencia M
ENDDO [nombre]
```

Todas las sentencias dentro de lazo `DO` y por arriba de la condición lógica se ejecutan al menos una vez. Si la condición es verdadera, se ejecuta el comando `EXIT` y se transfiere el control a la línea de programa por debajo del `ENDDO`. Si la expresión lógica es falsa, se siguen ejecutando todas las sentencias hasta el final y, luego, vuelve al comienzo del bucle.

Observaciones

- Si bien es posible tener más de una sentencia `EXIT`, no es conveniente.
- Si la expresión lógica nunca es verdadera, el bucle se ejecuta indefinidamente.
- La posición del `IF` lógico puede ser cualquiera dentro del bloque. Si se coloca al principio, antes de cualquier otra sentencia, y es evaluada como verdadera, ninguna instrucción dentro del bucle es ejecutada. Si se coloca al final, todas las instrucciones del bucle se ejecutarán al menos una vez.

1.18.3. Bucle DO WHILE

Existe otra estructura de repetición que es la siguiente:

```
[nombre:] DO WHILE (condicion logica)
   sentencia 1
   ...
   sentencia N
ENDDO [nombre]
```

Este bucle se repetirá mientras la condición lógica sea verdadera. Si no lo es, el flujo pasará a la sentencia por debajo de `ENDDO`.

Naturalmente, para evitar bucles infinitos, está claro que entre iteraciones debe suceder algo que modifique los valores de la condición lógica.

Este bloque puede considerarse un caso particular del (`DO` infinito) cuando en este último la condición lógica está al principio. ¿Qué diferencia hay con respecto a la condición lógica en ambos casos?

Ejemplo: Sea

$$y = y_0 - \frac{1}{2}gt^2 \tag{1.31}$$

la posición de un cuerpo en caída libre, soltado desde una altura $y_0 = 10$ m. Calcular la posición desde $t = 0$ hasta antes de impactar con el suelo, a intervalos de 0.1 s.

```
PROGRAM caida
IMPLICIT NONE
REAL :: t=0., dt=0.1, y0=10., y
REAL,PARAMETER :: g=9.8

y = y0
DO WHILE (y > 0.)
   WRITE(*,*)t,y
   t=t+dt
   y = y0 - 0.5*g*t**2
ENDDO
END PROGRAM caida
```

Noten el lugar donde coloqué la instrucción `WRITE`. ¿Qué podría haber pasado si la hubiese puesto dentro del bucle `DO WHILE` pero debajo de la sentecia que calcula el nuevo y?

1.18.4. Comandos EXIT y CYCLE

Ya vimos que ejecutando dentro de un bucle el comando `EXIT` el control del programa sale del lazo. Si en vez de `EXIT` utilizamos `CYCLE`, lo que sucede es que las instrucciones por debajo de él no son ejecutadas, el índice del bucle es incrementado y todas las instrucciones dentro del bucle, a partir de la primera línea, son ejecutadas.

Ejemplo:

```
PROGRAM ejemplo_cycle
IMPLICIT NONE
INTEGER :: i

DO i=1,5
   IF (i==4) CYCLE
   WRITE(*,*) i
ENDDO
END PROGRAM ejemplo_cycle
```

Salida:

```
1
2
3
5
```

En el caso de tener bucles anidados, conviene asignar un nombre a cada uno de ellos. Veamos primero el comportamiento del programa si no asignamos nombres.

Ejemplo 1 (usando bucles sin nombre):

```
PROGRAM sin_nombre
IMPLICIT NONE
INTEGER :: i,j

DO i=1,3
   DO j=1,3
      IF (j==2) CYCLE
      WRITE(*,*) i,'*',j,'=',i*j
   ENDDO
ENDDO
END PROGRAM sin_nombre
```

Salida:

```
1 * 1 = 1
1 * 3 = 3
2 * 1 = 2
2 * 3 = 6
3 * 1 = 3
3 * 3 = 9
```

Ejemplo 2 (usando bucles con nombre):

```
PROGRAM con_nombre
IMPLICIT NONE
INTEGER :: i,j

externo: DO i=1,3
   interno: DO j=1,3
      IF (j==2) CYCLE externo
      WRITE(*,*) i,'*',j,'=',i*j
   ENDDO interno
ENDDO externo
END PROGRAM con_nombre
```

Salida:

```
1 * 1 = 1
2 * 1 = 2
3 * 1 = 3
```

1.19. Lectura/escritura de datos desde/hacia archivos

En los programas vistos hasta el momento hemos utilizado

```
WRITE(*,*)
```

y

```
READ(*,*)
```

para "escribir" en pantalla (WRITE) y para "leer" de lo ingresado por teclado (READ). Limitar la escritura y la lectura de datos a estos dispositivos puede ser muy poco práctico, sobre todo en el caso de tratarse de muchos datos. En este caso, es mejor que la lectura/escritura de datos se produzca de uno o varios archivos.

Ejemplo:

```
PROGRAM elipsoide
IMPLICIT NONE

REAL, PARAMETER :: A=6378.38,ALPHA=0.003367
REAL :: R,FI,PI,FIR
!
! Calculo PI
!
PI = 4.*ATAN(1.)
!
! Abro archivo de datos
!
OPEN(UNIT=10,FILE='elipsoide.datos',STATUS='OLD')
READ(10,*) FI
WRITE(*,*)'El valor leido de FI es ',FI,' grados'
CLOSE(10)
!
! Convierto de grados a radianes
!
FIR = FI*PI/180.
!
! Calculo el radio
!
R = A*(1.-ALPHA*SIN(FIR)**2)
!
! Escribo el resultado en un archivo y en pantalla
!
OPEN(11,FILE='elipsoide.resultado',STATUS='NEW')
WRITE(11,*)'El radio a una latitud de ',FI,' grados es de ',R,' km'
WRITE(*,*)'El radio a una latitud de ',FI,' grados es de ',R,' km'
CLOSE(11)
END PROGRAM elipsoide
```

Observaciones

- Notar que para acceder a un archivo, primero hay que abrirlo con la sentencia OPEN.
- En el acto de abrir un archivo, se le asocia un número entero positivo (UNIT) que se usará para referir las operaciones de entrada/salida desde/hacia el mismo (observen cómo se usa el número asociado en los comandos READ y WRITE).
- Las unidades asociadas a archivos deben cerrarse mediante el comando CLOSE.

1.19.1. Sentencia OPEN

Sintaxis:

```
OPEN(especificaciones)
```

Existe una gran variedad de especificaciones posibles, acá nosotros sólo vamos a ver las más comunes.

Ejemplos:

```
OPEN(UNIT=10,FILE='elipsoide.datos')
```

```
OPEN(10,FILE='elipsoide.datos',STATUS='NEW')
```

```
OPEN(10,FILE='elipsoide.datos',FORM='UNFORMATTED')
```

A continuación se describen algunas especificaciones, comenzando por las más habituales:

- `UNIT`: debe ser un valor entero positivo o una expresión entera que dé por resultado un número entero positivo (usualmente entre 1 y 99). Además, debe ser la primera especificación de la sentencia `OPEN`. Se puede obviar el encabezado “`UNIT=`”. Es la única especificación que debe estar obligatoriamente siempre.
- `FILE`: expresión de tipo `CHARACTER`. Puede tratarse de una cadena constante (`'datos_de_temperatura'`), una variable de este tipo con valor previamente asignado o una cadena armada “pegando” (concatenando) cadenas.
- `STATUS`: indica el estado del archivo a ser abierto. El valor asignado debe ser una de las siguientes cadenas de caracteres: `'OLD'`, `'NEW'`, `'REPLACE'`, `'SCRATCH'`, `'UNKNOWN'` (valor por omisión). `'OLD'` indica que el archivo debe existir con anterioridad (si no existe, se produce un error en tiempo de ejecución); `'NEW'` que el archivo no debe existir y que, por lo tanto, debe ser creado (si existe con anterioridad, se produce un error en tiempo de ejecución); `'REPLACE'` indica que si el archivo existe, será reemplazado (y sus datos eliminados), si el archivo no existe, será creado; `'SCRATCH'` indica que el archivo es temporario (el sistema lo eliminará al finalizar el programa), no tiene nombre (esta especificación es incompatible con `'FILE'`).
- `IOSTAT`: espera un nombre de variable de tipo entera. Esa variable recibirá como resultado de la ejecución de la sentencia `OPEN`, un valor igual a cero si todo salió bien, o distinto de cero si se produjo algún error. La inclusión de esta especificación permite que el programa no aborte si se produce un error.
- `FORM`: puede asignársele uno de dos valores: `'FORMATTED'` (valor por omisión) o `'UNFORMATTED'`. En el caso con formato, el archivo es humanamente legible mientras que si no tiene formato, el archivo guarda la información en binario.

Existen otras especificaciones posibles, pero acá nos limitaremos a describir éstas, que son las más comunes. Si tienen curiosidad por conocer las demás, pueden encontrarlas en cualquier libro de Fortran (ver bibliografía de la materia).

1.19.2. Sentencia `CLOSE`

Sintaxis:

```
CLOSE(nro_de_unidad_abierta)
```

Esta sentencia se utiliza para cerrar un archivo y liberar la unidad. Ésta puede volver a ser utilizada para abrir el mismo u otro archivo. `CLOSE` tiene opciones pero no las vamos a ver.

1.19.3. Sentencia `READ` (sin formato)

Sintaxis (básica):

```
READ(nro_de_unidad_abierta,*) variables_de_entrada
```

Sintaxis (más completa:)

```
READ(nro_de_unidad_abierta,*,IOSTAT=variable_entera) vars_de_entrada
```

Esta sentencia permite leer datos de la unidad/archivo previamente abierto con `OPEN`. El asterisco señala que se deja “libre” el formato de lectura (lo veremos dentro de poco). La especificación `IOSTAT` es opcional pero muy útil para detectar errores en la lectura de datos (por ejemplo, si se llega al final del archivo y se intenta leer otro dato). En caso de error de lectura, usando `IOSTAT` el programa seguirá adelante (no

abortará) y guardará en la variable entera un valor que indicará que se ha producido un error de lectura.

Para tener éxito en la lectura de datos desde archivo con formato libre (*) el archivo deberá contener (al menos) tantos datos como se desee leer. Los datos deben estar separados por uno o más espacios en blanco, o estar en distintas líneas.

Ejemplos de uso de `READ` sin formato

Sean `A`, `B` y `C` variables de tipo `REAL`.

```
READ(10,*) A,B,C
```

Archivo de datos:

```
1. 2. 3.
```

Como resultado tendremos que `A=1.`, `B=2.` y `C=3.`.
Idéntico resultado se hubiera conseguido con el archivo de datos:

```
1. 2.
3.
```

* * *

Otra posibilidad:

```
READ(10,*) A
READ(10,*) B
READ(10,*) C
```

Archivo de datos:

```
1.
2.
3.
```

Cada sentencia `READ` lee datos desde el principio de una línea nueva. Si no encuentra suficientes datos en esa línea, los busca en la siguiente. Si `READ` no encuentra datos, se produce un error, como en el siguiente ejemplo.

```
READ(10,*) A
READ(10,*) B
READ(10,*) C
```

Archivo de datos:

```
1.  2.  3.
```

1.19.4. Sentencia `WRITE` (sin formato)

Sintaxis (básica):

```
WRITE(nro_de_unidad_abierta,*) variables_de_salida
```

Sintaxis (más completa:)

```
WRITE(nro_de_unidad_abierta,*,IOSTAT=variable_entera) vars_de_salida
```

Esta sentencia envía el valor de las variables a un archivo, previamente abierto con la sentencia `OPEN`. El asterisco (*) indica que el formato (apariencia) que va a tener la salida queda librada a la elección del Fortran. Así, si tenemos el siguiente programa

```
  ...
  OPEN(10,FILE='salida')
  X = 0.0079
  A = 'DIA'
  Y = 123.52189
  WRITE(10,*)X,A,Y
```

obtendremos en el archivo de salida:

```
   7.89999962E-03 DIA    123.521889
```

Noten que el formato de salida fue automáticamente establecido por el Fortran, lo que puede resultar en salidas poco prácticas o difíciles de leer. Para controlar el formato de salida hay que introducir *descriptores de edición* en lugar del asterisco.

1.19.5. Sentencia `WRITE` (con formato)

Modificamos el programa anterior utilizando descriptores de edición

```
  ...
  OPEN(10,FILE='salida')
  X = 0.0079
  A = 'DIA'
  Y = 123.52189
  WRITE(10,5)X,A,Y
5 FORMAT(F5.3,A4,1X,F6.2)
```

La salida será

```
0.008 DIA 123.52
```

En este programa, los descriptores aparecen en la sentencia `FORMAT` (cadena '`F5.3,A4,1x,F6.2`'). `F` se utiliza para valores de tipo `REAL`, `A` para los de tipo `CHARACTER` y `X` para dejar espacios en blanco. También existen descriptores para valores de tipo `INTEGER` y para organizar la salida de diferentes maneras (tabular, dejar renglones en blanco, etc). A continuación damos una explicación (breve) de los descriptores de edición más comunes. Para más detalles pueden consultar la bibliografía de la cátedra.

1.19.6. Descriptores de edición utilizados para SALIDA

F$n.d$	El número real se imprime justificado a derecha en n columnas. La parte fraccional es redondeada a d decimales. OJO, el punto decimal ocupa uno de los n lugares.
E$n.d$	Se utiliza para escribir números reales en formato exponencial ($\pm$0.$d_1d_2d_3$...E$\pm e_1e_2$). Debe tenerse en cuenta que n incluye los cuatro lugares del formato (E$\pm e_1e_2$).
In	Un número entero se escribe justificado a derecha en `n` columnas.
An	Se imprimen exactamente n caracteres. Si hay más de n caracteres en la cadena, los caracteres extra (a derecha) no son escritos. Si hay menos de n, la cadena es completada con blancos a izquierda, es decir, justificada a derecha.
A	El número de caracteres impresos coincide exactamente con la longitud de la cadena a imprimir.
nX	Deja n espacios en blanco.
n(/)	Indica que se deben bajar n líneas. Si sólo se desea bajar una línea se utiliza / sin paréntesis.
Tn	Tabula a la columna n.

Se puede repetir r veces un descriptor anteponiendo r al mismo. Ejemplos: rIn, rF$n.d$, rAn.

1.19.7. Descriptores de edición utilizados para ENTRADA

Las reglas y los descriptores son iguales que para el caso de la salida de datos, pero debe tenerse en cuenta lo siguiente:

1. No es necesario que los datos leídos (sean reales, enteros o de caracteres) ocupen todo el campo asignado a través del descriptor. En general, conviene seguir la siguientes recomendaciones:

   - los valores enteros deben escribirse en la porción derecha del lugar asignado. Ejemplo, el número entero 123 leído con descriptor `I7`, conviene ordenarlo así

				1	2	3

   - los valores reales pueden aparecer en cualquier lugar del campo asignado, siempre y cuando incluyan el punto decimal. Si no se incluye, el número debe ubicarse en la parte derecha del campo y el punto decimal se supondrá según indica el descriptor d de F$n.d$. Explicitar el punto decimal deja sin efecto al descriptor d. Ejemplo:

     ```
       READ(10,4) X,Y,Z
     4 FORMAT(F4.1,F4.2,F4.2)
     ```

     Archivo de datos

	−	8	5	6	.	3		.	5	8	9

     De acuerdo con las reglas señaladas más arriba, el programa asignará los siguientes valores: `x=-8.5`, `y=6.3`, `z=0.589`.

2. Si los descriptores se repiten

   ```
     READ(10,4) X,Y,Z,W
   4 FORMAT(F6.2,F6.2,F6.2,F6.2)
   ```

   pueden escribirse de manera más compacta:

   ```
     READ(10,4) X,Y,Z,W
   4 FORMAT(4F6.2)
   ```

   El número de repeticiones lo fija el usuario y puede ser utilizado con otros descriptores también.

3. ¿Qué sucede si la lista de entrada/salida no es satisfecha, es decir, si el número de variables difiere del número de descriptores?

   - Si hay más descriptores de edición que variables los descriptores adicionales son ignorados.
   - Cuando existen más nombres de variables en la lista que descriptores de edición, ésta se repite hasta completar el número de variables.
   - Si los descriptores de edición dan lugar insuficiente para la impresión de los valores de las variables, si éstas son cadenas de caracteres las trunca, si son reales o enteros completa el campo con asteriscos (*).

1.20. Otros comandos para lectura/escritura de archivos

1.20.1. Comando `BACKSPACE`

Sintaxis:

```
BACKSPACE(numero_de_unidad)
```

El puntero en el archivo asociado a la unidad "`numero_de_unidad`" es posicionado justo antes del último dato procesado. Si la posición del puntero está en el símbolo de fin de archivo, se posiciona justo delante del mismo.

1.20.2. Comando REWIND

Sintaxis:

```
REWIND(numero_de_unidad)
```

Este comando posiciona al puntero en el archivo asociado a la unidad "numero_de_unidad" justo antes del primer dato en él.

1.21. Arreglos

1.21.1. Introducción

En ciencias exactas, además de cantidades escalares solemos trabajar con vectores

$$\vec{v} = v_x\,\hat{\imath} + v_y\,\hat{\jmath} + v_z\,\hat{k} \tag{1.32}$$

y matrices

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} . \tag{1.33}$$

Fortran nos permite representar estos objetos usando arreglos (*arrays*), que son conjuntos de valores referenciados a través de un nombre común. Los arreglos requieren una declaración especial. Existen dos formas de declararlos, por un lado utilizando el atributo DIMENSION

```
REAL,DIMENSION(3) :: v       ! vector
REAL,DIMENSION(3,3) :: A     ! matriz (arreglo bidimensional)
```

y, por el otro, haciendo directamente

```
REAL :: v(3),A(3,3)
```

En el caso de vectores, la declaración anterior hará que se reserve espacio para 3 valores de tipo REAL asociados al nombre v. El primer espacio será referenciado como v(1), el segundo v(2) y el tercero v(3). Fortran reserva lugares contiguos de memoria para los tres números reales:

v(1)	v(2)	v(3)

Observaciones

- Siempre se deberá utilizar un número entero para indicar a qué elemento del grupo nos estamos refiriendo.
- Todos los elementos del arreglo serán de igual tipo de dato y podrá ser de cualquiera de los 5 tipos.

Cada elemento de un arreglo puede asignarse, o utilizarse, como una variable común:

```
vx = v(1)
vtot = SQRT(v(1)**2+v(2)**2+v(3)**2)
```

En el caso de matrices, necesitamos referirnos a cada elemento con dos números, llamados índices.

Existen casos en los que puede resultar más conveniente que los índices no comiencen en el valor 1. Por ejemplo, si estamos trabajando con polinomios

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3, \tag{1.34}$$

puede resultar conveniente declarar el vector para los coeficientes de la siguiente forma

```
REAL,DIMENSION(0:3) :: a
```

De esta manera, los 4 elementos de a serán a(0), a(1), a(2) y a(3), lo que resulta más natural. Lo mismo se puede hacer para cada índice de una matriz.

1.21.2. Algunas definiciones

- **Rango:** número de índices. En una matriz `A(3,3)` el rango es 2. Fortran permite declarar arreglos de rango 7, como máximo. El primer índice se menciona como la "dimensión 1", el segundo la "dimensión 2", etc.

- **Extensión:** cantidad de valores que recorre el índice en cada dimensión. Por omisión, los índices van desde 1 a la cantidad de elementos en esa dimensión (p. ej., un vector declarado como `v(3)` tiene 3 elementos, cada uno de los cuales se referencian como `v(1)`, `v(2)` y `v(3)`; su extensión será de 3). La declaración de arreglos puede ser menos rígida, para que el índice tome un rango de valores $(m{:}n)$. En este caso el índice recorrerá los valores enteros desde m hasta n. Un ejemplo con vectores sería declararlo como `v(-2:2)`, en cuyo caso sus elementos individuales serán `v(-2)`, `v(-1)`, `v(0)`, `v(1)` y `v(2)`. La extensión en este caso será de 5.

- **Forma:** es la combinación del rango y la extensión en cada dimensión. Así, dos arreglos tienen igual forma si tienen igual rango y la misma cantidad de elementos en cada dimensión.

1.21.3. Inicialización de arreglos

Una vez declarados, los arreglos deben ser inicializados antes de utilizarlos. Existen varias maneras de hacerlo.

1. Al momento de declararlos

Ejemplo (a):

```
REAL :: V(10)=0.
```

Ejemplo (b): usando "constructores" de arreglos.

```
REAL :: V(10)= (/1.,2.,3.,4.,5.,6.,7.,8.,9.,10./)
```

Es importante que el constructor contenga tantos elementos como el vector. Si esto no se cumple, la asignación generará un error al momento de compilar (*"Different shape for array assignement"*).

2. Usando asignaciones en la sección ejecutable

Ejemplo (a):

```
REAL :: V(10)
...
DO j=1,10
   V(j) = REAL(j)
ENDDO
```

Ejemplo (b):

```
V = 0.
```

Ejemplo (c):

```
V = (/ 0.,0.,0.,0.,0.,0.,0.,0.,0.,0./)
```

3. Leyendo datos con `READ`

```
REAL :: A(5)
READ(*,*) A(1),A(2),A(3),A(4),A(5)
```

La mayoría de estas formas son poco prácticas si el arreglo tiene muchos elementos. Para eso existe una estructura especial que veremos a continuación.

1.21.4. Lazos DO implícitos

Son estructuras de repetición, probablemente las más simples y eficientes, para leer y/o escribir los elementos de un arreglo.

Sintaxis:

$$(\texttt{arg1,arg2,...,i} = \texttt{i}_{\text{inicial}}\texttt{,i}_{\text{final}}\texttt{,}\Delta\texttt{i})$$

Al igual que en otros lazos, los valores inicial, final y el incremento pueden ser expresiones aritméticas (de resultado de tipo INTEGER).

Ejemplo 1)

```
INTEGER :: i,B(1000) = (/(i,i=1,1000)/)
```

que equivale a

```
INTEGER :: B(1000),i

DO i=1,1000
   B(i)=i
ENDDO
```

Ejemplo 2)

```
INTEGER :: B(1000) = (/(j,3*j,j=1,500)/)
```

Ejemplo 3) los lazos DO implícitos pueden estar anidados

```
INTEGER :: C(25) = (/((0,i=1,4),5*j,j=1,5)/)
```

que equivale a

```
INTEGER :: C(25)

C(1)=0; C(2)=0; C(3)=0; C(4)=0; C(5)=5; C(6)=0;...  ! notar el ;
```

Ejemplo 4)

```
INTEGER :: i,B(1000)

OPEN(10,FILE=...)
READ(10,*)(B(i),i=1,1000)
```

1.21.5. Un error común: salirse de rango

Un problema que surge frecuentemente cuando se utilizan arreglos es permitir que el índice de los mismos tome valores más allá de los permitidos por la declaración de los mismos.

Ejemplo 1)

```
INTEGER :: i
REAL :: A(3)

DO i=1,3
   A(i)=REAL(i)**2
ENDDO
DO i=1,5
   WRITE(*,*)i,A(i)
ENDDO
```

(En clase veremos qué hace el compilador en estos casos.)

Ejemplo 2)

```
INTEGER :: i
REAL :: A(3)

DO i=1,5
   A(i)=REAL(i)**2
   WRITE(*,*)i,A(i)
ENDDO
```

Comparar ambos ejemplos. ¿Qué diferencia importante existe entre ambos programas fuente?

En la mayoría de los casos el problema no genera un mensaje de error, lo que puede tener importantes consecuencias en el resultado. Para evitar este problema, hay que exigirle al compilador que realice un chequeo de índices al ejecutar el programa. Con `gfortran` esto se puede hacer compilando con la opción `-fbounds-check`. Si al momento de ejecutar el programa un índice se sale de rango, la corrida aborta y aparece un mensaje de error que indica la línea de código en la que se dio la situación. Esta opción sólo debe usarse cuando el código está siendo desarrollado y no para versiones definitivas, ya que hace más lenta la ejecución.

Ejemplo 3) Compilamos con `-fbounds-check` el programa del ejemplo 2)

```
gfortran -fbounds-check -o ejemplo ejemplo.f90
```

Ahora, al correr el programa obtendremos el siguiente mensaje:

```
fwachlin@marvin:~/ANyP$ ejemplo
           1   1.00000000
           2   4.00000000
           3   9.00000000
At line 7 of file pru.f90
Fortran runtime error: Index '4' of dimension 1 of array 'a' above upper bound of 3
fwachlin@marvin:~/ANyP$
```

1.21.6. Declarando arreglos usando constantes con nombre

Cuando el número de arreglos a declarar es grande, es conveniente utilizar constantes con nombre para establecer el tamaño de los mismo.

Ejemplo

```
INTEGER,PAREMETER :: N=100
REAL :: A(N)
CHARACTER(15) :: C(N)
COMPLEX :: M(3*N)
```

De esta manera, no hay que cambiar el tamaño de cada uno de los arreglos, simplemente cambiando `N` se ajustan todos automáticamente.

1.21.7. Operaciones con arreglos como un todo

Si dos arreglos tienen la misma forma, es posible utilizarlos con las operaciones aritméticas básicas, como si fuesen variables simples. Las operaciones se realizarán elemento a elemento.

Ejemplo: consideremos dos vectores enteros inicializados así

```
A(1)=1; A(2)=2; A(3)=3; A(4)=4
B(1)=5; B(2)=6; B(3)=7; B(4)=8
```

Utilizando un lazo `DO` podemos hacer la suma elemento a elemento

```
DO i=1,4
   D(i)=A(i)+B(i)
ENDDO
```

con lo que tendremos que D(1)=6, D(2)=8, D(3)=10 y D(4)=12. El mismo resultado se puede obtener haciendo

```
D = A+B
```

que es mucho más compacto. Esta misma operatoria se puede lograr para las demás operaciones aritméticas (-, *, /, **), siempre que los arreglos tengan la misma forma. Si hubiésemos hecho

```
D = A*B
```

el resultado habría sido D(1)=5, D(2)=12, D(3)=21 y D(4)=32. (Noten que el producto de matrices no tiene nada que ver con este tipo de producto elemento a elemento).

* * *

También es posible multiplicar un arreglo por un escalar

```
INTEGER :: i,A(4) = (/(i,i=1,4)/)
INTEGER :: C=5

WRITE(*,*) C*A
```

Salida

```
           5          10          15          20
```

* * *

Muchas de las funciones intrínsecas pueden ser aplicadas a arreglos. Como resultado se obtiene un nuevo arreglo cuyos elementos serán el resultado de aplicar la función a cada uno de los elementos originales. Por ejemplo, si A es el vector del ejemplo anterior, SQRT(REAL(A)) dará por resultado otro vector cuyos elementos serán $\{\sqrt{4.}, \sqrt{10.}, \sqrt{15.}, \sqrt{20.}\}$.

1.21.8. Algo más sobre arreglos bidimensionales (matrices)

Como mencionamos antes, en Fortran 90 es posible utilizar arreglos de hasta rango 7. Particularmente útiles son los bidimensionales (rango 2), también llamados matrices. Ya hemos visto la forma de declararlos:

```
INTEGER :: HIST(0:100,0:20)
CHARACTER :: MARCA(-3:3,10)
```

Su inicialización puede hacerse de diversas formas, una de las cuales es elemento a elemento

```
INTEGER :: i,j
REAL :: A(4,3)

DO i=1,4
   DO j=1,3
      A(i,j)=0.
   ENDDO
ENDDO
```

No es posible usar aquí “constructores” (A=(/0.,...,0./)) puesto que ellos sólo producen arreglos unidimensionales. Una posibilidad para subsanar este problema es utilizar la función RESHAPE, que permite cambiar la forma de un arreglo sin modificar su número de elementos. La sintaxis es la siguiente

arreglo_resultante = RESHAPE(vector_original,vector_de_forma)

El "vector_original" es el vector que contiene los elementos con los que queremos inicializar la matriz en cuestión y "vector_de_forma" es otro vector con los índices que dan forma a la matriz. Por ejemplo, supongamos que queremos inicializar la matriz

$$Z = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \quad (1.35)$$

utilizando un constructor. Lo hacemos de la siguiente forma

```
REAL :: Z(3,2)

Z = RESHAPE( (/1,3,5,2,4,6/),(/3,2/) )
```

Observen el orden de los elementos en el primer vector. Esto es así porque Fortran ubica en forma contigua, en memoria, los elementos de una matriz ordenados por columnas. Este tema será crítico más adelante, cuando trabajemos con funciones y subrutinas.

Otra manera de inicializar un arreglo bidimensional es leyendo datos de un archivo:

```
INTEGER :: i,j,K(4,3)

OPEN(10,FILE='DATOS',STATUS='OLD')
READ(10,*)((K(i,j),j=1,3,i=1,4)
```

que funciona adecuadamente si los datos tienen, por ejemplo, la siguiente estructura

```
1 2 3
4 5 6
7 8 9
10 11 12
```

1.21.9. Funciones intrínsecas para arreglos

Existen diversas funciones intrínsecas que nos permiten averiguar propiedades de los arreglos, como su tamaño, forma, etc. Para entender mejor su funcionamiento, vamos a aplicarlas a la matriz

```
REAL :: A(-2:2,3)
```

La función

SIZE(arreglo,dimensión)

nos devolverá un número entero con la extensión del arreglo a lo largo de la dimensión especificada como argumento. Si no se especifica ninguna dimensión, SIZE nos dará el número total de elementos en el arreglo.

```
INTEGER :: r1,r2,r3
REAL :: A(-2:2,3)

r1 = SIZE(A,1)
r2 = SIZE(A,2)
r3 = SIZE(A)
WRITE(*,*)'r1,r2,r3:␣',r1,r2,r3
```

Ejecutando el programa obtendremos como salida

```
 r1,r2,r3:           5           3          15
```

La función

LBOUND(arreglo,dimensión)

devolverá el índice inferior del arreglo en la dimensión especificada, o todos los índices inferiores del arreglo (en este caso, como un vector) si no se especifica la dimensión.

```
INTEGER :: r1,r2,r3(2)
REAL :: A(-2:2,3)

r1 = LBOUND(A,1)
r2 = LBOUND(A,2)
r3 = LBOUND(A)
WRITE(*,*)'r1,r2: ',r1,r2
WRITE(*,*)'r3:    ',r3
```

Ejecutando el programa obtendremos como salida

```
r1,r2:          -2          1
r3:             -2          1
```

La función

UBOUND(arreglo,dimensión)

opera de manera similar a LBOUND pero con respecto a los límites superiores.

Existen otras funciones intrínsecas que, a diferencia de las que vimos, operan sobre el arreglo como un todo devolviendo como resultado un arreglo que en general no tiene la misma forma que el argumento. Algunas de ellas son:

- DOT_PRODUCT(vector_a,vector_b): devuelve un escalar, resultado del producto escalar entre ambos vectores. Naturalmente, los vectores en el argumento deberán tener el mismo tamaño.
- MATMUL(matriz_a,matriz_b): realiza la multiplicación de matrices. La forma de las matrices debe ser compatible para la multiplicación.
- TRANSPOSE(matriz): da por resultado la matriz transpuesta.
- MAXVAL(arreglo,condición): devuelve el elemento de mayor valor del arreglo, tomado de entre los valores que cumplen la condición "máscara" (la condición es opcional). Ejemplo

```
MAXVAL(A,A<0)
```

- MINVAL(arreglo,condición): ídem anterior para el elemento de menor valor.
- MINLOC(arreglo,condición): vector con los índices correspondientes al elemento de valor mínimo, compatible con la condición.
- MAXLOC(arreglo,condición): ídem anterior, para el elemento de mayor valor.
- PRODUCT(arreglo): producto de todos los elementos del arreglo.
- SUM(arreglo): suma de todos los elementos del arreglo.

1.21.10. Asignación dinámica de memoria

En todos los ejemplos vistos hasta ahora el tamaño de los arreglos quedó establecido a la hora de declararlos. A partir de ese momento, el tamaño ha quedado fijo a lo largo de la ejecución del programa. Esta forma de declarar arreglos se denomina **asignación estática de memoria**. Un problema que puede darse con esta forma de declararlos es que el tamaño de los arreglos tiene que ser el mayor de todos los casos posibles, y en todos los casos en que el tamaño del arreglo sea menor, como la memoria ya fue asignada, se desperdicia.

Existe la posibilidad de utilizar una **asignación dinámica de memoria**, declarando arreglos cuyo tamaño es ajustado durante la ejecución del código, evitando de esta manera que se desperdicie memoria.

Para declarar un arreglo dinámico hacemos

```
REAL,ALLOCATABLE,DIMENSION(:,:) :: A
```

Un arreglo declarado de esta forma no puede ser utilizado hasta que no se le asigne un tamaño preciso, tarea que se realiza con la instrucción `ALLOCATE`, cuya sintaxis es

`ALLOCATE(arreglo,STAT=status)`

Ejemplo

```
ALLOCATE(A(10,20),STAT=resultado)
```

La entrada `STAT=` es opcional. Si es utilizada, devuelve un valor entero (`resultado` debe haber sido declarada INTEGER) con valor nulo si la asignación fue exitosa y no nulo si falló. Si `STAT=` no está y la asignación de memoria falla, el programa abortará.

La sentencia `ALLOCATE` puede asignar memoria para más de un arreglo

```
ALLOCATE(V(2345),A(0:500,5:20))
```

Cuando se trabaja con arreglos cuyo espacio en memoria es asignado dinámicamente, es importante liberar ese espacio antes de finalizar el programa, para devolverle esa memoria al sistema. Eso se realiza con la sentencia `DEALLOCATE`, cuya sintaxis es

`DEALLOCATE(lista_de_arreglos,STAT=status)`

Ejemplo

```
DEALLOCATE(A,B,C,D,STAT=resultado)
```

1.21.11. Sentencia WHERE

Permite ejecutar asignaciones de forma selectiva sobre los elementos de un arreglo. Sintaxis:

```
WHERE(condicion_sobre_arreglo) arreglo = expresion
```

donde "`condicion_sobre_arreglo`" es una condición lógica que opera sobre los elementos de un arreglo de igual forma que el de la asignación. La asignación sólo se ejecuta sobre los elementos del arreglo para los cuales la variable lógica asociada (por la posición en el arreglo de la condición) es verdadera.

Ejemplo. Si tenemos la matriz

$$\mathtt{B} = \begin{pmatrix} 1{,}2 & 7 & -3 \\ 0 & 1 & 5 \\ -1 & 4 & 6 \end{pmatrix} \quad (1.36)$$

declarada como

```
REAL :: B(3,3)
```

entonces

```
WHERE(B<0.) B=-1./B
```

da por resultado

$$\mathtt{B} = \begin{pmatrix} 1{,}2 & 7 & 0{,}33333333 \\ 0 & 1 & 5 \\ 1 & 4 & 6 \end{pmatrix} \quad (1.37)$$

1.21.12. Bloque WHERE

Consiste en la generalización de la sentencia WHERE, para que realice una asignación diferente si no se cumple la condición especificada.

Sintaxis:

```
WHERE(condicion_sobre_arreglo)
   sentencia_de_asignacion_1
ELSE WHERE
   sentencia_de_asignacion_2
END WHERE
```

Ejemplo

```
REAL :: B(3,3)

WHERE(B>0.)
  B=SQRT(B)
ELSE WHERE
  B=-10.
END WHERE
```

1.21.13. Otras funciones intrínsecas para arreglos

```
ALL(condicion_sobre_arreglo)
```

ALL da por resultado .TRUE. si la condición es verdadera para todos los elementos del arreglo.

```
ANY(condicion_sobre_arreglo)
```

ANY da por resultado .TRUE. si la condición es verdadera al menos para un elemento del arreglo.

```
COUNT(condicion_sobre_arreglo)
```

COUNT da el número de elementos que satisfacen la condición.

Ejemplo. Sea la matriz

$$\mathtt{A} = \begin{pmatrix} .\text{TRUE.} & .\text{FALSE.} \\ .\text{FALSE.} & .\text{FALSE.} \\ .\text{TRUE.} & .\text{TRUE.} \end{pmatrix} \tag{1.38}$$

declarada

```
LOGICAL :: A(3,2)
```

Las funciones intrínsecas aplicadas a A darán lo siguiente:

$$\mathtt{ALL(A)} \rightarrow \mathtt{.FALSE.}$$

$$\mathtt{ANY(A)} \rightarrow \mathtt{.TRUE.}$$

$$\mathtt{COUNT(A)} \rightarrow 3$$

Estas funciones admiten otras especificaciones que no veremos acá.

1.21.14. Bloque FORALL

La sentencia FORALL fue incorporada recién con el Fortran 95. Sirve para ejecutar una serie de instrucciones basándonos en una selección elemento a elemento de un arreglo. Opcionalmente, se puede incluir una condición lógica sobre los mismos.

```
[nombre:] FORALL (ind1=tripl1 [,ind2=tripl2]...[,condicion logica])
  sentencia1
  [sentencia2]
END FORALL [nombre]
```

Cada índice ind se especifica por un tripete de la forma

$$v_i:v_f:paso.$$

La sentencia o sentencias que forman el cuerpo de la estructura se ejecutarán sólo sobre aquellos elementos del arreglo que cumplan la condición lógica. Si se omite la condición lógica, las sentencias se ejecutan sobre todos los elementos.

Ejemplo

```
REAL :: A(10,20)
INTEGER :: i,j

! (inicializar A...)
FORALL (i=1:10:1,j=1:20:1,A(i,j)>2.)
   A(i,j) = SQRT(A(i,j))
END FORALL
```

Observen que esta construcción es equivalente a usar dos bucles DO anidados y un IF dentro de ellos. La diferencia es en el funcionamiento, puesto que el bucle DO se procesa estrictamente en el orden de los índices, mientras que el FORALL en cualquiera. Esto es ventajoso ya que permite al sistema operativo optimizar la ejecución del programa. Si un bucle FORALL tiene más de una instrucción, la segunda no es ejecutada hasta que la primera se complete totalmente.

Finalmente, FORALL puede utilizarse también como una sentencia (similar al IF lógico)

```
FORALL (ind1=tripl1 [,ind2=tripl2]...[,cond logica]) sentencia_unica
```

1.22. Procedimientos

1.22.1. Introducción

Cuando uno diseña un programa debe procurar estructurar por bloques las tareas a realizar. Hasta el momento sólo sabemos llevar a cabo esas tareas en un único programa, que puede ser muy largo. Para hacer el programa más claro y versátil, Fortran proporciona medios para que los bloques de tareas distintas de un mismo programa puedan ser escritos como *unidades de programa* separados, es decir, como porciones de código que pueden desarrollarse y compilarse en forma separada del programa principal. Las unidades de programa a las que nos estamos refiriendo son las llamadas SUBRUTINAS y FUNCIONES.

Las subrutinas son procedimientos llamados mediante la instrucción CALL y pueden devolver uno o más resultados a través de sus argumentos. Por su lado, las funciones son invocadas mediante la inclusión de su nombre en una expresión y dan por resultado un único valor.

Los beneficios de la utilización de estas unidades de programa incluyen:

1. testeo independiente de cada grupo de tareas,
2. código reutilizable: en muchos casos, el mismo grupo básico de tareas es necesario en distintas partes del programa. En vez de escribirlo una y otra vez, utilizar subrutinas y funciones nos permiten hacerlo sólo una, simplificando el código y la búsqueda de errores.
3. protección (encapsulamiento de variables): dado que las subrutinas y funciones se comunican con el programa principal a través de variables llamadas *argumentos*, sólo la modificación de éstas puede tener consecuencias sobre el programa principal.

1.22.2. Subrutinas

Como dijimos antes, este procedimiento es invocado mediante la utilización de su nombre en una sentencia CALL. Una subrutina recibe datos de entrada y provee resultados a través de su *lista de argumentos*. La forma general de una subrutina es:

```
SUBROUTINE nombre_subrutina(lista_de_argumentos_formales)
seccion de declaracion
...
seccion de ejecucion
...
RETURN
END SUBROUTINE [nombre_subrutina]
```

El nombre de la subrutina puede tener hasta 31 caracteres alfanuméricos y guiones bajos, pero debe comenzar con una letra. Los elementos de la lista de argumentos se llaman *argumentos formales*, ya que no se asigna memoria para ellos, son simplemente punteros a las direcciones de memoria donde se guardan las variables que constituyen los argumentos *verdaderos* (los que se pasa a la subrutina al llamarla con la sentencia CALL).

Observaciones

- Debe notarse que una subrutina es una unidad de programa que es compilada separadamente del programa principal y de otros subprogramas, por ello, los nombres de las variables locales podrán ser utilizados en distintos subprogramas sin causar conflicto alguno (*data hiding*).
- Una subrutina puede incluir llamadas a otras, excepto a ella misma (salvo que haya sido declarada como recursiva).

Para llamar a una subrutina:

```
CALL nombre_subrutina(lista_de_argumentos_verdaderos)
```

Ejemplo: Veamos una subrutina que calcula la superficie y el volumen de una esfera de radio r.

```
SUBROUTINE SUP_VOL_ESFERA(r,sup,vol)
REAL,INTENT(IN) :: r
REAL,INTENT(OUT) :: sup,vol
REAL :: pi

! Calculo de pi
pi = 4.*ATAN(1.)

! Calculo de la superficie
sup = 4.*pi*r**2

! Calculo del volumen
vol = 4./3.*pi*r**3

END SUBROUTINE SUP_VOL_ESFERA
```

A continuación escribimos un programa principal que hace uso de esta subrutina.

```
PROGRAM uso_subrutina
REAL :: radio,superficie,volumen

radio=1.
CALL SUP_VOL_ESFERA(radio,superficie,volumen)
WRITE(*,*)'Para r= ',radio,' Sup= ',superficie,' Vol= ',volumen

radio=2.
```

```
CALL SUP_VOL_ESFERA(radio,superficie,volumen)
WRITE(*,*)'Para r= ',radio,' Sup= ',superficie,' Vol= ',volumen

END PROGRAM uso_subrutina
```

Este programa es extremadamente rígido, pero cumple con el objetivo de calcular la superficie y el volumen para dos esferas de distinto radio a través de una subrutina.

Noten que:

- Los argumentos verdaderos y formales no tienen por qué llamarse igual.
- Hay una variable en la subrutina que tiene carácter local (¿pueden identificarla?).
- Los argumentos formales tienen un atributo `INTENT` asociado, que puede tomar uno de los siguientes valores: `IN`, `OUT` o `INOUT`. Si bien este atributo no es obligatorio, es muy recomendable incluirlo ya que le permite saber al compilador cómo se planean usar los argumentos formales y ayudar así a detectar errores; por ejemplo, si una variable en el argumento es declarada como `INTENT(IN)`, cualquier intento de modificarla dará lugar a un error.

En clase plantearemos mejoras al programa principal para hacerlo menos rígido y más útil.

1.22.3. Pasando variables: el esquema de pasaje por referencia

Cuando un programa se comunica con un subprograma, los VALORES de las variables de entrada NO son pasados a la subrutina, sino las DIRECCIONES de memoria de esas variables. Esta forma de comunicación nos obliga a prestar mucha atención por dos motivos:

1. Si nos equivocamos en el orden (y/o tipo) de las variables de entrada al llamar a una subrutina, los resultados pueden ser catastróficos. Ejemplo

```
PROGRAM ej1
REAL :: x=1.

CALL SUB_EJ1(x)
END PROGRAM ej1
```

```
SUBROUTINE SUB_EJ1(Z)
INTEGER,INTENT(IN) :: Z

WRITE(*,*)Z
END SUBROUTINE SUB_EJ1
```

   Al compila y correr este programa, un compilador Fortran estándar daría por resultado

$$\texttt{Z} \longrightarrow 1065353216$$

   Éste también sería el resultado que daría `gfortran` hasta hace algunos años, antes de que se implementara una opción de compilación (encendida por omisión) que detecta el problema y muestra un mensaje cuando no hay concordancia entre el tipo de dato del argumento verdadero y el formal:

   Type mismatch in argument 'z'...; passed REAL(4) to INTEGER(4) [-Wargument-mismatch]).

   Este control no forma parte de la norma del Fortran estándar, por lo que programar así provoca que se corra el riesgo de producir códigos no "portables". Veremos muy pronto cómo debemos programar para evitar este problema.

2. Otra posible fuente de problemas es la modificación de parámetros de entrada dentro de la subrutina. Esto es posible si no se utiliza el atributo opcional `INTENT`, o si el argumento se declara como `INTENT(INOUT)`. Una modificación de esa variable en la subrutina tiene por efecto inmediato la modificación de la variable verdadera en el programa principal. Veamos un ejemplo.

```
PROGRAM ej2
REAL :: x=1.

WRITE(*,*)'X␣inicial:␣',x
CALL SUB_EJ2(x)
WRITE(*,*)'X␣final:␣␣␣',x
END PROGRAM ej2
```

```
SUBROUTINE SUB_EJ2(Z)
REAL,INTENT(INOUT) :: Z

Z = Z+1
END SUBROUTINE SUB_EJ2
```

Al correrlo, el programa imprime

```
X final: 2.00000000
```

Estas dos situaciones representar errores comunes, que se pueden evitar de distintas formas:

(a) usando el atributo `INTENT`, separando los parámetros de entrada y salida,

(b) mirando con mucha atención el orden y tipo de los argumentos verdaderos al momento de llamar a una subrutina, de modo que coincidan con los argumentos formales de la subrutina,

(c) creando interfases explícitas (lo veremos pronto).

1.22.4. Funciones

Una función es un procedimiento cuyo resultado es un único escalar o arreglo de alguno de los tipos de dato aceptados por Fortran. Este resultado puede combinarse con variables y constantes para formar expresiones. Hay dos tipos de funciones:

$$\text{funciones} \begin{cases} \textit{intrínsecas} \\ \\ \textit{definidas por el usuario} \end{cases} \tag{1.39}$$

Sobre funciones intrínsecas ya hemos visto varias (trigonométricas, trigonométricas inversas, exponencial, para multiplicar matrices, etc.). Veamos ahora aquéllas que son definidas por el usuario. La sintaxis a utilizar es la siguiente:

```
FUNCTION nombre_funcion(argumentos_formales)
tipo_de_dato :: nombre_funcion
...
(seccion de declaracion)
...
(seccion ejecutable)
...
nombre_funcion = expresion
RETURN
END FUNCION nombre_funcion
```

El nombre de la función debe tener un tipo asociado. Alternativamente, las dos primeras líneas pueden resumirse en la siguiente:

```
tipo_de_dato FUNCTION nombre_funcion(argumentos_formales)
```

El tipo de dato de la función debe declararse tanto en la función misma como en el programa que la invoca.

Veamos como ejemplo, el programa del elipsoide, calculando ahora el radio mediante una función.

```
FUNCTION calculo_radio(FI)
IMPLICIT NONE
REAL :: calculo_radio
REAL,INTENT(IN) :: FI
REAL :: FIR,PI
REAL, PARAMETER :: A=6378.38,ALPHA=0.003367

PI = 4.*ATAN(1.)
FIR = FI*PI/180.
calculo_radio = A*(1.-ALPHA*SIN(FIR)**2)
RETURN
END FUNCTION calculo_radio
```

```
PROGRAM elipsoide3
IMPLICIT NONE
REAL :: FI,calculo_radio

WRITE(*,*)'Ingrese␣la␣latitud␣en␣grados:␣'
READ(*,*) FI
WRITE(*,*)'El␣radio␣a␣una␣latitud␣de␣',FI,'␣grados␣es␣de␣', &
calculo_radio(FI),'␣km'
END PROGRAM elipsoide3
```

(En clase veremos cómo funciona este programa.)

La idea principal de una función creada por el usuario es similar a la de una función intrínseca: realizar un cálculo que arroje un único valor por resultado, sin modificar en el proceso el valor de sus argumentos. Como los argumentos son pasados por referencia, puede ocurrir que un programador descuidado los modifique en la función provocando que la llamada tenga "efectos colaterales". Esta es una práctica que debe desalentarse, por lo que será cuasi-obligatorio para nosotros declarar los argumentos formales de toda función con el atributo `INTENT(IN)`.

Observaciones

- No debemos olvidarnos de aclarar el tipo de dato de la función (tanto en la función en sí como en la unidad de programa que la invoca).
- Declarar los argumentos formales con `INTENT(IN)` ayuda a evitar que las funciones tengan efectos colaterales.
- Siempre hay que recordar, antes de salir, asignarle un valor a la variable cuyo nombre es el de la función. El resultado de la función se transmitirá a través de ella.

1.22.5. Pasando arreglos como argumento

De acuerdo al pasaje por referencia, cuando se utiliza un arreglo como argumento de un llamado a una subrutina, sólo se envía la dirección de memoria del primer elemento del mismo. Obviamente, esta información puede resultar insuficiente para trabajar con él en forma segura, ya que necesitamos conocer su tamaño para no exceder sus límites. Vamos a ver dos formas de pasar esta información a una subrutina que le permiten al compilador efectuar los controles pertinentes.

La primera es pasando los arreglos explicitando su forma, esto es, haciéndole conocer a la subrutina su rango y extensión. Haciendo esto, el compilador puede chequear que no nos excedamos de sus límites. Veamos un ejemplo.

```
PROGRAM ej3
IMPLICIT NONE
INTEGER,PARAMETER :: N=10
REAL :: X(N),Z(N)
INTEGER :: J,NVAL

NVAL=5
DO J=1,NVAL
   X(J) = REAL(J)
ENDDO
CALL SUB_EJ3(X,Z,N,NVAL)
DO J=1,NVAL
   WRITE(*,*)Z(J)
ENDDO
END PROGRAM EJ3
```

```
SUBROUTINE SUB_EJ3(X,Z,N,NVAL)
IMPLICIT NONE
INTEGER,INTENT(IN) :: N,NVAL
REAL,INTENT(IN) :: X(N)
REAL,INTENT(OUT) :: Z(N)
INTEGER :: K

DO K=1,NVAL
   Z(K) = 10.*X(K)
ENDDO

END SUBROUTINE SUB_EJ3
```

Declarando los arreglos de esta forma, el compilador tiene toda la información necesaria sobre el arreglo, y dentro de la subrutina pueden usarse todas las herramientas para arreglos que hemos visto.

La segunda alternativa es usando arreglos formales "con forma supuesta", es decir, declarando los arreglos en la subrutina poniendo dos puntos (:) en lugar del tamaño, y creando una interfaz explícita. A continuación veremos de qué se tratan las interfaces explícitas.

1.23. Creando interfases explícitas

Hasta el momento hemos visto cómo crear subrutinas y funciones, lo que nos permite programar en forma más estructurada. Lamentablemente, creando procedimientos de la manera en que lo hicimos el compilador no tiene la capacidad de controlar el tipo y el número de variables con los que una función/subrutina es llamada. Para que este control pueda ser realizado es necesario dotar a los subprogramas de *interfases explícitas*. Existen dos formas alternativas de hacerlo:

$$\text{interfases explícitas} \begin{cases} \text{módulos} \\ \\ \text{bloques de interfaz} \end{cases} \tag{1.40}$$

Vamos a ver en qué consiste cada una.

1.23.1. Módulos

Un módulo es una unidad de programa que puede compilarse separadamente y que tiene la siguiente estructura:

```
MODULE nombre_del_modulo
IMPLICIT NONE
(seccion_de_declaraciones)
```

```
  CONTAINS
    (procedimientos)
  END MODULE nombre_del_modulo
```

Ejemplo:

```
  MODULE suma_y_promedio
  CONTAINS
    REAL FUNCTION suma(a,b,c)
      IMPLICIT NONE
      REAL,INTENT(IN) :: a,b,c
      suma = a+b+c
    END FUNCTION suma
    REAL FUNCTION promedio(a,b,c)
      IMPLICIT NONE
      REAL,INTENT(IN) :: a,b,c
      promedio=suma(a,b,c)/3.
    END FUNCTION promedio
  END MODULE nombre_modulo
```

Un módulo no se puede utilizar en solitario, necesita de un programa principal. Para invocar al módulo, el programa debe incluir la sentencia

```
  USE nombre_modulo
```

en la sección de declaración. Utilizando módulos el compilador es capaz de detectar errores que antes no veía. Supongamos que tenemos en dos archivos distintos lo siguiente:

```
! ******* archivo modulo_1.f90 **********
  MODULE modulo1                ! <---
  CONTAINS                      ! <---
    SUBROUTINE SUB_EJ1(Z)
    IMPLICIT NONE
    INTEGER,INTENT(IN) :: Z
    WRITE(*,*)Z
    END SUBROUTINE SUB_EJ1
  END MODULE modulo1            ! <---
```

```
! ******* archivo ej1.f90 **********
  PROGRAM EJ1
  USE modulo1                   ! <---
  IMPLICIT NONE
  REAL :: x=1.
  CALL SUB_EJ1(x)
  END PROGRAM EJ1
```

Para compilar el programa debemos hacer ahora

`gfortran -o ej1 modulo_1.f90 ej1.f90`

(OJO, el orden importa, deben ser compilados en primer término los archivos que contienen módulos y después los programas que los invocan.)

Como resultado, Fortran nos mostrará el siguiente mensaje de error

```
ej1.f90:6.15:

  CALL SUB_EJ1(x)
               1
Error: Type mismatch in argument 'z' at (1); passed REAL(4) to INTEGER(4)
```

que demuestra que ahora el compilador es capaz de detectar este tipo de errores.

También se podrá detectar si nos equivocáramos en el número de argumentos al llamar a la subrutina. Supongamos que cambiamos la llamada por

```
  CALL SUB_EJ1(x,k)
```

con k previamente declarada y habiendo arreglado que la subrutina reciba un valor real. Entonces, el mensaje al compilar sería:

```
ej1.f90:6.19:

  CALL SUB_EJ1(x,k)
                 1
Error: More actual than formal arguments in procedure call at (1)
```

1.23.2. Bloques de interfaz

Para utilizar bloques de interfaz debemos incluir en el programa que llama a la subrutina, luego de la zona de declaración y antes de la primera instrucción ejecutable, la estructura:

```
  INTERFACE
    SUBROUTINE nombre_subrutina(argumentos_formales)
      (declaracion de argumentos formales)
    END SUBROUTINE nombre_subrutina
  END INTERFACE
```

Así, en esta alternativa, el programa principal del ejemplo visto antes quedaría:

```
! ******* archivo ej1.f90 **********
  PROGRAM EJ1
  IMPLICIT NONE
  REAL :: x=1.
  INTERFACE
    SUBROUTINE SUB_EJ1(x)
    INTEGER,INTENT(IN) :: x
    END SUBROUTINE SUB_EJ1
  END INTERFACE

  CALL SUB_EJ1(x)
  END PROGRAM EJ1
```

Nuevamente el compilador será capaz de detectar el error por falta de concordancia en el tipo de argumento.

1.24. Nota sobre la representación de números reales

En la mayoría de las computadoras, el tipo REAL reserva 4 bytes para representar al número real. Hay una versión de reales, llamada de "doble precisión", donde a cada real se le asignan 8 bytes. El estándar del F90 dice que un compilador debe soportar al menos dos tipos de reales, pero *no establece cuántos bits debe tener cada tipo, ni cómo deben ser llamados.* Esto genera inconvenientes, ya que para escribir programas portables entre computadoras, debiera ser posible asegurar igual representación para los reales, independientemente del nombre.
La forma de nombrar al tipo de real con el que se desea trabajar se hace a través de la palabra clave KIND.
Se denomina el "KIND" de una variable a la etiqueta entera que le indica al compilador cuál de los tipos soportados debe utilizarse para dicha variable.
Una elección muy común para el parámetro KIND es que sea equivalente al número de bytes reservados para la variable con ese KIND. Lamentablemente esta convención no es parte del estándar.

Así, un gran número de compiladores (gfortran entre ellos) entiende lo siguiente

```
REAL(KIND=4)  :: xs   ! reserva 4 bytes para xs
REAL(KIND=8)  :: xd   !    "     8 bytes
REAL(KIND=16) :: xq   !         16 bytes
```

mientras que para otros:

```
REAL(KIND=1) :: xs   !  4 bytes
REAL(KIND=2) :: xd   !  8 bytes
REAL(KIND=3) :: xq   ! 16 bytes
```

Este problema se puede resolver con la función intrínseca `selected_real_kind`, pero no lo veremos en este curso.