

---

# Manual de Fortran 77

---

v0.7

LAS PALMAS, OCTUBRE 2000

AUTORES : JESÚS GARCÍA QUESADA

---

Reservados todos los derechos. La reproducción total o parcial de esta obra por cualquier medio o procedimiento comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares de ella mediante alquiler o préstamos públicos, queda rigurosamente prohibida sin la autorización escrita de los titulares del copyright, bajo las sanciones establecidas por las leyes.

Sin perjuicio de lo anterior, queda expresamente autorizada la reproducción reprográfica total o parcial de esta obra por miembros de la comunidad universitaria de la Universidad de Las Palmas de Gran Canaria para uso docente en el ámbito de la misma.

Edificio de Informática y Matemáticas

Campus Universitario de Tafira

35017 Las Palmas de Gran Canaria

ISBN-84-699-4379-0

Nº Registro: 762501

El FORTRAN es uno de los lenguajes de programación más ampliamente utilizados, especialmente entre la comunidad científica. Fué el primer lenguaje de alto nivel que fué aceptado. Su nombre deriva de FORmula TRANslation (traducción de fórmulas) y ha tenido varias revisiones.... Las sentencias ejecutables describen las “acciones” del programa. Las no ejecutables describen la distribución de los datos y sus características, o dan información sobre la edición y conversión de datos.

Una sentencia **PROGRAM** puede aparecer solo como la primera instrucción de un programa principal. La primera sentencia de un subprograma ha de ser una **FUNCTION**, **SUBROUTINE** o **BLOCK DATA**.... Los caracteres alfabéticos en minúsculas, el signo de admiración(!), el subrayado(\_) y las comillas (") son extensiones al standard ANSI FORTRAN 77.

Se pueden intercalar espacios para mejorar la legibilidad del código fuente. El compilador no hace distinción entre mayúsculas y minúsculas (excepto en constantes de tipo carácter).... Consisten en un par de constantes enteras o reales separadas por una coma y encerradas entre paréntesis. La primera cte. representa la parte real y la segunda la parte imaginaria.... Se representan por un nombre simbólico asociado a determinadas posiciones de memoria. Se clasifican, al igual que las constantes, por su tipo de datos.... El tipo de datos de una variable indica el tipo de datos que representa, su precisión y sus requerimientos de memoria. Cuando se asigna cualquier tipo de datos a una variable, estos son convertidos, si es necesario, al tipo de datos de la variable.

# Índice General

<b>1</b>	<b>Introducción</b>	<b>1</b>
1.1	Elementos de un programa fuente FORTRAN . . . . .	1
1.1.1	Nombres simbólicos . . . . .	2
1.1.2	Comentarios . . . . .	2
1.1.3	Conjunto de caracteres FORTRAN . . . . .	2
1.1.4	Estructura de las líneas de código . . . . .	3
1.1.5	Tipos de datos . . . . .	3
<b>2</b>	<b>Datos</b>	<b>4</b>
2.1	CONSTANTES . . . . .	4
2.1.1	ENTERAS . . . . .	5
2.1.2	CONSTANTES REALES . . . . .	5
2.1.3	CONSTANTES COMPLEJAS . . . . .	7
2.1.4	CONSTANTES OCTALES Y HEXADECIMALES . . . . .	7
2.1.5	CONSTANTES LÓGICAS . . . . .	7
2.1.6	CONSTANTES TIPO CARÁCTER . . . . .	8
2.2	Variables . . . . .	8
2.3	Arrays . . . . .	9
2.3.1	Declaradores de array . . . . .	9
2.3.2	Almacenamiento en memoria . . . . .	9
2.3.3	Subcadenas . . . . .	10
<b>3</b>	<b>EXPRESIONES</b>	<b>12</b>
3.1	Expresiones aritméticas . . . . .	12
3.1.1	Uso de paréntesis . . . . .	13
3.1.2	Tipos de datos en una expresión aritmética . . . . .	13
3.2	Expresiones de carácter . . . . .	15
3.3	Expresiones relacionales . . . . .	15
3.4	Expresiones lógicas . . . . .	16
<b>4</b>	<b>INSTRUCCIONES DE ASIGNACIÓN</b>	<b>18</b>
4.1	Asignación aritmética . . . . .	18
4.2	Asignación lógica . . . . .	18
4.3	Asignación de carácter . . . . .	19
4.4	Instrucción <b>ASSIGN</b> . . . . .	19

<b>5</b>	<b>INSTRUCCIONES DE ESPECIFICACIÓN</b>	<b>20</b>
5.1	BLOCK DATA . . . . .	20
5.2	COMMON . . . . .	21
5.3	DATA . . . . .	22
5.4	SENTENCIAS DE DECLARACIÓN DE TIPO DE DATOS . . . . .	23
5.4.1	DECLARACIONES TIPO NUMÉRICO . . . . .	23
5.4.2	DECLARACIONES TIPO CARÁCTER . . . . .	24
5.5	DIMENSION . . . . .	24
5.6	EQUIVALENCE . . . . .	25
5.6.1	ARRAYS EQUIVALENTES . . . . .	25
5.6.2	CADENAS EQUIVALENTES . . . . .	28
<b>6</b>	<b>INSTRUCCIONES DE CONTROL</b>	<b>28</b>
6.1	DO . . . . .	28
6.1.1	DO indexado . . . . .	28
6.1.2	DO WHILE . . . . .	32
6.1.3	END DO . . . . .	32
6.2	END . . . . .	32
6.3	GO TO . . . . .	33
6.3.1	GOTO incondicional . . . . .	33
6.3.2	GOTO calculado . . . . .	33
6.3.3	GOTO asignado . . . . .	34
6.4	Sentencias IF . . . . .	34
6.4.1	IF aritmético . . . . .	35
6.4.2	IF lógico . . . . .	36
6.4.3	Bloques IF . . . . .	36
6.5	PAUSE . . . . .	38
6.6	RETURN . . . . .	39
6.7	STOP . . . . .	39
<b>7</b>	<b>PROCEDIMIENTOS</b>	<b>40</b>
7.1	INTRÍNSECAS . . . . .	40
<b>8</b>	<b>Nombres Específicos de las Funciones Genéricas</b>	<b>42</b>
8.1	FUNCIONES SENTENCIA . . . . .	42
8.2	SUBPROGRAMAS FUNCTION . . . . .	43

8.3	SUBROUTINAS . . . . .	44
<b>9</b>	<b>ENTRADA/SALIDA</b>	<b>44</b>
9.1	Especificaciones de formato . . . . .	45
9.2	Descriptores de formato . . . . .	46
9.2.1	Descriptores A,E,F,G,I,L . . . . .	47
9.3	Descriptores de control . . . . .	49

# 1 Introducción

El FORTRAN es uno de los lenguajes de programación más ampliamente utilizados, especialmente entre la comunidad científica. Fué el primer lenguaje de alto nivel que fué aceptado.

Su nombre deriva de FORmula TRANslation (traducción de fórmulas) y ha tenido varias revisiones.

## 1.1 Elementos de un programa fuente FORTRAN

Una unidad de programa es una secuencia de instrucciones que terminan con una sentencia END. Puede ser un programa principal o un subprograma FUNCTION (8.2), subrutina (8.3) o BLOCK DATA (5.1).

En FORTRAN las sentencias se dividen en dos clases : ejecutables y no ejecutables.

Las sentencias ejecutables describen las “acciones” del programa . Las no ejecutables describen la distribución de los datos y sus características, o dan información sobre la edición y conversión de datos.

La siguiente figura muestra el orden en el que han de aparecer las instrucciones en una unidad de programa FORTRAN :

Líneas de comentario	PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA		
	FORMAT	PARAMETER	IMPLICIT
			<i>Instrucciones de tipo:</i> INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER <i>Otras instrucciones de especificación:</i> COMMON, DIMENSION, EQUIVALENCE, EXTERNAL, INTRINSIC, SAVE
	ENTRY	DATA	<i>Definición funciones sentencia</i>
			<i>Instrucciones ejecutables:</i> BACKSPACE, CALL, CLOSE, CONTINUE, DO, ELSE, ELSE IF, END IF, GO TO, IF, INQUIRE, OPEN, READ, RETURN, REWIND, STOP, WRITE, <i>instrucciones asignación.</i>
	END		

En ésta figura, las líneas verticales separan tipos de instrucciones que se pueden mezclar. Por ejemplo, las DATA se pueden mezclar con las sentencias ejecutables. Sin embargo, las

líneas horizontales indican tipos de sentencias que no pueden aparecer mezcladas. Por ejemplo, las instrucciones de declaración no se pueden solapar con las ejecutables.

Una sentencia **PROGRAM** puede aparecer solo como la primera instrucción de un programa principal. La primera sentencia de un subprograma ha de ser una **FUNCTION**, **SUBROUTINE** o **BLOCK DATA**.

Así, por ejemplo, dentro de una unidad de programa:

1. las **FORMAT** pueden aparecer en cualquier sitio.
2. todas las funciones sentencia han de preceder a todas las sentencias ejecutables.
3. las **DATA** pueden aparecer en cualquier parte después de las sentencias de especificación.
4. la última línea de una unidad de programa ha de ser una **END** .

#### 1.1.1 Nombres simbólicos

Identifican entidades dentro de una unidad de programa. Es una string de letras, dígitos y subrayado (-). El primer carácter ha de ser una letra.

Una gran parte de los compiladores permite una longitud de 32 caracteres para un nombre simbólico.

#### 1.1.2 Comentarios

Se pueden introducir en el fuente poniendo la letra **C** o un asterisco **\*** en la columna 1 de la línea.

Una línea conteniendo solo espacios es también considerada una línea de comentario.

#### 1.1.3 Conjunto de caracteres FORTRAN

El conjunto de caracteres admisible para FORTRAN es el siguiente :

- Las letras

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- los dígitos 0 1 2 3 4 5 6 7 8 9

- los caracteres especiales



Carácter	Nombre
	Espacio en blanco
=	Igual
+	Más
-	Menos
*	Asterisco
/	División (slash)
(	Paréntesis izquierdo
)	Paréntesis derecho
,	Coma
.	Punto decimal
\$	Símbolo dólar
:	Dos puntos
'	Apóstrofe

Los caracteres alfabéticos en minúsculas, el signo de admiración(!), el subrayado(\_) y las comillas (“) son extensiones al standard ANSI FORTRAN 77.

Se pueden intercalar espacios para mejorar la legibilidad del código fuente.

El compilador no hace distinción entre mayúsculas y minúsculas (excepto en constantes de tipo carácter).

#### 1.1.4 Estructura de las líneas de código

Una línea de una unidad de programa es una secuencia de 72 caracteres

<i>Columnas</i>	Campo
1–5	Etiqueta numéricas, cuando existan. De 1 a 99999
6	Indicador de continuación. Carácter no blanco o espacio
7–72	Cuerpo de la sentencia
73–80	Nº en secuencia. Se ignoran

#### 1.1.5 Tipos de datos

El tipo de datos puede ser inherente a su construcción, puede venir dado implícitamente por convención o ser explícitamente declarado.

Pueden ser

<i>Tipo</i>	Bytes
BYTE	1
LOGICAL	2, 4
LOGICAL*1	1
LOGICAL*2	2
LOGICAL*4	4
INTEGER	2 , 4
INTEGER*1	1
INTEGER*2	2
INTEGER*4	4
REAL	4
REAL*4	4
REAL*8	8
DOUBLE PRECISION	8
REAL*16 (cuádruple precisión)	16
COMPLEX	8
COMPLEX*8	8
COMPLEX*16	16
DOUBLE COMPLEX	16
CHARACTER*len	len
CHARACTER*(*)	

Si no se declara explícitamente (la variable, array, función, etc.), el compilador supone que es entero o real simple según sea la primera letra del identificador o nombre simbólico:

- Si comienza por I, J, K, L, M, N supone que es de tipo **entero**
- en otro caso, letras A--H, O--Z supone que es **real**

## 2 Datos

Pueden ser constantes, variables, arrays y subcadenas.

### 2.1 CONSTANTES

Existen ocho tipos :

Enteras, Reales, Doble precisión, Complejas, Octales, Hexadecimales, Lógicas, Tipo carácter y Tipo Hollerith. Las de tipo octal, hexadecimal y Hollerith no tienen tipo de datos. Asumen el tipo conforme al contexto en que aparecen.

### 2.1.1 ENTERAS

Son números enteros sin punto decimal. Forma :

<b>sn</b>
-----------

donde s es un signo opcional y n es una cadena de dígitos decimales. Su valor está en el rango  $-2147483648$  a  $2147483647$ .

**Ejemplo 1.** 0, -127, +32123

Si su valor está en el rango  $-32768$  a  $32767$  es tratada como tipo de datos `INTEGER*2` y si está fuera de éste rango como `INTEGER*4`.

Las constantes enteras también se pueden especificar de forma octal.

### 2.1.2 CONSTANTES REALES

Es un número con un punto decimal o exponente o ambos. Puede ser positiva, cero o negativa y puede tener precisión simple (`REAL*4`), doble precisión (`REAL*8`) o cuádruple precisión (`REAL*16`). Con mayor detalle :

- Precisión simple (`REAL*4`) puede ser cualquiera de
  - una constante básica real
  - una constante básica real seguida por un exponente decimal
  - una constante entera seguida por un exponente decimal

Una constante básica real es una string de dígitos decimales con una de las formas :

<b>s.n</b>	<b>sn.n</b>	<b>sn.</b>
------------	-------------	------------

donde s es un signo opcional y n una cadena de dígitos decimales. El exponente tiene la forma `Esn`, donde n es una constante entera.

El exponente representa una potencia de 10 por la que se ha de multiplicar.

Por tanto, una constante en simple precisión puede adoptar una de la siguientes formas

sn.nEsn	s.nEsn	sn.Esn	snEsn
---------	--------	--------	-------

**Ejemplo 2.**  $1.0E6 = 1.0 * 10 * *6 = 10^6$

Ocupa 4 bytes y representa un número real con un grado de precisión de siete u ocho dígitos decimales. El signo menos (-) puede aparecer delante de la mantisa y entre la letra E y el exponente.

La magnitud ha de estar en el rango

$$\pm 1.175494 \times 10^{-38} \text{ y } \pm 3.402823 \times 10^{+38}$$

**Ejemplo 3.** 3.14159, 6217. , -.00127, +5.0E3, 2E-3

- Doble precisión (REAL\*8, DOUBLE PRECISION)

Es una constante básica real o una constante entera seguida por un exponente decimal de la forma:

Dsn
-----

donde s es un signo opcional y n una cadena de dígitos decimales. Ocupa 8 bytes y el grado de precisión es de 14 a 17 dígitos decimales.

La magnitud ha de estar entre  $\pm 1.79769 \times 10^{+308}$  y  $\pm 2.22507 \times 10^{-308}$ .

Por tanto, una constante en doble precisión adoptará una de la siguientes formas

sn.nDsn	s.nDsn	sn.Dsn	snDsn
---------	--------	--------	-------

**Ejemplo 4.** 1234567890D+5, +2.71828182846182D00 , -72.5D-15, 1D0, 123456789.D0 +2.34567890123D-7, -1D+300

- Cuádruple precisión (REAL\*16)

Es una constante básica real o una constante entera seguida por un exponente decimal de la forma :

Qsn
-----

donde s es un signo opcional y n una cadena de dígitos decimales. Ocupa 16 bytes y el grado de precisión es de 32 a 34 dígitos decimales.

La magnitud ha de estar entre  $\pm 3.362103 \times 10^{-4932}$  y  $\pm 1.189731 \times 10^{4932}$ .

Por tanto, una constante en doble precisión tendrá una de la siguientes formas

sn.nQsn	s.nQsn	sn.Qsn	snQsn
---------	--------	--------	-------

**Ejemplo 5.** 123456789Q4000, -1.23Q-400 , +2.72Q0

### 2.1.3 CONSTANTES COMPLEJAS

Consisten en un par de constantes enteras o reales separadas por una coma y encerradas entre paréntesis. La primera cte. representa la parte real y la segunda la parte imaginaria.

- Complejo simple (COMPLEX\*8)

Tiene la forma

$$(c,c)$$

donde  $c$  es una constante entera (2 ó 4 bytes) o REAL\*4. Ocupa 8 bytes

**Ejemplo 6.** (1.7039, -1.70391), (+12739E3,0.), (1,2), (45.9,12)

- Complejo doble (COMPLEX\*16, DOUBLE COMPLEX)

Tiene la forma

$$(c,c)$$

donde  $c$  es una constante entera (2 ó 4 bytes) , REAL\*4 o REAL\*8 y *al menos una de las constantes del par será REAL\*8*. Ocupa 16 bytes

**Ejemplo 7.** (1.7039D0, -1.7039D0), (+12739D3,0.D0)

### 2.1.4 CONSTANTES OCTALES Y HEXADECIMALES

Representan una alternativa a la representación de constantes numéricas.

Una constante octal es una cadena de dígitos octales encerrados entre apóstrofes y seguidos por el carácter alfabético 'O'.

Tiene la forma:

$$O'c_1c_2 \dots c_n'$$
 donde  $0 \leq c_i \leq 7$ , para  $1 \leq i \leq n$ .

Una constante hexadecimal es una cadena de dígitos hexadecimales encerrados entre apóstrofes y seguidos por el carácter alfabético 'X'.

Tiene la forma:

$$X'c_1c_2 \dots c_n'$$
 donde  $c_i \in \{0, 1, \dots, 9, A, \dots, F\}$ , para  $1 \leq i \leq n$ .

Tanto en las constante octales como en las hexadecimales se ignoran los ceros iniciales si los hubiera. Se puede especificar hasta 128 bits (43 dígitos octales, 32 hexadecimales).

### 2.1.5 CONSTANTES LÓGICAS

Especifica un valor lógico, verdadero o falso. Por tanto, solo son posibles las dos constantes lógicas :

`.TRUE.` o `.FALSE.`

*Los puntos delimitadores son necesarios.*

### 2.1.6 CONSTANTES TIPO CARÁCTER

Es una cadena de caracteres ASCII imprimibles encerrados entre apóstrofes. Tiene la forma:

`'c1c2...cn'` donde  $c_i$  es un carácter imprimible.

Los apóstrofes se puede sustituir por comillas (' ').

Dentro de la constante tipo carácter, el carácter apóstrofe se representa por dos apóstrofes consecutivos (sin espacios entre ambos). La longitud de una constante tipo carácter es el número de caracteres existentes entre los dos apóstrofes delimitadores, salvo los dos apóstrofes consecutivos que representan un apóstrofe.

## 2.2 Variables

Se representan por un nombre simbólico asociado a determinadas posiciones de memoria.

Se clasifican, al igual que las constantes, por su tipo de datos.

El tipo de datos de una variable indica el tipo de datos que representa, su precisión y sus requerimientos de memoria.

Cuando se asigna cualquier tipo de datos a una variable, estos son convertidos, si es necesario, al tipo de datos de la variable.

Se puede establecer el tipo de datos de una variable mediante una instrucción de declaración de tipo, por una **IMPLICIT** o por implicación según reglas predefinidas. Pueden ser definidas antes de la ejecución del programa por una instrucción **DATA** o durante la ejecución por una sentencia de asignación o de entrada de datos.

- Por especificación, indicando explícitamente el tipo de datos, mediante una sentencia del tipo

`INTEGER{*2, *4}, REAL{*4, *8}, DOUBLE PRECISION, etc.`

Una declaración explícita tiene prioridad sobre el tipo definido por una sentencia **IMPLICIT**, y también sobre una declaración por implicación.

- Por implicación. En ausencia de declaraciones explícitas o de sentencias **IMPLICIT**, todas las variables con nombres que comiencen con las letras I,J,K,L,M ó N se suponen enteras. Y las variables que empiecen con cualquier otra letra se suponen **REAL\*4**.

## 2.3 Arrays

Es un grupo de posiciones de memoria contiguas asociadas a un solo nombre simbólico, el nombre del array. Pueden tener de una a siete dimensiones.

Se pueden declarar por sentencias explícitas o por instrucciones **DIMENSION**, **COMMON**. Estas instrucciones contienen declaradores de array que definen el nombre, n° de dimensiones y n° de elementos de cada dimensión del array en cuestión.

La definición, igual que para variables. La única salvedad es que se puede leer un array entero con una sola instrucción de lectura.

### 2.3.1 Declaradores de array

Especifica el nombre simbólico que identifica al array e indica las propiedades de éste array. Tiene la forma :

$$\boxed{\mathbf{a}(\mathbf{d}[\mathbf{d}]\dots)} \quad \text{donde}$$

- **a** es el nombre simbólico del array
- **d** es un declarador de dimensión, pudiendo especificar el límite inferior y superior del subíndice en la forma  $\boxed{[d_i:]d_s}$  donde  $d_i$  = límite inferior de la dimensión y  $d_s$ =límite superior de la dimensión especificada.

El número de declaradores de dimensión indica el n° de dimensiones del array. El valor del límite inferior puede ser negativo, cero o positivo. Y el valor del límite superior será mayor o igual al del límite inferior.

El número de elementos en la dimensión es de  $d_s - d_i + 1$ .

Si no se especifica límite inferior, se supone igual a 1.

El límite superior de la última dimensión puede ser un asterisco(\*) indicando que tomará un valor que será pasado en una lista de parámetros de un subprograma.

Cada límite de una dimensión ha de ser una expresión aritmética entera.

Límites que no son constantes solo se pueden utilizar en un subprograma para definir arrays ajustables.

El n° de elementos de un array es el producto del número de elementos en cada dimensión.

**Ejemplo 8.** `nombre(4,-5:5,6), lista(10), m(0:0)`

### 2.3.2 Almacenamiento en memoria

El FORTRAN siempre almacena un array en memoria como una secuencia lineal de valores.

Un array unidimensional se almacena con su primer elemento en la primera posición de almacenamiento y el último elemento en la última posición de almacenamiento de la secuencia.

Un array multidimensional se almacena de forma que *los primeros subíndices varían más rápidamente que los siguientes*. Esto se llama “orden de progresión de subíndices”. Véase figura 1.

El tipo de datos de un array se especifica igual que en las variables.

Todos los elementos de un array tienen el mismo tipo de datos .

Cualquier valor asignado a un elemento de array será convertido al tipo de datos del array. En las instrucciones **COMMON**, **DATA**, **EQUIVALENCE**, **NAMelist**, **SAVE** de entrada/salida y de declaración de tipo se puede especificar el nombre del array sin subíndices, indicando que se va a usar ( o definir) la totalidad del array.

Análogamente ocurre si se usa como argumentos formales de las sentencias **FUNCTION**, **SUBROUTINE** y **ENTRY**.

En cualquier otro tipo de instrucciones no está permitido.

### 2.3.3 Subcadenas

Es un segmento contiguo de una variable tipo carácter o de un elemento de un array tipo carácter. Una referencia de subcadena tiene una de las siguientes formas :

$$\boxed{\mathbf{v}([e_1] : [e_2])} \quad \text{ o } \quad \boxed{\mathbf{a}(s[, s] \dots)([e_1] : [e_2])}$$

donde

- **v** es un nombre de una variable tipo carácter
- **a** es un nombre de un array tipo carácter
- **s** es una expresión de subíndice
- **e<sub>1</sub>** es una expresión numérica que especifica la posición del primer carácter
- **e<sub>2</sub>** es una expresión numérica que especifica la posición del último carácter

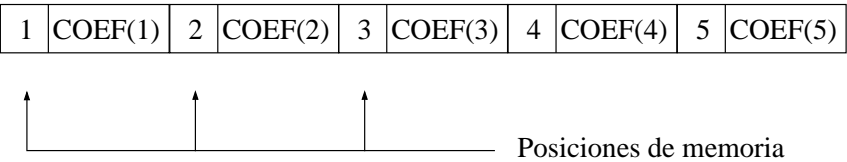
**Ejemplo 9.** si  $LABEL='XVERSUSY'$  entonces  $LABEL(2:7)$  sería *VERSUS*

Si los valores de las expresiones numéricas  $e_1$  o  $e_2$  no son de tipo entero se truncan sus partes fraccionarias antes de su uso, a efectos de convertirlos en un valor entero. Los valores de  $e_1$  y  $e_2$  cumplirán las condiciones :

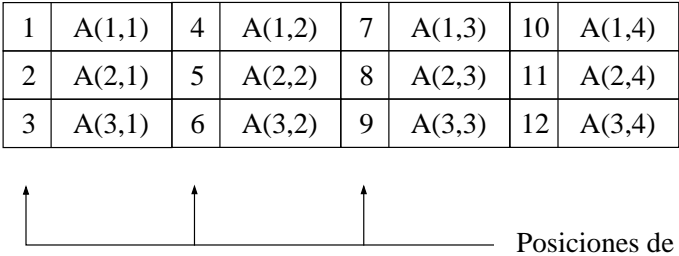
$$1 \leq e_1 \leq e_2 \leq long$$



Array unidimensional COEF(5)



Array bidimensional A(3,4)



Array tridimensional P(3,3,3)

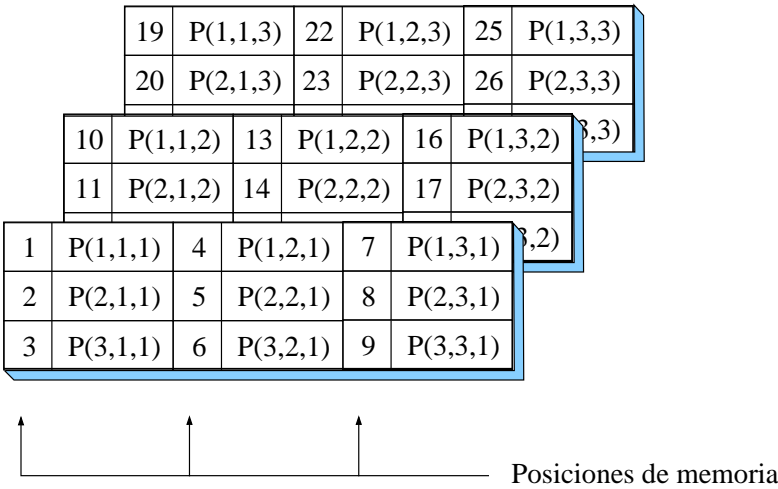


Figura 1: Almacenamiento de arrays

donde long es la longitud de la variable tipo carácter o elemento de array.

Si se omite  $e_1$ , el compilador supone que es 1. Y si se omite  $e_2$ , supone que  $e_2$  es igual a long.

**Ejemplo 10.** *NAME(1,3)(:7) especifica la subcadena que comienza en la posición 1 y termina en la 7ª del elemento de array tipo carácter NAME(1,3).*

## 3 EXPRESIONES

Consiste en una combinación de operadores con constantes, variables y/o referencias a función.

Pueden ser aritméticas, de carácter, relacionales o lógicas, produciendo respectivamente valores aritméticos, de carácter, o lógicos.

### 3.1 Expresiones aritméticas

Se forman con elementos aritméticos y operadores aritméticos. Su evaluación produce un único valor numérico. Un elemento aritmético puede ser :

- una referencia numérica escalar
- una expresión aritmética entre paréntesis
- una referencia a función numérica

El término “numérico” incluye datos lógicos ya que estos son tratados como enteros cuando se usan en un contexto aritmético. Los operadores numéricos son :

**	exponenciación
*	multiplicación
/	división
+	suma (o más unario)
-	resta (o menos unario)

Cualquier variable o elemento de array tendrá un valor definido antes de ser usado en una expresión. Las expresiones aritméticas son evaluadas en un orden determinado por la prioridad asignada a cada operador. Estas son :

OPERADOR	PRECEDENCIA
**	primera
*, /	segunda
+, -	tercera

Cuando dos o más operadores de igual prioridad (tales como + y -) aparecen en una expresión, la evaluación se realiza de izquierda a derecha, salvo la exponenciación, que se realiza de derecha a izquierda.

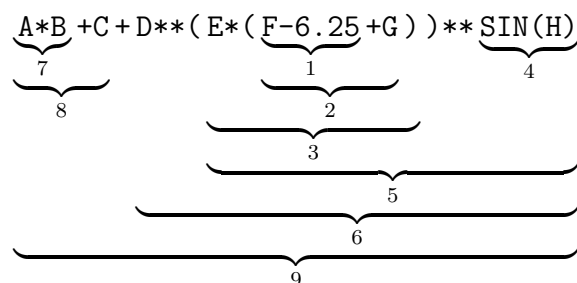
$A**B**C$  se evalúa como  $A**(B**C)$ .

Normalmente no se permite que dos operadores aparezcan en sucesión. Cuando el segundo operador es unario (+ ó -), el FORTRAN lo permite.

### 3.1.1 Uso de paréntesis

Se pueden usar paréntesis para cambiar el orden de evaluación. Cuando aparecen, la parte encerrada entre paréntesis se evalúa primero, y su valor es usado en la evaluación del resto de la expresión.

Así,  $A*B+C+D**(E*(F-6.25+G))**SIN(H)$  se evalúa en el siguiente orden:



### 3.1.2 Tipos de datos en una expresión aritmética

Si cada elemento de una expresión aritmética es del mismo tipo de datos, el valor de la expresión será del mismo tipo. Sin embargo, si se combinan diferentes tipos, el valor resultante tendrá un tipo que depende del rango asociado a cada tipo de datos, siendo del tipo de datos de mayor rango que aparece en la expresión.

Así, p.e., el tipo de datos resultante de una expresión entre un entero y un real será real. Si un tipo es `COMPLEX*8` y otro `REAL*8` o `REAL*16` su resultado será `COMPLEX*16`.

Tipo de datos	Rango
LOGICAL*1	1 (Menor)
LOGICAL*2	2
LOGICAL*4	3
INTEGER*1	4
INTEGER*2	5
INTEGER*4	6
REAL*4 (REAL)	7
REAL*8 (DOUBLE PRECISION)	8
REAL*16	9
COMPLEX*8	10
COMPLEX*16 (DOUBLE COMPLEX)	11 (Mayor)

El tipo de datos de una expresión será del tipo de datos del resultado de la última operación en dicha expresión. El tipo de datos se determina de acuerdo con las siguientes convenciones :

**operaciones enteras** se realizan solo con elementos enteros (las entidades lógicas se tratan como enteros). En aritmética entera se trunca la parte fraccionaria de la división :

Ejemplo: El valor de  $1/4+1/4+1/4+1/4$  es 0.

Para ser exactos, en el caso de la división entera, su valor es un entero, que se obtiene así:

1. si la magnitud del cociente matemático es menor que uno (1), entonces el cociente entero es cero. Por ejemplo, el valor de  $11/12$  es cero.
2. si la magnitud del cociente matemático es mayor o igual a 1, el cociente entero es el mayor entero que no excede la magnitud del cociente matemático y cuyo signo es el mismo que el del cociente matemático. Por ejemplo, el valor de  $-7/2$  es  $-3$ .

**operaciones reales** se realizan solo con elementos reales o combinaciones de reales, enteros y lógicos. Los elementos enteros se convierten a reales y la expresión se evalúa entonces usando aritmética real.

**operaciones con REAL\*8 y REAL\*16** los otros elementos de la expresión se convierten al tipo correspondiente de alta precisión, haciendo su valor a la parte más significativa

de la nueva representación y haciendo cero su parte menos significativa. La expresión se evalúa en aritmética de alta precisión.

**convertir** un elemento real a alta precisión no incrementa su precisión. P.e., una variable real con valor 0.3333333 se convierte a 0.3333333432674408 y no a 0.3333333000000000D0 ó a 0.3333333333333333D0.

**operaciones complejas** los elementos enteros se convierten a reales. El elemento **REAL** o **REAL\*8** así obtenido será la parte real de un número complejo, con parte imaginaria cero. La expresión se evalúa entonces usando aritmética compleja siendo entonces el resultado de tipo complejo.

## 3.2 Expresiones de carácter

Consisten en elementos de carácter y operadores de carácter. Su evaluación conduce a un único valor de tipo carácter.

Un elemento de carácter puede ser :

- una referencia escalar de carácter
- una subcadena
- una expresión tipo carácter, opcionalmente entre paréntesis
- una referencia a una función de carácter

El único operador tipo carácter es el operador de concatenación (//)

Los paréntesis no afectan al valor de una expresión de carácter.

Si los elementos contienen espacios, estos son incluidos en el valor de la expresión de carácter. P.e. :

'ABC '// 'D E' '// 'F '    tiene el valor    'ABC D EF '

## 3.3 Expresiones relacionales

Consiste en dos expresiones aritméticas o dos expresiones de carácter separadas por un operador relacional. El valor de la expresión es *verdadera* o *falsa*.

Los operadores relacionales son :

operador	significado
.LT.	Menor que (Less than)
.LE.	Menor o igual (Less than or equal to)
.EQ.	Igual a (Equal to)
.NE.	Distinto a (Not equal to)
.GT.	Mayor que (Greater than)
.GE.	Mayor o igual (Greater than or equal to)

En expresiones complejas pueden intervenir solo los operadores .EQ. y .NE..

En expresiones relacionales aritméticas, las expresiones aritméticas se evalúan primero.

Análogamente ocurre con las expresiones de carácter.

En las expresiones tipo carácter “menor que” y “mayor que” significan “precede” y “sucede” en la tabla de caracteres ASCII respectivamente.

Si dos expresiones de carácter no tienen la misma longitud, la más pequeña se rellena a espacios por la derecha.

Todos los operadores relacionales tienen la misma prioridad.

Si se comparan dos expresiones numéricas de diferente tipo de datos, la expresión de menor rango se convierte antes de la comparación a la de rango superior.

### 3.4 Expresiones lógicas

Puede ser un único elemento lógico o una combinación de elementos lógicos y operadores lógicos. Una expresión lógica produce un único valor lógico, verdadero o falso.

Un elemento lógico puede ser :

- una referencia escalar lógica o entera
- una expresión relacional
- una expresión lógica o entera entre paréntesis
- una referencia a función lógica o entera

Los operadores lógicos son :

operador	significado
.AND.	conjunción lógica
.OR.	disyunción inclusiva
.NEQV.	disyunción exclusiva
.XOR.	disyunción exclusiva (extensión al ANSI)
.EQV.	equivalencia lógica
.NOT.	negación lógica (unario)

Cuando un operador lógico actúa con elementos lógicos, el tipo de datos resultante es lógico (es lógico).

Cuando opera con elementos enteros, la operación lógica se realiza bit a bit en las correspondientes representaciones binarias y el tipo de datos resultante es entero.

Una expresión lógica se evalúa de acuerdo al orden de precedencia de sus operadores, que aparece en la siguiente lista.

Esta lista relaciona los operadores que pueden aparecer en una expresión, y el orden en que son evaluados :

Operadores	Precedencia
**	(mayor)
*, /	
+, -	
//	
operadores relacionales	↓
.NOT.	
.AND.	
.OR.	
.XOR., .EQV., .NEQV.	(menor)

Los operadores de igual rango se evalúan de izquierda a derecha, salvo la exponenciación, que se evalúa de derecha a izquierda.

Como en las expresiones aritméticas, se pueden usar los paréntesis para alterar la secuencia normal de evaluación.

Dos operadores lógicos no pueden aparecer consecutivamente, salvo que el segundo sea un .NOT..

**Ejemplo 11.** *La expresión*

`A*B +C*ABC .EQ. X*Y+OM/ZZ .AND. .NOT. X+8 .GT. TT`

*se evalúa en el siguiente orden:*

`((A*B)+(C*ABC)) .EQ. ((X*Y)*(OM/ZZ))) .AND. (.NOT. ((X+8) .GT. TT))`

## 4 INSTRUCCIONES DE ASIGNACIÓN

Definen el valor de una variable, elemento de array, registro, elemento de registro o subcadena.

### 4.1 Asignación aritmética

Tiene la forma

$$\boxed{v = e}$$

donde

**v** es una referencia numérica escalar (variable, elemento de array, etc. ).

**e** es una expresión aritmética

Ejemplos : `PI=3.14159`, `CONT=CONT+1`, `ALFA=-1./(2.*X)+A`

Si **v** tiene el mismo tipo de datos que la expresión aritmética de la derecha, se asigna el valor directamente.

Si los tipos de datos son diferentes, el valor de la expresión se convierte al tipo de datos de la entidad de la izquierda.

### 4.2 Asignación lógica

Asigna el valor de la expresión lógica de la derecha a la referencia lógica escalar que está a la izquierda. Véase la tabla anterior para las reglas de conversión.

Tiene la forma

$$\boxed{v = e}$$

donde

**v** es una referencia lógica escalar

**e** es una expresión lógica

Ejemplos :

`FINPAG=.FALSE.`

`AGORDO=A.GT.B .AND. A.GT.C .AND. A.GT.D`

`IMPRIME=LINEA.LE.65 .AND. .NOT. FINPAG`



### 4.3 Asignación de carácter

Tiene la forma

$$\boxed{v = e}$$

donde

**v** es una referencia de carácter escalar

**e** es una expresión de carácter

Si  $\text{LEN}(e) > \text{LEN}(v) \implies$  e se trunca por la derecha

Si  $\text{LEN}(e) < \text{LEN}(v) \implies$  e se rellena a blancos por la derecha

Ejemplos :

```
FILE='NEWTON'  
NOMBRE81)=X// ' RODRIGUEZ'  
TEXTO(I, J+1)(2:N-1)=LINEA(I)//A
```

### 4.4 Instrucción ASSIGN

Asigna una etiqueta de instrucción a una variable entera. La variable puede ser usada entonces para bifurcar con una instrucción **GOTO** asignado o para asignar un especificador de formato a una instrucción de E/S.

Tiene la forma

$$\boxed{\text{ASSIGN } s \text{ TO } v}$$

donde

**s** es la etiqueta de una instrucción ejecutable o una **FORMAT**.

**v** es una variable entera

La instrucción **ASSIGN** asigna el número de instrucción a una variable. Es similar a una expresión aritmética, con una excepción: la variable queda indefinida como variable entera, no pudiendo entrar en operaciones aritméticas.

Ejemplos :

```
ASSIGN 10 TO NSTART  
ASSIGN 99999 TO KSTOP  
ASSIGN 250 TO ERROR
```

## 5 INSTRUCCIONES DE ESPECIFICACIÓN

Son instrucciones no ejecutables que se usan para reservar memoria, e inicializar variables, arrays, records y estructuras, y también para definir otras características de los nombres simbólicos usados en el programa. Son las siguientes:

### 5.1 BLOCK DATA

Esta instrucción, seguida por una serie de instrucciones de especificación, asigna valores iniciales a entidades de bloques comunes con nombre y a la vez, establece y define estos bloques.

Tiene la forma

BLOCK DATA nomb

donde

**nomb** es un nombre simbólico.

En una BLOCK DATA se pueden usar COMMON, DATA, DIMENSION, EQUIVALENCE, IMPLICIT, PARAMETER, RECORD, SAVE, declaraciones de estructuras y sentencias de declaración de tipo.

Las instrucciones de especificación que siguen a la BLOCK DATA establecen y definen bloques comunes, asignan variables, arrays y records a estos bloques, y asignan valores iniciales a las variables, arrays y records.

Constituye una clase especial de subprograma, en el que no pueden haber instrucciones ejecutables y ha de finalizar con una END.

Se usa principalmente para inicializar las entidades de una o varias COMMON con nombre.

```
BLOCK DATA BLOQUE1
INTEGER S,X
LOGICAL T,W
DOUBLE PRECISION U
DIMENSION R(3)
COMMON /AREA1/R,S,T,U/ AREA2/W,X,Y
DATA R/1.0,2*2.0/,T/.FALSE./, U/0.21D-7/,W/.TRUE./, Y/3.5/
END
```

## 5.2 COMMON

Define una o más áreas contiguas de memoria, o bloques. También define el orden en el que las variables, arrays y records aparecen en un bloque común.

Dentro de un programa, puede haber un bloque **COMMON** sin nombre, pero si existen más, se les ha de asignar un nombre. Esta instrucción, seguida por una serie de instrucciones de especificación, asigna valores iniciales a entidades de bloques comunes con nombre y a la vez, establece y define estos bloques.

Proporciona un medio de asociar entidades en diferentes unidades de programa. Esto permite a diferentes unidades definir y referenciar los mismos datos sin tener que pasarlos como argumentos en llamadas subrutinas, y también compartir unidades de almacenamiento.

La sintaxis es:

**COMMON** [/nomb/] list [[,]/[nomb1]/list1] ...

donde

**nomb** es un nombre simbólico. Pueden ser blancos, en cuyo caso se omiten el par de /'s.

**list** es una lista de nombres de variables, nombres de arrays y declaradores de array.

Cuando se declaran bloques comunes con el mismo nombre en diferentes unidades de programa, estos comparten la misma área de memoria cuando se combinan en un programa ejecutable.

Las entidades que aparecen en una **COMMON** de una unidad de programa han de coincidir en tipo y número con las que aparezcan en una **COMMON** con el mismo nombre de otra unidad de programa.

Ejemplo:

<u>Programa principal</u>	<u>Subprograma</u>
<b>COMMON</b> COLOR,X/BLOQUE1/KILO,Q	<b>SUBROUTINE</b> FIGURA
.	<b>COMMON</b> /BLOQUE1/LIMA,2/ /ALFA,BETA
.	.
.	.
<b>CALL</b> FIGURA	<b>RETURN</b>
	<b>END</b>

en el que hay dos bloques, uno sin nombre que asociaría las variables **COLOR,X**  $\iff$  **ALFA,BETA** y otro con nombre **BLOQUE1** que asociaría **KILO,Q**  $\iff$  **LIMA,R**.

### 5.3 DATA

Asigna valores iniciales a variables, arrays, elementos de array y subcadenas antes de la ejecución del programa.

Sintaxis:

`DATA nlista/clista/[[,]nlista/clista/] ...`

donde

**nlista** es una lista de nombres de variables, arrays, elementos de array, nombres de sub-strings, o DO implícitos, separados por comas. La forma de un DO implícito en una DATA es

`(dlista, i=n1,n2[,n3])`

donde

**dlista** es una lista de uno o más nombres de elementos de array, nombres de sub-strings o DO implícitos, separados por comas.

**i** es el nombre de una variable entera.

**n1,n2,n3** son expresiones enteras constantes, salvo que contenga variables de un DO implícito.

(Ejemplo: DATA ((ELEM(I,J,K), I=1,5), J=1,5), K=1,5)/125\*0.0/)

**clista** es una lista de constantes, con una de las formas  $c$ ,  $n * c$ , donde

**c** es una constante o nombre de una constante

**n** define el número de veces que se va a asignar el mismo valor a las sucesivas entidades en la nlista asociada. Ha de ser una constante entera ( $\neq 0$ ) o nombre de constante entera.

En una DATA , los valores son asignados uno a uno en el orden en que aparecen, de izquierda a derecha.

Puede aparecer nombres de array sin subíndices. En éste caso, habrá el número suficiente de valores en la lista asociada de constantes. Los elementos del array son inicializados en el orden de *sus posiciones de memoria*.

Ejemplos:

```
DATA A,B,C,D/3*5.0,7.2/
```

```
DIMENSION SUM(20)
```

```
LOGICAL A,B
```

```
DATA ((X(J,I), I=1,J), J=1,5) /15*0./
```

```
DATA SUM/20*0.0/,I,J,K,L/4*5,2/
```

```
DATA A,B/2*.TRUE./
```

## 5.4 SENTENCIAS DE DECLARACIÓN DE TIPO DE DATOS

Definen explícitamente el tipo de datos de los nombres simbólicos especificados. Existen dos formas: declaraciones de tipo numérico y declaraciones de tipo carácter.

Se pueden inicializar datos en el momento de la declaración de tipo poniendo el valor entre barras (/) inmediatamente después del nombre de la variable o array que se quiere inicializar. La asignación es igual que en `DATA`.

Las sentencias de declaración siguen las siguientes reglas de sentido común:

- precederán a cualquier instrucción ejecutable
- el tipo de datos se declara una sola vez
- una declaración d tipo no puede cambiar el tipo de un nombre simbólico que ha sido usado en un contexto que implícitamente supone un tipo diferente.

### 5.4.1 DECLARACIONES TIPO NUMÉRICO

Tienen la forma:

`tipo v[/clista/][,v[/clista/]] ...`

donde

**tipo** es cualquiera de `BYTE`, `LOGICAL`, `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `DOUBLE COMPLEX`.

**v** es el nombre simbólico de una constante, variable, array, función sentencia, subprograma `FUNCTION` o declarador de array.

**clista** es una lista de constantes, como en `DATA`.

El nombre simbólico puede ser seguido por un especificador de longitud de la forma `*s`, donde `s` es una de las longitudes aceptables del tipo de datos declarado.

```
INTEGER CONTAD,MATRIZ(4,4),SUMA
```

```
REAL INTEGR,NDERIV
```

LOGICAL SWITCH

INTEGER\*2 I,J,K, M12\*4, Q, IVEC\*4(10)

REAL\*8 WX1,WXZ,WX3\*4,WX5,WX6\*8

REAL\*16 PI/3.14159Q0/, E/2.72Q0/, QARRAY(10)/5\*0.0,5\*1.0/

## 5.4.2 DECLARACIONES TIPO CARÁCTER

Son de la forma:

`CHARACTER[*long[,]] v[*long][/clista/][,v[*long][/clista/]] ...`

donde

**v** es el nombre de una constante, variable, array, función sentencia, subprograma FUNCTION o declarador de array.

**long** es una constante entera sin signo, una expresión entera constante entre paréntesis o un asterisco (\*) entre paréntesis. Su valor especifica la longitud.

CHARACTER SOC(100)\*9, NOMB\*4/'PEPE' /

PARAMETER (LONGIT=4)

CHARACTER\*(4+LONGIT) A,B

## 5.5 DIMENSION

Define el número de dimensiones de un array y el número de elementos de cada dimensión.

Tiene la forma:

`DIMENSION a(d)[,a(d)] ...`

donde

**a(d)** es un declarador de array.

Ejemplo: DIMENSION ARRAY(4,-4:4), MATRIX(5,0:5,5)

Se pueden utilizar declaradores de array en las sentencias de declaración de tipo y las COMMON.

Ejemplos:

DIMENSION X(5,5), Y(4,85), Z(100)

subroutine aproc(a1,a2,n1,n2,n3)

DIMENSION MARCA(4,4,4,4)

dimension a1(n1:n2), a2(n3:\*)

## 5.6 EQUIVALENCE

Dadas dos o más entidades de la misma unidad de programa, ésta instrucción las asocia total o parcialmente a las mismas posiciones de memoria.

Tiene la forma:

`EQUIVALENCE (lista)[,(lista)] ...`

donde

**lista** es una lista de variables, nombres de array, elementos de array o referencias a subcadenas, separadas por comas. Se han de especificar al menos dos entidades en cada lista.

Se pueden hacer equivalentes variables de diferente tipo de datos. Por ejemplo, si se hace a una variable entera equivalente a una variable compleja, la entera comparte las mismas posiciones de memoria con la parte real de la variable compleja.

Ejemplos:

1. `DOUBLE PRECISION DVAR`

`INTEGER*2 IARR(4)`

`EQUIVALENCE (DVAR,IARR(1))`

hace que los cuatro elementos del array entero `IARR` ocupen las mismas posiciones que `DVAR` (doble precisión).

2. `CHARACTER CLAVE*16, DNI*10`

`EQUIVALENCE (CLAVE,DNI)`

hace que el primer carácter de las variables `CLAVE` y `DNI` tengan las mismas posiciones de memoria. `DNI` es equivalente a `CLAVE(1:10)`.

### 5.6.1 ARRAYS EQUIVALENTES

Cuando se hacen equivalentes un determinado elemento de un array con otro determinado de otro array, `EQUIVALENCE` también hace equivalentes los otros elementos de los dos arrays. Así, si el tercer elemento de un array con siete elementos se hace equivalente al primer elemento de otro array, los últimos cinco elementos del primer array se solapan con los primeros cinco elementos del segundo array ( si los elementos de ambos arrays tienen igual “tamaño”).

Por ejemplo:

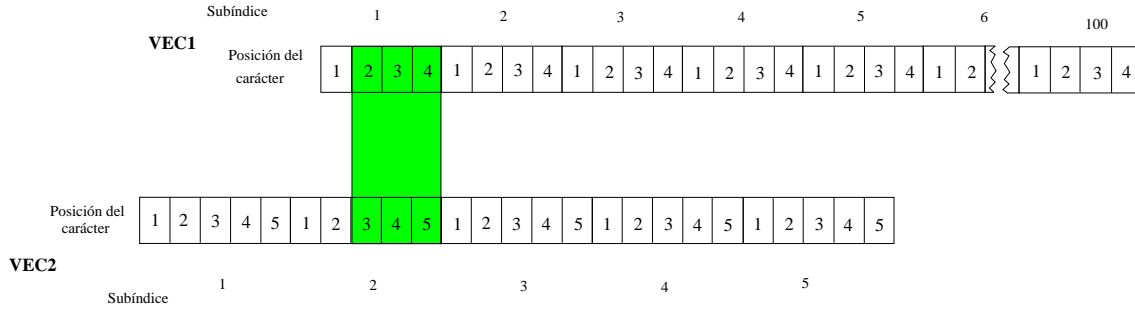


Figura 2: Almacenamiento de arrays

```
DIMENSION DOBLE(2,2), TRIPLE(2,2,2)
EQUIVALENCE (DOBLE(2,2),TRIPLE(1,2,2))
```

origina la siguiente equivalencia:

Array TRIPLE		Array DOBLE	
Elemento array	Número de elemento	Elemento array	Número elemento
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		
TRIPLE(2,2,1)	4	DOBLE(1,1)	1
TRIPLE(1,1,2)	5	DOBLE(2,1)	2
TRIPLE(2,1,2)	6	DOBLE(1,2)	3
TRIPLE(1,2,2)	7	DOBLE(2,2)	4
TRIPLE(2,2,2)			

Hacen lo mismo:

```
EQUIVALENCE(DOUBLE,TRIPLE(2,2,1))
EQUIVALENCE(TRIPLE(1,1,2),DOBLE(2,1))
```

Análogamente, se pueden hacer equivalentes a arrays que no tengan la unidad como subíndice más pequeño. Por ejemplo, un array  $A(2:3,4)$  se puede hacer equivalente a  $B(2:4,4)$  con `EQUIVALENCE (a(3,4),b(2,4))`.



Array B		Array A	
Elemento array	Número de elemento	Elemento array	Número elemento
B(2,1)	1		
B(3,1)	2		
B(4,1)	3	A(2,1)	1
B(2,2)	4	A(3,1)	2
B(3,2)	5	A(2,2)	3
B(4,2)	6	A(3,2)	4
B(2,3)	7	A(2,3)	5
B(3,3)	8	A(3,3)	6
B(4,3)	9	A(2,4)	7
B(2,4)	10	A(3,4)	8
B(3,4)	11		
B(4,4)	12		

Unicamente en una **EQUIVALENCE** se pueden referenciar un elemento de array con un solo subíndice, incluso si éste array es multidimensional.

Por ejemplo, la instrucción **EQUIVALENCE (DOBLE(4), TRIPLE(7))** hace lo mismo que en la primera tabla.

### 5.6.2 CADENAS EQUIVALENTES

Análogamente al caso anterior, cuando se hacen equivalentes dos subcadenas, asociando un carácter de una con otro de la segunda, EQUIVALENCE también asocia los otros caracteres de las entidades correspondientes.

Por ejemplo,

```
CHARACTER NOMB*16, CLAVE*9
EQUIVALENCE (NOMB(10:13), CLAVE(2:5))
```

o también EQUIVALENCE (NOMB(9:9), CLAVE(1:1)).

Si las referencias a subcadenas son elementos de array, EQUIVALENCE también establece equivalencias en todo el array.

Por ejemplo,

```
CHARACTER VEC1(100)*4, VEC2(5)*5
EQUIVALENCE (VEC1(1)(2:4), VEC2(2)(3:5))
```

No se puede usar EQUIVALENCE para asignar las mismas posiciones a dos o más subcadenas que comienzan en diferentes posiciones de carácter dentro de una misma variable ipo carácter o array tipo carácter.

**NOMB**

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

**CLAVE**

1
2
3
4
5
6
7
8
9

## 6 INSTRUCCIONES DE CONTROL

Se utilizan para transferir el control a un punto de la misma unidad de programa o a otra unidad de programa. También controlan el procesamiento iterativo, la suspensión de la ejecución del programa y la terminación del mismo.

### 6.1 DO

#### 6.1.1 DO indexado

Controla el procesamiento iterativo, o sea, las instrucciones de su rango se ejecutan un número especificado de veces. Tiene la forma:

DO [s[,]] v = e <sub>1</sub> , e <sub>2</sub> [,e <sub>3</sub> ]
--

donde

**s** es la etiqueta de una instrucción ejecutable, que ha de estar en la misma unidad de programa.

**v** es una variable entera o real, que controla el bucle (índice).

**e<sub>1</sub>, e<sub>2</sub>, e<sub>3</sub>** son expresiones aritméticas

La variable **v** es la variable de control, **e<sub>1</sub>** es el valor inicial que toma **v**, **e<sub>2</sub>** es el valor final y **e<sub>3</sub>** es el incremento o paso, que no puede ser cero. Si se omite **e<sub>3</sub>**, su valor por defecto es 1.

El rango de una **DO** incluye todas las instrucciones que siguen a la misma **DO** hasta la instrucción terminal, la última del rango.

La instrucción terminal no puede ser:

- una **GOTO** incondicional o asignada
- un **IF** aritmético
- un bloque **IF**
- **ELSE** , **ELSE IF** , **END IF** , **RETURN** , **STOP**, **END** , otra **DO** .

El número de ejecuciones del rango de una **DO**, llamado contador de iteraciones viene dado por :

$$\text{MAX}(\text{INT}((e_2 - e_1 + e_3)/e_3), 0)$$

donde **INT(x)** representa la función parte entera de **x**.

Y las etapas seguidas en la ejecución son las siguientes :

1. Se evalúa  $\text{contador} = \text{INT}((e_2 - e_1 + e_3)/e_3)$
  2. Se hace  $v = e_1$
  3. Si contador es mayor que cero, entonces:
    - (a) Ejecutar las instrucciones del rango del bucle
    - (b) Asignar  $v = v + e_3$
    - (c) Decrementar el contador ( $\text{contador} = \text{contador} - 1$ ). Si contador es mayor que cero, repetir el bucle.
- No se puede alterar el valor de la variable de control dentro del rango de la **DO**.

- Se pueden modificar los valores inicial, final e incremento dentro del bucle sin que quede afectado el contador de iteraciones ( no afecta al n° de iteraciones del bucle, ya que son establecidas al inicio).
- El rango de una DO puede contener otras instrucciones DO, existiendo algunas reglas de anidación.
- Se puede transferir el control hacia afuera de un bucle DO, pero no al revés.

**Ejemplo 12.** DO 100 K=1,50,2 DO 350 J=50,-2,-2 DO 25 IVAR=1,5

```

      N=0
      DO 100 I=1,10
        J=I
        DO 100 K=1,5
          L=K
100    N=N*1
101    CONTINUE

```

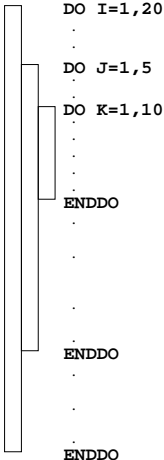
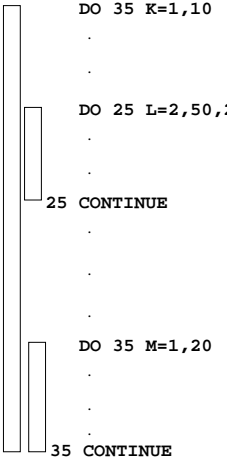
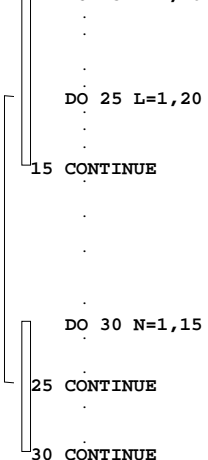
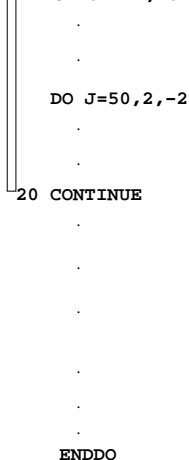
```

      N=0
      DO 200 I=1,100
        J=I
        DO 200 K=5,1
          L=K
200    N=N+1
201    CONTINUE

```

en el primer caso, los valores que quedan, después de la ejecución, son : I=11, K=6, J=10, L=5, N=50. Y en el segundo caso, I=11, K=5, J=10, N=0, L queda no definida. DO anidados

Una DO puede contener en su rango uno o más bucles DO completos. El rango de una DO más interna ha de estar totalmente incluido en el rango de cualquier otra DO más externa. Las DO anidadas pueden compartir la misma instrucción final (etiquetada numéricamente) pero no una ENDDO . Algunos ejemplos de anidación correcta e incorrecta son:

CORRECTAMENTE ANIDADOS		INCORRECTAMENTE ANIDADOS	
 <pre>       DO I=1,20         .         .         DO J=1,5           .           .         DO K=1,10           .           .         ENDDO         .         .       ENDDO         .         .       ENDDO </pre>	 <pre>       DO 35 K=1,10         .         .         DO 25 L=2,50,:           .           .         25 CONTINUE         .         .         DO 35 M=1,20           .           .         35 CONTINUE       35 CONTINUE </pre>	 <pre>       DO 15 K=1,10         .         .         DO 25 L=1,20           .           .         15 CONTINUE         .         .         DO 30 N=1,15           .           .         25 CONTINUE         .         .       30 CONTINUE </pre>	 <pre>       DO 20 I=1,10         .         .         DO J=50,2,-2           .           .         20 CONTINUE         .         .       ENDDO </pre>

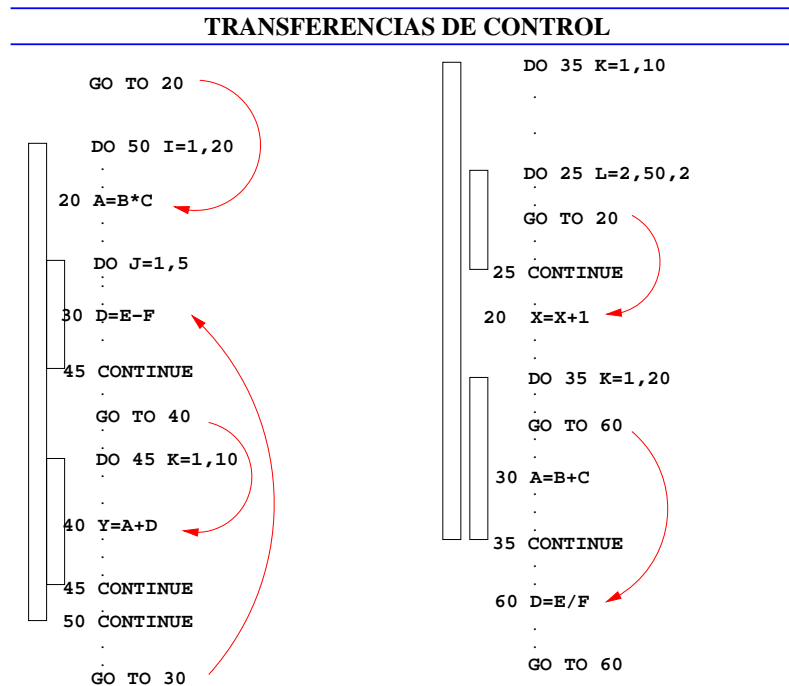


Figura 3: Transferencias y rangos

En una anidación de DO 's, se puede transferir el control desde uno más interno a otro más externo. pero en ningún caso se puede bifurcar hacia el rango de una DO desde fuera de éste rango.

Si dos o 'mas DO 's anidados comparten la misma instrucción terminal, se puede bifurcar a ésta instrucción solo desde el rango del bucle más interno.

#### Rango extendido

Un bucle DO tiene un rango extendido si contiene una instrucción de control que bifurca a un parte fuera de su rango, y después de la ejecución de una o varias instrucciones, otra instrucción ( de control) retorna el control al bucle. De ésta forma, el rango del DO se extiende para incluir las instrucciones ejecutables que están entre la de destino de la primera transferencia y la que retorna el control al bucle.

Existen un par de reglas que gobiernan su uso :

1. Una transferencia hacia el rango de una instrucción DO está permitida solo si la transfrecia se hace desde el rango extendido de ésta DO .
2. El rango extendido de una DO no debe cambiar el valor de la variable de control de la DO .

### 6.1.2 DO WHILE

Es similar al DO indexado, pero mientras que éste se ejecuta un número fijo de veces, el DO WHILE se ejecuta mientras sea verdadera la expresión lógica contenida en la instrucción. Tiene la forma:

DO [s[,]] WHILE (e)

donde

s es la etiqueta de una instrucción ejecutable que está en la misma unidad de programa.

e es una expresión lógica

Si no se pone etiqueta en la DO WHILE, ha de terminar con una ENDDO .

Se puede bifurcar hacia afuera del bucle, pero no al revés (desde el exterior hacia el interior del bucle).

### 6.1.3 END DO

Finaliza el rango de una DO o una DO WHILE, si éstas no contienen una etiqueta que indique la instrucción final del bucle. Tiene la forma:

END DO

```
DO WHILE (I.GT.J)
  ARRAY(I,J)=1.0
  I=I-1
ENDDO
```

```
DO 10 WHILE (I.GT.J)
  ARRAY(I,J)=1.0
  I=I-1
10 END DO
```

## 6.2 END

Determina el fin de una unidad de programa. Ha de ser la última línea fuente de cada unidad de programa. Tiene la forma:

END

- Puede tener etiqueta, pero no puede continuarse la ejecución.
- En un programa principal, la END provoca la finalización de la ejecución.
- En un subprograma, se ejecuta implícitamente una RETURN .

## 6.3 GO TO

Las instrucciones `GOTO` transfieren el control dentro de una unidad de programa. Dependiendo del valor de una expresión, el control se transfiere, bien a la misma instrucción siempre, o bien a una de un determinado conjunto de instrucciones.

Los tres tipos de instrucciones `GOTO` son:

- `GOTO` incondicional
- `GOTO` calculado
- `GOTO` asignado

### 6.3.1 `GOTO` incondicional

Transfiere el control a la misma instrucción cada vez que se ejecuta. Tiene la forma:

`GOTO s`

donde `s` es la etiqueta de una instrucción ejecutable que está en la misma unidad de programa de la `GOTO`.

`GOTO 99999 GOTO 100`

### 6.3.2 `GOTO` calculado

Transfiere el control a la misma instrucción según sea el valor de la expresión que aparece en la `GOTO`. Tiene la forma:

`GOTO (lista) [,] e`

donde

**lista** es una lista de una o más etiquetas de instrucciones ejecutables separadas por comas.

Se le llama *lista de transferencia*.

**e** es una expresión aritmética cuyo valor cae en el rango de 1 a  $n$  (siendo  $n$  el número de etiquetas de la lista de transferencia).

Por ejemplo, si la lista es `(30,20,30,40)` y el valor de `e` es 2, se transfiere el control a la primera sentencia ejecutable después de la `GOTO`.

`GOTO (12,24,36), INDEX GOTO (320,330,340,350,360), V(J,K)+2*B`

### 6.3.3 GOTO asignado

Transfiere el control a una etiqueta representada por una variable. Dicha variable ha intervenido previamente en una instrucción **ASSIGN** “asignándole” un valor a la variable. De ésta forma, se puede cambiar el destino de la transferencia dependiendo de la sentencia **ASSIGN** que se ha ejecutado más recientemente. Tiene la forma:

GOTO v[, ] (lista) ]

donde

**v** es una variable entera

**lista** es una lista de una o más etiquetas de sentencias ejecutables separadas por comas que la variable **v** podría tener como valor. No afecta a la ejecución de la **GOTO** y se puede omitir. Si se usa, el compilador podría generar código más eficiente.

- La **GOTO** y su **ASSIGN** asociada han de estar en la misma unidad de programa.
- Si el valor de la variable no es ninguna de las de la lista, el resultado es indefinido, salvo que se haya puesto la directiva **RANGE** a **ON** en fase de compilación, la cual daría error.

```
ASSIGN 450 TO IG
```

```
  .
```

```
  .
```

```
GOTO IG
```

que sería equivalente a **GOTO 450**.

```
ASSIGN 200 TO IB
```

```
  .
```

```
  .
```

```
GOTO IB, (300,200,100,25)
```

que sería equivalente a **GOTO 200**

## 6.4 Sentencias IF

Transfieren el control condicionalmente, o bien ejecutan condicionalmente una instrucción o bloque de instrucciones. Existen tres tipos :



- IF aritmético
- IF lógico
- bloque IF ( IF THEN , ELSE IF THEN , ELSE , END IF )

Para cada tipo, la decisión de transferir el control o ejecutar la sentencia o bloque de sentencias está basada en la evaluación de una expresión en la IF .

#### 6.4.1 IF aritmético

Transfiere el control condicionalmente a una de tres sentencias, según sea el valor de la expresión que aparece en la IF . Tiene la forma

IF (e) s <sub>1</sub> ,s <sub>2</sub> ,s <sub>3</sub>
---

donde:

e es una expresión aritmética (de cualquier tipo salvo compleja, lógica o carácter).

s<sub>1</sub>,s<sub>2</sub>,s<sub>3</sub> son etiquetas de instrucciones ejecutables de la misma unidad de programa.

- las tres etiquetas s<sub>1</sub>,s<sub>2</sub>,s<sub>3</sub> son obligatorias, aunque no tienen que ser distintas.
- se evalúa la expresión e y se transfiere el control a una de las tres etiquetas:

<i>si el valor de e es</i>	<b>el control pasa a</b>
menor que cero	etiqueta s <sub>1</sub>
igual a cero	etiqueta s <sub>2</sub>
mayor que cero	etiqueta s <sub>3</sub>

#### Ejemplo 13.

IF (THETA-CHI) 50,50,100

el control pasa a la sentencia 50 si la variable real THETA ≤ la variable real CHI, y pasa a la 100 si THETA > CHI.

IF (NUMERO/2\*2-NUMERO) 20,40,20

transfiere el control a la sentencia 40 si el valor de la variable entera NUMERO es par, y a la de la sentencia 20 si su valor es un número impar.

### 6.4.2 IF lógico

Ejecuta condicionalmente una única sentencia dependiendo del valor de la expresión lógica que aparece en la misma IF. Tiene la forma

IF (e) sentencia

donde:

**e** es una expresión lógica.

**sentencia** es una sentencia FORTRAN completa, ejecutable, excepto una DO , ENDDO , bloque IF u otro IF lógico.

- se evalúa la expresión lógica “ e “. Si su valor es verdadero, se ejecuta “ sentencia”. Si es falso, se transfiere el control a la siguiente instrucción ejecutable después del IF , sin ejecutarse “sentencia”.

#### Ejemplo 14.

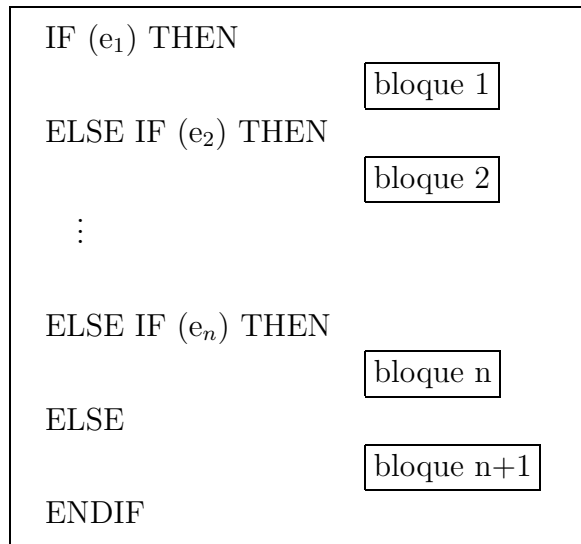
```
IF (J.GT.4 .OR. J.LT.1) GO TO 250
IF (REF(J,K) .NE. HOLD) REF(J,K)=REF(J,K)+(-1.5D0)
IF (ENDRUN) CALL EXIT
```

### 6.4.3 Bloques IF

Ejecutan condicionalmente bloques completos de sentencias. Las cuatro sentencias que se usan en la construcción de los bloques IF son :

- IF THEN
- ELSE IF THEN
- ELSE
- END IF

Un bloque IF tiene la forma:



donde:

$e_i$  es una expresión lógica

**bloque i** es una secuencia de cero o más sentencias FORTRAN completas.

- la sentencia **IF THEN** inicia una construcción de bloque **IF** . El bloque que sigue se ejecuta si es verdadero el valor de la expresión lógica en la **IF THEN** .
- la **ELSE IF THEN** es opcional, y una construcción de bloque **IF** puede tener varias **ELSE IF THEN** .
- la **ELSE** es también opcional y especifica un bloque de sentencias a ejecutar si no se ha ejecutado previamente ningún bloque de sentencias del bloque **IF** . Si está presente, ha de seguirle inmediatamente una **END IF** .
- la **END IF** termina una construcción de bloque **IF** .
- después de que se ha ejecutado la última sentencia de un bloque de instrucciones, el control pasa a la siguiente sentencia ejecutable que sigue a la **END IF** . Por tanto, solo un bloque de sentencias se ejecuta cuando una **IF THEN** es ejecutada.
- un bloque de instrucciones puede contener a su vez otro bloque **IF** , permitiendo construcciones de la complejidad que se requiera.
- **ELSE IF THEN** y **ELSE** pueden tener etiquetas, pero éstas no pueden ser referenciadas.

- la END IF puede tener etiqueta a la que se puede bifurcar, pero solo desde el bloque inmediatamente precedente.

### Ejemplo 15.

```
IF (ABS(ADJ).GE. 1.0E-5) THEN
    TOT=TOT+ABS(ADJ)
    Q=ADJ/VAL
END IF
```

```
IF (NAME.LT.'N') THEN
    IFR=IFR+1
    FRL(IFR)=NAME(1:2)
ELSE
    IB=IB-1
END IF
```

```
if (a.gt.b) then
    d=b
    f=a-b
else if (a.gt.b/2.) then
    d=b/2.
end if
```

```
if (a.ge.b) then
    if (c.gt.d) then
        x=c
    else
        x=d
    endif
    x=x+a
else
    x=b+max(c,d)
endif
```

## 6.5 PAUSE

Visualiza un mensaje en la terminal y suspende temporalmente la ejecución del programa. Tiene la forma:

PAUSE [mensaje]

donde **mensaje** es una constante tipo carácter o una cadena de números decimales de 1 a 5 dígitos. Es opcional.

- Si el programa se está ejecutando en modo interactivo, el contenido de **mensaje** se visualiza en la terminal, seguido por el prompt usual, indicando que se ha suspendido el programa y espera un comando.

### Ejemplo 16.

```
PAUSE 777    PAUSE 'Preparar impresora'
```

## 6.6 RETURN

Transfiere el control desde un subprograma a la unidad de programa que le llamó. Se usa solo en unidades de subprograma. Tiene la forma:

RETURN [i]

donde **i** indica un retorno alternativo desde el subprograma y se puede especificar únicamente en subprogramas tipo subrutina (**??**), no en subprogramas tipo función (**??**).

Cuando se especifica, indica que se ha de tomar el **i**-ésimo retorno alternativo en la lista actual de argumentos de la subrutina.

- Cuando una sentencia **RETURN** se ejecuta en un subprograma de función, el control se devuelve al programa que lo llamó, en la instrucción que contiene la referencia a función.
- Cuando una **RETURN** se ejecuta en una subrutina, el control se devuelve a la primera sentencia ejecutable posterior a la **CALL** que originó la llamada, o bien a la etiqueta que se especificó como **i**-ésimo retorno alternativo en la **CALL** .

### Ejemplo 17.

```
SUBROUTINE CONVERT(N,ALFA,DATOS,K)
INTEGER ALFA(*),...
.
.
RETURN
END
```

## 6.7 STOP

Finaliza la ejecución del programa. Tiene la forma:

STOP [mensaje]

donde **mensaje** es una constante tipo carácter o una cadena de números decimales de 1 a 5 dígitos. Es opcional.

- Si se especifica **mensaje**, su contenido se visualiza en la terminal, termina la ejecución y devuelve el control al sistema operativo.
- Si no se especifica, el programa termina sin ningún mensaje.

### Ejemplo 18.

```
STOP 100    STOP 'Fin ejecución!'
```

## 7 PROCEDIMIENTOS

Permite encapsular cualquier conjunto de cálculos .

Existen cuatro tipos:

1. Funciones intrínsecas
2. Funciones sentencia
3. Subprogramas **FUNCTION**
4. Subrutinas

### 7.1 INTRÍNSECAS

Son las funciones de biblioteca, librerías.

Algunas funciones tienen nombres genéricos: según sea el tipo de argumento el compilador selecciona la apropiada.

Sus nombres no son globales, ni son palabras reservadas.

Una relación de funciones intrínsecas viene dada por la tabla que viene a continuación. En la tabla aparece el número de argumentos para cada función y qué tipos de datos permite. Los códigos de los tipos son: **I** = Entero, **R** = Real, **D** = Doble precisión, **X** = Complejo, **C** = Carácter, **L** = Lógico, **\*** significa que el resultado tiene el mismo tipo que su(s) argumento(s). Cuando exista más de un argumento, han de ser todos del mismo tipo.

<i>Función</i>	<b>Explicación</b>
R = ABS(X)	Calcula el módulo de un número complejo.
* = ACOS(RD)	Arco-coseno; el resultado está en el intervalo 0 to $+\pi$
R = AIMAG(X)	Extrae la parte imaginaria de un número complejo. Usar REAL para obtener la parte real.
* = ANINT(RD)	Redondea al entero más próximo.
* = ATAN2(RD, RD)	Arco tangente de $arg_1/arg_2$ , el resultado en el rango $-\pi$ a $+\pi$ .
C = CHAR(I)	Retorna el I-ésimo carácter ASCII.
X = CMPLX(IRDX, IRD)	Convierte a complejo, segundo argumento opcional.
X = CONJG(X)	Complejo conjugado de un número complejo.
* = COS(RDX)	Coseno del ángulo en radianes.
D = DBLE(IRDX)	Convierte a doble precisión.
D = DPROD(R, R)	Calcula el producto de dos valores reales en doble precisión .
* = EXP(RDX)	Retorna la exponencial, o sea, $e$ elevado a la potencia del argumento. Es la función inversa del logaritmo natural.
I = ICHAR(C)	Retorna la posición del carácter en la tabla ASCII .
I = INDEX(C, C)	Busca en la primera cadena y retorna la posición de la primera ocurrencia de la segunda cadena, cero si no está presente.
I = INT(IRDX)	Convierte a entero truncando.
I = LEN(C)	Retorna la longitud de la cadena.
L = LGE(C, C)	Comparación léxica “mayor o igual”, verdadero si $arg_1 \geq arg_2$ .
L = LGT(C, C)	Comparación léxica “mayor que”, verdadero si $arg_1 > arg_2$ .
L = LLE(C, C)	Comparación léxica “menor o igual”, verdadero si $arg_1 \leq arg_2$ .
L = LLT(C, C)	Comparación léxica “menor que”, verdadero si $arg_1 < arg_2$ .
* = LOG(RDX)	Logaritmo en base e (donde $e=2.71828182845904\dots$ ).
* = LOG10(RD)	Logaritmo en base 10.
* = MAX(IRD, IRD,...)	Retorna el mayor de sus argumentos.
* = MIN(IRD, IRD,...)	Retorna el menor de sus argumentos.
* = MOD(IRD, IRD)	Retorna $arg_1$ módulo $arg_2$ , o sea el resto de dividir $arg_1$ por $arg_2$ .
R = REAL(IRDX)	Convierte a real.
* = SIGN(IRD, IRD)	Realiza una transferencia de signo: si $arg_2$ es negativo el resultado es $-arg_1$ , si $arg_2$ es cero o positivo el resultado es $+arg_1$ .
* = SQRT(RDX)	Raíz cuadrada (error si arg es negativo).
* = TAN(RD)	Tangente del ángulo en radianes.

## 8 Nombres Específicos de las Funciones Genéricas

Son necesarios cuando el nombre de la función intrínseca se usa como argumento real de otro procedimiento. En éste caso, el nombre específico se ha de declarar en una instrucción `INTRINSIC`. La siguiente tabla lista los nombres específicos en Fortran77 de algunas funciones comunes. EL resto de funciones o no tienen nombres genéricos o no se pueden pasar como argumentos reales.

Nombre genérico	Nombres específicos			
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX
ABS	IABS	ABS	DABS	CABS
ACOS		ACOS	DACOS	
AINT		AINT	DINT	
ANINT		ANINT	DNINT	
ASIN		ASIN	DASIN	
ATAN		ATAN	DATAN	
ATAN2		ATAN2	DATAN2	
COS		COS	DCOS	
COSH		COSH	DCOSH	
DIM	IDIM	DIM	DDIM	CCOS
EXP		EXP	DEXP	
LOG		ALOG	DLOG	
LOG10		ALOG10	DLOG10	
MOD	MOD	AMOD	DMOD	CEXP
NINT		NINT	IDNINT	
SIGN	ISIGN	SIGN	DSIGN	CLOG
SIN		SIN	DSIN	
SINH		SINH	DSINH	
SQRT		SQRT	DSQRT	
TAN		TAN	DTAN	CSIN
TANH		TANH	DTANH	
				CSQRT

### 8.1 FUNCIONES SENTENCIA

Se definen por una única sentencia

Pueden tener cualquier número de argumentos formales, y todos han de aparecer en la parte derecha de la definición.



```
REAL M1,M2,G,R
NEWTON(M1,M2,R)=G*M1*M2/R**2
```

y la invocación sería, por ejemplo FUERZA=NEWTON(X,Y,DIST)

Las siguientes funciones calcularían el seno, coseno y tangente de ángulos expresados en grados sexagesimales:

```
PARAMETER(PI=3.14159265, GAR=PI/180.0)
SING(ALFA)=SIN(ALFA*GAR)
COSG(ALFA)=COS(ALFA*GAR)
TANG(ALFA)=SING(ALFA)/COSG(ALFA)
```

```
LOGICAL MAT,DIGITO,DOM
CHARACTER C*1
DIGITO(C)=LGE(C,'0').AND.LLE(C,'9')
MAT(C)=INDEX('+-*/',C).NE.0
DOM(C)=DIGITO(C).OR.MAT(C)
```

la sintaxis general es la siguiente:

func(arg1,arg2,...,argn)=expresión
------------------------------------

donde func y expresión tendrán normalmente el mismo tipo de datos.

## 8.2 SUBPROGRAMAS FUNCTION

la sintaxis general es la siguiente:

(tipo) FUNCTION nombre(arg1,arg2,...,argn) ... END
--

Pueden retornar un solo valor

```
REAL FUNCTION TSEGS(NHORAS,MINS,SEGS)
INTEGER NHORAS,MINS
REAL TSEGS
TSEGS=((NHORAS*60)+MINS)*60+SEGS
```

Invocacion : TSEGS(12,30,0.0)

Los argumentos reales se transfieren a los argumentos formales

## 8.3 SUBROUTINAS

la sintaxis general es la siguiente:

SUBROUTINE nombre(arg1,arg2,...,argn) ... END

o bien

SUBROUTINE nombre ... END

pueden retornar cualquier numero de valores, controlado por el programador (cualquier argumento puede ser de entrada, salida o de entrada/salida).

```
SUBROUTINE HMS(HORA,NHORAS,MINS,SEGS)
REAL HORA,SEGS
INTEGER NHORAS,MINS
NHORAS=INT(HORA/3600.0)
SEGS=HORA-3600.0*NHORAS
MINS=INT(SEGS/60.0)
SEGS=HORA-60.0*MINS
END
```

La invocación podría ser

```
CALL HMS(45000.0,NHORAS,MINS,SEGS)
WRITE(UNIT=*,FMT=*)NHORAS,MINS,SEGS
```

Es importante tener en cuenta que *el paso de parámetros es siempre por referencia* por lo que estamos pasando los punteros a las direcciones de memoria de las variables que estamos pasando por parámetros, y si modificamos en la subrutina los valores de dichas variables, quedarán con esos valores después de salir de la subrutina.

La recursividad no está permitida por el estándar, pero la mayoría de los compiladores la permite. Aunque hay que tener en cuenta que el principal objetivo de los programas Fortran es ser veloces, ya que se trata de problemas de cálculo, y si bien la formulación recursiva de un algoritmo puede ser más clara y elegante que su formulación iterativa, en último término la ejecución se ha de transformar en iterativa por parte del compilador. Por tanto, a efectos de cálculo, siempre es preferible la versión iterativa del algoritmo.

## 9 ENTRADA/SALIDA

- En Fortran el término fichero se usa para cualquier cosa que se pueda manejar con `READ` o `WRITE`: el término cubre no solo los ficheros de datos almacenados en disco o cinta sino también periféricos tales como impresoras o terminales.

- Antes de que pueda usar un fichero externo se ha de conectar via una `OPEN` a una unidad de I/O ( entre 1 y 99).

Los ficheros son referenciados via sus números de unidad.

```
OPEN(UNIT=1, FILE='B:INPUT.DAT', STATUS='OLD')
OPEN(UNIT=9, FILE='PRINTOUT', STATUS='NEW')
```

- Las unidades de E/S son un recurso global que puede ser utilizado por cualquier unidad de programa, que usarán todas el mismo número de unidad ( se le puede pasar a un procedimiento como un argumento).
- La conexión entre un fichero y una unidad persiste hasta que:
  - el programa termina (`STOP,END`).
  - otra `OPEN` conecta otro fichero a la misma unidad.
  - se ejecuta una `CLOSE` para esa unidad.
- Todos los periféricos se han de procesar secuencialmente (terminales e impresoras serán tratadas siempre como ficheros secuenciales con formato)
- Unidades preconectadas: 5=teclado, 6=pantalla)
- Manejo de errores: se realiza via `IOSTAT` o bien `ERR`.
- Fin de fichero: `END`=etiqueta para tratarlo cuando se alcance y prara ponerlo : `END FILE`.

## 9.1 Especificaciones de formato

Toda `READ` o `WRITE` que use un fichero con formato interno o externo ha de incluir un identificador de formato:

`FMT=*` Especifica un atransferencia dirigida por listas y está permitida solo a ficheros secuenciales externos.

`FMT=etiqueta` la etiqueta ha de ser una `FORMAT` en la misma unidad de programa.

`FMT=expresión_tipo_carácter` el valor de la expresión tipo carácter es una especificación de formato completa.

FMT=*array\_tipo\_carácter* los elementos del array carácter contienen la especificación de formato, que puede ocupar tantos elementos del array como sea necesario.

Es también posible “calcular” un formato en tiempo de ejecución: salida de un número real en formato de punto fijo(F10.2) cuando es pequeño, cambiarlo a formato exponencial (E18.6) cuando es mayor (de un millón)

```
CHARACTER F1*(*), F2*12, F3*(*)  
PARAMETER(F1='(1X, ''MAGNITUD='', ''')  
PARAMETER(F3='')  
IF (PESO .LT. 1.0E6) THEN  
    F2='F10.2'  
ELSE  
    F2='E18.6'  
ENDIF  
WRITE(UNIT=*,FMT=F1//F2//F3) PESO
```

La senetencia **FORMAT** es no ejecutable y puede continuarse hasta ocupar 20 lineas como máximo.

La misma **FORMAT** se puede utilizar por varias **READ** o **WRITE**, pero como es fácil equivocarse con el matching de la lista de datos tiene lógica poner la **FORMAT** lo más cerca posible de la **READ** o **WRITE**

## 9.2 Descriptores de formato

- Existen dos tipos de descriptores de edición: descriptores de datos y descriptores de control.
- Un descriptor de datos ha de existir por cada item de datos transferido; comienzan con una letra que indica el tipo de datos seguido por un entero sin signo que denota el ancho del campo(Ej.: I5, F9.2)
- un descriptor de control se usa para varios propósitos, tales como tabulación a columnas concretas, etc.

### 9.2.1 Descriptores A,E,F,G,I,L

Los descriptores ELSE ,F,G se pueden utilizar por reales, doble precisión y complejos; en cualquier otro caso el tipo de datos ha de coincidir (para cada complejo se necesitan dos descriptores de coma flotante)

Tipo	Descriptores
Enteros	Iw, Iw.m
Real, Doble precisión, o complejo	Ew.d, Ew.dEe, Fw.d, Gw.d, Gw.dEe
Lógico	Lw
Carácter	A, Aw

Las letras **w**, **m**, **d**, y **e** representan constantes enteras sin signo y han de verificar  $w > 0, e > 0$ .

- w** ancho total del campo.
- m** número mínimo de dígitos a producir en la salida.
- d** número de dígitos después del punto decimal.
- e** número de dígitos usados por el exponente.

Cualquier descriptor puede estar precedido de un entero sin signo que indica el número de repeticiones:

3F6.0 es equivalente a F6.0,F6.0,F6.0

que es de utilidad en el manejo de arrays.

#### Enteros (Iw, Iw.m)

En salida, un entero con Iw aparece justificado a la derecha con blancos a la izquierda, los que sean necesarios.

Ej.:

```
NHORAS = 8
MINUTOS = 6
WRITE(UNIT=*, FMT='(I4.2, I2.2)') NHORAS, MINUTOS
```

tendrá por salida:

□□0806

En entrada  $I_w$  y  $I_{w.m}$  son iguales.

Nótese que un campo entero no puede contener punto decimal, exponente o cualquier otro simbolo de puntuación como una coma.

Ejs.:

```
A=+425.25
```

```
B=+6234
```

```
C=-11.92
```

```
D=3
```

```
WRITE(UNIT=*, FMT='(I4,I2,I3,I6.3)') A,B,C,D
```

tendrá por salida:

```
 452**-12  003
```

### **Punto flotante( $E_{w.d}$ , $E_{w.dEe}$ , $F_{w.d}$ , $G_{w.d}$ , $G_{w.dEe}$ )**

En la salida, los números son redondeados al número especificado de digitos.

La salida con  $E_{w.d}$  produce un número en notación exponencial o “cientifica”. La mantisa estará entre 0.1 1.

$G_{w.d}$  es el descriptor de propósito general: si el valor es mayor que 0.1 pero no demasiado largo, el campo será escrito usando un formato de punto fijo con  $d$  digitos en total y con 4 blancos al final del campo; en otro caso es equivalente al formato  $E_{w.d}$ .

Exactamente la cosa es asi : Si la magnitud es menor que 0.1 o mayor o igual que  $10^d$  (después de redondear a  $d$  digitos) se usa  $E_{w.d}$ ; en otro caso se usa  $F_{w.d}$ .

La forma  $E_{w.dEe}$  especifica que habrán exactamente  $e$  digitos en el exponente.

La forma  $G_{w.dEe}$  permite especificar también la longitud del exponente, si se elige el formato de punto fijo existen  $e + 2$  blancos al final.

En los casos  $E_{w.dEe}$  y  $G_{w.dEe}$  se ha de verificar  $w \geq d + e + 5$ , donde el 5 surge de sumar el blanco inicial, el signo del valor, el punto decimal, la letra  $E$  y el signo del exponente.

Ejemplo:

```
X = 123.456789
```

```
Y = 0.09876543
```

```
WRITE(UNIT=*, FMT='(E12.5, F12.5, G12.5)') X,X,X, Y,Y,Y
```

produce dos registros (con  $\square$  para representar el blanco):

`□0.12346E+03□□□123.45679□□123.46`  
`□0.98766E-01□□□□□0.09877□0.98766E-01`

En entrada todos los descriptores **E**, **F**, y **G** tienen efectos idénticos : si el campo de entrada contiene explícitamente un punto decimal, éste siempre toma precedencia, en otro caso los últimos *d* dígitos se toman como la fracción decimal. Si se usa un exponente puede estar precedido de *E* o *D* (la letra del exponente es opcional si el exponente tiene signo). Si el campo de entrada proporciona más dígitos que los que puede utilizar el almacenamiento interno, la precisión extra se ignora. Es mejor usar **Fw.d** que encaja con todas las formas de punto flotante o incluso enteros.

### Lógicos (Lw)

Cuando se escribe un valor lógico con **Lw** el campo contendrá la letra **T** o **F** precedida por un punto decimal y cualquier número de blancos. caracteres después de **T** o **F** se ignoran. Por tanto las formas **.TRUE.** y **.FALSE.** son aceptables.

### Tipo Carácter (A y Aw)

Si se usa **A** (sin especificar *w*) entonces la longitud del ítem de carácter en la lista de transferencia de datos lo determina. Nótese que la posición del resto de los ítems en el registro cambiará.

Es importante usar diseños con columnas fijas de la forma **Aw**: siempre usará un campo con ancho *w*.

en salida si la longitud real es menor que *w* se justifica a la derecha, con  $w - \text{long}$  blancos a la izquierda; en otro caso, solo se imprimen *w* caracteres.

En entrada si la longitud es menor que *w*, solo se utilizan los caracteres más a la derecha; en otro caso se leen *w* caracteres y se le añaden  $\text{long} - w$  blancos a la derecha.

## 9.3 Descriptores de control

No se corresponden con ningún ítem de la lista y son los siguientes:

A format specification consisting of nothing but control descriptors is valid only if the **READ** or **WRITE** statement has an empty data-transfer list.

Función	Descriptor
Saltar al siguiente registro/linea	/
Moverse a un columna (salida)	Tn, TLn, TRn, nX
Imprimir una constante tipo carácter	'cualquier cadena'
Parar si la lista es vacia	:
Controlar signo + antes de números positivos	SP, SS, S
Tratar blancos como nullos/ceros	BN, BZ
Poner un factor de escala para números	kP

donde **n** and **k** son constantes enteras, **k** puede tener signo.

Los descriptors **SP**, **BN**, **kP** afectan a todos los números tratados a partir de ese momento, pero los defaults del sistema se restauran al comienzo de la siguiente **READ** o **WRITE**.

Cualquier lista puede ir precedida por una constante entera, p.e.

2(I2.2, '-'), I2.2

es equivalente a

I2.2, '-', I2.2, '-', I2.2

Ejemplo de /:

```
WRITE(UNIT=LP, FMT=95) (NYEAR(I), POP(I), I=1, NYEARS)
95  FORMAT(1X, 'Year  Population', //, 100(1X, I4, F12.0, /))
```

**Tn** la salida comienza en la columna **n**

**TRn** desplazamiento de **n** columnas a la derecha

**TLn** desplazamiento de **n** columnas a la izquierda

**nX** equivalente a **TRn**

**SP** todo número será escrito con un + delante (a partir de cuando se use)

**SS** todo número será escrito sin un + delante (a partir de cuando se use)

**S** restaura el default, que es dependiente del sistema

**BN** todos los blancos son tratados como nullos, o sea, ignorados (después de que se use)

**BZ** todos los blancos son tratados como ceros (después de que se use)



El default inicial (que depende de **BLANK=** en la **OPEN**) es restablecido al comienzo de cada nueva transferencia con formato.

El factor de escala (**kP**) se utiliza para introducir una escala por una potencia de 10 entre valores internos y externos cuando se usan **E,F,G**.

En principio podría ser útil cuando se manejan datos demasiado grandes o demasiado pequeños.

El factor de escala inicial en cada transferencia ( entrada o salida) es cero.

valor interno= valor externo /  $10^k$

valor externo= valor interno \*  $10^k$

## Referencias

- [Bez89] H.C. Bezner. *Fortran 77*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1989.
- [Bor89] G.J. Borse. *Programación en FORTRAN 77, con Aplicaciones de Cálculo Numérico en Ciencias e Ingeniería*. Anaya Multimedia, 1989.
- [CS90] I.D. Chivers and J. Sleighthome. *Interactive Fortran 77*. Ellis Horwood Limited, New York, second edition, 1990.
- [Hewa] Hewlett-Packard. *HP FORTRAN 77-HP-UX Programmer's*.
- [Hewb] Hewlett-Packard. *HP FORTRAN 77-HP-UX Reference Volume 1*.
- [Hewc] Hewlett-Packard. *HP FORTRAN 77-HP-UX Reference Volume 2*.
- [JC88] R.K. Jones and T. Crabtree. *Fortran Tools*. John Wiley & Sons, New York, 1988.
- [Lig88] P. Lignelet. *Les Fichiers en Fortran 77*. Masson, Paris, 1988.
- [Mer86] F. García Merayo. *Programación en Fortran 77*. Ed. Paraninfo, Madrid, 1986.
- [Met85] M. Metcalf. *Fortran Optimization*. Academic Press, London, 1985.
- [MS88] D.D. McCracken and W.I. Salmon. *Computing for Engineers and Scientists with Fortran 77*. John Wiley & Sons, New York, second edition, 1988.
- [Ter87] P.D. Terry. *FORTTRAN from Pascal*. Addison-Wesley Publishing Co., Wokingham, England, 1987.