Departamento de Física Aplicada
Universidad de Granada

FORTRAN

EN 6 LECCIONES

Juan Antonio Morente Chiquero.

Departamento de Física Aplicada.

Universidad de Granada.

Fortran en 6 lecciones.

ISBN: 84-607-6535-0

Depósito Legal: GR-176-2003.

Lo que encontraremos a continuación es un subconjunto del conjunto completo de instrucciones que hoy día forma el lenguaje de programación Fortran. Con esta versión reducida de este lenguaje de programación, se pretende que en un tiempo corto, los alumnos de la asignatura "Introducción a los Métodos Numéricos en Física" (cuatrimestral, de sólo 6 créditos en 2º curso de Física en la Universidad de Granada) tengan a su disposición una herramienta que les permita realizar sus prácticas de programación. Es claro que han quedado fuera muchas proposiciones Fortran y, que de las nombradas, han quedado al margen muchas de sus opciones. Damos a continuación una lista de libros en los que se puede aprender mucho más sobre el lenguaje de programación Fortran.

Bibliografía.

- G.B. Davis y T.R. Hoffman. FORTRAN 77: Un estilo estructurado y disciplinado. McGraw-Hill, 1985.
- F. García Merayo. Programación en FORTRAN 77. Editorial Paraninfo, 1996.
- F. García Merayo. Lenguaje de Programación en FORTRAN 90. Editorial Paraninfo, 1998.
- B.D. Hahn. Introduction to FORTRAN 90 for Scientists and Engineers. Cambridge University Press, 1994.
- Microsoft FORTRAN Optimizing Compiler. Language Reference. Microsoft Corporation, 1987.
- C.G. Page. Profesional Programmer's Guide to Fortran. University of Leicester, U.K., 2001.

Código máquina, lenguajes ensambladores y lenguajes de alto nivel.

A principio de los años 50, cuando los procesadores eran muy simples (hoy ni se les daría el nombre de procesadores), se programaban estos con un lenguaje, lista de unos y ceros, que la máquina entendía directamente. Este tipo de lenguaje presentaba serias dificultades para el programador que debía aprender y recordar multitud de reglas nemotécnicas para llevar a cabo operaciones tan simples como sumas o multiplicaciones. Este tipo de lenguaje, en código máquina o un poco más evolucionado, pero más cerca de la máquina que del hombre, se llama lenguaje ensamblador. Con el tiempo, los lenguajes de programación evolucionan y se van acercando a la cultura y forma de expresión del hombre. Estos lenguajes, fácilmente utilizables por el hombre, se llaman lenguajes de alto nivel. Al programa que traduce el lenguaje de alto nivel a código máquina se le llama compilador.

Estrictamente hablando, llamaremos lenguaje ensamblador a aquel que sólo genera una instrucción máquina por línea de programa, y llamaremos lenguajes de alto nivel a aquellos que generan muchas instrucciones máquina por cada línea del programa.

FORTRAN es un lenguaje de alto nivel. Su nombre procede del acrónimo FORmula TRANslator. Con él, el programador puede escribir su programa en lenguaje de alto nivel, semejante al empleado en fórmulas o expresiones matemáticas. El compilador/encuadernador se encargará posteriormente de traducir a lenguaje máquina y nos devolverá un programa ejecutable. El primer compilador Fortran nació entre 1954 y 1955, desarrollado por la compañía IBM.

El lenguaje Fortran es el lenguaje de programación propio de los científicos, que convive con otros lenguajes de programación, como C, C++, Java, Visual Basic, Pascal, Cobol, etc. Algunos de estos lenguajes permiten controlar el ordenador más directamente (C, C++), otros permiten diseñar con facilidad un entorno gráfico que permita una relación fácil entre máquina y hombre (Visual Basic), otros están más volcados a la compatibilidad entre diferentes plataformas (Java), pero el lenguaje científico por excelencia sigue siendo el Fortran. Tampoco debemos despreciar el lenguaje BASIC (Beginner's All Purpose Symbolic Instruction Code) que casi todos los ordenadores con sistema operativo Windows tienen en su interior (véase la carpeta \tools\oldmsdos del CD-ROM de instalación del sistema operativo) y al que podemos acceder ejecutando el comando qbasic. Con este lenguaje también podemos llevar a cabo operaciones numéricas de envergadura.

Conjunto de caracteres Fortran.

a) Caracteres alfabéticos: 26 letras.

No está la ñ, pero sí la w. Fortran es indiferente a la letra mayúscula o minúscula (interpreta ambas de la misma forma). Lo ortodoxo es escribir con letras mayúsculas.

b) Caracteres numéricos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (los diez dígitos).

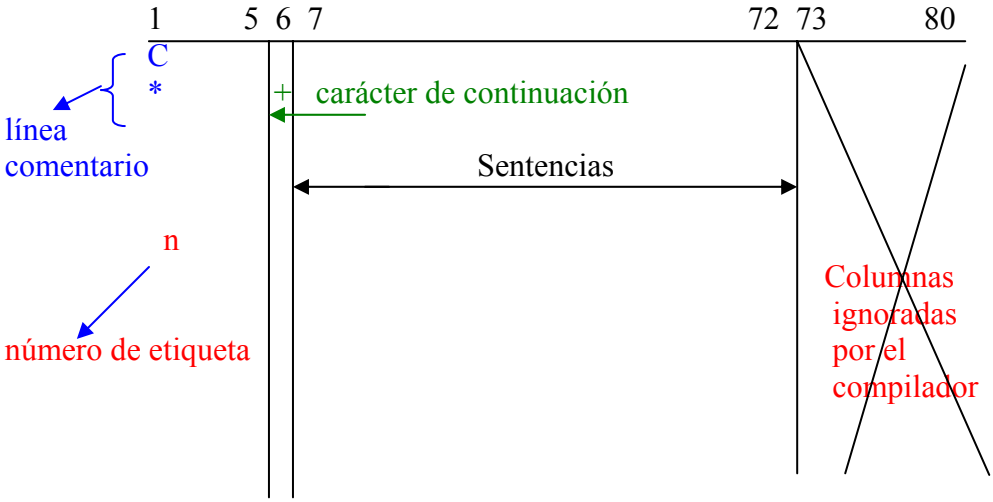
Al conjunto de caracteres alfabéticos y numéricos se les denomina caracteres alfanuméricos.

c) Caracteres especiales: (blanco) = + - * / ( ) . , ' : $

Codificación de un programa Fortran

Dividimos las 80 columnas del monitor del ordenador en bloques.

Distribución de la Pantalla en FORTRAN :



- Las sentencias, instrucciones o proposiciones Fortran se colocan entre al columna 7 y la 72.

- Las columnas 1 a 5 se dedican para enumerar o etiquetar las sentencias o proposiciones Fortran. Las sentencias no tienen porque ir etiquetadas obligatoriamente (sólo si el programador cree que es necesario o el programa lo requiere). Tampoco hay que etiquetar por orden.

- Si en la columna 1 se pone una C o un *, la línea se convierte en un comentario y no se compila.

- La mayor parte de las instrucciones se pueden escribir en una línea, pero si no podemos o no queremos escribirla en una línea, se coloca un carácter que no sea ni el blanco ( ), ni el cero (0) en la columna 6. Este carácter se llama carácter de continuación. Aunque depende del compilador, el número máximo de caracteres de una sentencia Fortran es de 1320 caracteres (20 líneas).

- Las columnas 73 a 80 son ignoradas por el compilador. Se pueden utilizar para identificar el programa, enumerar líneas o cualquier otro propósito.

- Con Fortran 77, en cada fila sólo se coloca una instrucción o sentencia.

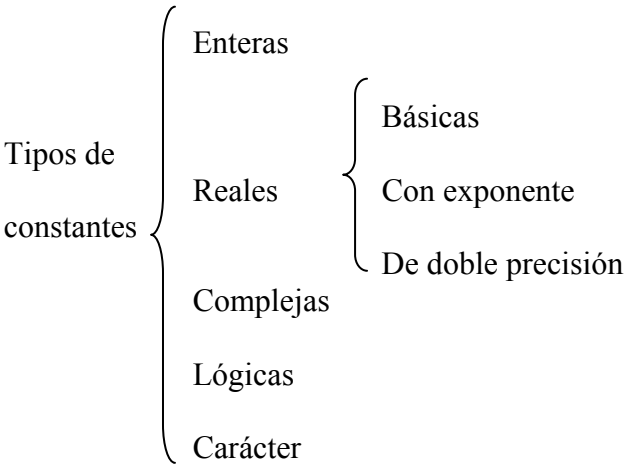
- En Fortran 90 hay algunas innovaciones que modifican algunos aspectos anteriores:

  - Se pueden escribir varias sentencias en una línea, separando las sentencias mediante ;

  - Cualquier línea cuyo primer carácter no blanco sea una exclamación (!), es una línea de comentario. Se puede añadir un comentario a continuación de una sentencia, poniendo ! como primer carácter del comentario

Constantes Fortran.

Designan un valor específico y determinado que se define al hacer el programa.



Constantes enteras.

- No llevan parte decimal ⇒ Tienen que ir sin punto.

- Si es negativa debe llevar el signo menos (-). En las positivos, el signo más (+) es opcional.

- Ejemplos de constantes enteras: 45 , -2 , -132, 0

Constantes reales básicas.

- Su forma es: Un signo (si es +, el signo es opcional) seguido por una serie de dígitos con un punto delante, entre o al final de ellos.

- Ejemplos: +37.85, -3., –3.0, .12

Constantes reales con exponente. (Simple precisión)

- Se forman con una constante real básica multiplicada por una potencia de diez que se expresa por un exponente entero precedido por la letra E. La forma general es mantisaEexponente.

- El exponente tiene que ser entero.

- Ejemplos: 5.23E-2 cuyo significado es $5.23 \cdot 10^{-2}$
  $0.35E-27 = 0.35 \cdot 10^{-27}$

Constantes reales de doble precisión. (8 bytes)

- Se expresan de idéntica forma a las constantes reales con exponente, pero cambiando la E por D.

- La diferencia entre las reales con exponente y las de doble precisión es que el compilador reserva doble memoria (8 bytes) para las constantes de doble precisión, con lo que el número de dígitos disponibles es mayor y se pueden guardar constantes con mayor número de decimales y mayor rango en el exponente.

- Ejemplos: $1.24D-3 = 1.24 \cdot 10^{-3}$, $+5.25D+2 = 5.25D2 = 5.25 \cdot 10^{2}$

Constantes complejas.

- Se representan a través de una pareja de números reales dentro de un paréntesis y separadas por una coma (,). El primer número representa la parte real, y el segundo, la parte imaginaria.

- La mayoría de los compiladores admiten el empleo de dos constantes de doble precisión para formar una constante compleja.

- Ejemplos: (63.0,-5.02) = 63.0 - j5.02
  (65.7E-3, 10.2E+5) = $65.7 \cdot 10^{-3} + j10.2 \cdot 10^{5}$

Constantes lógicas.

- Un dato lógico puede poseer únicamente dos valores:

  .TRUE. que significa verdadero.
  .FALSE. que significa falso.

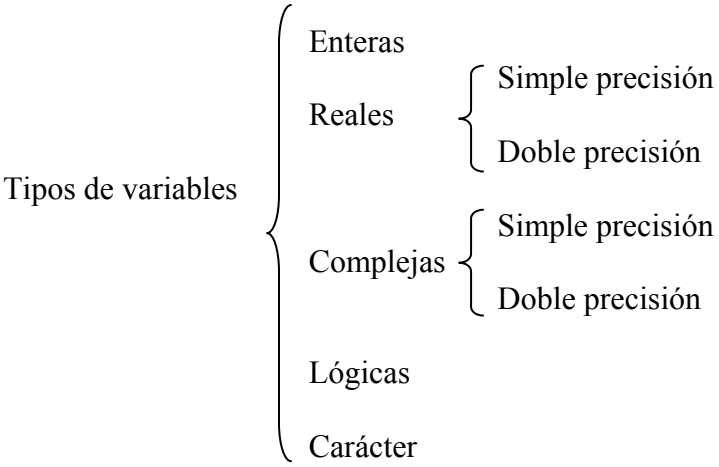
- Obsérvense los puntos que preceden y siguen a estas constantes.

Constantes carácter.

- Conjunto de caracteres Fortran precedidos y seguidos del carácter especial apóstrofe (').
- Ejemplos: 'UNIVERSIDAD', 'LA SOLUCION ES ='.

Variables Fortran.

El concepto de variable en Fortran coincide con el concepto habitual de esta palabra: Nombre simbólico con el que se designa una magnitud que puede tomar valores diversos.



Reglas generales para los nombres de las variables

1ª) El 1er carácter debe ser alfabético.

```
(6)| (7)    (escribimos las proposiciones Fortran a partir de la columna 7)
   | PI = 3.1415
   | 2PI = 6.28    (incorrecto)
```

2ª) No se admiten caracteres especiales o no contemplados por el Fortran (sólo caracteres alfanuméricos).

~~PI* =~~
~~π =~~

* , π no son caracteres admitidos.

3ª) En lenguaje Fortran ortodoxo, el número total de caracteres de una variable no debe de exceder de seis. Actualmente los compiladores admiten variables de hasta 32 caracteres.

4ª) Si no se declaran explícitamente, las variables que comienzan por las letras I, J, K, L, M, N (letras comprendidas entre las dos primeras letras de INteger) son enteras y el resto reales.

Ejemplo: Si escribimos el siguiente programa, lo compilamos, encuadernamos y ejecutamos

```
PROGRAM ENTEROS
J = 1/2
PRINT * ,  'J = ' , J
I=7/2
PRINT * ,  'I = ' , I
STOP
END
```

Escribiría J= 0 y I=3 porque I y J son enteros al no haber sido declaradas como reales de manera explicita.

```
PROGRAM REALES
REAL I, J  <-------  (declaración explicita como reales)
J = 1/2
PRINT * ,  'J = ' , J
I=7/2
PRINT * ,  'I = ' , I
STOP
END
```

Muestra en pantalla J= 0.5 y I=3.5

Variables Enteras

Cualquier variable que comience por las letras I, J, K, L, M, N es una variable entera si no se especifica lo contrario. Si no comienza por estas letras puede declararse como entera mediante la proposición de declaración con la siguiente sintaxis:

```
INTEGER  variable, variable
```

Ejemplo:

```
INTEGER    ro, z, k
```

Si no se especifica lo contrario, una variable entera se almacena en 4 bytes. Podemos almacenarla en menos bytes especificándolo en su declaración

```
INTEGER*1  lista variables          (utiliza  1 byte:  ± 127 )
INTEGER*2  lista variables          (utiliza 2 bytes:  ±32767 )
INTEGER*4  lista variables          (utiliza 4 bytes: ±2 147 483 647)
```

La declaración INTEGER e INTEGER*4 son equivalentes.

Variables Reales

Todas las variables que no empiecen por I, J, K, L, M, N son reales si no se especifica lo contrario. Se declara una variable como real mediante las siguientes proposiciones:

```
REAL  lista de variables separadas por comas

REAL  lista de variables
                                  } equivalentes (se almacenan en 4 bytes)
REAL*4  lista de variables

REAL*8   lista de variables
                                  } equivalentes (se almacenan en 8 bytes)
DOUBLE  PRECISION   lista
```

Ejemplo:

```
REAL   f, x, pi, i
```

Estas tres no hace falta declararlas

Esta la tenemos que declarar explícitamente

Las declaraciones REAL y REAL*4 son equivalentes. Si una variable no se declara y por su primera letra le corresponde ser real, pasa a ser REAL*4. También son equivalentes las declaraciones DOUBLE PRECISION y REAL*8.

Las variables complejas, lógicas y carácter hay que declararlas siempre.

Variables Complejas

Hay que declararlas obligatoriamente mediante las proposiciones

```
COMPLEX lista de variables
                                  } equivalentes
COMPLEX*8 lista de variables

COMPLEX*16 lista de variables
```

Ejemplos:

```
COMPLEX   TENSIÓN, INTEN
COMPLEX*8   FEM, CORR
COMPLEX*16    POTEN, ENERG   (se almacena en 16 (8+8) bytes )
```

Variables lógicas

Contendrán una constante lógica. Hay que declararlas mediante la sentencia:

```
LOGICAL   lista de variables
```

Sólo pueden tener los valores .TRUE. ó .FALSE.

Ejemplo:

```
LOGICAL  si,  no,  nose
```

Variables carácter

Una variable carácter contiene un conjunto de caracteres Fortran. Su declaración es obligatoria. Debemos decir su longitud (número de caracteres máximo que puede incluir) al declararla. La sintaxis de declaración es la siguiente:

```
CHARACTER*n lista de variables

CHARACTER  variable*n, variable*m
```

En ambas sentencias m y n son números enteros. En la primera sentencia todas las variables de la lista tienen la misma longitud de n caracteres. En la segunda sentencia, cada variable tiene su longitud.

Ejemplos:

```
CHARACTER*6     TIT1,  TIT2
CHARACTER     PEPE
CHARACTER L*9, LL*4
...
L= 'resultado'
LL=  'nulo'
...
TIT1 = 'METODO'
TIT2 = 'NUMERO'

PEPE = 'S'
```

Si no se especifica la longitud, por defecto se toma un solo carácter, como sucede con la variable PEPE del ejemplo anterior.

Debemos de tener en cuenta que todas las declaraciones de variables deben de ir al inicio del programa antes de cualquier sentencia ejecutable, de lo contrario se generará un error en la compilación.

Sentencias Iniciales y Finales

En Fortran no es necesario que un programa tenga nombre, pero está permitido. El nombre se le da mediante la proposición

```
PROGRAM   nombre
```

Esta proposición no es obligatoria, pero de existir debe ser la primera, salvo comentarios. Por ejemplo, es válido y tal vez bonito comenzar un programa así:

```
**************************************************************
        PROGRAM  RAIZ
**************************************************************
```

Sentencia STOP

Su sintaxis es la siguiente:

```
STOP
STOP  n                          (p.e.,  STOP 2)
STOP  'variable carácter'        (p.e.,  STOP 'FINAL')
```

La sentencia STOP, en cualquiera de sus variables, detiene la ejecución del programa sin posible reinicio. En un programa podemos encontrar varias sentencias STOP. Si ponemos un número entero después de STOP y el programa finaliza en ese STOP, el programa mostrará en pantalla una referencia a ese número. Si después de STOP, ponemos una variable carácter, al finalizar el programa aparecerá esta variable carácter. La proposición STOP no es una sentencia obligatoria.

Sentencia END

Sentencia obligatoria. Debe ser la última proposición o sentencia de un programa. Sólo puede haber una sentencia END en un programa. Indica al compilador que ya no hay más sentencias. Su sintaxis es simple:

```
END
```

En Fortran 90, puede finalizarse el programa con el nombre dado al inicio y ahora colocado a continuación de END:

```
END PROGRAM nombre
```

Sentencia PAUSE

Provoca que el programa se detenga hasta que se vuelva a pulsar la tecla *ENTER*. Puede haber varias sentencias PAUSE en un programa. Su sintaxis es:

```
PAUSE

PAUSE    n
```

Sentencias de asignación

Cada variable tiene reservado su lugar en la memoria por el compilador. Mediante las sentencias de asignación damos valores a las variables.

Asignación aritmética.

Las sentencias de asignación aritmética sirven para indicar al ordenador la realización de cálculos aritméticos y la asignación de valores a variables. La sintaxis general es:

```
V= expresión aritmética
```

donde V es una variable entera o real.

Ejemplos:

```
V =5.5
PI = 3.14
F = (2*X) +3
```

Esta sentencia de asignación no tiene el mismo sentido que una igualdad o ecuación en el cálculo matemático. La primera de las sentencias anteriores quiere decir que en lugar reservado en la memoria para la variable V, hemos puesto la constante real 5.5. Por ejemplo, podemos tener la siguiente secuencia:

```
I= 3
I= I+1
```

Esta última proposición sería una incorrección matemática, pero no así en Fortran y otros lenguajes de programación, donde quiere decir que la posición de memoria ocupada por I es sustituida por otro número que es el que había anteriormente más una unidad.

A la izquierda de sino igual (=) sólo puede existir una variable. No son correctas las siguientes instrucciones:

```
X*A = 4*Y

X +1 =7
```

Asignación compleja.

En la asignación compleja, la parte real y la imaginaria están dentro de un paréntesis y separadas por comas.

Ejemplo:

```
COMPLEX  C, D
C=(4.3, -2.1)
D=(4.0E-2, -5.774E4)
```

Asignación carácter y operador de concatenación //.

La forma general de asignación carácter tiene la forma:

```
CHARACTER*n  V
...
V = 'expresión'
```

donde V es una variable definida carácter y 'expresión' es una expresión carácter válida.

Ejemplo:

```
CHARACTER A*5, B*5, C*13
...
A='LISTA'
B='TABLA'
```

En las sentencias anteriores, se asignan las constantes carácter 'LISTA' y 'TABLA' a las variables carácter A y B, de 5 caracteres cada una.

El operador concatenación, //, sirve para unir variables carácter.

Ejemplo enlazado con el anterior:

```
C= A// 'y' //B
```

daría en C, el carácter LISTA y TABLA.

Asignación lógica.

Tiene la misma forma ya estudiada, pero ahora con variables y sentencias lógicas.

```
LOGICAL  V
...
V= expresión o variable lógica
```

Ejemplo:

```
LOGICAL  NOSE
NOSE= .TRUE.
```

Operaciones aritméticas

Símbolo	Significado	Ejemplo
+	Suma	S=A+B
-	Sustracción o negación	D=A-B B= -A
*	Multiplicación	P=A*B
/	División	C=A/B
**	Exponenciación	E= A**2

Prioridades.

1º. En primer lugar las operaciones indicadas entre paréntesis.
2º. Exponenciación (**).
3º. Multiplicación y división (*, /).
4º. Sumas y restas (+,-).

Si el programa se encuentre con dos o más expresiones de la misma prioridad, procede a ejecutar de izquierda a derecha.

Ejemplo: La expresión $y = ax^2+bx+c$ la podemos expresar en Fortran como

```
Y=(A*(X**2))+(B*X)+C
```

pero también daría el mismo resultado

```
Y=A*X**2+B*X+C
```

ya que, por las prioridades de las operaciones, primero se eleva al cuadrado, después se hacen los productos y finalmente se suma.

Ejemplo: La expresión $y = a(b-c)$ se expresaría como

```
Y= A*(B-C)
```

y no como

```
Y=A*B-C
```

que sería equivalente a la expresión $y=ab-c$, distinta de la que inicialmente intentamos implementar en Fortran.

Debemos de tener en cuenta los siguientes puntos:

- Los paréntesis se realizan desde el interior al exterior.
- No se permiten dos signos de operación seguidos, salvo los dos asteriscos de la exponencial. Es incorrecto la secuencia A*-B, tendríamos que poner A*(-B).
- La tabla siguiente indica el resultado que se obtiene cuando se combinan variables de diferentes tipo con los operadores aritméticos (+, -, *, /, **).

	Entero	Real	Doble precisión
Entero	Entero	R	DP
Real	R	R	DP
Doble precisión	DP	DP	DP

Es de destacar el hecho, puesto ya de manifiesto en el apartado "Reglas generales para el nombre de las variables", de que cuando se mezclan dos enteros aparece otro entero.

Funciones intrínsecas para la conversión de datos.

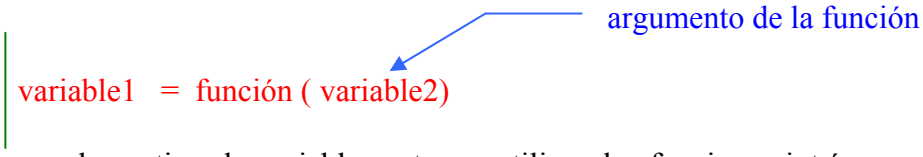
Fortran posee muchas funciones intrínsecas de conversión de datos, para datos carácter, aritméticas, trigonométricas, etc., que facilitan mucho la programación. Destacamos, en este apartado y en los siguientes, las más fundamentales, comenzando por las de conversión de datos.

En la proposición A=B, si A es de tipo diferente a B, los resultados de B se convertirán al tipo de A. Así, si B es real y A es entero, se perderá la parte fraccionaria de B que no pasará a A. El programa

```
REAL   B
INTEGER    A
B= 7.83
A = B
PRINT *,   A, B  ------>          7, 7.83
STOP
END
```

asigna el valor 7 a la variable A, perdiéndose la parte decimal.

La sintaxis general de las funciones intrínsecas es:



Para pasar de un tipo de variable a otra se utilizan las funciones intrínsecas de conversión, las más usuales son:

INT → Convierte su argumento real a entero por truncamiento.

Ejemplo:

```
NN=INT(VAR)
```

Si VAR es real con valor 3.8, NN es entera con valor 3.

REAL o FLOAT → Convierten un argumento entero a real.

Ejemplo:

```
R1= REAL (J)
```

Si J es entero de valor 3, R1 es real con valor 3.0.

DBLE → Convierte su argumente entero o real a real de doble precisión.

Ejemplo:

```
D=DBLE(C)
```

Si C=7.2, D=7.2D0 (el mismo valor, pero en doble precisión).

CMPLX→ Convierte un argumento entero o real a complejo, introduciendo el argumento en la parte real. Si le damos dos argumentos reales, forma una variable

compleja cuya parte real coincide con el primer argumento y la parte imaginaria con el segundo argumento.

Ejemplo:

```
C = CMPLX (I)
C4= CMPLX (A, B)
```

En la primera instrucción, si I = 5, C = (5.0, 0.0). En la segunda instrucción formamos una variable compleja, C4, a partir de las reales A y B.

Otras funciones implícitas básicas, relacionadas con conversión de tipo de variable, son:

NINT→ Obtiene el entero más próximo del argumento, expresado como entero.

Ejemplo:

```
J = NINT (4.3)      ⟶   J = 4
J2 = NINT ( 5.6)    ⟶   J2 = 6
```

ANINT→ Obtiene el entero más próximo del argumento, pero ahora expresado como real.

Ejemplo:

```
P = ANINT (4.3)     ⟶   P = 4.0
P2 = ANINT(5.6)     ⟶   P2 = 6.0
```

El paso de variables complejas a reales lo podemos hacer mediante las siguientes funciones intrínsecas:

REAL→ Genera un número real con la parte real del argumento complejo. Es la misma función que pasa enteros a reales. En esta y en otras ocasiones, la función actúa según sea su argumento.

Ejemplo:

```
R = REAL (7.5E2, -2.3E0)   ⟶   R = 7.5E2
```

AIMAG→ Genera un número real con la parte imaginaria del argumento complejo.

Ejemplo:

```
R = AIMAG (7.5E2, -2.3E0)   ⟶   R = -2.3E0
```

CONJG→ Obtiene el complejo conjugado del argumento.

Ejemplo:

```
C = CONJG (3.2, 2.7)  ──────►  C = (3.2, -2.7)
```

ABS→ Devuelve el módulo de un argumento complejo.

Ejemplo:

```
R = ABS (4.0, 3.0)  ──────►  R = ( 4.0^2+3.0^2)^½ = 5.0
```

Funciones intrínsecas aritméticas.

La forma general o sintaxis de estas funciones es la que ya hemos visto para las funciones intrínsecas de conversión de datos. Las funciones intrínsecas más básicas son las que comentamos a continuación.

ABS → Calcula el valor absoluto de argumentos reales o enteros, devolviendo en número real. Como vimos anteriormente, si el argumento es complejo devuelve el módulo del argumento.

Ejemplo:

```
Y= ABS (-7)  ──────►  Y=7.0
```

MAX→ Escoge el máximo entre la lista de sus argumentos.
MIN → Escoge el mínimo entre la lista de sus argumentos.

Ejemplo:

```
Y = MAX (a1, a2, a3)
ZZ = MIN (7,12, 23, f, g)
```

SQRT → Realiza y devuelve la raíz cuadrada de su argumento.

Ejemplo:

```
Y = SQRT(4.0)  ──────►  Y= 2.0
```

EXP → Eleva el número e a la potencia indicada.

Ejemplo:

```
R = 7.2
Y = EXP (R)          ⟶  y = e^7.2
Z = X + EXP (X+3.0)  ⟶  z = x + e^(x+3)
```

LOG → Calcula el logaritmo natural.

Ejemplo:

```
Y = LOG (24.32)  ⟶  y = ln 24.32
```

LOG10 → Calcula el logaritmo decimal o en base 10.

Ejemplo:

```
Y= LOG10 (10.)          ⟶  y = 1.0
Tdb = 20.0*LOG10  (Vout/Vin)
```

MOD → Calcula el resto de dividir el primer argumento entre el segundo.

Ejemplo:

```
Y= MOD (14, 7)  ⟶  y = 0.0
```

Funciones intrínsecas trigonométricas.

SIN → seno	ASIN → arcoseno	SINH → seno hiperbólico
COS → coseno	ACOS → arcococoseno	COSH → coseno hiperbólico
TAN → tangente	ATAN → arcotangente	TANH → tangente hiperbólica

- Los argumentos de las funciones SIN, COS y TAN siempre hay que darlos en radianes y entre paréntesis. Si utilizamos las variantes SIND, COSD y TAND, el argumento debe estar en grados.
- Las funciones ASIN y ATAN devuelven radianes en el intervalo $\left[-\frac{\pi}{2},\frac{\pi}{2}\right]$ y ACOS en $[0,\pi]$. Las variantes ASIND, ACOSD y ATAND devuelven grados en el mismo intervalo en que lo hacen ASIN, ACOS y ATAN .
- Estas funciones, como otras funciones intrínsecas, siempre deben de ir a la derecha del igual en la instrucción de asignación.

Ejemplos:

```
PI = 4.0*ATAN(1.0)
Y = SIN(X) * ACOS (2*PI)
```

Sentencias READ y PRINT en formato libre.

En muchas ocasiones, para completar nuestro programa, necesitamos introducir datos. Y es claro, que al finalizar la ejecución de un programa, debemos indicar al ordenador que nos facilite el resultado de la ejecución. La entrada y salida de datos las hacemos con las proposiciones READ, WRITE y PRINT. En este apartado vamos a ver las formas más sencillas de estos comandos, las formas de formato libre.

Para introducir datos se emplea la instrucción READ, cuya sintaxis es:

```
READ  *,  V1, V2, ..., VN
```

En la anterior sentencia, el carácter especial * significa en formato libre. V1, V2,..., VN son las variables a las que vamos a dar valores a través del teclado y la pantalla del ordenador. Los valores deben de ser tecleados separados por comas o espacios en blanco. Los valores enteros no deben tener punto decimal, los valores reales pueden no tenerlo, pero se recomienda ponérselo.

Ejemplo:

```
READ  *, A, B, BB, Y
```

La sentencia READ en formato libre también puede escribirse:

```
READ (*,*) V1, V2, ..., VN
```

En esta forma, el primer asterisco (*) del paréntesis indica que el dispositivo de entrada de datos es el dispositivo por defecto, en nuestro caso, el teclado y la pantalla. El segundo asterisco (*) indica que el formato es libre y podemos introducir reales con o sin exponente y con el número de decimales que estimemos oportuno.

Para sacar datos del programa mediante escritura en la pantalla, utilizamos la sentencia PRINT, con la siguiente sintaxis:

```
PRINT  *,  V1, V2, ..., VN
```

Como en la sentencia READ, el asterisco (*) significa en formato libre y V1, V2, ..., VN es la lista de variables o conjunto de caracteres entre apóstrofes que queremos imprimir.

La primera columna de la pantalla no es utilizada por el comando PRINT cuando escribe variables enteras, reales o complejas. El resto de columnas las divide en partes iguales y en cada parte coloca una variable. Si las variables no caben en una fila, el compilador utilizará la siguiente fila. Cuando se escriben variables carácter sí se colocan también en la primera línea.

Ejemplo:

```
PRINT *, A, 'Hola', B, 'C=', C
```

Las dos sentencias siguientes son equivalentes:

```
PRINT  *,  V1, V2, ..., VN
WRITE (*,* ) V1, V2, ..., VN
```

Delante de una instrucción READ en formato libre, se suele colocar una sentencia PRINT de la forma

```
PRINT *, 'Escribe el valor de la variable Vin de entrada'
READ *, Vin
```

para que sepamos que el ordenador está esperando a que introduzcamos unos datos. Suele ocurrir con programadores novatos despistados que al llegar a la ejecución de una sentencia READ por pantalla, el ordenador queda esperando la introducción de estos datos y el programador mira impaciente la pantalla esperando una indicación del ordenador.

La sentencia siguiente genera una línea en blanco.

```
PRINT *
```

PARAMETER.

Proporciona un nombre simbólico a un valor constante que no puede ser modificado durante la ejecución del programa. La sintaxis es:

```
PARAMETER  (nombre1 = valor1, nombre2 = valor2, ...)
```

Ejemplo:

```
PARAMETER  (PI = 3.14,  C = 3.0E8, CINCO = 5.0)
```

El valor del parámetro no puede ser alterado a lo largo del programa mediante una proposición de asignación o cualquier otro procedimiento. Por ejemplo, no está permitido y el compilador no admitirá las siguientes instrucciones:

```
PARAMETER (TASA = 4.3)
...
TASA = TASA + 1.2
```

Las variables no declaradas de forma natural (es decir, enteras que comiencen por I, J, K, L, M, N, y reales que comiencen por el resto de las letras) deben de declararse antes de darle valor con el comando PARAMETER.

Ejemplo:

```
CHARACTER*5 A, B
PARAMETER (PI = 3.14, A = 'HOLA', B = 'ADIOS')
```

MATRICES, CONJUNTOS, ARRAYS O AGRUPACIONES.

Con las cuatro formas anteriores, designan diferentes autores a las agrupaciones de variables que vamos a tratar en este apartado y las que nosotros nos referiremos con las palabras matriz o matrices.

Una matriz es un conjunto de variables ordenadas de forma que nos podemos referir a cualquier elemento del conjunto por el lugar que ocupa.

La forma general de la matriz es :

A (i, j, k, ...)

donde :

A → es el nombre de la matriz. Como sucede con las variables, en Fortran ortodoxo el nombre de la matriz no puede exceder de seis caracteres. La primera letra del nombre de la matriz identifica a las componentes de la matriz como enteras o reales, siguiendo las mismas reglas que hemos visto para las variables. Si no queremos el tipo por defecto (real o entera) debemos declarar la matriz al principio del programa.

i, j, k, … → contadores enteros que actúan como subíndices. Deben estar separados por comas y su número está limitado a siete. Un elemento de la matriz queda determinado a través de su posición y es denominado como A(i, j, k, ...).

- Es obligatorio dimensionar una matriz antes de utilizarla. Cada vez que el programa compilador se encuentra con el nombre de una variable, le asigna una posición de memoria. Para permitir al compilador asignar a una matriz el número correcto de posiciones de memoria hay que dimensionarla, mediante la proposición DIMENSION, para que el compilador sepa la memoria que tiene que reservar para la matriz.

```
DIMENSION   A(n1), B(n2, n3, n4), C(n6, n7)
```

donde A, B, C son los nombres de las matrices y n1, n2, ..., las dimensiones máximas de las matrices.

Ejemplo:

```
DIMENSION  VECTOR (15)  ⟶  Conjunto de 15 elementos:
VECTOR(1), VECTOR(2), ..., VECTOR (15)

DIMENSION AA (3, 3)  ⟶  Matriz cuadrada 3x3 (9 elementos)
```

• Si no queremos que el límite inferior de los contadores sea 1, debemos especificar el límite inferior y el superior separados por dos puntos (:), como en los siguientes ejemplos:

```
DIMENSION P(0:10) ⟶ Conjunto de 11 elementos: P(0), ..., P(10)
DIMENSION E(-10:20)⟶ Conjunto de 31 elementos: E(-10), ..., E(20)
DIMENSION  Q(1:3, 2:4, -1:4)⟶ Conjunto de 3x3x6 elementos
```

• En una proposición DIMENSION podemos englobar la declaración de varias matrices. Por ejemplo, las tres declaraciones anteriores son equivalentes a

```
DIMENSION   P(0:10),  E(-10,20), Q(1:3, 2:4, -1:4)
```

• En la proposición DIMENSION, también pueden utilizarse expresiones y operaciones con constantes enteras.

Ejemplo:

```
DIMENSION H(5+30, 10), K2(10**2)
```

• De una matriz hay que especificar el tipo (para el que se siguen las mismas reglas que para las variables) y sus dimensiones. Ejemplo:

```
INTEGER  V
DIMENSION V(3), MI(5,2), R(3,3)
```

En el anterior ejemplo V es entera porque así la hemos declarado, sino la hubiésemos declarado sería real. La matriz MI es entera por estar su primera letra en el conjunto de letras I-N que dan entero por defecto, mientras que la matriz R es real por defecto.

• Pueden unirse la declaración de tipo con la de dimensión. Por ejemplo:

```
REAL  X(10,10)
```

especifica una matriz real de 10x10 componentes. Algunos autores desaconsejan esta práctica, por parecerles menos clara que una proposición de tipo combinada con la proposición DIMENSION, pero nadie sigue este consejo que parece bastante absurdo, utilizándose proposiciones combinadas para definir y dimensionar matrices, como

```
REAL  IG(3,3), A2(2,1200)
COMPLEX  CC(9,9), Z5(2)
INTEGER V(3), IV(3)
DOUBLE PRECISION  E2E (12)
```

En Fortran 90, puede utilizarse una sentencia mezcla de la de declaración de tipo y la de dimensionar matrices, como la siguiente:

```
REAL, DIMENSION A(20), B(0:3, 6)
```

• Si tenemos varias matrices, todas ellas con dimensiones máximas del mismo valor o valores relacionados, a las que frecuentemente cambiamos este valor máximo en sucesivas ejecuciones, podemos utilizar la proposición PARAMETER para simplificar la situación.

Ejemplo: Supongamos que tenemos las matrices B (6,6,6), C(6,6), D(7), podemos dimensionarlas de la siguiente forma:

```
PARAMETER  (I =6 )
DIMENSION  B(I,I,I),  C(I,I), D(I+1)
```

• Los elementos de una matriz pueden ser utilizados a lo largo del programa como cualquier otra variable.

Ejemplos:

```
Z= R(3,2)  ⟶  Asigna a Z el valor del elemento R(3,2)
ANT= SIN(A(I,J))**2
```

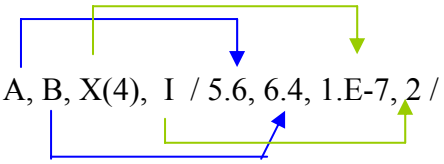
Sentencia DATA.

Para dar valores inicialmente a las variables o matrices, además de una proposición de asignación o una proposición READ, puede utilizarse también una sentencia DATA. Se coloca después de las proposiciones de especificación, tales como DIMENSION. Su sintaxis general es la siguiente:

```
DATA  lista de variables separadas por comas /lista de valores /, variables/valores/, ...
```

Ejemplo:

```
DATA  A, B, X(4),  I  / 5.6, 6.4, 1.E-7, 2 /
```



que es equivalente a

```
DATA  A,B, X(4) / 5.6, 6.4, 1.0E-7 /, I /2/
```

y ambas son equivalentes a

```
A = 5.6
B = 6.4
X(4) = 1.E-7
I = 2
```

• Un valor puede repetirse utilizando un entero que especifica el número de repeticiones y un asterisco delante del valor que vamos a repetir. Con frecuencia se utiliza esta opción para poner a cero matrices o variables.

Ejemplo:

```
DIMENSION G(100)
DATA C1, C2 /2*0.0/, G/100*0.0/
```

Otro ejemplo:

```
LOGICAL  A, B
CHARACTER  NOM*5
DATA  I, J, K /3, 5, 7/, A, B /2*.TRUE./, NOM /'MAPAS'/
```

• En la sentencia DATA puede utilizarse lo que se llama un DO implícito.

Ejemplo:

```
DATA (J(I), I =1, 3) /10, 20, 30/
```

es equivalente a

```
DATA  J(1), J(2), J(3) /10, 20, 30/
```

Otro ejemplo:

```
DIMENSION  IGAMMA (10,15)
DATA ((IGAMMA(I, J), I = 6, 10), J=1, 15) /75*0/
```

La segunda sentencia de este ejemplo inicializa a cero las últimas cinco filas de la matriz dimensionada en la primera instrucción. Para cada valor de J, el contador I, que está en un paréntesis más profundo, toma todos sus valores. El orden de la asignación sería:

```
(6, 1)   (7, 1)   (8, 1)   (9, 1)   (10, 1)
(6, 2)   (7, 2)   (8, 2)   (9, 2)   (10, 2)
                  ...
(6, 15)  (7, 15)  (8, 15)  (9, 15)  (10, 15)
```

Proposición GOTO incondicional.

Sintaxis:

```
GOTO  número de etiqueta
```

Esta sentencia transfiere el control del programa a la etiqueta cuyo número se especifica en el GOTO.

Ejemplo:

El siguiente programa pone en pantalla una sucesión indefinida de números enteros y sus cuadrados.

```
      I = 0
15    I = I+1
      J = I**2
      PRINT *, I, J
      GOTO 15
      STOP
      END
```

El programa entra en un bucle cerrado del que no sale nunca. Para parar la ejecución (abortar el programa) pulsar simultáneamente las teclas Control y c.

Operadores de relación.

Operador F77 ó F90	Significado	Operador F90
.LT.	menor que	<
.LE.	menor o igual que	<=
.EQ.	igual a	= =
.NE.	diferente a (no igual a)	/=
.GT.	mayor que	>
.GE.	mayor o igual que	>=

Operadores lógicos.

Operadores booleanos básicos. (Sirven, entre otras cosas, para relacionar los operadores anteriores).

Operador	Nombre	La expresión es verdadera cuando las relaciones a la izquierda y derecha del operador son:
.AND.	Intersección	Ambas verdaderas
.OR.	Unión	Una o ambas son verdaderas
.NOT.	Negación	Lo opuesto es verdadero

Operadores de equivalencia:

Operador	Nombre	La expresión es verdadera cuando las relaciones a la izquierda y derecha del operador son:
.EQV.	Equivalencia lógica	Tienen el mismo valor de verdad (ambas son verdaderas o ambas son falsas).
.NEQV.	No equivalencia lógica	Tienen distinto valor de verdad (una es verdadera y la otra falsa).

La sintaxis general con estos operadores es:

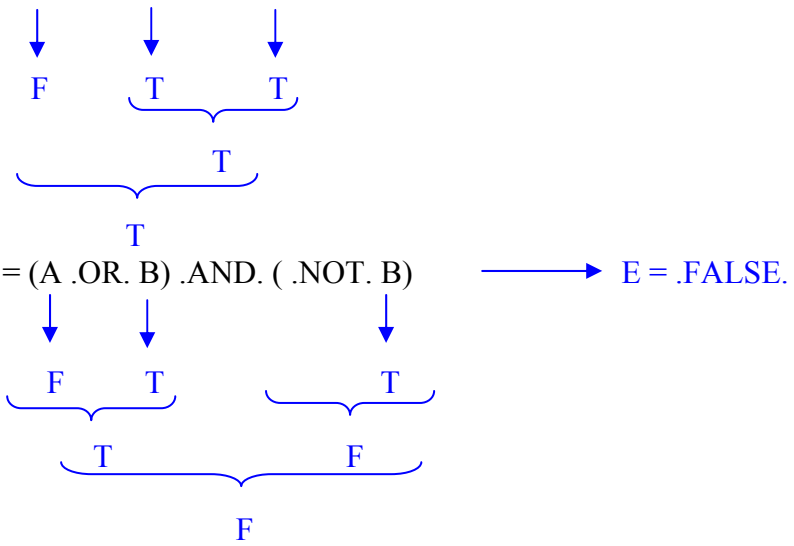
(expresión lógica 1) .operador. (expresión lógica 2)

para todos los operadores, excepto .NOT. que tiene la forma:

.NOT. (expresión lógica)

Ejemplo:

```
LOGICAL A, B, C, D, E
A = .FALSE.
B = .TRUE.
C = .TRUE.
D = A .OR. (B .AND. C)
E = (A .OR. B) .AND. ( .NOT. B)
```



Otro ejemplo:

```
IF (( A .GT. B) .EQV. (C .LT. D)) GOTO 301
```

En la proposición anterior, si A>B y C<D, el control pasa a la línea etiquetada con el 301. Pero también pasará a esta línea, si lo que hay a la derecha e izquierda de .EQV. es falso. Es decir, si $A \leq B$ y $C \geq D$, también se pasa el control a 301.

• Los operadores lógicos .EQV. y .NEQV. tienen un preferencia más baja que los .AND., .OR. y .NOT.. Por tanto, el ejemplo anterior es equivalente a

```
IF ( A .GT. B .EQV. C .LT. D) GOTO 301
```

Bloque IF.

Las sentencias IF forman en Fortran las estructuras de selección. Comenzamos estudiando el bloque IF, que tiene la siguiente sintaxis:

```
IF  ( expresión lógica ) THEN          <-------  obligatoria

        Proposiciones a ejecutar si se cumple la expresión lógica

ELSE                                   <-------  opcional

        Proposiciones a procesar si no se cumple la expresión lógica

END IF                                 <-------  obligatoria
```

- Puede omitirse ELSE y las proposiciones que van debajo de ELSE si no se necesitan.

- De existir, ELSE irá solo en una línea.

Ejemplo:

```
IF  (A.GT.B) THEN
        PRINT*,  'Mayor = ' , A
ELSE
        PRINT*,     'Mayor = ' , B
END IF
```

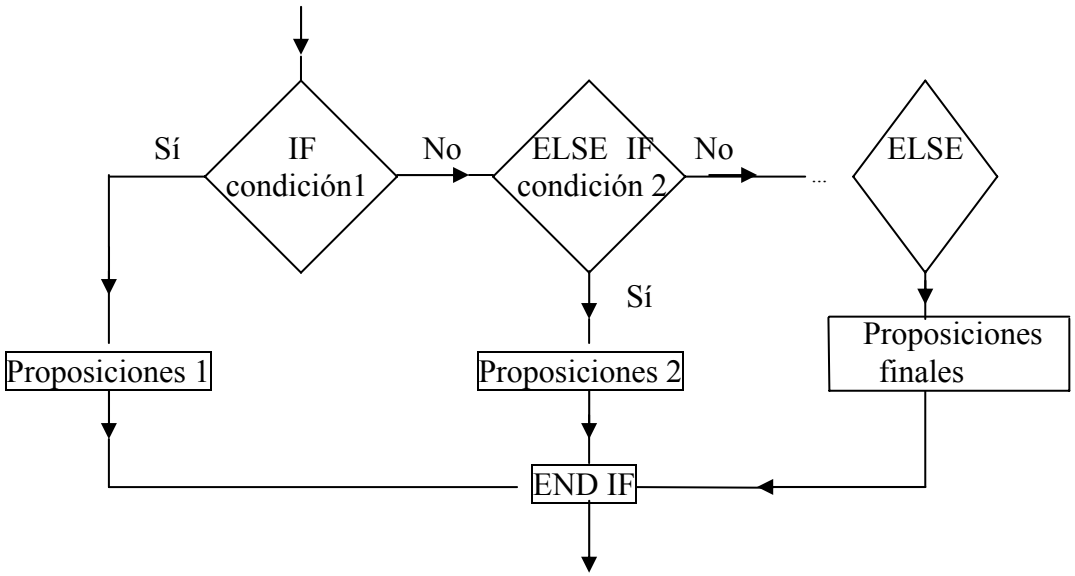
y en el caso de que A y B sean iguales, B aparecería como mayor.

Otro ejemplo:

```
IF (PLBR .EQ. 'FIN') THEN
        PRINT *, 'FIN DE DATOS'
END IF
```

Bloque IF anidado.

El bloque IF permite elegir entre dos posibilidades, pero a veces son más de dos los caminos o bifurcaciones que podemos seguir. Dentro de un bloque IF (pero no dentro de un bloque ELSE) puede programarse una secuencia de pruebas con la declaración ELSE IF que permite generar un camino con múltiples bifurcaciones de entre las cuales sólo elegimos una. El diagrama de flujo es el siguiente:



La sintaxis del bloque IF anidado es:

```
IF    (expresión lógica)  THEN

      Proposiciones a realizar si la expresión lógica anterior es
      verdadera. Si  se realizan estas proposiciones pasamos a END IF.

ELSE IF  (expresión lógica )  THEN

      Proposiciones a realizar si la expresión lógica anterior es
      verdadera. Si se realizan estas proposiciones pasamos a END IF.

ELSE IF  ( expresión lógica)  THEN

                ….        ….        ….

ELSE IF  ( expresión lógica)  THEN

                ….        ….        ….

ELSE

      Proposiciones a realizar si no se cumple ninguna de las
      expresiones lógicas anteriores.

END IF
```

- Cada bloque IF debe finalizar obligatoriamente con una sentencia END IF.
- El bloque ELSE es opcional y puede omitirse.
- Podemos poner tanto bloques ELSE IF como sea necesario.

- Si dentro de un bloque IF anidado, existen varias expresiones lógicas de las sentencias IF o ELSE IF que son verdaderas, sólo se ejecutarán las proposiciones asociadas a la primera expresión lógica que sea verdadera.

Ejemplo:

```
IF  ( A .GT. B )  THEN
        PRINT*,    'Mayor= ' , A
ELSE IF  ( A .EQ. B ) THEN
        PRINT*, 'Son   Iguales'
ELSE   IF   ( A .LT. B ) THEN
        PRINT*,  'Mayor = ', B
ELSE
        PRINT*, 'Imposible llegar aquí'
END IF
```

IF lógico.

Es una versión simplificada de las proposiciones IF anteriores. Su sintaxis es la siguiente:

```
IF  ( expresión lógica )  proposición ejecutable
```

Ejemplos:

```
IF ( A .LE. B )  X = A+B
```

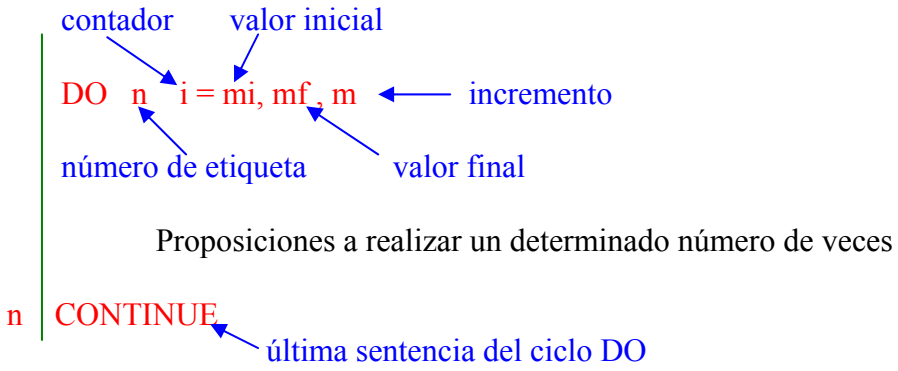
```
IF (X .LE.100 )   GOTO  25
```

```
IF  (A .GT. B )   PRINT*, 'Mayor= ' , A
IF  (A .LT. B )   PRINT*,  'Mayor = ', B
IF  (A .EQ. B )   PRINT*,  'Son   Iguales'
```

Proposición cíclica de repetición DO.

Un ciclo es un conjunto de una o más instrucciones que se ejecutan cierto número de veces, alterando cada vez el valor de una o varias de las variables que intervienen en el ciclo.

Aunque un ciclo puede implementarse con otras proposiciones, generalmente se prefiere la proposición DO. Una posible sintaxis, de entre las varias existentes, para el ciclo DO es la siguiente:



donde

- n → número de etiqueta de la última proposición del ciclo DO.
- i → contador o variable de control del ciclo DO. Generalmente es entera, aunque puede ser real.
- mi → valor inicial del contador o variable del ciclo.
- mf → valor final del contador o variable del ciclo.
- m → incremento del contador en cada ciclo. Si no se especifica vale 1.

La última proposición del ciclo DO (marcada con la etiqueta n) puede ser cualquier proposición susceptible de ser ejecutada, con excepción de una proposición de transferencia de control GOTO, otra sentencia DO, un IF, STOP, END o RETURN. Para evitar problemas se recomienda terminar el ciclo DO con la proposición CONTINUE, llevando esta instrucción, en el caso de que se utilice, la etiqueta n que marca el final del ciclo DO. La proposición CONTINUE no es obligatoria.

Ejemplo:

```
      DO  40   I = 1, 8
      A = A+2.0
      B = B+3.0
40    CONTINUE
```

```
      DO  40   I = 1, 8
      A = A+2.0
40    B = B+3.0
```

Las dos opciones anteriores son igualmente válidas, pero es recomendable la primera opción por mostrar más claramente el final del ciclo. Al finalizar ambos ciclos tendremos A(ahora)=A(inicial)+16.0 y B(ahora)=B(inicial)+24.0. Como puede verse en este ejemplo, el contador o variable del ciclo (en nuestro caso la variable entera i) no

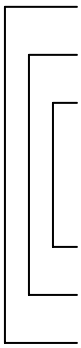
tiene porque intervenir en las sentencias del ciclo, aunque sí puede intervenir, como sucede en el ejemplo siguiente, en el que calculamos sucesivas potencias de un número:

```
      PRINT*,  'Leer A'
      READ  *,  A
      DO  7    I = 1, 10, 1
      B = A**I
      PRINT *,  I, A, B
7     CONTINUE
      STOP
      END
```

Ciclos DO anidados.

Un ciclo DO puede contener otros ciclos DO, pero debe de contenerlos completamente, pudiendo coincidir únicamente la última proposición de los ciclos. Cuando un ciclo contiene a otro, se recomienda utilizar proposiciones CONTINUE diferentes para cada ciclo, así como sangrías (márgenes más interiores) Para las sentencias comprendidas entre DO y CONTINUE.

No existe límite específico a los ciclos que pueden anidarse.



Esquema de anidamiento de ciclos DO

```
      DO  15  I = 10, 1, -2

      DO 15 J = 12, 14
         16
16    CONTINUE
15    CONTINUE
```

(Se recomienda cambiar 15 por 16 y añadir un nuevo CONTINUE con el número de etiqueta 16)

Ejemplo:

```
      SUM = 0.0
      DO 110 I =1, N
            DO 120 J =1, 5
                  SUM = SUM + A(I, J)
120         CONTINUE
110   CONTINUE
```

La variable de control puede o no ser utilizada en las sentencias interiores del ciclo DO, pero al finalizar el ciclo esta dispuesta para ser utilizada con el valor mf+1. Por ejemplo, al finalizar un ciclo DO que comience con la instrucción

```
      DO  I =1, 10, 1
```

la variable I tiene el valor 11 y puede ser empleada con este valor en lo que se necesite.

Proposición END DO.

En Fortran 90 y muchos compiladores de Fortran 77, puede sustituirse la proposición CONTINUE por la proposición END DO. Ambas proposiciones son equivalentes.

Cuando se utiliza END DO, el número de etiqueta n, que aparece tanto en la proposición inicial del ciclo DO como en la final, puede suprimirse. Así son equivalentes los siguientes esquemas de sintaxis:

```
      DO  n   i= mi, mf, m
            Proposiciones del ciclo
n     END DO
```

```
      DO    i = mi,mf, m
            Proposiciones del ciclo
      END DO
```

Proposición DO WHILE.

En Fortran 90 es posible utilizar un nuevo bucle DO WHILE que corresponde a la sintaxis general:

```
      DO  n  WHILE  (condición)
        Proposiciones a ejecutar
n     END DO
```

donde el número de etiqueta n es opcional.

Ejemplo:

```
A = 0
I = 0
DO   WHILE  ( I .LE.10 )
          A = A+I
          I = I+1
END DO
PRINT *, I, A
```

Al finalizar este grupo de instrucciones se escribe en pantalla 11, 55.

Proposiciones EXIT y CYCLE.

Dentro de un ciclo DO podemos utilizar las sentencias EXIT Y CYCLE.

EXIT → Para salir de un ciclo. Esta orden transfiere el control a la 1ª sentencia ejecutable fuera del ciclo DO y nos permite termina el ciclo sin que la variable de control haya podido tomar todos sus valores).

Ejemplo:

```
      DO  J = 1, 100
      …

      IF  ( A .EQ. 0 ) EXIT      (GOTO 111)
      …

      END DO
111
```

Si la variable A toma el valor cero en la ejecución del ciclo, éste se interrumpe y se pasa el control a la 1ª sentencia fuera del ciclo DO.

CYCLE → Devuelve el control al principio del ciclo.

Ejemplo: Con el siguiente bloque de instrucciones ponemos a cero los elementos no diagonales de una matriz.

```
      DO  I = 1, N
          DO  J = 1, N
              IF ( I .EQ. J ) CYCLE     (GOTO 112)
                 A (I, J) = 0.0
112       END DO
      END DO
```

Proposión FORMAT.

Las sentencias FORMAT, OPEN, CLOSE, WRITE y READ, que vamos a ver a continuación, están relacionadas las unas con las otras. No podremos entender la utilización de todas y cada una de ellas hasta que no las hayamos visto todas globalmente.

La proposición FORMAT establece el formato en el cual los datos van a ser escritos o leídos en los ficheros o pantalla. Su sintaxis es la siguiente:

n | FORMAT (lista de especificaciones separadas por comas y entre paréntesis)

Las especificaciones de la lista de especificaciones pueden ser de varios tipos:

a) Para datos.
b) De posicionamiento.
c) Códigos especiales.

En lo que sigue, w, d y n son números enteros que representarán respectivamente:

w → ancho del campo o número de caracteres o posiciones que van a leerse o a escribirse.

d → número de caracteres ó posiciones decimales que siguen al punto decimal.

n → número de repetición o número de veces que se repite lo que va detrás del propio n.

Especificaciones para datos

Fw.d → Entrada ó salida para números reales sin exponente con w caracteres de los cuales d son decimales. El punto decimal ocupa un carácter. Debemos también contar el signo. Ejemplo: El numero -37.125 está en formato F7.3

Ew.d → Entrada ó salida para números reales con exponente. De los w caracteres totales, d son decimales. En w debemos contar el exponente de la potencia de 10, simbolizado con la letra E, el signo del exponente y de la mantisa, así como el punto decimal. Ejemplos: el número -37.125E-2 tiene formato E10.3, y el número 1.25E2, tiene formato E6.2.

Dw.d → Análogo al caso anterior, pero para números reales en doble precisión. Ejemplo: El número +0.456243D-12 está en formato D13.6. En esta especificación de formato y en la anterior, se recomienda que entre el campo total y el número de decimales se debe dejar una diferencia de 6 ó 7 caracteres para poder representar sin problemas el punto decimal, los signos y las letras E ó D.

Iw → Entrada ó salida para números enteros, con un campo de w caracteres que deben incluir el signo. No pueden aparecer puntos decimales ni exponentes. Ejemplo: El número +335 tiene formato I4.

Aw → Entrada ó salida para una variable carácter que dispone de un campo de w posiciones.

Lw → Entrada ó salida de variables lógicas, para las que se dispone de un campo de w posiciones.

Especificaciones de posicionamiento.

X → Salta una posición dejándola en blanco.

nX → Salta n posiciones. Dejándolas en blanco. (Ejemplo: 3X)

/ → Salta a la siguiente línea (escribiendo ó leyendo), posicionándose al principio de la línea.

Ejemplo: Si suponemos dos valores en memoria, tales como A= 88.3 y B= 5.22, empleando las sentencias

```
      WRITE (1, 100) A, B
100   FORMAT (2X, F4.1, 5X, F4.2)
```

se producirá la siguiente salida:

```
_ _ 8 8 . 3 _ _ _ _ _ 5 . 2 2
```

Códigos especiales.

Para producir títulos y encabezamientos, además de la impresión en formato libre utilizando un conjunto de caracteres entre apóstrofes ('), tenemos la opción de poner la frase deseada entre comillas dentro del paréntesis de una proposición FORMAT. Si la salida se realiza por pantalla o impresora, debe tenerse en cuenta que la primera columna de la proposición FORMAT no se imprime, utilizándose como control de la consola (monitor) o del carro de la impresora. Así, si escribimos, compilamos, encuadernamos y ejecutamos el siguiente programa,

```
      WRITE (*, 100)
100   FORMAT ('probado el Fortran')
      STOP
      END
```

la máquina nos mandará un mensaje que seguramente sea cierto.

Observaciones:

- Fortran escribe en la pantalla utilizando 132 caracteres.

- Repetición: Todas las especificaciones de datos pueden repetirse colocando un número entero delante de ella. Por ejemplo, 7F10.3 significa que la especificación F10.3 debe repetirse 7 veces. Si son varias las especificaciones que se quieren repetir, se encierran en un paréntesis y se coloca el número de repetición delante del paréntesis, por ejemplo 3(F10.3, I3)

- Si el número a imprimir es de tamaño más pequeño que el campo, se carga hacia la derecha. También sucede así, si es una variable carácter la que se está imprimiendo. Por ejemplo, 'PIO' en formato A6 se imprimirá ___PIO.

- Si el número no se puede imprimir correctamente en la especificación de formato que se le intenta dar, se produce un error por derrama y se imprimen asteriscos (****). Si es una variable carácter sólo se imprimirán los caracteres que quepan. Por ejemplo, 'PIO' en formato A1 queda P al imprimir.

- En la especificación E y D, deberemos de poner un ancho de campo de 6 ó 7 caracteres mayor que el número de decimales deseados, ya que hay que tener en cuenta los signos, el punto decimal y las letras E o D.

- Reutilización de la proposición FORMAT: Cuando se agotan las especificaciones de FORMAT y la proposición WRITE o READ continúa diciendo que hay más variables para leer o escribir, se comienza un nuevo registro (línea) repitiendo las especificaciones de FORMAT desde el paréntesis izquierdo más cercano al final.

- El paréntesis de cierre de la proposición FORMAT es equivalente a un salto de línea.

- La especificación de formato (1P, Ew.d) hace que se escriba un dígito delante del punto decimal (forma científica de coma flotante).

Ejemplo:

Supongamos que tenemos el siguiente fichero, formado por 20 filas

_ _ 1_ _ 35.42E-2 _ _ 35.44E-2

_ _ 2_ _ 12.12E-3 _ _ 14.22E-1

.....

_ 20_ _ 05.89E+1 _ _ 11.11E-1

El formato para leer éste fichero sería :

```
100 | FORMAT  ( 1X, I2, 2 (2X,E8.2) )
```

El formato siguiente sirve para leer sólo la última columna:

```
300 | FORMAT  (15X, E8.2)
```

Otro ejemplo: Las líneas

```
_ _PEPE_ _22.7_ _12
_ _ _ _ _ _ 14.3E+12_ _
```

salto de línea

se leen o escriben con el formato:

```
200 FORMAT  (2X, A4, 2x, F4.1, 2X, I2, /, 6x, E8.1, 2X)
```

Proposición OPEN.

La instrucción OPEN permite abrir un fichero. Fortran asocia o encadena a cada fichero que se abre un número de unidad, de forma que el número de unidad y el fichero denominan el mismo ente. Antes de escribir o leer de un fichero hay que abrirlo.

La sintaxis general es

```
| OPEN ( lista de opciones separadas por comas)
```

Lista de opciones ó especificaciones

De entre las especificaciones posibles, nosotros sólo comentaremos las más básicas. Estas son:

UNIT = expresión entera

UNIT indica el número de unidad a la que se va a conectar el fichero especificado por FILE. Si UNIT se coloca en primer lugar en la lista de especificaciones, no es necesario escribir UNIT= y el primer número entero después del paréntesis se sobreentiende que es la unidad.

FILE = nombre de fichero

FILE especifica el nombre del fichero que se conecta a la unidad especificada por UNIT. Si el fichero no existe, la sentencia OPEN lo crea. Si el fichero está en el mismo directorio o carpeta que el programa ejecutable Fortran, no es necesario especificar nada más que el nombre del fichero; pero si el fichero está o lo queremos en

otro directorio, hay que especificar su *pathname* completo. Al nombre del fichero se le puede añadir la extensión deseada (.txt, .dat), o puede no llevar extensión.

ERR = número de etiqueta de sentencia

Si se produce un error al abrir el fichero, ERR transfiere el control del programa a la sentencia indicada por el número de etiqueta. Esta especificación no es obligatoria.

Ejemplo:

```
OPEN (UNIT = 12, FILE = 'A:\SALIDA.TXT', ERR = 25)
```

Se asocia la unidad 12 con el fichero SALIDA.TXT de la disquetera. Si hubiese problemas al abrir el fichero, se transfiere el control a la sentencia con la etiqueta 25.

Ejemplo:

Las tres instrucciones siguientes son equivalentes:

```
OPEN  ( UNIT = 8, FILE = 'RESULTADOS')
OPEN  ( 8, FILE = 'RESULTADOS')
OPEN  (FILE = 'RESULTADOS', UNIT = 8)
```

Proposición CLOSE.

Esta sentencia rompe la conexión de un fichero con la unidad asignada. Es aconsejable cerrar los ficheros antes de finalizar el programa. Pero si no se cierran con la proposición CLOSE, los cerrará el programa Fortran al llegar a su final.

La sintaxis de esta proposición es:

```
CLOSE  ( lista opciones )
```

Lista de opciones ó especificaciones

De entre las especificaciones posibles, las más básicas son:

UNIT = expresión entera

UNIT indica la unidad de entrada o salida que se va a desconectar. Si UNIT se coloca en primer lugar en la lista de especificaciones, no es necesario escribir UNIT=, y el primer número entero después del paréntesis se sobreentiende que es la unidad que se va a cerrar.

ERR = número de etiqueta de sentencia

Si se produce un error al desconectar o cerrar el fichero, ERR transfiere el control del programa a la sentencia indicada por el número de etiqueta. Esta especificación no es obligatoria

- Entre la lista de opciones de CLOSE no se encuentra la especificación FILE.

Ejemplos:

```
CLOSE (8)
CLOSE (ERR=32, UNIT = 12)
CLOSE (12, ERR=32)
CLOSE (UNIT =12, ERR=32)
```

} equivalentes

Sentencias READ y WRITE.

Ya vimos las sentencias READ y PRINT (WRITE) en formato libre que nos permitieron una comunicación con el programa a través del monitor y el teclado. Ahora vamos a ver la forma general de las proposiciones READ y WRITE. Las estudiaremos conjuntamente ya que tiene la misma sintaxis y prácticamente la misma lista de especificaciones.

READ permite leer y transferir información desde un fichero o desde el conjunto monitor-teclado a la memoria principal donde se está ejecutando el programa.

WRITE permite escribir, en un fichero o en el monitor, la información procedente de la memoria principal.

Para leer o escribir en un fichero, éste debe de ser previamente abierto y conectado a una unidad mediante una proposición OPEN.

Sintaxis:

```
READ  ( lista de especificaciones) lista de variables
WRITE  ( lista de especificaciones) lista de variables
```

Lista de especificaciones

De entre las especificaciones posibles, las más básicas son:

UNIT = expresión entera

UNIT indica la unidad de entrada o salida en la que vamos a escribir o leer. Si es el conjunto teclado-monitor (dispositivo por defecto), utilizamos el asterisco (*) en lugar de un número entero.

FMT = número de etiqueta

El número de etiqueta que acompaña a la especificación FMT es el correspondiente a la proposición FORMAT que contiene el formato que vamos a utilizar en nuestra lectura o escritura. Si leemos o escribimos en formato libre, utilizaremos el asterisco (*) en lugar de un número de etiqueta.

ERR = número de etiqueta de sentencia

ERR especifica el número de etiqueta a la que se transfiere el control del programa en caso de que se produzca un error en la lectura o escritura. Esta especificación no es obligatoria.

Observaciones:

- Entre la lista de opciones de READ y WRITE no se encuentra la especificación FILE. Ya se conoce el fichero en el que vamos a escribir o del que vamos a leer a través de la proposición OPEN que conecta número de unidad y fichero.

- Obligatoriamente debemos de especificar la unidad y el formato.

- En la lista de especificaciones, el identificador alfabético del número de unidad, UNIT=, y el de etiqueta de formato, FMT=, son optativos, Si se utilizan estos identificadores alfabéticos, la lista de especificaciones puede tener cualquier orden, pero si se omite, el número de unidad debe ocupar el primer lugar en la lista y la etiqueta de formato, el segundo.

  Ejemplo: Son equivalentes, las tres siguientes proposiciones de lectura:

  ```
  READ  (UNIT = 8, FMT = 100 ) X, Y, Z
  READ ( 8,100) X, Y, Z
  READ (8, FMT = 100) X, Y, Z
  ```

- No se puede omitir FMT=, si no se omite también UNIT=. Así, sería incorrecto escribir:

  ~~WRITE (UNIT = 4, 700) ALFA~~

- Lectura y escritura utilizando el dispositivo por defecto o estándar.
  Si utilizamos el dispositivo estándar, normalmente el conjunto monitor-teclado, podemos utilizar una versión reducida de las instrucciones de lectura y escritura, con la siguiente sintaxis:

```
READ número de etiqueta de formato, lista de variables

PRINT número de etiqueta de formato, lista de variables
```

  Ejemplo:

```
READ 200, A, BETA, I

PRINT 101, COF, F
```

  Obsérvese, que en esta versión reducida PRINT sustituye a WRITE. Es incorrecto:

```
WRITE 100, COF, F
```

- Si el número de etiqueta de formato se sustituye por * (formato libre), estas instrucciones se convierten en escritura y lectura con formato libre que ya vimos anteriormente.

- Ciclo DO implícito en entrada y salida.
  Una característica Fortran sumamente útil, que puede simplificar la entrada o salida de variables con subíndices (matrices), es el ciclo DO implícito que ya vimos en la proposición DATA. Ejemplos de utilización de este ciclo DO implícito con sentencias READ y WRITE son los siguientes:

```
READ  (5, 700)   ( A( I ), I =1, N )

WRITE ( 6, 200)  ( ( F( I, J ) , J = 1, 25 ), I = 1, 25 )
```

Esta estructura puede ser sumamente útil en la lectura y escritura adecuada de matrices si se tiene en cuenta que, al finalizar una instrucción de lectura o escritura en un fichero, la posición o cursor dentro del fichero pasa al principio de la siguiente línea. En la siguiente instrucción se lee una matriz vector y una matriz cuadrada:

```
READ  (5, 999)   ( P( I ), (Q(I, J), J=1, 10), I =1, 5 )
```

Función interna o de una sola línea.

Fortran permite definir funciones de una sola proposición dentro del programa principal, a las que llamamos funciones internas. Una vez definida, las funciones internas se utilizan como las funciones intrínsecas.

La sintaxis de definición de una función es la siguiente:

```
nombre-función  ( argumentos entre comas ) = expresión
```

Ejemplo:

```
FUERZA (K, X) = -K*X          ← proposición de definición
...
TRABAJO = FUERZA  ( 1.5, Y )*Y      ← uso de la función
```

Otro ejemplo:

```
F ( X, Y, Z) = 3*X+5*Y+Z          ←  (definición de la función)
...
A = 5.0
READ *, B
C = SIN (ATAN(0.56))
                                  3*A+5*B+C
...

SALIDA = 100.0 * F (A, B, C )     ←  (uso de la función)
```

- La función tiene que declararse en una sola proposición.

- En la expresión de definición de una función puede llamarse a otra función declarada anteriormente.

Ejemplo:

```
ALFA(A)  = A**5 – 6 * SQRT(A)
BETA(X, Y, Z) = X * Y * ALFA(Z)
```

- En la proposición de definición, los argumentos son ficticios ya que son genéricos y no tienen asignado ningún valor.

- La definición de la función debe de ser anterior, lógicamente, a su utilización. Pero también debe de preceder a cualquier proposición ejecutable. Es decir, la definición de una función debe de ir al principio del programa con la declaración de variables.

- Si de manera implícita no es del tipo deseado, hay que declararla.

Ejemplo:

```
REAL  IVELC
IVELC(A) = A*T
```

- Aunque no es recomendable, las variables ficticias que sirven para definir una función pueden ser utilizadas posteriormente a lo largo del programa como variables reales, sin que tengan ninguna relación las ficticias con las reales.

Función externa.

Las funciones externas y las subrutinas son programas aparte que son utilizados y llamados por el programa principal. Las funciones externas y las subrutinas se compilan de manera independiente y en la encuadernación hacen un único bloque con el programa principal.

Una función externa es utilizada por el programa principal como una función intrínseca o una función interna, salvo que hay que definirla en un programa independiente y hay que hacer constar en el programa principal que haremos uso de esa función externa.

En la definición de una función externa podemos hacer uso de las sentencias o instrucciones que sean necesarias. La estructura de sintaxis es la siguiente:

En el programa principal

```
EXTERNAL   nombre
...
Se utiliza como  nombre(argumentos)
```

Definición de la función externa

```
FUNCTION   nombre (argumentos)
...
Nombre =  expresión o expresiones de definición
...
RETURN
END
```

Ejemplo:

En el programa principal:

```
EXTERNAL  SUMA
...
Y= EXP (SUMA(3.0, B))
...
```

Definición de la función externa:

```
FUNCTION SUMA (A, B)
SUMA = A+B
RETURN
END
```

Subrutinas.

Las subrutinas son programas en los que se apoya el programa principal para hacer cálculos u otras tareas que, porque se repiten con frecuencia o por cualquier otra razón, decidimos desgajar del programa principal. Las subrutinas se llaman utilizando la instrucción CALL. La estructura de sintaxis es la siguiente:

Programa Principal

```
...
CALL  nombre (argumentos)
…
```

Definición de la subrutina

```
SUBROUTINE nombre (argumentos)
...
RETURN
END
```

Ejemplo:

Programa principal:

```
READ *,  A,B
CALL  SUMA  (A, B, SUM)
PRINT*,  A, B, SUM
STOP
END
```

Subrutina:

```
SUBROUTINE  SUMA (X, Y, RESULT)
RESULT = X+Y
RETURN
END
```

Utilización de la librería IMSL

La libraría IMSL es un conjunto de más de 1000 subrutinas, escritas en Fortran, para resolver numéricamente problemas matemáticos mediante análisis numérico. Están divididas en tres grandes grupos:

- IMSL Math Library (Análisis numérico).
- IMSL Stat Library (Estadística).
- IMSL Math Library Special Functions (Funciones especiales).

Para ver el contenido de los tres apartados anteriores, una vez abierto la utilidad informática para el desarrollo de programas Fortran, pulsamos Help → Contents y en Contenidos pulsamos sobre el signo + que aparece delante de IMSL Libraries Referente (Pro Edition). Posteriormente, podemos desplegar los diferentes apartados, apareciendo, básicamente, el siguiente contenido:

IMSL Math Library:
Sistemas lineales
Sistemas de autovalores y autovectores
Interpolación y aproximación
Integración y diferenciación
Ecuaciones diferenciales
Transformadas
Ecuaciones no lineales
Optimización
Operaciones con matrices y vectores

IMSL Stat Library:
Regresión
Correlación
Análisis de varianza
Análisis categórico y discreto de datos
Estadística no paramétrica
Pruebas de efectividad para ajustes y aleatoriedad
Análisis de series temporales y predicción
Análisis discriminante
Muestreo
Estimación de riesgo. Confiabilidad
Distribuciones de probabilidad
Numero aleatorios

IMSL Math Library Special Functions:
Funciones Elementales
Funciones trigonométricas e hiperbólicas
Integrales exponenciales y funciones relacionadas
Funciones Gamma y afines
Funciones de Error y afines
Funciones de Bessel

Funciones de Kelvin
Funciones de Airy
Integrales elípticas
Funciones elípticas y afines
Funciones de distribución de probabilidades
Funciones de Mathieu

Una vez localizada la subrutina que mejor se adapta a nuestras necesidades de cálculo, debemos leer detenidamente las instrucciones de utilización, el significado de las diferentes variables que intervienen y podemos apoyarnos en el ejemplo que aparece con todas y cada una de las subrutinas, pudiéndose construir el programa de cálculo haciendo corta/pega sobre el ejemplo.

Las instrucciones que aparecen con la IMSL son las mismas para todas las plataformas en las que se puede utilizar esta librería. No debemos olvidar añadir a nuestro programa la sentencia

```
USE MSIMSL
```

para que haga uso de esta librería. La anterior sentencia no aparece en los ejemplos de la IMSL por no ser propia de la IMSL, sino de la aplicación informática que estamos utilizando.

Ejemplo 1: Cálculo de la integral $\int_0^1 \ln(x)\, x^{-1/2} dx$, utilizando la subrutina QDAGS de la IMSL.

En primer lugar buscamos la subrutina QDAGS y vemos las variables que emplea y la forma de utilización. La función a integrar debe ser introducida como una función externa que se compila separadamente del programa principal. El siguiente programa es una implementación de este cálculo:

```
      USE MSIMSL  !Sentencia que permite que usemos la IMSL.
C     Se declaran las variables de la subrutina de integración.
      REAL F, A, B, ERRABS, ERRREL, ERREST, RESULT
C     La función a integrar se declara como externa.
      EXTERNAL F
C     Límites del intervalo de integración.
      A = 0.0       !Límite inferior
      B = 1.0       !Límite superior
C     Errores que exigimos a la integración
      ERRABS=0.0 !Error absoluto. Si es cero no se considera.
      ERRREL=0.001 !Error relativo deseado.
C     Se llama a la subrutina de integración de la IMSL.
      CALL QDAGS (F,A,B,ERRABS,ERRREL,RESULT,ERREST)
C     Se imprimen resultados.
      WRITE (*,100) RESULT, ERREST
100   FORMAT ('Integral =',F8.3,//,'Error estimado =', 1PE10.3)
      STOP
      END
```

La función externa, que se defina a continuación, debe ser introducida en el mismo proyecto en que está el programa principal.

```
FUNCTION F(X)
F=LOG(X)/SQRT(X)
RETURN
END
```

El resultado que se obtiene después de compilar separadamente, encuadernar conjuntamente para crear un archivo .exe y después ejecutar, es el siguiente:

```
Integral =  -4.000

Error estimado = 2.708E-04
Stop - Program terminated.

Press any key to continue
```

El resultado exacto es: $\int_0^1 \ln(x)\, x^{-1/2} dx = -4$.

Ejemplo 2: Calcular el espectro en amplitud, mediante la transformada discreta de Fourier, de la función:

$$f(t) = \frac{e^{-t}}{4}\left[\cos(2\pi f_1 t) + \cos(2\pi f_2 t) + \cos(2\pi f_3 t) + \cos(2\pi f_4 t)\right]$$

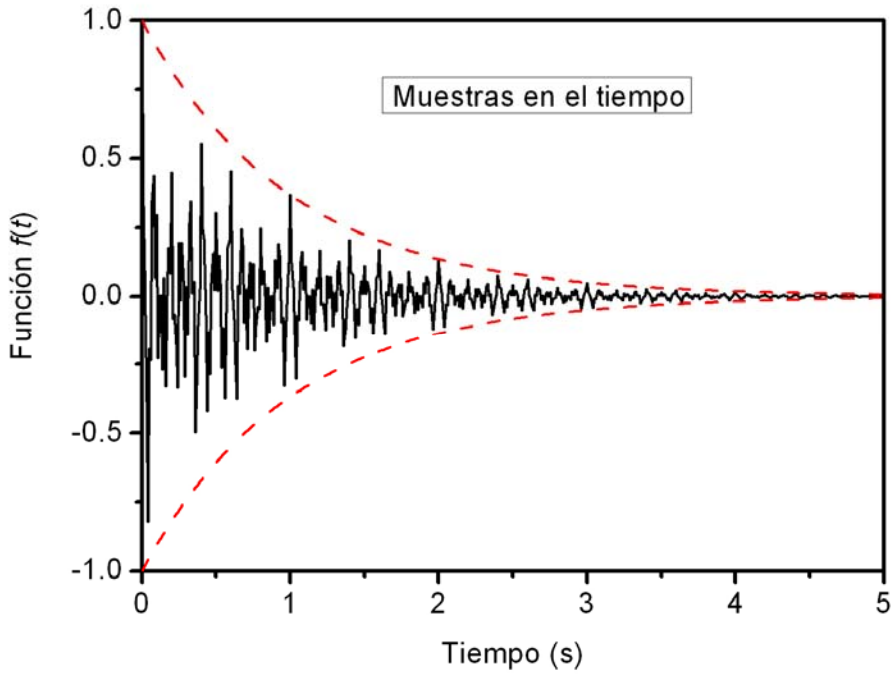
con f_1=10 Hz, f_2=12 Hz, f_3=15 Hz y f_4=40 Hz, que está compuesta por cuatro cosenos amortiguados.

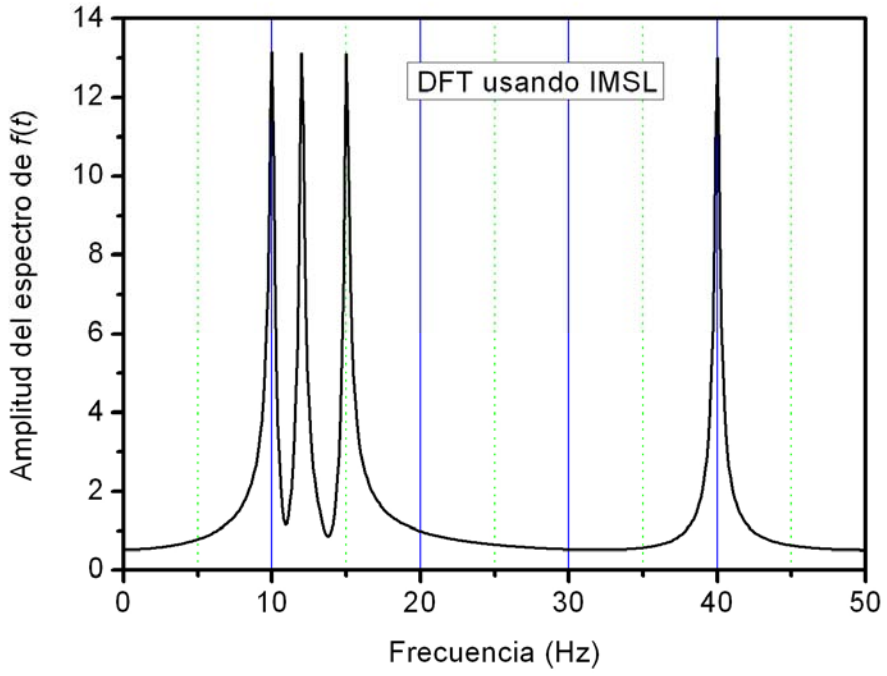
Dentro del apartado Transform-Routines de la IMSL Math Library, vamos a utilizar la primera subrutina que aparece en la lista, la FFTRF, que permite hacer la DFT de una señal real. La DFT asocia una lista de números en el dominio del tiempo con otra lista de números en el dominio de la frecuencia, pero sin relacionar el paso del tiempo con el de la frecuencia, que debemos llevar a cabo nosotros de manera paralela. Ya que esperamos cuatro resonancias, una a cada una de las frecuencias que hemos introducido en la señal, y que estas están entre los 10 y los 40 Hz, un paso adecuado en la frecuencia es $\Delta f = 0.1$ Hz. Esta Δf la podemos conseguir con $\Delta t = 0.01$ s y $N = 1000$ muestras, ya que $\Delta f = \frac{1}{N\Delta t}$.

También deberemos leer detenidamente la información que la IMSL suministra con la subrutina FFTRF ya que en ella se nos indica como relacionar la salida aportada por la subrutina con el espectro de amplitud que buscamos. Todo lo anterior se lleva a cabo en el siguiente programa:

```
      Program DFT
*     Calcula la TF en amplitud usando la librería IMSL
      use msimsl
      parameter (nin=1000,pi2=3.1415926*2.)
*     nin es el número de datos
*     Definición de la función
      f(f1,f2,f3,f4,b,t)=0.25*exp(-b*t)*
     &(cos(f1*pi2*t)+cos(f2*pi2*t)+cos(f3*pi2*t)+cos(f4*pi2*t))
*     SEQ tiene los datos de entrada y COEF los de salida
      dimension SEQ(nin),COEF(nin),P(nin),tm(nin),fm(nin)
*     dt es el incremento del tiempo
      dt=0.01
*     Se definen lo parámetros de la función
      f1=10.0
      f2=12.0
      f3=15.0
      f4=40.0
      b=1.0
*     Se muestrea la función y los datos se introducen en SEQ
*     También se muestrea el tiempo y la frecuencia
      do n=1,nin
      t=(n-1)*dt
      SEQ(n)=f(f1,f2,f3,f4,b,t)
      tm(n)=t
      fm(n)=(n-1)/(nin*dt)
      end do
*     Se llama a la subrutina que realiza la DFT
      call FFTRF (nin,SEQ,COEF)
*     Siguiendo la información dada por la IMSL, el espectro
*     de amplitud viena dado por:
      P(1)=abs(COEF(1))
      do k=2,nin/2
      P(k)=sqrt(COEF(2*k-2)**2+COEF(2*k-1)**2)
      end do
*     Se guardan en ficheros las muestras de las señales
      open (11, file='tiempo.txt')
      open (12, file='frecuencia.txt')
      do i=1,nin/2
      write(11,fmt='(2(2x,e12.6))') tm(i),SEQ(i)
      write(12,fmt='(2(2x,e12.6))') fm(i),P(i)
      end do
      close (11)
      close (12)
      stop
      end
```

Las muestras, tanto en el dominio del tiempo como en el de la frecuencia, se representan en las siguientes figuras, apreciándose en el espectro cuatro picos que se corresponden respectivamente con cada una de las cuatro oscilaciones que forman la señal.





Ejemplo 3: Resolución de un sistema de ecuaciones algebraicas utilizando la librería matemática IMSL.

Como ejemplo concreto y por sencillez, resolveremos el sistema 3×3 siguiente:

$$\begin{pmatrix} 33.0 & 16.0 & 72.0 \\ -24.0 & -10.0 & -57.0 \\ 18.0 & -11.0 & 7.0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 129.0 \\ -96.0 \\ 8.5 \end{pmatrix}$$

En primer lugar, deberemos acudir a la Ayuda de la IMSL y elegir una subrutina. Dentro del aparatado Linear Systems-Routines, de la IMSL Math Library, existen más de 120 subrutinas relacionadas con las diferentes operaciones matemáticas que pueden realizarse son sistemas lineales. Estas subrutinas están clasificadas según su función y el tipo de matrices y coeficientes que intervengan en el sistema de ecuaciones. Dentro del primer apartado, titulado Real General Matrices, la segunda subrutina es la LSLRG cuya finalidad es resolver un sistema lineal y que, por tanto, será la que vamos a utilizar.

Leídas las instrucciones que acompañan a la subrutina LSLRG, debemos dar los siguientes parámetros:

N — Contiene el número de ecuaciones.
A — Matriz cuadrada N×N con los coeficientes del sistema.
LDA — Parámetro entero que, para nuestros propósitos, coincide con N.
B — Vector de longitud N conteniendo los términos independientes.
IPATH — Parámetro que toma el valor 1 para resolver $Ax = b$, y 2 para $Ax^t = b$.
X — Vector de longitud N conteniendo la solución.

El siguiente programa resuelve el sistema:

```
      PROGRAM SISTEMAIMSL
      USE MSIMSL
      PARAMETER (IPATH=1, LDA=3, N=3)
      REAL A(LDA,LDA), B(N), X(N)
C
C     Introducimos las siguientes matrices
C             A = ( 33.0  16.0  72.0)
C                 (-24.0 -10.0 -57.0)
C                 ( 18.0 -11.0   7.0)
C             B = (129.0 -96.0   8.5)
C
      DATA A/33.0,-24.0,18.0,16.0,-10.0,-11.0,72.0,-57.0,7.0/
      DATA B/129.0,-96.0,8.5/
C
      CALL LSLRG (N, A, LDA, B, IPATH, X)
C     Se imprime el resultado
      PRINT *, 'X=(',X,')'
      STOP
      END
```

El resultado aportado por el anterior programa es:

```
X=(   9.999992E-01        1.499999        1.000001)
Stop - Program terminated.

Press any key to continue
```

que salvo por el error numérico asociado a la simple precisión da el resultado $x_1 = 1.0,$ $x_2 = 1.5,$ $x_3 = 1.0,$ solución del sistema.

Sentencias, proposiciones o estamentos Fortran

ALLOCATE (*array*([*I :*]*u*[, [*I :*]*u* ...]) [, **STAT**=ierr])...

ASSIGN *label* **TO** *variable*

AUTOMATIC [*names*]

BACKSPACE { *unitspec* | ([**UNIT** =]*unitspec*{, **ERR**=*errlabel*]
[, **IOSTAT**=*iocheck*])}

BLOCK DATA [*blockdataname*]

BYTE *vname* [[*attrs*]] [(*dim*)] [/*values*/ [, *vname* [[*attrs*]] [(*dim*)] [/*values*/]...

CALL *sub* [([*actuals*])]

CASE (*expressionlist*)

CASE DEFAULT

CHARACTER [**chars*] *vname*[[*attrs*]][**length*][(*dim*)][/*values*/] [,*vname* [[*attrs*]]
[**length*][(*dim*)]][/*values*/]...

CLOSE ([**UNIT**=]*unitspec*[,**ERR**=*errlabel*][,**IOSTAT**=*iocheck*][,**STATUS**=*status*])

COMMON [/[*cname*][[*attrs*]]/]*nlist*[[,]/[*cname*][[*attrs*]]/*nlist*]...

COMPLEX [**bytes*]*vname* [[*attrs*]][**length*][(*dim*)][/*values*/] [, *vname*[[*attrs*]]
[**length*][(*dim*)][/*values*/]]...

CONTINUE

CYCLE

DATA *nlist*/*clist*/[[,]*nlist*/*clist*/]...

DEALLOCATE (*arraytist*[,**STAT**=*ierr*])

DIMENSION *array*[[*attrs*]]({[*lower:*]*upper*}[,{[*lower:*]*upper*}...])...

DO [*label*[,]] *dovar=start, stop*[,*inc*]

DO [*label*[,]] **WHILE** (*logicalexpr*)

DOUBLE COMPLEX *vname* [[*attrs*]][(*dim*)][/*values*/][,*vname*[[*attrs*]]
[(*dim*)][/*values*/]]...

DOUBLE PRECISION *vname*[[*attrs*]][(*dim*)][/*values*/][,*vname*[[*attrs*]]
[(*dim*)][/*values*/]]...

ELSE

ELSE IF (*expression*) **THEN**

END

END DO

END IF

END MAP

END SELECT

END STRUCTURE

END UNION

ENDFILE {*unitspec* | ([**UNIT** =]*unitspec* [, **ERR=***errlabel*][, **IOSTAT=***iocheck*)]}

ENTRY *ename* [[*eattrs*]][([*formal*[[*attrs*]][, *formal* [[*attrs*]]]...])]

EQUIVALENCE (*nlist*)[, (*nlist*)]...

EXIT

EXTERNAL *name*[[*attrs*]][, *name*[[*attrs*]]]...

FORMAT [*editlist*][*type*] **FUNCTION** *func*[[*fattrs*]]([*formal* [[*attrs*]]] [, *formal* [[*attrs*]]]...)

GOTO *label*

GOTO (*labels*) [,] n

GOTO *variable* [[,](*labels*)]

IF (*expression*) *label1, label2, label3*

IF (*expression*) *statement*

IF (*expression*) **THEN**

IMPLICIT {*type*(*letters*)[,*type*(*letters*)]…|**NONE**}

INCLUDE '*filename*'

INQUIRE ({[**UNIT=**]*unitspec*, FILE=*file*}[,**ACCESS=**access][, **BINARY=***binary*]
[,**BLANK=***blank*] ,**BLOCKSIZE=***blocksize*] [,**DIRECT** =*direct*]
[, **ERR=***errlabel*] [,**EXIST=***exist*] ,**FORM**=*form*]
[,**FORMATTED=***formatted*][,**IOFOCUS=***iofocus*][,**IOSTAT=***iocheck*]

[,**MODE**=*mode*][,**NAME**=*name*] [,**NAMED**=*named*] [,**NEXTREC**=*nextrec*]
[,**NUMBER**=num][,**OPENED**=*opened*][,**RECL**=*recl*] [,**SEQUENTIAL**=*seq*]
[,**SHARE**=*share*] [,**UNFORMATTED**=*unformatted*])

INTEGER {[**bytes*] | [[C]]} *vname*[[*attrs*][**length*][(*dim*)][/*values*/][,*vname*[[*attrs*]]
[**length*][(*dim*)][/*values*/)]...

INTERFACE TO {*function-declaration* | *subroutine-declaration*}

INTRINSIC *names*

LOCKING ([**UNIT**=]*unitspec*[,**ERR**=*errlabel*] [,**IOSTAT**=*iocheck*]
[,**LOCKMODE**=*lockmode*][,**REC**=*rec*] [,**RECORDS**=*recnum*])

LOGICAL [**bytes*] *vname*[[*attrs*]][**length*][(*dim*)](/*values*/] [,*vname* [[*attrs*]]
[**length*][(*dim*)][/*values*/]]...

MAP

NAMELIST /*namlst*/ *varlst* [/*namls*/ *varlst*]

OPEN ([**UNIT**=]*unitspec* [,**ACCESS**=*access*] [,**BLANK**=*blank*]
[,BLOCKSlZE=*blocksize*][,**ERR**=*errlabel*] [, **FILE**=*file*]
[,**FORM**=*form*][,**IOSTAT**=*iocheck*] [,**MODE**=*mode*][,**RECL**=*recl*]
[,**SHARE**=*share*] [, **STATUS**=*status*])

PARAMETER (*name=constexpr*[,*name=constexpr*]...)

PAUSE [*prompt*]

PRINT {*,| *formatspec*| *namelist*}[, *iolist*]

PROGRAM *program-name*

READ {{*fonnatspec,*| *nmlspeq*}| ([**UNIT**=]*unitspec*[,[{[**FMT** =]*formatspec*]|
[**NML=]**nmlsped}][,**END**=*endlabel*] [,**ERR**=*errlabel*][,**IOSTAT**=*iocheck*][,
REC=rec])}*iolist*

REAL [**bytes*] *vname*[[*attrs*]][**length*][(*dim*)](/*values*/] [,*vname* [[*attrs*]] [**length*]
[(*dim*)][/*values*/]]...

RECORD /*type-name*/ *vname*[[*attrs*]][(*dim*) [,*vname* [[*altrs*]][(*dim*)]]…

RETURN [*ordinal*]

REWIND { *unitspec*| ([**UNIT**=]*unitspec* [, **ERR**=*err1abel*][, **IOSTAT**=*iocheck*])}

SAVE [*names*]

SELECT CASE (*testexpr*)

STOP [*message*]

STRUCTURE /*type-name*/

SUBROUTINE *subr*[[*sattrs*]] [([*fomal* [[*attrs*]][, *formal* [[*attrs*]]]...])]

UNION

WRITE ([**UNIT=**]*unitspec*[,[{[**FMT =**]*formatspec*]|

[**NML=]**nmlsped}][,**ERR=***errlabel*][,**IOSTAT=***iocheck*]

[,**REC=**rec])}*iolist*

Funciones intrínsecas Fortran

ABS (*gen*)
ACOS (*real*)
AIMAG (*cmp8*)
AINT (*real*)
ALLOCATED (*array*)
ALOG (*real4*)
ALOG10 (*real4*)
AMAX0 (*intA*, *intB* [,*intC*]...)
AMAX1 (*real4A*, *real4B* [, *real4C*]...)
AMIN0 (*intA*, *intB* [,*intC*]...)
AMIN1 (*real4A*, *real4B* [, *real4C*]...)
AMOD (*real4A*, *real4B*)
ANINT (*real*)
ASIN (*real*)
ATAN (*real*)
ATAN2 (*realA*, *realB*)
BTEST (*intA*,*intB*)
CABS (*cmp*)
CCOS (*cmp8*)
CDABS (*cmp16*)
CDCOS (*cmp16*)
CDEXP (*cmp16*)
CDLOG (*cmp16*)
CDSIN (cmp16)
CDSQRT (*cmp16*)
CEXP (*cmp8*)
CHAR (*int*)
CLOG (*cmp8*)
CMPLX (*genA* [,*genB*])
CONJG (*cmp8*)
COS (*gen*)
COSH (*real*)
COTAN (*real*)
CSIN (*cmp8*)
CSQRT (*cmp8*)
DABS (*dbl*)
DACOS (*dbl*)
DASIN (*dbl*)
DATAN (*dbl*)
DATAN2 (*dblA*, *dblB*)
DBLE (*gen*)
DCMPLX (*genA* [,*genB*])
DCONJG (*cmp16*)
DCOS (*dbl*)
DCOSH (*dbl*)
DCOTAN (*dbl*)
DDIM (*dblA*, *dblB*)
DEXP (*dbl*)
DFLOAT (*gen*)
DIM (*genA*, *genB*)
DIMAG (*cmp16*)
DINT (*dbl*)
DLOG (*dbl*)
DLOGl0 (*dbl*)
DMAX1 (*dblA*, *dblB* [,*dblC*]...)
DMIN1 (*dblA*, *dblB* [,*dblC*]...)
DMOD (*dblA*, *dblB*)
DNINT (*dbl*)
DPROD (*real4A*, *real4B*)
DREAL (*cmp16*)
DSIGN (*dblA*, *dblB*)
DSIN (*dbl*)
DSINH *(dbl*)
DSQRT (*dbl*)
DTAN (*dbl*)
DTANH (*dbl*)
EOF (*int*)
EPSILON (*gen*)
EXP (*gen*)
FLOAT (*int*)
HFIX (*gen*)
HUGE (*gen*)
IABS (*int*)
IAND (*intA*, *intB*)
IBCHNG (*intA*, *intB*)
IBCLR (*intA*, *intB*)
IBSET (*intA*, *intB*)
ICHAR (*char*)
IDIM (*intA*, *intB*)
IDINT (*dbl*)
IDNINT (*dbl*)
IEOR (*intA*, *intB*)
IFIX (*real4*)
IMAG (*cmp*)
INDEX (*charA*, *charB* [, *log*])
INT (*gen*)
INTL (*gen*)
INT2 (*gen*)

INT4 (*gen*)
INTC (*gen*)
IOR (*intA*, *intB*)
ISHA (*intA*, *intB*)
ISHC (*intA*, *intB*)
ISHFT (*intA*, *intB*)
ISHL (*intA*, *intB*)
ISIGN (*intA*, *intB*)
JFIX (*gen*)
LEN (*char*)
LEN_TRIM (*char*)
LGE (*charA*, *charB*)
LGT (*charA*, *charB*)
LLE (*charA*, *charB*)
LLT (*charA*, *charB*)
LOC (*gen*)
LOG (*gen*)
LOG10 (*real*)
MAX (*genA*, *genB* [,*genC*]...)
MAX0 (*intA*, *intB* [,*intC*]...)
MAX1 (*real4A*, *real4B* [, *real4C*]...)
MAXEXPONENT (*real*)
MIN (*genA*, *genB* [,*genC*]...)
MIN0 (*intA*, *intB* [,*intC*]...)
MIN1 (*real4A*, *real4B* [, *real4C*]...)
MINEXPONENT (*real*)
MOD (*genA*, *genB*)
NEAREST (*real*, *director*)
NINT (*real*)
NOT (*intA*)
PRECISION (*gen*)
REAL (*gen*)
SCAN (*charA*, *charB* [,*log*])
SIGN (*genA*, *genB*)
SIN (*gen*)
SINH (*real*)
SNGL (*dbl*)
SQRT (*gen*)
TAN (*real*)
TANH *(real*)
TINY (*real*)
VERIFY (*charA*, *charB* [,*log*])

ÍNDICE