Guía de Programación en Fortran 90/95

Ing. Arturo J. López García

2004

Tantas veces me mataron
cuantas veces me morí
sin embargo estoy aquí
resucitando.

Gracias doy a la desgracia
y a la mano con puñal
porque me mató tan mal
y seguí cantando.

Tantas veces me borraron
tantas desaparecí
a mi propio entierro fui
sola y llorando.

Hice un nudo en el pañuelo
pero me olvidé después
que no era la única vez
y seguí cantando.

Como la cigarra
María Elena Walsh

CONTENIDO

INTRODUCCIÓN

FORTRAN fue el primer lenguaje de programación de computadoras usado ampliamente por la comunidad científica en todo el mundo.

La compañía IBM en 1953 empezó a diseñar un lenguaje de alto nivel que fuera una alternativa eficiente y económica a los lenguajes ensambladores usados en aquel tiempo. Como resultado de este esfuerzo, a mediados de 1954, un grupo de trabajo encabezado por John Backus había generado ya un lenguaje de programación flexible, poderoso y eficiente que recibió el nombre de FORTRAN, un acrónimo de FORmula TRANslation (traducción de fórmulas). El primer compilador FORTRAN fue liberado en abril de 1957 para la computadora IBM 704. Un año más tarde, IBM publicó una versión mejorada que llamó FORTRAN II. Sin embargo, FORTRAN II contenía muchas características dependientes del diseño de la computadora, por lo que los usuarios presionaron a IBM para que diseñara un FORTRAN independiente de la arquitectura de la computadora. Resultado de esa presión fue el FORTRAN IV, liberado en 1962 para la computadora IBM 7030 (conocida también como Stretch).

Una vez superada la resistencia al cambio de algunos programadores acostumbrados al uso de ensambladores, el FORTRAN se convirtió en un éxito rotundo de tal manera que los demás fabricantes de computadoras implementaron un compilador de este lenguaje. Esto ocasionó que surgieran muchos dialectos FORTRAN con características incompatibles entre sí, restando portabilidad a los programas. Afortunadamente, un comité de la Asociación Americana de Estándares (ahora Asociación Nacional Americana de Estándares ANSI), definió un estándar basado en FORTRAN IV que recibió posteriormente el nombre de FORTRAN 66.

Dado que el lenguaje FORTRAN siguió evolucionando, en 1969 ANSI instruyó al comité X3J3 encargado de la estandarización del FORTRAN, para que iniciara la elaboración de un nuevo estándar. Ese nuevo estándar fue aceptado hasta 1978 y se conoce como FORTRAN 77 e incluye al tipo CHARACTER, a la construcción IF-THEN-ELSE y algunas facilidades adicionales de lectura/escritura.

Casi inmediatamente después de la publicación oficial del FORTRAN 77, el comité X3J3 se avocó a producir un futuro estándar. Este nuevo estándar recibió el nombre de FORTRAN 90 y es un ejemplo muy logrado de un lenguaje que combina la legendaria eficiencia del código obtenida por un compilador FORTRAN con los últimos avances en la teoría de la programación de las computadoras. Por otra parte, FORTRAN 77 es un subconjunto de FORTRAN 90, lo que implica que los viejos

programas FORTRAN 77 son compatibles con el nuevo estándar. Entre la característica más sobresalientes del FORTRAN 90 se encuentran:

- Forma libre para la escritura del código fuente
- Atributos de los tipos de datos
- Tipos de datos con parámetros
- Tipos derivados
- Asignación dinámica de memoria
- Apuntadores
- Operaciones con arreglos
- Nuevas estructuras de control
- Módulos y bloque de interfases
- Procedimientos internos
- Uso de parámetros opcionales y/o parámetros clave
- Recursividad
- Enmascaramiento de operadores
- Definición de nuevos operadores
- Procedimientos genéricos
- Mejoras en lectura/escritura
- Nuevas funciones intrínsecas

Todas estas nuevas características fueron adoptadas sin sacrificar la facilidad de uso y de aprendizaje tradicionales del lenguaje FORTRAN.

FORTRAN 90 fue adoptado como estándar ISO a nivel mundial en 1991 y como estándar ANSI en Estados Unidos en 1992.

El cómite X3J3 siguió trabajando y publicó en 1997 el último estándar oficial: FORTRAN95. Este consiste del estándar FORTRAN 90 más agregados tales como la estructura FOR ALL, los procedimientos puros y elementales y algunas mejoras en ciertas funciones intrínsecas. FORTRAN 95 también es una estándar ANSI en Estados Unidos e ISO a nivel mundial.

El próximo estándar ya está definido en su totalidad y sólo faltan algunos detalles menores para su aceptación formal y será conocido como FORTRAN 2003. Se espera su adopción para finales de 2004 o principios de 2005 y consiste básicamente en FORTRAN 95 más un soporte total a la programación orientada a objetos.

ELEMENTOS DE FORTRAN 90

Conjunto de caracteres

Letras (alfabeto inglés):

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

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

Dígitos:

0 1 2 3 4 5 6 7 8 9

Caracteres especiales:

= + - * / ( ) , . ' : ! " % & ; < > ? $ _ espacio

Las letras mayúsculas son equivalentes a las minúsculas, excepto en el valor de las constantes de caracteres:

ABC es equivalente a abc

'IPN' es diferente a 'ipn'

En FORTRAN 77, los caracteres especiales válidos son:

= + - * / , . ' : $ espacio

? y $ no tienen algún significado especial

Nombres

Los nombres son usados en FORTRAN para identificar entidades tales como variables, unidades de programa, tipos definidos por el usuario, estructuras, etc.. Un nombre comienza con una letra, puede ser hasta de 31 caracteres de longitud y consiste de una combinación de letras, dígitos y el carácter _. Ejemplos:

ARC FS0 Presion_de_vapor I Y MIN

En FORTRAN 77, los nombres tienen una longitud máxima de 6 caracteres y no pueden incluir al carácter _. Los nombres también son conocidos como identificadores.

Etiquetas

Las instrucciones FORTRAN pueden tener etiquetas para su identificación. Una etiqueta consiste de 1 a 5 dígitos, al menos uno de ellos, diferente de 0. Algunas etiquetas válidas son:

123 5000 9 0037 37

Las etiquetas 0037 y 37 son equivalentes.

Formas de escritura del código fuente

FORTRAN 90 ofrece dos formas de escribir el código fuente: fija y libre.

Forma fija:

Las columnas de la 1 a la 5 están reservadas para etiquetas. Las etiquetas pueden contener espacios.
La columna 6 es usada para indicar la continuación de la línea anterior mediante un carácter de continuación (cualquiera, diferente del espacio o del 0). Puede haber hasta 19 líneas de continuación. Las líneas de continuación no pueden estar etiquetadas. Una instrucción END no puede ser continuada.
Las columnas 7 a la 72 son usadas para las instrucciones FORTRAN.
La información escrita después de la columna 72 es ignorada.
Los comentarios se indican comenzando una línea con una C o un * en la columna 1. Adicionalmente, los comentarios al final de una instrucción pueden ser escritos en cualquier columna (excepto la 6), antecediendo a su texto un !. Las líneas de comentarios no pueden ser continuadas.
El ; puede ser usado para separar varias instrucciones escritas en una misma línea.
Los espacios son ignorados, excepto en constantes de caracteres.

Forma libre:

Una línea puede tener una longitud hasta de 132 caracteres.
Los espacios son usados para separar nombres, constantes o etiquetas de nombres, constantes o etiquetas adyacentes. Los espacios también son utilizados para separar los nombres de las instrucciones FORTRAN.

Un ! inicia un comentario, incluyendo a los escritos al final de una instrucción.
El ; puede ser usado para separar instrucciones en una misma línea.
El & , empleado como último carácter (no espacio, no comentario) de una línea, indica que ésta va a ser continuada en la siguiente línea distinta de un comentario. Si un nombre, instrucción o etiqueta no son terminados en una línea, el primer carácter diferente del espacio en la línea de continuación deberá ser un & seguido de los caracteres restantes. Se permiten hasta 39 líneas de continuación.
Los comentarios no pueden ser continuados en otra línea.

FORTRAN 77 sólo admite el código fuente en forma fija, pero excluye a los comentarios con ! y no es posible escribir más de una instrucción en una línea.

```
c  Ejemplo de código en forma fija
*234567
        read *, a,b
        c= (a+
     1     b)/2   ! promedio de a y b
        print *, a,b,c
        end

!  Ejemplos de código en forma libre
!
read *,a,b;c=(a+b)&
/2   ! promedio de a y b
print *,a,b,c
end

! Otro ejemplo de código en forma libre
! Cálculo de la raíz de a/b
a= 5 ; b= 7.87
c= sqr&
&t(a/b)    ! continuación de la función sqrt (raíz cuadrada)
print *,a,b,c
end
```

Tipos de datos intrínsecos

Los 5 tipos de datos intrínsecos de FORTRAN son: INTEGER, REAL, COMPLEX, LOGICAL y CHARACTER. Los tipos de datos intrínsecos tienen una o más clases. Para los tipos CHARACTER,

INTEGER, REAL y LOGICAL, la clase indica el número de bytes usados para representar cada dato. En el caso del tipo COMPLEX, la clase es el número de bytes usados para representar tanto a la parte real como a la parte imaginaria. En la siguiente tabla se muestran las clases disponibles para una computadora personal, así como los intervalos dentro de los cuales es posible representar información y el número aproximado de dígitos decimales de precisión para los tipos REAL y COMPLEX:

Tipo	Clase	Notas
INTEGER	1	intervalo:-127 a 127
INTEGER	2	intervalo:-32767 a 32767
INTEGER	4*	intervalo:-2147483647 a 2147483647
REAL	4*	intervalo:$1.18X10^{-38}$ a $3.4X10^{38}$,7-8 dígitos de precisión
REAL	8	intervalo:$2.23X10^{-308}$ a $1.79X10^{308}$,15-16 dígitos de precisión
COMPLEX	4*	intervalo:$1.18X10^{-38}$ a $3.4X10^{38}$,7-8 dígitos de precisión
COMPLEX	8	intervalo:$2.23X10^{-308}$ a $3.4X10^{308}$,15-16 dígitos de precisión
LOGICAL	1	valores: .TRUE. y .FALSE.
LOGICAL	4*	valores: .TRUE. y .FALSE.
CHARACTER	1*	conjunto de caracteres ASCII

* clase por omisión

Literales

Una literal es una constante sin nombre. En FORTRAN 90 es posible indicar la clase de la literal escribiendo el valor de la constante seguida de _clase. Ejemplos de literales:

Literales enteras: 34, -578, +27, 0, 15872, 345_4, -12_1, 14_clase_entera (clase_entera es una constante entera con nombre cuyo valor representa la clase del tipo de dato)

Literales reales: -7.87, .0003, 4.9e7, 1.51_8, -17.831_4, 1.25d-23, 3_class. La constante 4.9e7 representa a $4.9X10^{7}$,mientras que 1.25d-23 equivale a $1.25X10^{-23}$ (así se escriben en FORTRAN 77 las constantes de precisión doble)

Literales complejas: (-3.1, .07), (-1, +7), (4.9d5,-3.42_8). El primer número es la parte real y el segundo la parte compleja.

Literales lógicas: .false. , .true., .true_1.

Literales de caracteres: 'ESIQIE IPN', "FORTRAN 90","I'am", ' ',cl_'xy'. Los caracteres que forman la literal deben encerrarse entre " o entre '. FORTRAN 77 solo permite utilizar al ' como delimitador en constantes de caracteres.

Parámetros de tipo

En FORTRAN 90 los tipos de datos pueden ser parametrizados. Lo anterior implica que en una declaración de tipo se puede especificar su clase en función de ciertos parámetros. Las siguientes son funciones intrínsecas o de biblioteca que manejan dichos parámetros:

KIND(X) Devuelve un resultado igual a la clase del tipo que corresponde al dato X.

SELECTED_INT_KIND(r) El resultado es un entero igual a la clase del tipo entero que "acomoda" todos los enteros n entre $-10^{r}<n<10^{r}$. Si tal clase no está disponible, el resultado es -1. Si más de una clase está disponible, el resultado corresponde a la clase con el intervalo de representación menos amplio.

SELECTED_REAL_KIND(p,r) El resultado es un entero igual a la clase del tipo real con precisión al menos de p dígitos decimales e intervalo de representación de por lo menos $10^{-r}<n<10^{r}$.El resultado es -1 si la precisión no es alcanzable y -2 si el intervalo no está disponible. Ejemplos:

x=kind(1.75_8)

i=selected_int_kind(5)

f=selected_real_kind(10,200)

Para el tipo CHARACTER, existe un parámetro adicional: (LEN= número máximo de caracteres que puede representar un objeto de datos tipo CHARACTER).

Objetos de datos con nombre

Un objeto de datos con nombre, tal como una variable, una constante o el resultado de una función, tiene las propiedades del tipo de dato correspondiente. La información adicional acerca de un objeto de datos con nombre, se conoce como atributo. Los atributos pueden ser especificados en una declaración de tipo o en declaraciones ex-profeso. Las siguientes son declaraciones de tipo de variables o literales, con o sin atributos:

```
!n es un nombre asignado a la constante 50
!parameter es un atributo
integer (kind=4),parameter::n=50

integer (kind=4)::n  !equivalente a la instrucción anterior
parameter (n=50)
real (kind=8)::uno,dos   !uno y dos son nombres de variables

!equivalente a la instrucción anterior
double precision uno,dos

real x,y,z

complex a,b,c,d

logical respuesta

!len es la longitud en caracteres de la variable
character (len=50)::nombre
real (kind=selected_real_kind(12,200))::a,precision_final

integer (kind=selected_int_kind(10))::i,j,k,iteraciones
integer,parameter::clase=selected_real_kind(5,50)
real (kind=clase)::x,y,z
read *,x,y
z=x+y+.007_clase
```

En FORTRAN 77 no hay clases ni atributos: el número de bytes empleados para representar un dato se indican precedidos por

un *. Los valores por omisión son 4 bytes para los tipos REAL, INTEGER o LOGICAL y 1 byte para el tipo CHARACTER:

```
real x,y,z

integer a,b,c

double precision z

parameter (n=75)

logical respuesta

character *40 direccion

real *4 a,b

real *8 z

integer *2 m,n

logical *1 logic

complex angulo,omega
```

Tipo implícito de objetos de datos con nombre

En ausencia de una instrucción de declaración explícita, el tipo de un objeto de datos con nombre es determinado por la primera letra de su nombre; si es I,J,K,L,M o N el tipo es entero de la clase por "default" u omisión, si es diferente de I,J,K,L,M o N entonces el tipo es real de la clase por "default". De esta manera:

```
a,x_10,f,presion_osmotica
```

son variable reales, mientras que

```
i,n_50,n_alumnos,K
```

son variables enteras.

Instrucción IMPLICIT

Permite especificar la regla implícita para determinar el tipo de un objeto de datos con nombre.

```
implicit real (i-m)   ! los objetos cuyo nombre comience con i,j,k,l,m son reales

! los objetos cuyo nombre inicie con a,b,c,...,i,h,o,p,...,z   !son de tipo double precision

implicit double precision (a-h,o-z)     ! los objetos cuyo nombre inicie
                                        ! con a,b,c,...,i,h,o,p,...,z son de tipo double precision

implicit real(i-n),integer(o-z)         ! los objetos cuyo nombre comience con i,j,k,l,m,n
                                        ! son reales, mientras que los que inician con
                                        ! o,p,q,r,...,z son enteros

implicit complex (f,v)                  ! los objetos cuyo nombre empiece con f o v son del
                                        ! tipo complex

implicit logical (a-c,f)                ! los objetos cuyo nombre comience con a,b,c, o f son
                                        ! del tipo lógico

implicit real (kind=8) (d-p,z)          ! los objetos con nombre cuya primera letra es
                                        ! d,e,f,...,p,z son reales de clase 8

implicit character (kind=1,len=50) (n,s)    ! los nombres que empiezan con n,o,p,q,r,s son
                                            ! de tipo character (len=50)
```

Para inhabilitar cualquier regla para el tipo implícito de variables, use:

```
implicit none
```

En FORTRAN 77 no se permite la declaración IMPLICIT NONE.

Operadores aritméticos

Los operadores aritméticos disponibles en FORTRAN son, en orden de prioridad descendente:

Exponenciación	**
Multiplicación, división	*,/
Suma, resta	+,-

En expresiones con dos o más operadores de igual jerarquía, la

evaluación es de izquierda a derecha. La exponenciación, la cual es evaluada de derecha a izquierda, es la excepción de la regla anterior. Ejemplos que muestran el orden de ejecución de las operaciones aritméticas:

A + B/2

1) División de B entre 2
2) Al resultado de 1), se le suma A

Y**X/5*F

1) Se eleva Y a la potencia X
2) División del resultado de 1) entre 5
3) Multiplicación del resultado de 2) por F

3**4**2

1) 4 es elevado a la segunda potencia
2) 3 es elevado a una potencia igual al resultado de 1)

6 + G/N - K

1) División de G entre N
2) Al resultado de 1) se le suma 6
3) Resta de K al resultado de 2)

El orden de ejecución puede cambiarse con el uso de paréntesis:

(A + B)/2

1) Suma de A y B
2) el resultado de 1) es dividido entre 2

Y**X/(5*F)

1) Multiplicación de 5 por F
2) Y es elevado a la potencia X
3) División del resultado de 2) entre el de 1)

(3**4)**2

1) 3 es elevado a la cuarta potencia
2) el resultado de 1) es elevado al cuadrado

6 + G/(N - K)

1) Resta de K a N
2) División de G entre el resultado de 1)
3) 6 es sumado al resultado de 2)

Asignación

El valor de la expresión, constante o variable colocada a la derecha del operador de asignación = es asignado a la variable especificada a la izquierda del mismo:

```
A=B+15.7;C='ESIQIE'

I=I+1

7=45/A;F=(15.7_4,3.1E2);L=H

J=2/3          !0 es asignado a J (división entera)

F=2./3         !.6666... es asignado a F

complex Z
Z=(-15.1)      !-15.1+0i es asignado a Z

X=A**3         !A*A*A  es asignado a X

X=A**3.0       !e^3lnA  es asignado a X

real (kind=8)::F10,Z
Z=95.7
A=F10

real(kind=selected_real_kind(12,300))::f
complex::n
n=(3,1)*(15+f-A)
```

Funciones de biblioteca

Son aquellas provistas por el compilador FORTRAN 90. También se conocen como funciones intrínsecas.

```
Y=SIN(4*ATAN(1))
```

```
prueba=dexp(1.27_8)

Li=4*atan(1.0)

mayor_elemento=max(a,b,c,d,)

menor_elemento=min(x,y,z)

a=abs(10/n)

prueba=dexp(1.27_8)

z=mod(5,2)
```

El compilador supone que los argumentos de las funciones trigonométricas y los resultados devueltos por sus inversas, están dados en radianes. En el apéndice aparece una lista de las funciones de biblioteca soportadas por FORTRAN 90.

Entrada/Salida dirigidas por lista

La instrucción read * permite leer los valores que serán asignados a una lista de variables. Los datos serán proporcionados a través de la unidad estándar de lectura (el teclado de la computadora), separados entre sí por comas o por espacios. Print * o write(*,*) imprimen en la unidad estándar de escritura (la pantalla de la computadora) los valores de una lista de variables, constantes o expresiones.

```
read *,a,b,j

print *,a,b,k_minima

read * c,h,temperatura

print *,'Pi=',Pi,'ángulo=',&
      asin(x13)

write(*,*)a+b,"FORTRAN 90",17.5;print *,'ESIQIE'

print *     ! impresión de una línea en blanco
```

Instrucción PROGRAM

Especifica un nombre lógico para un programa principal. Por ejemplo:

```
program Eliminacion_de_Gauss
```

```
program principal
```

El nombre dado en una instrucción program no podrá ser empleado como nombre de un subprograma externo o de un módulo en el mismo programa ejecutable. Si se omite la instrucción PROGRAM, en Lahey FORTRAN 90 el compilador le asignará el nombre _LF_MAIN.

Instrucciones STOP y END

La instrucción STOP termina la ejecución de un programa:

```
stop
```

```
stop 325
```

```
stop 'FIN DE PROGRAMA'
```

En los dos últimos ejemplos se especifica; ya sea un número de 1 a 5 dígitos o una cadena de caracteres, que son impresos al ser ejecutado el STOP. El STOP que precede a la instrucción END es opcional.
El END termina un programa y siempre debe ser la última instrucción de este. Solo se permite un END en un programa:

```
end
```

```
end program
```

```
end program muestra
```

```
program hipotenusa
read *,a,b;c=sqrt(a**2+b**2)        ! cálculo de la hipotenusa c de un triángulo
print *,'a=',a,'b=',b,'c=',c        ! con catetos a y b
end program hipotenusa
```

```
program abc
real::x=3.15,y=-7.38,z              ! En una declaración de tipo, es posible asignar
z=x+y                               ! asignar valores a las variables. En este caso es
```

```
                                        ! obligatorio el separador ::
print *,x,y,z
end program
```

En FORTRAN 77 sólo se permiten STOP y END sin especificar más información.

Ejercicio

Escriba un programa que determine el área total y volumen de un cilindro de radio y altura conocidos.

área total = 2*área de la base + área lateral

área de la base = $\pi * radio^2$

área lateral = $2*\pi*radio*altura$

volumen = area de la base*altura

Pseudocódigo[1]

```
!Este programa determina el volumen y el área total de
!un cilindro, conocidos el radio r y la altura h
real Pi,r,h,area_base,area_lateral,area_total,volumen
Pi=3.145192
lee r,h    !lectura del radio y la altura
area_base=Pi*r↑2
area_lateral=2*Pi*r*h
area_total=2*area_base+area_lateral !cálculo del área total
volumen=area_base*h                 !y del volumen
!impresión de resultados
escribe "radio=",r,'altura=',h,'área total=',area_total,
        'volumen=',volumen
fin
```

Codificación en FORTRAN 77

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the
80386/80486) Wed Mar  5 01:33:37 1997   Page:     1
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS
RESERVED

PROGRAM AREA , Opciones de Compilación:
```

[1]En la elaboración de los pseudocódigos, se utilizan las reglas propuestas pro Guillermo Levine G. en su libro "Computación y Programación Moderna", Pearson Educación, México 2001.

```
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/
NV/W/NX/NZ1
Listado de fichero fuente

         1         program area
         2   c
         3   c       Este programa determina el volumen y el area
             c       total de
         4   c       un cilindro,conocidos el radio r y la altura
             c       h
         5   *
         6         parameter (Pi=3.141592)
         7         real r,h,vol,areat,areab,areal
         8   c       lectura del radio y la altura
         9         read *,r,h
        10         areab=Pi*r**2
        11         areal=Pi*2*r*h
        12   c       calculo del area total
        13         areat=2*areab+areal
        14   c       calculo del volumen
        15         vol=areab*h
        16   c       impresion de resultados
        17         print *,'radio=',r,'altura=',h
        18         print *,'area total',areat,
        19        1        'volumen=',vol
        20         stop
        21         end
```

Codificación en FORTRAN 90

```
Lahey Fortran 90 Compiler Release 3.00c                                    Wed Mar  5
01:39:58 1997  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

PROGRAM AREA    Compiling Options:     -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin    -inln

                                       -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing -----------------------------------------
- -
      1  program area
      2  !  Este programa determina el volumen y el area total de
      3  !  un cilindro,conocidos el radio r y la altura h
      4  !
      5  real,parameter::Pi=3.141592_4
      6  real::r,h,volumen,area_total,area_base,area_lateral
      7  read *,r,h          !lectura del radio y la altura
      8  area_base=Pi*r**2
      9  area_lateral=Pi*2*r*h
     10  area_total=2*area_base+area_lateral   !calculo del area total
     11  volumen=area_base*h                   !calculo del volumen
     12  ! impresion de resultados
     13  print *,'radio=',r,"altura",h;print *,'area total=',area_total,&
     14           'volumen=',volumen
     15  stop 'FIN DE PROGRAMA'
     16  end program area
Bytes of stack required for this program unit: 28.
```

ESTRUCTURAS DE DECISIÓN

Expresiones lógicas

Sólo pueden tomar un valor: verdadero (.TRUE.) o falso (.FALSE.). Los operadores de relación permiten construir expresiones lógicas.

Operadores de relación, en orden descendente de prioridad:

.EQ., ==	igual
.NE., /=	diferente
.LT., <	menor que
.LE., <=	menor o igual que
.GT., >	mayor que
.GE., >=	mayor o igual que

En FORTRAN 77, no son permitidos los operadores de relación ==, /=, <, <=, > y >=. Ejemplos:

X.GT.10.15

(X**2-10*4)==(2+10)

Y.NE.ABS(X-X0)

NOMBRE/='IPN'

X>=2.5

ABC<(1,2.5)

!una variable lógica constituye por sí sola una expresión lógica
logical var_uno

Cuando se comparan cadenas de caracteres, es conveniente recordar que las letras minúsculas son distintas a las mayúsculas:

'abc' es diferente de 'ABC'

Es posible comparar dos cadenas de caracteres de diferente longitud:

'IPN es igual a 'IPN '

Operadores lógicos

Con los operadores lógicos se elaboran expresiones lógicas compuestas.

Operadores lógicos, en orden descendente de prioridad:

.NOT.	negación
.AND.	conjunción
.OR.	disyunción inclusiva
.EQV.,.NEQV.	equivalencia, no equivalencia

Tablas de verdad para los operadores lógicos; X y Y son expresiones lógicas

X	Y	.NOT.X	X.AND.Y	X.OR.Y	X.EQV.Y	X.NEQV.Y
V	V	F	V	V	V	F
F	V	V	F	V	F	V
V	F	F	F	V	F	V
F	F	V	F	F	V	F

En una expresión, primero se evalúan los operadores aritméticos, luego los operadores de relación y por último los operadores lógicos. Ejemplos de evaluación de expresiones:

real::X=7,y=3
.
.
.
...(X+3).EQ.4.OR.Y**2>8

1) Calcular X+3
2) Elevar Y al cuadrado

```
3) Evaluar (X+3).EQ.4
4) Evaluar Y**2>8
5) Evaluación del operador lógico .OR.
```

CHARACTER (KIND=1,LEN=4)::NOMBRE
X=1
Y=2
NOMBRE='JOSE'
.
.
.
...NOMBRE=='RAUL'.AND.(X+Y)**2<10.OR.(X+1)/=2

```
1) Calcular X+Y
2) Determinar X+1
3) Elevar X+Y al cuadrado
4) Evaluar NOMBRE=='RAUL'
5) Evaluar (X+1)/=2
6) Evaluar (X+Y)**2<10
7) Evaluación del operador .AND.
8) Evaluación del operador .OR.
```

IF lógico

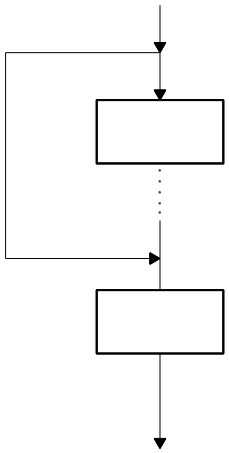
```
IF (expresión lógica) instrucción ejecutable

Si la expresión lógica encerrada entre paréntesis en el IF
es verdadera, entonces se ejecuta la instrucción indicada.
Si la expresión es falsa, no se ejecuta dicha instrucción.
La  instrucción debe ser diferente a otro IF o al END de un
programa, función o subrutina. Ejemplos:
```

if(a>b)print *,a,' es mayor que',b

IF(NOMBRE_EMPLEADO/="FIN DE ARCHIVO")N=N+1

Instrucción GO TO

```
GO TO etiqueta


Transfiere, en forma incondicional, el control a la
instrucción identificada por una etiqueta
```

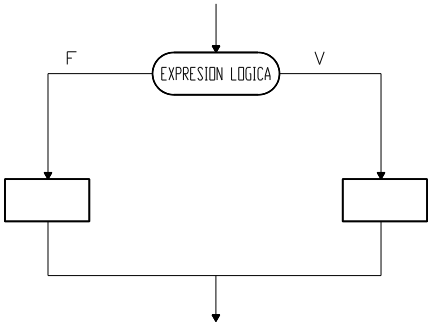


```
read *,x,y
if(x==y)go to 100
  print *,x,' es diferente de ',y
go to 110
100 print *,x,' es igual a',y
110 z=x+y
.
.
.
```

Construcción IF - THEN - ELSE

```
IF(expresión lógica)THEN
      .
      .
      .
      .
ELSE
      .
      .
      .
      .
END IF
```

Si la expresión lógica tiene un valor verdadero, se ejecutarán las instrucciones después del THEN y antes del ELSE. Si el valor de la expresión lógica es falso, entonces se ejecutará el bloque de instrucciones después del ELSE y antes del END IF.




```
read *,x,y
if(x==y)then
   print *,x,' es igual a',y
else
   print *,x,' es diferente de',y
endif
z=x+y
```

Si no hubiera instrucciones por ejecutar en el caso de que la expresión lógica tuviera un valor falso:

```
IF(expresión lógica)THEN
        .
        .
        .
        .
ENDIF
```

Ejemplo:

```
read *,a
if(a>0.0_4)then
  print *,a,'es mayor que cero'
  al=log(a);f=sqrt(a)
endif
```

Cuando no hay instrucciones por ejecutar si la expresión lógica es verdadera:

```
IF(expresión lógica)THEN
ELSE
      .
      .
      .
      .
ENDIF
```

Ejemplo:

```
read *,a
if(a>0.0_4)then
else
   print *,a,' es negativo'
   f= sqrt(abs(a))
endif
```

El siguiente es un ejemplo del uso de construcciones IF-THEN_ELSE en cascada:

```
real (kind=selected_real_kind(12,250))::T,K,Cp
read *,T
if(T>1500)then    !T es la temperatura en grados centígrados
   K=.043
   Cp=.50
else
   if(T>1350)then
      K=.032*.005*T;Cp=.20+.001*T
   else
     if(T>1100)then
        K=.015+.008*T+5.7e-6*T**2;Cp=.15+2.57*sqrt(T)
     else
        K=.007+1.5d-3*T-0.1e-6*T**2;Cp=.10+4.77**1.8
     endif
   endif
endif
```

Frecuentemente, las estructuras IF-THEN-ELSE en cascada, pueden ser escritas en una forma más compacta mediante el uso de ELSEIF:

```
real (kind=selected_real_kind(12,250))::T,K,Cp
read *,T
if(T>1500)then   !Este código es equivalente al anterior
   K=.043
```

```
   Cp=.50
elseif(T>1350)then
   K=.032+.005*T;Cp=.20+.001*T
elseif(T>1100)then
   K=.015+.008*T+5.7e-6*T**2;Cp=.15+2.57*sqrt(T)
else
   K=.007+1.5d-3*T-0.1e-6*T**2;Cp=.10+4.77*T**1.8
endif
```

En FORTRAN 90, ahora es posible proporcionar un nombre a una estructura IF-THEN-ELSE o equivalente:

```
real mayor,a,b,c
read *,a,b,c      !Determinación del mayor de tres números
uno:if(a>b)then
     dos:if(a>c)then  !dos es el nombre de la construcción
          mayor=a    !IF-THEN-ELSE
     else dos
          mayor=c
     endif dos
else uno
     tres:if(b>c)then
          mayor=b
     else tres
           mayor=c
     endif tres
endif uno

read *,Presion
estructura:if(Presion>100)then  !estructura es el nombre de la construcción
     h=37.5
elseif(Presion>80)then estructura
     h=16.4
else estructura
     h=7.32
endif estructura
```

En FORTRAN 77 no se permite especificar un nombre para una construcción IF-THEN-ELSE.

Tipos derivados

Una de las características más novedosas del FORTRAN 90 es

la de incluir tipos derivados. Los tipos derivados son definidos por el programador y están basados en los tipos intrínsecos INTEGER, REAL, COMPLEX, LOGICAL Y CHARACTER. Un objeto de tipos derivados es también conocido como una estructura de datos:

```
!Definición del tipo COEFICIENTES mediante un constructor
!de estructuras. A, B y C son los componentes del tipo coeficientes
type coeficientes
  real a,b,c
end type coeficientes
        .
        .
!coef es una variable del tipo coeficientes
real::z
type(coeficientes)::coef
        .
        .
read *,coef  !se leen tres valores reales
        .
        .
!mediante un %, se especifican a las componentes de la estructura
z=coef%a+coef%b-cos(coef%c)
        .
        .
print *,coef,z        !escritura de la variables COEF y Z

type empleado
  character(len=40)nombre
  real salario
  integer antiguedad
end type empleado
type(empleado)::trabajador_1,trabajador_2
!
!asignación con un constructor de estructuras:
!
trabajador_1=empleado('PANCHO LOPEZ',3419.18,17)
!lectura del nombre de trabajador_2
read *,trabajador_2%nombre
!asignación del salario y de la antigüedad de trabajador_2
trabajador_2%salario=2923.08
trabajador_3%antiguedad=10
print *,trabajador_1;print *,trabajador_2
```

```
type nombre
character *20 nombre_de_pila,apellido_paterno,&
                        apellido_materno
end type nombre
type registro
  !uso del tipo derivado definido arriba
  type(nombre)::alumno
  integer::primera_calificacion,segunda_calificacion
end type registro
type(registro)::alumno_inscrito
!
!se leen el nombre de pila, apellidos paterno y materno,
!primera y segunda calificaciones
!
read *,alumno_inscrito
print *,alumno_inscrito%alumno%apellido_paterno
print *,alumno_inscrito%segunda_calificacion
```

FORTRAN 77 no permite el uso de tipos derivados.

Ejercicio

Desarrolle un programa que calcule las raíces reales de la ecuación cuadrática $ax^2+bx+c=0$

Pseudocódigo

```
!Este programa calcula las raíces reales de una ecuación
!cuadrática a*x**2+b*x+c=0
!definición de los tipos coeficientes y raices_reales
tipo coeficientes
  real a,b,c
tipo raices_reales
  real X1,X2
coeficientes coef
raices_reales solucion
real disc
!lectura y escritura de los coeficientes
lee coef
escribe 'a=',coef%a,'b=',coef%b,'c=',coef%c
disc=coef%b↑2-4*coef%a*coef%c
si disc<0 entonces
  escribe 'raices imaginarias'  !raíces imaginarias
otro
  solucion%X1=(-coef%b+sqrt(disc))/(2*coef%a)
```

```
  solucion%X2=(-coef%b-sqrt(disc))/(2*coef%a)
  escribe 'X1=',solucion%X1,'X2=',solucion%X2
fin
```

Codificación en FORTRAN 77

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486) Sun May 11 01:22:02 1997        Page:
1
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

PROGRAM CUADRA , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

         1        program cuadra
         2   c
         3   c       Este programa calcula las raices reales de una ecuacion
         4   c       cuadratica a*x**2+b*x+c=0
         5   c
         6   c       lectura e impresion de los coeficientes
         7   c
         8        read *,a,b,c
         9        print *,'a=',a,'b=',b,'c=',c
        10        disc=b**2-4*a*c
        11        if(disc.lt.0.0)then
        12   c         raices imaginarias
        13          print *,'Raices complejas'
        14        else
        15   c         raices reales
        16          x1=(-b+sqrt(disc))/(2*a)
        17          x2=(-b-sqrt(disc))/(2*a)
        18          print *,'x1=',x1,'x2=',x2
        19        end if
        20        stop
        21        end
```

Codificación en FORTRAN 90

```
Lahey Fortran 90 Compiler Release 3.00c                                                  Sun May 11
01:34:45 1997  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

PROGRAM ECUACION_CUADRATICA    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed
-nin    -inln
                                    -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
-------------------------------------------------------------
      1  program ecuacion_cuadratica
      2  ! Este programa calcula las raices reales de una ecuacion cuadratica
      3  ! a*x**2+b*x+c=0
      4  !
      5  type coeficientes          !definicion de los tipos coeficientes y
      6    real a,b,c               !raices_reales
      7  end type coeficientes
      8
      9  type raices_reales
     10    real::x1,x2
     11  end type raices_reales
     12
     13  type(coeficientes)::coef
     14  type(raices_reales)::solucion
     15
     16  read *,coef       !lectura e impresion de los coeficientes
     17  print *,'a=',coef%a,'b=',coef%b,'c=',coef%c
     18  disc=coef%b**2-4*coef%a*coef%c
     19  if(disc<0.0)then
     20    print *,'Raices complejas'        !raices imaginarias
     21  else
     22    solucion%x1=(-coef%b+sqrt(disc))/(2*coef%a)      !raices reales
     23    solucion%x2=(-coef%b-sqrt(disc))/(2*coef%a)
     24    print *,'x1=',solucion%x1,'x2=',solucion%x2
     25  end if
     26  end program ecuacion_cuadratica
```

```
Bytes ofstack required for this program unit: 44.
```

IF aritmético (característica obsoleta)

IF(expresión aritmética)etiqueta1,etiqueta2,etiqueta3

La expresión aritmética puede ser tipo entero o real. Si la expresión aritmética tiene un valor negativo, se hará una transferencia de control hacia la instrucción rotulada con etiqueta1. Si la expresión tiene un valor cero, la transferencia de control será hacia la instrucción identificada con etiqueta2. Por último, si la expresión aritmética tiene un valor positivo, la transferencia de control será hacia la instrucción rotulada con etiqueta3. Ejemplos:

```
IF(15*a-b+cos(x))50,60,10

IF(TAN(X-X0))50,50,90

IF(K)10,15,20
```

```
real *,valor
if(sin(valor))10,20,30
10 print *,'el seno de',valor,' es negativo'
goto 40
20 print *,'el seno de',valor,' es cero'
goto 40
30 print *, 'el seno de',valor,' es positivo'
40 X=valor**2/15.1
```

Es posible simular el funcionamiento de un IF aritmético con un IF-ELSEIF-ELSE y sin necesidad de usar instrucciones GO TO:

```
read *,valor
if(sin(valor)<0.0)then
  print *,'el seno de',valor,' es negativo
elseif(sin(valor)==0.0)then
  print *,'el seno de', valor,' es cero'
else
  print *,'el seno de', valor,' es positivo
endif
```

Cabe aclarar que una característica marcada como obsoleta en

FORTRAN 90 ya no aparecerá en el siguiente estándar del lenguaje.

GO TO asignado e instrucción ASSIGN (características obsoletas)

Mediante la instrucción ASSIGN,a una variable de tipo entero se le asigna un valor que representa un número de etiqueta:

```
INTEGER CAM
      .
      .
ASSIGN 15 CAM
      .
      .
GO TO CAM

ASSIGN 100 TO K
      .
      .
GO TO K,(80,100,120) !la coma después de K es opcional
      .
      .
80    .
      .
100   .
      .
120   .
```

Construcción CASE

```
SELECT CASE(expresión)
      .
      .
CASE(selector)
      .
      .
CASE DEFAULT
      .
      .
END SELECT
```

La construcción CASE es usada para seleccionar entre varios bloques de código ejecutable, basándose en el valor de una

expresión. Necesariamente, la expresión debe ser del tipo INTEGER, LOGICAL o CHARACTER.

```
! el selector puede tener las formas:
! (valor inicial:valor final)
! (:valor final)
! (valor inicial:)
! (valor)
select case(k)
case (:-1)
  print *,k,' es negativo
case(0)
  print *,k,' es igual a cero'
case (1:)
  print *,k," es positivo"
end select

integer y
          .
          .
select case(y+15)
case(:-1)
   print *,'la expresión',y+15,' es negativa'
case(1:100)
    print *,'la expresión',y+15,&
      ' es mayor o igual que 1 y menor o igual que 100'
case default !no se cumple ninguno de los casos anteriores
    print*,'la expresión',y+15,' es mayor que 100 o igual a 0'
end select

character llave
          .
          .
read *,llave
!También se puede especificar un nombre para una construcción !CASE
opcion:select case(llave)
      case('A') opcion        !'A'
        x=50.17
      case('B':'F') opcion    !'B','C','D','E' o 'F'
        x=-17.14
      case('T','Y')           ! 'T' o Y'
        x=39.72
      case('Z') opcion        ! 'Z'
        x=83.26
```

```
    end select opcion
print *,x
```

En FORTRAN 77 no existe la construcción CASE.

GO TO computado (característica obsoleta)

GO TO (etiqueta1,etiqueta2,etiqueta3,...),expresión entera

Si el valor de la expresión entera es 1, la transferencia de control será hacia la instrucción rotulada con etiqueta1. En caso de que el valor de la expresión entera sea 2, la transferencia de control será hacia la instrucción señalada con etiqueta 2, y así, sucesivamente.

```
GO TO(10,20),K       !la , antes de la expresión es opcional

GO TO(100,30,80,999)NUL-K*L

read*,k
go to(10,20,30,55)k
go to 60
10 print *,k,' es igual a 1'
x=17.0*k
go to 60
20 print *,k,' es igual a 2'
x=k**2
go to 60
30 print *,k,' es igual a 3'
x=k/1000.0
go to 60
55 print *,k,' es igual a 4'
x=k/cos(k)
60 y=2*x
```

En el caso de que la expresión tenga un valor mayor que el número de etiquetas especificadas en el GO TO o un valor menor o igual que 0, entonces el GO TO computado es ignorado y se ejecuta la siguiente instrucción después de éste.
El GO TO computado también puede ser simulado con instrucciones IF-THEN-ELSEIF-ELSE o con estructuras case.

```
integer k ; real x
read *,k
```

```
if(k.eq.1)then
  print *,k,' es igual a 1'
  x=17.0*k
elseif(x.eq.2)then
  print *,k,' es igual a 2'
  x=k**2
elseif(x.eq.3)then
  print *,k,' es igual a 3'
  x=k/1000.0
elseif(x.eq.4)then
  print *,k,' es igual a 4'
  x=k/cos(k)
endif
y=2*k

integer k ; real x
read *,k
select case(k)
case(1)
  print *,k,' es igual a 1'
  x=17.0*k
case(2)
  print *,k,' es igual a 2'
  x=k**2
case(3)
  print *,k,' es igual a 3'
  x=k/1000.0
case(4)
  print *,k,' es igual a 4'
  x=k/cos(k)
end select
y=2*k
```

CICLOS

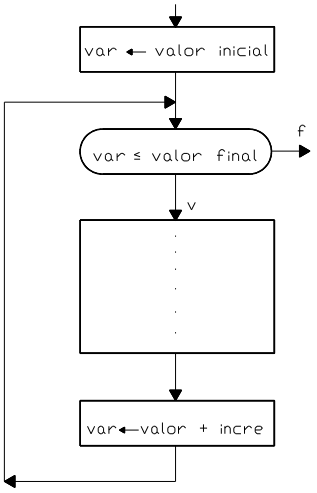
Ciclo DO

Un ciclo DO permite repetir un número conocido de veces un conjunto de instrucciones. Dicho conjunto de instrucciones se conoce como rango del ciclo DO:

DO etiqueta,var=valor inicial,valor final,incremento

donde:

Etiqueta identifica a la última instrucción del rango del ciclo. En FORTRAN 90, la etiqueta es opcional. La , después de la etiqueta también es opcional.
Var es una variable escalar del tipo INTEGER o REAL que actúa como el índice del ciclo DO. Sin embargo, en el estándar FORTRAN 90, el empleo de una variable de tipo real como índice del ciclo es marcado como un característica obsoleta.
Valor inicial, valor final e incremento son constantes, variables o expresiones de tipo INTEGER o REAL. Si es omitido el incremento, el compilador supone que es igual a 1. El incremento debe ser diferente de cero. Valor inicial, valor final e incremento controlan el número de veces que será ejecutado el rango del ciclo.



Ejemplos:

```
DO 10 I=1,5

DO 80 J=1,13,2

DO  5,ABC=-1,1,.1

DO 999 X=1.5,12,.5

DO 80 n=10,1,-1

DO 50 R=SIN(0.0),COS(0.0),.1

DO 710 L=N,K+3,J-1
```

Las siguientes son observaciones sobre el uso del ciclo DO:

Cuando el valor inicial es mayor que el valor final y el incremento es positivo, entonces el ciclo no es ejecutado:

```
DO 10 J=4,3
  X=J+17
  PRINT *,X,J
10 Y=2*X
```

Var, valor inicial, valor final o incremento no se pueden modificar dentro d e un ciclo DO:

```
DO 88 K=2,N,2
 J=K/2+N
 K=K+1
88 PRINT *,K

DO 100 K=2,N,2
 J=K/2+N
 N=N-J
100 PRINT *,K
```

No es posible ejecutar una transferencia de control hacia el interior de un ciclo DO:

```
GO TO 20
      .
      .
```

```
DO 5 K=1,10
      .
      .
      .
20    .
      .
      .
      .
5     .......
```

Sin embargo, la transferencia puede ser hacia la instrucción DO:

```
GO TO 20
      .
      .
20 DO 5 K=1,10
      .
      .
      .
5     .......
```

Es legal ejecutar una transferencia de control desde el ciclo hacia afuera de él:

```
DO 25 L=1,N
      .
      .
      .
GO TO 30
      .
      .
25    .
      .
      .
30    .
```

La última instrucción del rango del ciclo DO debe ser instrucción ejecutable diferente de un GO TO, RETURN, STOP, EXIT, CYCLE, GO TO asignado, IF aritmético, DO, bloque IF, ELSE y ENDIF. En caso de conflicto, siempre es posible terminar el rango del ciclo con una instrucción CONTINUE:

```
! la función intrínseca mod(i,j) determina el residuo que resulta de dividir i entre j
do 90 i=1,100
```

```
        if(mod(i,2)==0)then
                print *,i,' es par'
        else
                print *,i,' es impar'
        endif
90 continue
```

El terminar un ciclo DO con una instrucción diferente de un CONTINUE es una característica obsoleta en el estándar FORTRAN 90, como en:

```
suma=0.0
do 10 k=1,100
 read *,x
10 suma=suma+x
```

En lugar de lo anterior, se puede emplear un END DO para cerrar el rango del ciclo. El END DO no lleva etiqueta:

```
suma=0.0
do k=1,100
 read *,x
 suma=suma+x
end do
```

En un DO sin etiqueta que termina con END DO, es posible proporcionar un nombre para el ciclo:

```
ciclo1:do l=1,20
      x=sin(l*.1);y=cos(l*.1)
  print *,l,x,y
    end do ciclo1
```

En FORTRAN 77, el DO debe terminar necesariamente en una instrucción ejecutable con etiqueta. Tampoco se permite dar un nombre a la construcción DO

Dos ciclos están anidados si uno está contenido totalmente dentro del otro:

```
uno:do i=1,m
        .
        .
        .
         dos:do j=1,n
```

```
            .
            .
            .
         end do dos
      .
      .
      .
   end do uno
```

También se considera como característica obsoleta que dos o más ciclos anidados compartan la misma instrucción terminal:

```
do 793 i=1,l
      .
      .
      .
      do 793 k=n,m
        .
        .
        .
793 x=l+k-15.7
```

Ejercicio

Desarrolle un programa que calcule el promedio de N números

Pseudocódigo

```
!Este programa calcula el promedio prom de N números
real x,prom,SUMA
entero N,i
lee N
SUMA=0
ejecuta i=1,N
empieza                    !lectura de los N números y
      lee x;escribe i,x    !cálculo de su suma
      SUMA=SUMA+X
termina
prom=SUMA/N                !Obtención del promedio
escribe 'El promedio es:',prom
fin
```

Codificación en FORTRAN 77

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486) Wed May 28 00:34:32 1997        Page:
1
```

```
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

PROGRAM PROMED , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

         1         program promed
         2   c
         3   c       Este programa calcula el promedio prom de N numeros
         4
         5         integer N,i
         6         real X,SUMA,prom
         7         read *,N
         8         SUMA=0.0
         9         do 10 i=1,N
        10   c
        11   c         lectura de los N numeros y calculo de su suma
        12   c
        13           read *,X
        14           print *,i,X
        15           SUMA=SUMA+X
        16       10 continue
        17   c
        18   c     obtencion del promedio
        19   c
        20         prom=SUMA/N
        21         print *
        22         print *,'El promedio es:',prom
        23         end
```

Codificación en FORTRAN 90

```
Lahey Fortran 90 Compiler Release 3.00c                                          Wed May 28
00:29:02 1997  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

PROGRAM PROMEDIO_DE_N_NUMEROS    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed
-nin    -inln
                                   -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
------------------------------------------------------------
      1  program promedio_de_N_numeros
      2
      3  ! Este programa calcula el promedio prom de N numeros
      4
      5  integer N,i
      6  real X,SUMA,prom
      7
      8  read *,N
      9  suma=0.0
     10  ciclo:do i=1,N
     11          read *,X;print *,i,X    ! lectura de los N numeros y calculo
     12          SUMA=SUMA+X             ! de su suma
     13        end do ciclo
     14  prom=SUMA/N                     ! obtencion del promedio
     15  print *;print *,'El promedio es:',prom
     16
     17  end program promedio_de_N_numeros
Bytes of stack required for this program unit: 52.
```

Instrucción CYCLE

La instrucción CYCLE termina la ejecución de una iteración de un ciclo DO. Por ejemplo, el código

```
do i=1,5
  if(i==3)cycle
  print *,i,i**2
```

```
end do
```

produciría los valores:

```
1     1
2     4
4     16
5     25
```

Es posible especificar en el CYCLE el nombre del ciclo DO para el cual se termina la ejecución de la iteración:

```
read *,a,b
externo:do i=1,10
interno:  do j=1,8
        if(i>a)cycle externo
          if(j>b)cycle    !se toma el ciclo más interno
                        .
                        .
                end do interno
                .
                .
           end do externo
```

En FORTRAN 77 no existe la instrucción CYCLE, pero puede simularse con una transferencia de control al CONTINUE que finaliza el rango del ciclo:

```
  do 17 i=1,5
    if(i.eq.3)go to 17
    print *,i,i**2
17 continue
```

Instrucción EXIT

Termina la ejecución de un ciclo DO. Por ejemplo:

```
read *,k
do i=1,10
if(i==k)exit
  print *,i,i**2
end do
```

En el EXIT se puede indicar el nombre de la construcción DO

que terminará su ejecución:

```
read *,a,b
externo:do i=1,10
interno:  do j=1,8
                if(i>a)exit externo
                  if(i>b)exit !termina el ciclo interno
              .
              .
       end do interno
        .
                .
     end do externo
```

En FORTRAN 77 no existe la instrucción EXIT, pero puede ser sustituida por un GO TO hacia afuera del ciclo:

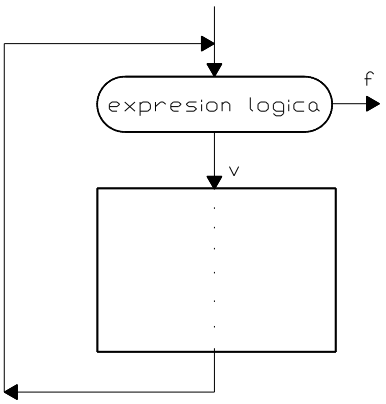
```
     read *,k
  do 34 i=1,10
    if(i.eq.k)go to 80
    print *,i,i**2
34 continue
        .
        .
80 .....
```

Ciclo DO WHILE

Permite la repetición de un conjunto de instrucciones (rango del ciclo) bajo el control de una expresión lógica:

```
DO etiqueta, WHILE(expresión lógica)
```

El rango del ciclo se ejecutará mientras la expresión lógica sea verdadera. Si la expresión lógica es falsa, entonces el control se transferirá a la siguiente instrucción ejecutable después del rango del ciclo. La etiqueta identifica a la instrucción terminal del rango del ciclo y es opcional en FORTRAN 90. La , antes del WHILE es también opcional.



Ejemplos:

```
do 15 while(x>100)
        .
        .
        .
15 continue

do 80 while(x.lt.n)
        .
        .
        .
80 x=x+10.0

i=1
do while(i<=10)
  x=i**2
  print *,i,x
  i=i+1
end do

character (len=2)::resp
pi=4*atan(1.0);resp='si'
do while (resp=='si'.or.resp=='SI')
        read *,radio
        area_del_circulo=pi*radio**2
        print*,'radio=',radio,&
```

```
        'area del circulo=',area_del_circulo
        print *,"desea continuar (si/no)?";read *,resp
end do
print *,'Fin del proceso'

C=0.0_4
!mientras es el nombre de la construcción while
mientras:do while (C<=100)
                F=1.8*C+32
                print *,C,F
                C=C+5.0
          end do mientras
```

En FORTRAN 77 no está contemplado el ciclo DO WHILE. Pero, es posible simularlo con un IF y dos instrucciones GO TO.

```
  i=1
5  if(i.gt.10)go to 25
        x=i**2
        print *,i,x
        i=i+1
  go to 5
25 continue
```

Ejercicio

Elabore un programa FORTRAN que genere una lista de calificaciones. Utilice la técnica del centinela para controlar el proceso de lectura de datos.

Pseudocódigo

```
!Este programa genera una lista de calificaciones
tipo registro
comienza
  hilera de caracteres nombre
  entero calificacion_1,calificacion_2,calificacion_3
termina
registro alumno
real promedio
entero promedio_definitivo
lee alumno   !lectura de los datos del primer alumno
!técnica del centinela
mientras(alumno%nombre≠'fin de archivo')
comienza
```

```
        promedio=(alumno%calificacion_1,+alumno%calificacion_2&
                  +alumno%calificacion_3)/3
      si promedio≥6 entonces
        promedio_definitivo=NINT(promedio)
      otro
         promedio_definitivo=INT(promedio)
      escribe alumno,promedio_definitivo
      lee alumno     !lectura de datos adicionales
    termina
    fin
```

Codificación en FORTRAN 77

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486) Wed May 28 01:02:36 1997        Page:
1
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

PROGRAM LISTA , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

         1         program lista
         2   c
         3   c       Este programa genera una lista de calificaciones
         4   c
         5         character *40 nombre
         6         integer calif1,calif2,calif3
         7         real prom
         8         integer promf
         9
        10   c
        11   c       lectura de los datos del primer alumno
        12   c       tecnica del centinela
        13   c
        14         read *,nombre,calif1,calif2,calif3
        15      10 if(nombre.eq.'fin de archivo')go to 20
        16           prom=(calif1+calif2+calif3)/3.0
        17   c         calculo del promedio
        18           if(prom.ge.6.0)then
        19             promf=nint(prom)
        20           else
        21             promf=int(prom)
        22           end if
        23           print *,nombre,calif1,calif2,calif3,promf
        24   c         lectura de datos adicionales
        25           read *,nombre,calif1,calif2,calif3
        26           go to 10
        27      20 continue
        28         stop
        29         end
```

Codificación en FORTRAN 90

```
Lahey Fortran 90 Compiler Release 3.00c                                                Wed May 28
00:54:37 1997  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

PROGRAM LISTA_DE_CALIFICACIONES    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed
-nin    -inln
                                   -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo    -nxref

------------------------------------------------  Source Listing
-------------------------------------------------------------
      1  program lista_de_calificaciones
      2  ! Este programa genera una lista de calificaciones
      3
      4  type registro
      5    character (len=40)::nombre
      6    integer::calificacion_1,calificacion_2,calificacion_3
```

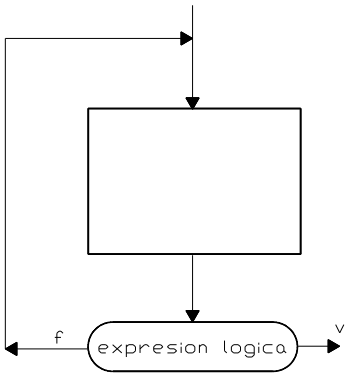
```
      7  end type registro
      8  real::promedio
      9  integer::promedio_definitivo
     10  type (registro)::alumno
     11
     12  read *,alumno      ! lectura de los datos del primer alumno
     13  do while(alumno%nombre/='fin de archivo')  ! tecnica del centinela
     14    promedio=(alumno%calificacion_1+alumno%calificacion_2+&
     15              alumno%calificacion_3)/3.0
     16    if(promedio>=6.0)then      ! calculo del promedio
     17      promedio_definitivo=nint(promedio)
     18    else
     19      promedio_definitivo=int(promedio)
     20    end if
     21    print *,alumno,promedio_definitivo
     22    read *,alumno              ! lectura de datos adicionales
     23  end do
     24  stop 'FIN DE PROGRAMA'
     25  end program lista_de_calificaciones
Bytes of stack required for this program unit: 92.
```

Ciclo DO until

El ciclo DO until también permite la repetición de un grupo de instrucciones (rango del ciclo):

DO etiqueta

El rango del ciclo se seguirá ejecutando hasta que cierta expresión lógica tenga el valor verdadero. Dicha expresión lógica es probada en un IF colocado siempre al final del rango. Después del IF, se escribe ya sea un CONTINUE con etiqueta o un END DO.



Ejemplos:

i=0
do 18

```
  i=i+1
  x=i**2
  print *,i,x
  if(i==10)exit
18 continue

i=0
do
  i=i+1
  x=i**2
  print *,i,x
  if(i==10)exit
end do

hasta:do     !hasta es el nombre de la construcción
       read *,x,y
       z=sin(x**2+y**2)
       print *,x,y,z
      if(abs(x)<=1.0e-8.and.abs(y)<=1.0e-8)exit hasta
 end do hasta
```

FORTRAN 77 no permite usar ciclos UNTIL. Sin embargo, es posible simularlos con instrucciones GO TO:

```
i=0
7 continue
    i=i+1
    x=i**2
    print *,i,x
    if(i==10)go to 19
    go to 7
19 continue
```

ARREGLOS

Declaración de arreglos

Un arreglo es un conjunto de datos, todos del mismo tipo y con los mismos parámetros de tipo, dispuestos en forma rectangular en una o más dimensiones. Un objeto de datos que no es un arreglo es un escalar. Los arreglos son especificados en una instrucción DIMENSION o bien con el atributo DIMENSION en una declaración de tipo. Un arreglo tiene un rango (rank)igual a su número de dimensiones. Un escalar tiene rango cero. Un arreglo de una dimensión o de rango 1 es conocido como un vector, mientras que las matrices son arreglos de dos dimensiones o de rango 2. La forma (shape) de un arreglo es su extensión (extent) en cada dimensión. El tamaño (size) de un arreglo es el número de elementos en un arreglo. Los siguientes ejemplos muestran declaraciones de arreglos:

```
!atributo dimension
integer,dimension(3,2)::l   !l tiene rango 2, forma (3,2) y tamaño 6
integer k
dimension k(3,2)                    !instrucción DIMENSION

real (kind=8),dimension(50)::X

complex,dimension(100,100)::vector,arco

!el subíndice del primer elemento es 0 (el "default" es 1)
real,dimension(0:20)::x,y,z

!len y kind son los parámetros del tipo
character (len=10),dimension(1958:2000)::fecha

real (kind=selected_real_kind(5,30)),&
dimension(0:5,1:7):: matriz,matriz_alfa

real,dimension(10,10)::coeficientes

dimension arreglo(-2:2)

logical,dimension(10)::respuesta,confirmación

type especie_quimica
```

```
        character (len=20)::nombre
        real Tc,Pc,omega
end type especie_quimica
!compuesto es un vector del tipo especie_quimica
type (especie_quimica),dimension(25)::compuesto
```

El atributo DIMENSION aparta, en tiempo de compilación, las localidades de memoria que forman el arreglo (asignación estática de memoria). En este caso, ya no es posible modificar el tamaño del arreglo en tiempo de ejecución:

```
real (kind=8),dimension(6)::x
```

x(1)
x(2)
x(3)
x(4)
x(5)

Los elementos de una matriz son guardados columna por columna, es decir, el subíndice más a la izquierda cambia más rápidamente)

```
real,dimension(2,3)::Spline
```

Spline(1,1)
Spline(2,1)
Spline(1,2)
Spline(2,2)
Spline(1,3)
Spline(2,3)

En FORTRAN 77, una declaración de tipo puede ser también empleada para declarar arreglos. Entonces

```
real mx(40,20),k23(60)
```

es equivalente a

```
real mx,k23
dimension mx(40,20),k23(60)
```

El máximo número de dimensiones es siete. En FORTRAN 77 no se permite el atributo DIMENSION, pero si la instrucción DIMENSION.

Subíndices

A través de un sistema de subíndices es posible especificar a un elemento en particular de un arreglo. Ejemplos:

```
real,dimension (10)::x,y
        .
        .
! un subíndice puede ser una constante, una variable o una expresión
y(4)=x(k)
y(5)=x(l+n-1)/150
y(6)=x(l)+2
! el valor numérico de la expresión a-b es truncado, porque los subíndices deben ser ! !
enteros
y(7)=x(a-b)

type prueba
        real a,b,c
end type prueba
type (prueba),dimension(100)::var
var(8)%b=.01;var(15)%c=1.20
var(37)%a=3.1415/var(15)%c
```

En arreglos de dos dimensiones; el primer subíndice (de izquierda a derecha) corresponde al número de fila y el segundo subíndice al número de columna:

```
dimension v(10,8),x(10),ind(10)
        .
        .
v(i,j+1)=x(ind(j))
```

Constructores de arreglos

Mediante el uso de los constructores de arreglos es posible proporcionar valores a los elementos de un arreglo. En su forma más simple, un constructor de arreglo es una lista de

valores encerrados entre los delimitadores (/ y /). Ejemplos de constructores de arreglos:

```
! en el constructor de arreglos para el vector c, i toma un valor inicial de 4, un
! valor final de 6 con un incremento de 1
integer,dimension(3)::a,b=(/13,-5,14/),c=(/(i,i=4,6)/)
```

b(1)=13
b(2)= -5
b(3)=14

c(1)=4
c(2)=5
c(3)=6

La función RESHAPE se usa para dar valores a los arreglos de rango mayor que 1:

```
! el argumento shape=(/2,2/) indica que se trata de un arreglo de 2 filas y 2 columnas
real,dimension(2,2)::matriz_ejemplo=&
reshape((/35.1,-12.4,5.7,4.21/),shape=(/2,2/))
```

matriz_ejemplo(1,1)= 0.3510000E+02
matriz_ejemplo(2,1)=-0.1240000E+02
matriz_ejemplo(1,2)= 0.5700000E+01
matriz_ejemplo(2,2)= 0.4210000E+01

Se emplea una tabla como la anterior para indicar el contenido de un arreglo de tipo real, escribiendo los valores numéricos en notación normalizada, para recordar que en la memoria de una computadora precisamente de esa manera son representados los números de punto flotante.

```
character,dimension(2)::carac=(/'A','B','C','D'/)

type prueba
        logical::a
        real::b,c
end type prueba
```

```
type(prueba),dimension(2)::var=&
(/prueba(.true.,1.0_4,2.9_4),prueba(.false.,3.7,-4e-2)/)
```

En FORTRAN 77 no son permitidos los constructores de arreglos.

Lectura/escritura de arreglos

La forma más obvia de leer a los elementos de un arreglo es emplear un ciclo DO:

```
real,dimension(4)::x
      .
      .
n=4
do i=1,n
      read *,x(i)
end do
```

Cada vez que se ejecuta la instrucción read *, la computadora pide una línea de datos. Por esta razón, en el ejemplo anterior, será necesario proporcionar cuatro líneas de datos,como las siguientes:

```
100.4
718
362.9
-3.78E2
```

x(1)= 0.1004000E+03
x(2)= 0.7180000E+03
x(3)= 0.3629000E+03
x(4)= -0.3780000E+02

Mediante una lectura implícita es posible proporcionar los valores de un arreglo en una sola línea de datos, ya que la instrucción read * es ejecutada una sola vez, como en:

```
real,dimension(4)::x
      .
      .
n=4
```

```
read *,(x(i),i=l,n,1)   !lectura implícita del arreglo x
```

con una posible línea de datos:

100.4,718,362.9,-3.78E2

Una lectura implícita se puede indicar también mediante una tríada de subíndices: (valor inicial:valor final:incremento)

```
real,dimension(4)::x
      .
      .
n=4
read *,x(1:n:1)
```

Al omitir el incremento en una lectura implícita o en una tríada de subíndices, el compilador supondrá que es la unidad:

read *,(a(j),j=1,8,1) es equivalente a read *,(a(j),j=1,8)

read *,vec(1:k:1) es equivalente a read *,vec(1:k)

Ejemplos adicionales de lecturas de arreglos:

```
real  abc(0:10),xyz(0:10)
      .
      .
read *,n,(abc(l),l=0,n)  !n<=10
read *,xyz(0:n)

dimension array(20)
! lectura del arreglo completo
read *,array

real,dimension(11)::x
! lectura de x(1),x(3),x(5),x(7),x(9) y x(11)
read *,x(1:11:2)

dimension x(20),y(20),x10(20),y10(20)
read *,n
      .
      .
! lectura de x(1),x(2),...,x(n),y(1),y(2),...,y(n)
read *,(x(i),i=l,n),(y(i),i=l,n)
      .
```

```
        .
! lectura de x(1),y(1),x(2),y(2),...,x(n),y(n)
read *,(x(i),y(i),i=l,n)

integer,dimension(4)::vector=(/1,3,12,17/)
real,dimension(100)::datos
        .
        .
! lectura de datos(1),datos(3),datos(12),datos(17)
read *,datos(vector)    ! direccionamiento indirecto

dimension ab(3,3)
        .
        .
read *,numero_de_filas,numero_de_columnas
        .
! lectura de la matriz ab, fila por fila
do i=1,numero_de_filas
        do j=l,numero_de_columnas
                read *,ab(i,j)
        end do
end do
```

Si los datos correspondientes al último ejemplo son:

```
3,2
-9.6
-7.1
4.3
2.4
8.0
-5.7
```

entonces el arreglo ab los almacenaría de esta manera:

ab(1,1)=-0.9600000E+01
ab(2,1)= 0.4300000E+01
ab(3,1)= 0.8000000E+01
ab(1,2)=-0.7100000E+01
ab(2,2)= 0.2400000E+01
ab(3,2)=-0.5700000E+01

ab(1,3)=
ab(2,3)=
ab(3,3)=

```
dimension w(3,3)
        .
        .
read *,numero_de_filas,numero_de_columnas
        .
        .
! lectura de la matriz w,columna por columna
do k=1,numero_de_columnas
        do l=1,numero_de_filas
                read *,w(l,k)
        end do
end do
```

suponiendo los siguientes datos para el último ejemplo

```
3,2
-9.6
-7.1
4.3
2.4
8.0
-5.7
```

la matriz w queda así:

w(1,1)=-0.9600000E+01
w(2,1)=-0.7100000E+01
w(3,1)= 0.4300000E+01
w(1,2)= 0.2400000E+01
w(2,2)= 0.8000000E+01
w(3,2)=-0.5700000E+01
w(1,3)
w(2,3)
w(3,3)

A continuación, ejemplos de lectura implícita de matrices:

```
dimension a(3,3)
        .
        .
read *,numero_de_filas,numero_de_columnas
        .
! lectura implícita por filas
read *,((a(i,j),j=1,numero_de_columnas),i=1,numero_de_filas)
! los datos se proporcionan de la siguiente manera:
! 3,2
! -9.6,-7.1,4.3,2.4,8.0,-5.7

dimension w(3,3)
        .
        .
! lectura implícita por columnas
read *,numero_de_filas,numero_de_columnas
read *,((a(l,k),l=1,numero_de_filas),k=1,numero_de_columnas)
! los comentarios de abajo muestran dos posibles líneas de datos
! 3,2
! -9.6,-7.1,4.3,2.4,8.0,-5.7

! la lectura implícita por columnas también se puede indicar con tríadas de subíndices
        .
        .
read *,numero_de_filas,numero_de_columnas
read *w(1:numero_de_filas,1:numero_de_columnas)
```

Las ideas mostradas anteriormente en la lectura de arreglos son aplicables a la escritura de los mismos:

```
dimension x(10)
        .
        .
! escritura del arreglo completo
print *,x

real,dimension(5)::a,b,c
        .
        .
! una constante es asignada a todos los elementos de un arreglo
a=50.0
b=10.0
```

```
c=-87.0
! escritura implícita del vector a
print *,(a(k),k=1,4)
! escritura empleando una tríada de subíndices
print *,b(2:5)
!escritura con un ciclo DO
do i=1,5
        print *,c(i)
end do
```

La salida impresa producida por el ejemplo anterior sería similar a la siguiente:

```
50.0000   50.0000   50.0000   50.0000
10.0000   10.0000   10.0000   10.0000
-87.000
-87.000
-87.000
-87.000
-87.000
```

```
complex freq(10,20)
        .
        .
! escritura del arreglo completo, columna por columna, y en una sola línea
print *,freq

real (kind=8),dimension vec(100)
        .
        .
read *,vec(1:10)
do i=1,10
        print *,'vec(',i,')=',vec(i)
end do

real,dimension(50,50)::matriz
        .
        .
! la palabra clave shape es opcional, siempre y cuando se trate del segundo argumento
matriz(1:3,1:3)=reshape((/1,2,3,4,5,6,7,8,9/),(/3,3/))
! escritura de la matriz en formato rectangular
do k=1,3
        print *,matriz(k,1:3)
end do
```

Dado que el print * se ejecuta tres veces; serán impresas tres líneas y cada una de ellas constará de tres valores:

```
1.0000   4.0000   7.0000
2.0000   5.0000   8.0000
3.0000   6.0000   9.0000
```

```
real (kind=selected_real_kind(10,150)),dimension(30,20)::x
        .
        .
do 10 i =1,30
        print *,'fila',i
        print *,(x(i,m),m=1,20)
10 continue
```

El estándar FORTRAN 77 no incluye el uso de tríadas de subíndices.

Arreglos ALLOCATABLE

El atributo ALLOCATABLE o la instrucción ALLOCATABLE le indican al compilador que la forma y el tamaño del arreglo serán determinados en tiempo de ejecución (asignación dinámica de memoria). Los arreglos ALLOCATABLE son conocidos como "de forma diferida". Ejemplos:

```
! : es el indicador de forma diferida
real (kind=selected_real_kind(12,25)),allocatable::y(:,:)

complex f50
allocatable f50(:,:)
```

La instrucción ALLOCATE determina, en tiempo de ejecución, la forma y el tamaño de arreglos ALLOCATABLE y asigna las localidades de memoria correspondientes:

```
real,allocatable::b(:),a(:,:)
        .
        .
read *,m,n
allocate(a(m,n),b(n))

allocatable presion(:),temperatura(:),entalpia(:)
        .
        .
! el subíndice inicial es 0
```

```
allocate(presion(0:50),temperatura(0:50),entalpia(0:m+n-1))

dimension viscosidad(:,:);allocatable viscosidad
m=15;n=12;allocate(viscosidad(m,n))

character(len=10);allocatable::nombres_de_compuestos(:)
        .
        .
read *,n
```

La cláusula STAT=variable entera, permite saber si fue posible construir los arreglos con atributo allocatable: un valor diferente de cero devuelto en dicha variable implica un error en la ejecución de ALLOCATE.

```
allocate(nombres_de_compuestos(n),stat=ierr)
if(ierr==0)then
        print *,"Si fue posible apartar la memoria del arreglo"
        stop
else
        print *,'No fue posible apartar la memoria del arreglo'
endif
```

Si no se especifica STAT= variable entera, al haber un problema con ALLOCATE, termina la ejecución del programa. La instrucción DEALLOCATE libera la memoria ocupada por arreglos ALLOCATABLE:

```
logical, dimension(:)::respuestas
allocatable  respuestas
        .
        .
read *,n
m=max(50,n)
allocate(respuestas(m+50))
        .
deallocate(respuesta)
```

También es posible usar la cláusula STAT=variable entera en un DEALLOCATE:

```
real,allocatable;dimension(:,:,:)::matriz_de_frecuencias
        .
        .
read *,n
allocate(matriz_de_frecuencias(50,60,n+1))
```

```
        .
deallocate(matriz_de_frecuencias,stat=ierror)
if(ierror/=0)then
        print *, 'Error. No se libera la memoria ocupada'
        print *,'por matriz_de_frecuencias'
end if
```

Nuevamente, si no se especifica stat=variable entera y ocurriera algún error en el DEALLOCATE, finalizaría la ejecución del programa.
La función ALLOCATED determina si a un arreglo se le han asignado sus localidades de memoria. El resultado de ALLOCATED es de tipo LOGICAL:

```
real (kind=8),allocatable::vec(:,:,:)
        .
        .
if (allocated(vec))then
        print *,'El vector vec tiene asignadas sus '
        print *,'localidades de memoria'
else
        print *,"El vector vec no tiene asignadas sus "
        print *,"localidades de memoria"
end if
```

En FORTRAN 77 no se permite la asignación dinámica de memoria. Por tanto, no es posible usar ALLOCATABLE, ALLOCATE y DEALLOCATE.

Operaciones con arreglos

Un elemento de un arreglo puede ser empleado de la misma forma que una variable escalar es usada en expresiones, asignaciones o en instrucciones de lectura/escritura:

```
a(5)=b(7)-c(m-1)

print *,x(12)
```

Al cambiar el valor de uno o más subíndices de un arreglo, es posible hacer referencia a varios elementos del mismo:

```
do i=1,20
        c(i)=a(i)*b(i)
end do
```

FORTRAN 90 permite manejar un arreglo como un simple objeto, es decir, como si fuera una variable escalar:

```
integer,dimension(5)::a=(/1,2,3,4,5/),b,c,d,e
        .
        .
        .
b=(/10,20,30,40,50/)
! multiplicación por un escalar y suma de arreglos
c=2*a-b
        .
        .
! multiplicación, elemento por elemento, de dos arreglos
d=a*b
        .
        .
! división, elemento por elemento, de dos arreglos
e=b/a
```

Los arreglos a,b,c,d y e del ejemplo anterior tendrían los siguientes valores:

a(1)= 1	b(1)= 10	c(1)=-8	d(1)= 10	e(1)= 10
a(1)= 2	b(2)= 20	c(2)=-16	d(2)= 40	e(2)= 20
a(3)= 3	b(3)= 30	c(3)=-24	c(3)= 90	e(3)= 10
a(4)= 4	b(4)= 40	c(4)=-32	c(4)= 160	e(4)= 10
a(5)= 5	b(5)= 50	c(5)=-40	c(5)= 250	e(5)= 10

```
real,dimension(3,:)::x=reshape((/1.,2.,3.,4.,5.,6./), (/3,2/)),&
                y=reshape((/7.,8.,9.,10.,11.,12./),&
                        (/3,,2/)),zsum,zdiv
        .
        .
zsum=x+y
zdiv=x/y      !operaciones elemento por elemento
```

Los resultados de las últimas operaciones con arreglos quedan almacenados como a continuación se muestra:

x(1,1)= 0.1000000E+01	y(1,1)= 0.7000000E+01

x(1,1)= 0.2000000E+02	y(2,1)= 0.8000000E+01
x(3,1)= 0.3000000E+03	y(3,1)= 0.9000000E+01
x(1,2)= 0.4000000E+01	y(1,2)= 0.1000000E+02
x(2,2)= 0.5000000E+01	y(2,2)= 0.1100000E+02
x(3,2)= 0.6000000E+01	y(3,2)= 0.1200000E+02

zsum(1,1)= 0.8000000E+01	zdiv(1,1)= 0.1428571E+00
zsum(2,1)= 0.1000000E+02	zdiv(2,1)= 0.2500000E+00
zsum(3,1)= 0.1200000E+02	zdiv(3,1)= 0.3333333E+00
zsum(1,2)= 0.1400000E+02	zdiv(1,2)= 0.4000000E+00
zsum(2,2)= 0.1600000E+02	zdiv(2,2)= 0.4545454E+00
szum(3,2)= 0.1800000E+02	zdiv(3,2) = 0.500000E+00

```
real,dimension(5)::uno,dos
real,parameter::Pi=3.141592
read *,uno
dos=Pi*uno+(/7.1,-4.8,2.0,.07,-3.91/)
```

Se dice que dos arreglos son conformables si tienen la misma forma. Un escalar, inclusive una constante, es conformable con un arreglo. Las operaciones entre arreglos son permitidas solamente cuando son conformables. Por ejemplo, la siguiente operación no es válida:

```
real,dimension(3,2)::y_dato=50.0_4
real,dimension(6)::z_dato=(/4.5,6.0,-2.3,+0.56,-4.9/)
! y_dato y z_dato no son conformables
print *,0.56*y_dato-z_dato              ! operación no válida
```

En FORTRAN 77 las operaciones con arreglos se implementan variando los valores que toman sus subíndices con algún tipo de ciclo. Por ejemplo:

```
real,dimension(100)::x,y,z
      .
      .
z=x-y
```

en FORTRAN 77 se tiene que escribir como:

```
  dimension x(100),y(100),z(100)
  do 10 i=1,100
        z(i)=x(i)-y(i)
10 continue
```

Secciones de arreglos

Es factible hacer referencia a una parte de un arreglo como si se tratara de un arreglo completo. Dicha porción de un arreglo es conocida como una sección. Una sección de un arreglo es indicada mediante una lista de subíndices:

```
a(2:8:2)     !a(2),a(4),a(6) y a(8)

(a(k),k=2,8,2)              ! equivalente a la instrucción anterior

matriz(6,1:n_columnas)      ! sexta fila de matriz

x(1:n_filas,1)              ! primera columna de x

b(1,3:1:-1)                 ! b(1,3),b(1,2) y b(1,1)

vect(8:5)                   ! esta sección es un arreglo de tamaño cero
```

Funciones intrínsecas relacionadas con arreglos

FORTRAN 90 incluye nuevas funciones intrínsecas diseñadas para el manejo de arreglos.
La función MAXVAL encuentra a los elementos de mayor valor en un arreglo, a lo largo de una dimensión dada y tomando en cuenta solo aquellos para los cuales se cumple cierta condición o máscara(mask):

```
maxval(array,dim,mask)
```

donde array es un arreglo de tipo INTEGER o REAL, dim es un entero menor o igual al rango de array y mask es una expresión de tipo LOGICAL conformable con array. Tanto dim como mask son argumentos opcionales. El resultado es del mismo tipo y clase que array. Respecto al rango del resultado; es un escalar si dim está ausente o si array es de rango uno, de cualquier otra forma es de rango n-1. Si dim es especificado, el valor del resultado es el máximo de

todos los elementos de array. Por otra parte, si dim está presente, el resultado es el mayor valor de todos los elementos a lo largo de la dimensión dim. Cuando mask no es omitido, los elementos de array para los cuales mask tiene un valor falso no serán considerados.
MAXLOC determina la localización de los mayores valores de los elementos de un arreglo:

```
maxloc(array,mask)
```

en este caso array es un arreglo de tipo INTEGER o REAL y mask es un argumento opcional de tipo LOGICAL conformable con array. El resultado es de tipo INTEGER. Si mask está ausente, el resultado es un vector cuyo contenido son los subíndices del primer elemento en array que tiene el máximo valor de todos los elementos en dicho arreglo. Si mask está presente, los elementos de array para los cuales tiene un valor falso no serán tomados en cuenta:

```
integer(kind=selected_int_kind(6)),dimension(6)::arreglo
integer(kind=selected_int_kind(6)),dimension(1)::j,l
arreglo=(/1,5,3,9,0,-7/)
      .
      .
! i es el mayor elemento de arreglo (9)
i=maxval(arreglo)
      .
! j es el subíndice del mayor elemento de arreglo (4)
j=maxloc(arreglo)
```

Las funciones MINVAL y MINLOC trabajan de forma idéntica a como lo hacen MAXVAL y MAXLOC, excepto que ahora se buscan los valores mínimos de un arreglo:

```
! k es el mínimo valor en arreglo (-7)
k=minval(arreglo)
      .
! l el subíndice del mínimo valor en arreglo (6)
l=minloc(arreglo)

! m es el menor valor de arreglo, considerando solamente a los elementos mayores
! de cero (1)
m=minval(arreglo,mask=arreglo>0)
```

Otra muestra del uso de maxval,maxloc, minval y minloc:

```
real (kind=selected_real_kind(6,35)),dimension(3,3)::f
```

```
integer j(2),l(2)
real (kind=selected_real_kind(6,35)),dimension(3)::m,n
real (kind=selected_real_kind(6,35)) p,q,o
f=reshape((/10,5,-3,9,4,-5,12,0,9/),(/3,3/))
do i=1,3                    ! impresión de la matriz f
        print *,f(i,1:3)
end do
print *                     ! escritura de una línea en blanco
p=maxval(f)
! j es un vector de dos elementos con la localización del mayor valor del arreglo f
j=maxloc(f)
q=minval(f)
! l es un vector con la localización del menor elemento de f
l=minloc(f)
print *,p,j
print *,q,l
print *
! búsqueda de los máximos valores de f a lo largo de la dimensión 1, es decir, se
! determinan los máximos valores de cada columna
m=maxval(f,dim=1)
print *,m
! búsqueda de los máximos valores de f a lo largo de la dimensión 2; se determinan los
! valores más grandes de cada fila
n=maxval(f,dim=2)
print *,n
print *
o=maxval(f,mask=f<=10)      !sólo se toman en cuenta los elementos de f que son menores
                            ! o iguales a 10
print *,o
```

Los resultados generados por las instrucciones anteriores serían:

```
10.0000         9.0000         12.0000
 5.0000         4.0000          0.0000
-3.0000        -5.0000          9.0000

12.0000           1              3
-5.0000           3              2

10.0000         9.0000         12.0000
12.0000         5.0000          9.0000

10.0000
```

El producto escalar de dos vectores puede ser calculado con la función DOT_PRODUCT:

```
real,dimension(5)::a,b
a=(/1.0,2.0,0.0,-3.5,7.3/)
b=(/12.0,4.0,8.5,-5.2,9.6/)
! dot_product determina el producto punto de dos vectores
print *,dot_product(a,b)
print *,dot_product(a(1:3),b(1:3))
```

La función SUM calcula la suma de los elementos de un arreglo, a lo largo de una dimensión dada, para los cuales una condición (máscara) es verdadera:

```
sum(array,dim,mask)
```

los argumentos dim y mask son opcionales. Array debe ser un arreglo de tipo INTEGER, REAL o COMPLEX:

```
integer,dimension(2,2)::m=reshape((/1,2,3,4/),(/2,2/))
integer k(2),l(2),i
do i=1,2
        print *,m(i,1:2)
end do
! cálculo de la suma de todos los elementos del arreglo
i=sum(m)
! cálculo de la suma de los elementos, a lo largo de la dimensión 2
k=sum(m,dim=2)
! cálculo de la suma de los elementos mayores que 2, a lo largo de la dimensión 1
l=sum(m,mask=m>2,dim=1)
print *,i,k,l
```

los resultados producidos por las instrucciones arriba indicadas son:

```
1       3
2       4
10      4       6       0       7
```

FORTRAN 90 incluye la función MATMUL para determinar el producto de dos matrices como se calcula en álgebra lineal. El producto de dos matrices está definido solamente cuando el número de columnas de la primera es igual al número de filas de la segunda. Por ejemplo; si A es de m filas y n columnas y B es de n filas y p columnas, entonces el producto C=AB es una matriz de m filas y p columnas:

```
real a(2,3),b(3,3),c(2,3)
a=reshape((/1,2,3,4,5,6/),(/2,3/))
b=reshape((/1,2,3,4,5,6,7,8,9/),(/3,3/))
do i=1,2
        print *,a(i,1:3)
end do
print *;do i=1,3;print *,b(i,1:3);end do
c=matmul(a,b)            ! producto matricial c=ab
print *
do k=1,2
        print *,c(k,1:3)
end do
```

Las instrucciones anteriores producen la siguiente salida:

```
 1.0000        3.0000        5.0000
 2.0000        4.0000        6.0000

 1.0000        4.0000        7.0000
 2.0000        5.0000        8.0000
 3.0000        6.0000        9.0000

 22.0000       49.0000       76.0000
 28.0000       69.0000      100.0000
```

Dado que un vector puede considerarse como una matriz de una sola columna o de una sola fila, es posible multiplicar vectores con matrices usando MATMUL:

```
real a(2,3),b(3),c(2)
a=reshape((/1,2,3,4,5,6/),(/2,3/))
b=(/1,2,3/)
c=matmul(a,b)                ! producto matricial c=ab
```

PRODUCT es una función de uso similar al de SUM; la diferencia es que PRODUCT determina el producto de los elementos de un arreglo:

```
integer,allocatable::test(:,:),prod(:),suma(:)
read *,m,n
allocate(test(m,n),prod(n),mult(m))
k=m*n
! se proporcionan valores a la matriz test
test=reshape((/(i,i=1,k)/),shape=(/m,n/))
do i=1,m
        print *,test(i,1:n)
```

```
end do
print *
! operaciones sobre el arreglo completo
lp=product(test);ls=sum(test)
print *,lp,ls
print *
! producto de los elementos de test a lo largo de la dimensión 1
prod= product(test,dim=1)
print *,prod
print *
! producto de los elementos de test diferentes de 4 y a lo largo de la dimensión 2
mult=product(test,dim=2,mask=test/=4)
print *,mult
```

Para m=2 y n=3, la salida impresa por las instrucciones del ejemplo anterior sería:

```
  1       3       5
  2       4       6

720      21

  2      12      30

 15      12
```

Cuando en una función RESHAPE no se especifican todos los elementos de un arreglo, a los restantes se les puede asignar un valor de "relleno". Esto es posible mediante el uso del argumento opcional PAD, que debe ser un arreglo del mismo tipo y con los mismos parámetros de tipo que el arreglo especificado como primer argumento en la función:

```
integer x(3,2)
integer relleno(2)
relleno=(/10,60/)
x=reshape((/1,2,3,4,/),(/3,2/),pad=relleno)
! pad=relleno,asignó 10 a x(2,2) y 60 a x(3,2)
```

Otro argumento opcional de RESHAPE es ORDER, que es de tipo entero y una permutación de (1,2,3,..,n) donde n es el tamaño del segundo argumento de RESHAPE. ORDER indica el orden en que es cambiado el valor de cada subíndice del arreglo:

```
real,dimension (2,2)::uno
```

```
uno=reshape((/3.0,4.9,-7.5,37.2/),(/2,2/),order=(/2,1/))
do i=1,2; print*,uno(i,1:2); end do
```

El último ejemplo imprimiría:

```
  3.0000      4.9000
 -7.5000     37.2000
```

como una consecuencia de order=(/2,1/);el segundo subíndice varía más rápido que el primero (asignación por filas). El valor por omisión para ORDER es (/1,2,3,..,n/), que en el caso de matrices, es equivalente a una asignación por columnas.
SHAPE es una función que determina la forma de un arreglo:

```
real,allocatable::arreglo(:,:)
read *,m,n
allocate (arreglo(0:m,0:n))
arreglo=12.5
! shape da la forma de arreglo
print*,shape(arreglo)
! para m=3 y n=4, shape es el vector (/4,5/)
```

Con COUNT es posible contar cuántos elementos de un arreglo cumplen con cierta condición(máscara). COUNT acepta también al argumento opcional DIM:

```
real x(2,3);integer m(3)
x=reshape((/1.0_4,0.0_4,-4.0_4,17.0_4,9.0_4,3.0_4/),(/2,3/))
! n es el número de elementos mayores o iguales a 5(2)
n=count(x>=5)
! m es el número de elementos menores o iguales a 3, a lo largo de la dimensión 1, en este
! caso, (/2,1,1/)
m=count(x<=3,1)
```

La transpuesta de un arreglo de rango 2 de cualquier tipo puede ser generada mediante el uso de la función TRANSPOSE:

```
complex,dimension(3,2)::a=reshape&
        ((/(1,2),(3,4),(5,6),(7,8),(9,10),(11,12)/),(/3,2/))
complex b(2,3)
b=transpose(a)          ! b es la transpuesta de a
```

PACK es una función que "empaqueta" a un arreglo de cualquier tipo dentro de un vector, bajo el control de una máscara. Ejemplos:

```
real *8 q(3,3)
real (kind=8),dimension(9)::v
! (i*2-4,i=1,9) produce la secuencia -2,0,2,4,6,8,10,12,14
q=reshape((/(i*2-4,i=1,9)/),(/3,3/))
v=pack(q,mask=q>10.0_8)
do i=1,3;print *;q(i,1:3);end do
print *
print *,v
```

Las últimas instrucciones producirían la siguiente impresión:

```
-2.00000000000000    4.00000000000000    10.00000000000000
 0.00000000000000    6.00000000000000    12.00000000000000
 2.00000000000000    8.00000000000000    14.00000000000000

12.00000000000000  14.00000000000000
```

```
integer,dimension(2,2)::datos=reshape((/1,-5,6,12/),(/2,2/))
print *,pack(datos,mask=reshape&
          ((/.true.,.false.,.true.,.true./),(/2,2/))
print *
! se empaca todo el arreglo datos en un vector
print *,pack(datos,.true.)
```

Los resultados del ejemplo anterior son:

```
1       6       12

1      -5    6    12
```

PACK permite un argumento opcional:VECTOR. Si este argumento está presente y su tamaño es mayor que el número de elementos de valor .true. en MASK, entonces los elementos del resultado con subíndices más allá de dicho número, serán llenados con los elementos correspondientes de VECTOR. A manera de ilustración, el código

```
integer,dimension(2,2)::mat
integer,dimension(6)::vec
mat=reshape((/1,2,3,4/),(/2,2/))
vec=(/10,20,30,40,50,60/)
print *,pack(mat,vector=vec,mask=.true.)
```

genera los resultados:

1 2 3 4 50 60

La función UNPACK desempaca un arreglo dentro de otro, bajo el control de una máscara:

unpack(vector,mask,field)

donde vector es un arreglo de cualquier tipo y de rango uno, mask es un arreglo de tipo lógico y field es del mismo tipo y parámetros de tipo que mask. El resultado es un arreglo del mismo tipo y parámetros de tipo de vector y con la forma de mask. Los elementos del arreglo resultado de UNPACK a los que no les corresponde un valor .true. en mask tendrán el valor field, si field es un escalar, o el correspondiente elemento de field si este es un vector:

```
integer prueba
integer,dimension(2,2)::mat
logical,dimension(2,2)::mascara
allocatable prueba(:)
allocate(prueba(4))
prueba=(/1,6,-23,12/)
mascara=reshape((/.true.,.false.,.true.,.false./),(/2,2/))
mat=unpack(prueba,mask=mascara,field=-5000)
! la instrucción anterior produce al arreglo mat(1,1)=1, mat(2,1)=-5000,mat(1,2)=6
! y mat(2,2)=-5000
```

LBOUND y UBOUND determinan a los subíndices primero y último de un arreglo, respectivamente. Si el argumento opcional DIM está presente, el resultado es un escalar con el valor del primero o del último subíndice en la dimensión DIM. Si DIM no es especificado, entonces el resultado es un arreglo de rango uno que contiene a los primeros o a los últimos subíndices del arreglo:

```
real,dimension(0:10,3,-1:5)::multi
integer sub_inferior(3),sub_superior(3)
multi=3.15
sub_inferior=lbound(multi)          ! sub_inferior=(/0,1,-1/)
indice=lbound(multi,dim=3)          ! indice=-1
sub_superior=ubound(multi)          ! sub_superior=(/10,3,5/)
k=ubound(multi,dim=2)               ! k=3
```

Es posible añadir una dimensión a un arreglo mediante la adición de un objeto de datos a lo largo de una dimensión dada. SPREAD es la función diseñada para tal efecto:

```
! b es un vector de 2 elementos
real,dimension(2)::b=(/1,2/)
real,dimension(2,3)::nuevo1
real,dimension (4,2)::nuevo2
! spread(b,2,3) forma un arreglo con 3 copias de b en la dimensión 2
nuevo1=spread(b,2,3)
do i=1,2
        print *,nuevo1(i,1:3)
end do
print *
! spread(b,dim=1,copies=4) forma otro arreglo con 4 copias de b en la dimensión 1
nuevo2=spread(b,dim=1,copies=4)
do index=1,4
        print *,nuevo2(index,1:2)
end do
```

Los arreglos impresos en el ejemplo anterior son:

```
1.00000       1.00000       1.00000
2.00000       2.00000       2.00000

1.00000       2.00000
1.00000       2.00000
1.00000       2.00000
1.00000       2.00000
```

Fortran 90 introduce dos funciones para el "avance" (shift) de elementos dentro de un arreglo: CSHIFT y EOSHIFT. Para comprender a CSHIFT puede ser útil pensar que esta función trabaja con una plantilla formada por los elementos del arreglo repetidos indefinidamente arriba y abajo de los valores originales. Por ejemplo, para el arreglo x generado por las instrucciones

```
! 3 y 4 son constantes con nombre
integer,parameter::m=3,n=4
integer,dimension(m,n)::x,y
x=reshape((/(ind,ind=1,m*n)/),(/m,n/))
```

la plantilla sería:

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

```
-----------------------------          ⇑ -
1      4      7      10
2      5      8      11
3      6      9      12
-----------------------------          ⇓ +
1      4      7      10
2      5      8      11
3      6      9      12
       .
       .
```

en la cual un cambio hacia abajo tiene signo positivo y un cambio hacia arriba un signo negativo. El arreglo original está encerrado entre líneas punteadas. Por lo tanto;

```
y=cshift(x,2)
```

cambia a los elementos del arreglo dos renglones hacia abajo en la plantilla, de tal forma que

```
do k=1,m;print *,y(k,1:n);end do
```

imprimiría

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

Si ahora las instrucciones son

```
y=cshift(x,-1)
print *
do i=1,m;print *,y(i,1:n);end do
```

entonces el arreglo escrito sería:

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

El número de posiciones que se avanzan no necesariamente es constante. Aplicando

```
y=cshift(x,(/1,2,3,1/))
print
do l=1,m;print *,y(l,1:n);end do
```

al arreglo x del último ejemplo, el resultado de CSHIFT es:

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

El cambio circular realizado por CSHIFT puede ser a lo largo de cualquier dimensión. Por supuesto, el valor por omisión es dim=1.

```
integer,parameter::m=3,n=4
integer,dimension(m,n)::x,y
x=reshape((/(i=1,m*n)/),(/m,n/))
do i=1,m;print *,x(i,1:n);end do
! avance circular a lo largo de la dimension 2. El default es dim=1
y=cshift(x,2,dim=2)
print *
do i=1,m;print *,x(i,1:n);end do
```

En este ejemplo, la plantilla se debe imaginar repitiendo indefinidamente los valores del arreglo, tanto a la izquierda como a la derecha de los elementos originales:

```
                    ⇐ -                          + ⇒
1     4     7     10    |1     4     7     10|    1     4     7     10
2     5     8     11    |2     5     8     11|    2     5     8     11
3     6     9     12    |3     6     9     12|    3     6     9     12
```

El avance hacia la izquierda es negativo y hacia la derecha es positivo. Tomando en cuenta las consideraciones anteriores, el último ejemplo produce la salida:

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

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

EOSHIFT funciona de manera parecida a CSHIFT; la diferencia es que la plantilla imaginaria ahora se forma con ceros arriba, abajo, a la izquierda y a la derecha de los elementos originales del arreglo (para el caso de dos dimensiones). Un ejemplo del uso de EOSHIFT:

```
integer,dimension(2,3)::a,b
integer,dimension(3)::array=(/1,2,3/)
a(1:2,1)=(/1,2/)
a(1:2,2)=(/3,4/)
a(1:2,3)=(/5,6/)
do i=1,2;print *,a(i,1:3);end do
! eoshift elimina elementos al efectuar un avance
b=eoshift(a,-1)
print *
do i=1,2
        print *,b(i,1:3)
end do
! cambio a lo largo de la dimension 2
b=eoshift(a,1,dim=2)
print *
do k=1,2
        print *,b(k,1:3)
end do
! boundary es un parámetro opcional
! boundary=99 sustituye los ceros de la plantilla por 99
b=eoshift(a,-array,boundary=99)
print *
do i=1,2;print*,b(i,1:3);end do
```

La plantilla imaginaria queda ahora:

```
                    0     0     0
                    0     0     0
                   ---------------------
0     0     0      |1     3     5|     0     0     0
0     0     0      |2     4     6|     0     0     0
                   ---------------------
                    0     0     0
                    0     0     0
```

y los resultados impresos por las instrucciones anteriores son:

```
 1       3       5
 2       4       6

 0       0       0
 1       3       5

 3       5       0
```

```
 4       6       0

99      99      99
 1      99      99
```

Tanto en CSHIFT como en EOSHIFT, el arreglo por cambiar puede ser de cualquier tipo.
La función MERGE escoge valores alternativos en base al valor de una máscara. El siguiente ejemplo muestra el uso de MERGE en la construcción de arreglos:

```
real,dimension(2,3)::uno, dos,tres
uno=reshape((/2.4,7.8,.01,-5.3,12.5,-9.17/),(/2,3/))
dos=reshape((/3.4,23.9,.12,17.3,8.2,-45.0/),(/2,3/))
print *;do k=1,2;print *,uno(k,1:3);end do
print *
print *;do l=1,2;print *,dos(l,1:3);end do
! tres será una matriz con elementos iguales a los de uno si la máscara uno>dos es
! verdadera. Si es falsa, los elementos de tres serán iguales a los de dos.
tres=merge(uno,dos,uno>dos)
print *
do i=1,2
        print *,tres(i,1:3)
end do
```

Las matrices uno, dos y tres son:

```
-2.40000        0.01000        12.50000
 7.80000       -5.30000        -9.17000

 3.40000        0.12000         8.20000
23.90000       17.30000       -45.00000

 3.40000        0.12000        12.50000
23.90000       17.30000        -9.17000
```

La función lógica ANY determina si algún valor en una máscara es cierto:

```
integer,dimension(3,2)::uno,dos
logical prueba1,prueba2(3),prueba3(2)
uno=reshape((/2,8,0,12,-4,17/),(/3,2/))
dos=reshape((/2,8,-5,23,-4,17/),(/3,2/))
! el resultado de any es verdadero, si hay algún elemento de uno que sea igual al
! correspondiente elemento de dos
```

```
prueba1=any(uno==dos)
! a lo largo de la dimensión 2
prueba2=any(uno>=dos,dim=2)
! a lo largo de la dimensión 1
prueba3=any(uno==dos.and.dos>10,dim=1)
print *,prueba1
print *,prueba2
print *,prueba3
```

Las instrucciones anteriores imprimen:

```
T
TTT
FT
```

ALL es una función lógica que averigua si todos los valores en una máscara son ciertos:

```
integer,dimension(2,3)::x,y
logical respuesta
logical,dimension(3)::answer1
logical,dimension(2)::answer2
x=reshape((/1,2,3,4,5,6/),(/2,3/))
y=reshape((/1,2,3,5,6,4/),(/2,3/))
respuesta=all(x==y)                    ! a respuesta se le asigna .false.
answer1=all(x==y,dim=1)
!answer1=(/.true.,.false.,.false./)
answer2=all(x=>y,dim=2)
!answer2=(/.false.,.false./)
```

Ninguna de las funciones intrínsecas que manejan arreglos es soportada por el estándar FORTRAN 77.

Ejercicios

1) Elabore un programa FORTRAN que calcule la media y la desviación estándar de un conjunto de datos.

$$\text{media} = \frac{\sum_{i=1}^{N} X_i}{N}$$

$$\text{desviación estándar} = \sqrt{\frac{\sum_{i=1}^{N} (X_i - \text{media})^2}{N}}$$

Pseudocódigo

```
!Este algoritmo determina la media y la desviación estándar
!de un conjunto de datos X
real X[100],media,sd,suma
entero  ndatos,i
leer ndatos
ejecuta i=1,ndatos  !lectura del vector x
     lee x(i)
escribe 'los', ndatos, ' son:'
escribe
ejecuta i=1,ndatos
     escribe x(i)
! cálculo de la media y la desviación estándar sd
suma=0.0
ejecuta i=1,ndatos
     suma=suma+x(i)
media=suma/ndatos
suma=0.0
ejecuta i=1,ndatos
     suma=suma+(x(i)-media)↑2
sd=SQRT(suma/ndatos)
escribe
escribe 'media=',media
escribe 'desviación estándar= ',sd
fin
```

Codificación en FORTRAN 77

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486) Sun Jan √ 0√:■ⁿ:√√ 1962          Page:  1
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

PROGRAM ESTAD , Opciones de Compilación:  /N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

    1       program estad
    2  c
```

```
 3  c      Este programa calcula la media y la desviacion estandar sd de un
 4  c      conjunto de datos x
 5  c
 6        real x,media,sd,suma
 7        integer ndatos,i
 8        dimension x(100)
 9        read *,ndatos
10  c
11  c      lectura del vector x
12  c
13        read *,(x(i),i=1,ndatos)
14        print *,'Los',ndatos,' datos son:'
15        print *
16        print *,(x(i),i=1,ndatos)
17  c
18  c      calculo de la media y la desviacion estandar
19  c
20        suma=0.0
21        do 10 i=1,ndatos
22       10 suma=suma+x(i)
23        media=suma/ndatos
24        suma=0.0
25        do 20 i=1,ndatos
26       20 suma=suma+(x(i)-media)**2
27        sd=sqrt(suma/ndatos)
28  c
29  c      impresion de resultados
30  c
31        print *
32        print *,'media=',media
33        print *,'desviavion estandar=',sd
34        end
```

Codificación en FORTRAN 90

```
Lahey Fortran 90 Compiler Release 3.00c                                Sat Jul  8 17:54:04 1961  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

PROGRAM ESTADISTICA    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin    -inln
                                  -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo    -nxref
------------------------------------------------- Source Listing -----------------------------------------------------
     1  program estadistica
     2  !
     3  !  Este programa calcula la media y la desviacion estandar de un
     4  !  conjunto de datos x
     5  !
     6  implicit none
     7  real,dimension(100)::x
     8  real media,desviacion_estandar
     9  integer numero_de_datos
    10  read *,numero_de_datos
    11  read *,x(1:numero_de_datos)     !lectura del vector x
    12  print *,'Los',numero_de_datos,' datos son:'
    13  print *;print *,x(1:numero_de_datos)
    14  media=sum(x(1:numero_de_datos))/numero_de_datos   !calculo de la media y la desviacion estandar
    15  desviacion_estandar=sqrt(sum((x(1:numero_de_datos)-&
    16                   media)**2)/numero_de_datos)
    17  print *
    18  print *,'media=',media    !impresion de resultados
    19  print *,'desviacion estandar=',desviacion_estandar
    20  end program estadistica
Bytes of stack required for this program unit: 56.
```

2) Diseñe un programa FORTRAN que obtenga la norma 1 $\|X\|_1$ de una matriz cuadrada de orden n.

$$\|X\|_1 = \max_{1 \le j \le n} \sum_{i=1}^{n} |X_{i,j}|$$

Pseudocódigo

```
!Cálculo de la norma 1 de una matriz X de nXn
entero n,i,j,k
real X[20,20],sumac[20],norma1
lee n
!lectura por columnas de la matriz X
lee X[1:n,1;n]
escribe         !impresión de X
ejecuta i=1,n
     escribe X[i,1:n]
!cálculo de las sumas parciales, columna por columna
ejecuta j=1,n
empieza
     sumac[j]=0
     ejecuta i=1,n
          sumac[j]=sumac[j]+ABS(X[i,j])
termina
!obtención de la norma 1 de X
norma1=sumac(1)
ejecuta j=2,n
     si sumac[j]>norma1 entonces norma1=sumac[j]
escribe "La norma 1 es:",norma1
fin
```

Codificación en FORTRAN 77

F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486) Page: 1

PROGRAM NORMA , Opciones de Compilación: /N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

```
 1       program norma
 2  c
 3  c     Calculo de la norma 1 de una matriz X de n x n
 4  c
 5       real X(20,20),sumac(20),norma1
 6       read *,n
 7  c
 8  c     lectura por columnas de la matriz X
 9  c
10       read *,((X(i,j),i=1,n),j=1,n)
11  c
12  c     impresion de X
13  c
14       do 10 i=1,n
15    10 print *,(X(i,k),k=1,n)
16  c
17  c     calculo de las sumas parciales, columna por columna
18  c
19       do 30 j=1,n
```

```
      sumac(j)=0.0
      do 20 i=1,n
   20 sumac(j)=sumac(j)+abs(X(i,j))
   30 continue
c
c     obtencion de la norma 1 de X
c
      norma1=sumac(1)
      do 40 j=2,n
        if(sumac(j).gt.norma1)norma1=sumac(j)
   40 continue
      print *
      print *,'La norma 1 es:',norma1
      stop
      end
```

Codificación en FORTRAN 90

```
Lahey Fortran 90 Compiler Release 3.00c                    Thu Jul 13 18:21:14 1961  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

PROGRAM NORMA_1_DE_UNA_MATRIZ    Compiling Options: -nchk  -ndal  -ndbl  -nf90  -nfix  -ng   -hed   -nin   -inln
                          -lst   -pca   -stchk  -nsyn  -ntrap  -nvax  -w    -nwo   -nxref
------------------------------------------------- Source Listing -----------------------------------------------
    1  program norma_1_de_una_matriz
    2  !
    3  !  Calculo de la norma 1 de una matriz X de n x n
    4  !
    5  real,dimension(20,20)::X
    6  real,dimension(20)::suma_columnas
    7  real norma_1
    8  read *,n
    9  read *,X(1:n,1:n)     !lectura por columnas de la matriz X
   10  do i=1,n
   11    print *,X(i,1:n)    !impresion de X
   12  end do
   13  ! calculo de las sumas parciales, columna por columna
   14  suma_columnas(1:n)=sum(abs(X(1:n,1:n)),dim=1)
   15  norma_1=maxval(suma_columnas(1:n))  !obtencion de la norma 1 de X
   16  print *
   17  print *,'La norma 1 es:',norma_1
   18  stop "FIN DE PROGRAMA"
   19  end program norma_1_de_una_matriz
Bytes of stack required for this program unit: 92.
```

Construcción WHERE

La construcción WHERE decide qué elementos de un arreglo serán afectados por un bloque de instrucciones de asignación (asignación de arreglos mediante una máscara).

```
WHERE(máscara)
      .
      instrucciones de asignación
      .
ELSEWHERE
      .
      instrucciones de asignación
      .
END WHERE
```

WHERE opera en forma selectiva sobre uno o más arreglos de la misma forma; las instrucciones de asignación serán ejecutadas dependiendo del valor de la máscara. El segmento de código:

```
real,dimension(3,3)::a=reshape((/1,2,3,4,5,6,7,8,9,/),(/3,3/),order=(/2,1/))
do k=1,3
        print*,a(k,1:3)
end do
where(a>=3.and.a<=6)          !construcción where
        a=0
elsewhere
        a=-a
end where
print *
do l=1,3
        print *,a(l,1:3)
end do
end
```

imprimiría la siguiente información:

```
 1.00000        2.00000        3.00000
 4.00000        5.00000        6.00000
 7.00000        8.00000        9.00000

-1.00000       -2.00000        0.00000
  0.00000        0.00000        0.00000
-7.00000       -8.00000       -9.00000
```

En algunas ocasiones, bastará con emplear la instrucción WHERE:

```
WHERE(máscara)instrucción de asignación
```

Un ejemplo del uso de la instrucción WHERE es:

```
real,dimension(5)::vec
vec=(/3.01,4.5,-7.8,12.3,157.5/)
print *,vec
where (vec>100)vec=100.0_4
print *; print *,vec
end
```

con resultados:

3.01000	4.50000	-7.80000	12.30000	157.50000
3.01000	4.5000	-7.80000	12.30000	100.00000

LECTURA / ESCRITURA

Las operaciones de lectura y escritura son realizadas sobre unidades lógicas. Una unidad lógica puede ser:
1)Un entero no negativo el cual es asociado con un dispositivo físico tal como la consola (teclado/pantalla), una impresora o un archivo en disco. La unidad se "conecta" a un archivo o a un dispositivo a través de una instrucción OPEN, excepto en el caso de archivos preconectados.
2)Un * que indica a los dispositivos estándar de lectura y escritura.
3)Una variable de caracteres que corresponde al nombre de un archivo interno.

Unidades Preconectadas

Las unidades 5,6 y * están automáticamente conectadas a un dispositivo, es decir, no hay necesidad de conectarlas explícitamente a través de una instrucción OPEN. La unidad 5 está conectada al dispositivo estándar de lectura, generalmente el teclado, y la unidad 6 está conectada al dispositivo estándar de escritura, frecuentemente la pantalla de la terminal. Tanto la unidad 5 como la unidad 6 pueden ser conectadas a otros dispositivos o archivos mediante un OPEN. La unidad * siempre está conectada a los dispositivos estándar de lectura y escritura.

Archivos

FORTRAN trata a todos los dispositivos físicos como si fueran archivos. Un archivo en una secuencia de cero o más registros. Los archivos en FORTRAN son de dos tipos: internos y externos.
Un archivo interno existe sólo dentro de una unidad de programa y es guardado en la memoria de la computadora. Un archivo externo puede residir en un disco y puede ser guardado para uso futuro y utilizado por otros programas.

Instrucción OPEN

Conecta o reconecta un archivo externo a una unidad de lectura/escritura. Ejemplos de instrucciones OPEN:

```
! el archivo c:\lf90\src\equi.dat se conecta a la unidad 19
open(unit=19,file='c:\lf90\src\equi.dat')

En Lahey FORTRAN 90, el número de unidad es un entero mayor
o igual que 0 y menor o igual que 32767

open(unit=24789,file='prog.dat')

open(78,file='prog.dat')

! en este caso, la cláusula FILE= es opcional
open(78,'prog.dat')

! se reconecta la unidad 5 con el archivo datos
open(5,'datos')

! en este ejemplo, bajo el sistema operativo DOS, se conecta una impresora con
! la unidad 17
open(17,file='prn')

! en DOS, CON representa a la consola (teclado/pantalla)
open(38,file='con')

! es posible conectar una unidad con un archivo "scratch"
read *,x;open(19,file='nul')
write(19,*)a   !se desecha la impresión de a (en DOS)

! arch es una variable de caracteres cuyo contenido es el nombre del archivo
character *10 arch;read *,arch;open(j+1,file=arch)
```

Instrucciones READ y WRITE

```
La instrucción READ transfiere los valores proporcionados a
través de una unidad de lectura a las entidades
especificadas en una lista de lectura o en un grupo
NAMELIST. Los siguientes son ejemplos del uso de la
instrucción READ:

! 100 es una etiqueta que identifica a una instrucción FORMAT
read(unit=5,fmt=100)a,i

read(5,100)a,i                  ! instrucción equivalente a la anterior

Un * en el lugar de la etiqueta  correspondiente a un
FORMAT, indica que la lectura se llevará a cabo con un
```

formato predefinido; los valores se proporcionarán separados por una ,o por espacios:

```
read(5,*)a,i
```

Las cláusulas FMT= y FILE= pueden ser especificadas en cualquier orden:

```
read(fmt=827,unit=38)k,l,m,n
```

Sin embargo, si estas se omiten, entonces el compilador supone que el primer número es la unidad y el segundo corresponde a la etiqueta del FORMAT:

```
read(38,827)k,l,m,n
```

La cláusula END=etiqueta transfiere el control a la instrucción etiquetada con ese número, siempre y cuando se detecte la marca de fin de archivo:

```
open(80,file='prueba.dat')
read(80,*,end=45)a,b
do while((a+b)<=100)
print *,a,b
        read (80,*,end=45)a,b
end do
stop  '(a+b)>100'
45 stop 'marca de fin de archivo'
```

En un READ es posible omitir el número de unidad y entonces el compilador tomará el valor por omisión *, es decir, la entrada estándar:

```
! 17 es la etiqueta de la instrucción FORMAT
read 17,a,b,x

! lectura en la unidad estándar con un formato predefinido (lectura dirigida por lista)
read *,a,b,x

! instrucción equivalente a la anterior
read(*,*)a,b,x
```

ERR=etiqueta transfiere el control hacia la instrucción señalada con tal etiqueta, en caso de que se detecte algún error durante la ejecución de la instrucción READ:

```
read(5,*,err=34)a,b,c
```

```
x=sqrt(2*a-b+c**2)
y=cos(x-a)
z=x+y
print *,a,b,c,x,y,z
stop
34 print *, 'error en la lectura de datos'
end
```

Una información más detallada sobre el tipo de error encontrado en la ejecución del READ es obtenida con el uso de la cláusula IOSTAT=variable entera. Si el valor devuelto en tal variable es negativo; ocurrió una condición de fin de archivo o de fin de registro. Otro tipo de error distinto a los anteriores corresponde a un valor positivo. En ausencia de error; el valor devuelto en la variable entera es cero:

```
integer status_de_lectura
read(*,*,iostat=status_de_lectura)a,b,c
do while(status_de_lectura==0)
        promedio=(a+b+c)/3
        print *,a,b,c,promedio
        read(5,*,iostat=status_de_lectura)a,b,c
end do
if(status_de_lectura>0)print *,'Error en la lectura'
```

La cláusula ADVANCE permite econtrolar el avance en la lectura de un registro. Por ejemplo, para la instrucciones

```
implicit none
integer x,suma,n_datos,i
n_datos=3
suma=0
do i=1,n_datos                  ! i2 implica que la variable entera x será leída en un campo
                                ! de 2 columnas
        read(*,'(i2)',advance='yes')x           ! lectura con avance
        suma=suma+x
end do
print *,'suma=',suma
end
```

los registros de lectura (líneas de datos) podrían ser:

```
10
12
17
```

mientras que para el código

```
implicit none
integer x,suma
integer n_datos,i
n_datos=3
suma=0
do i=1,n_datos                  ! i2 implica que la variable entera x será leída en un campo
                                ! de 2 columnas
        read(*,'(i2)',advance='no')x            !lectura sin avance
        suma=suma+x
end do
print *,'suma=',suma
end
```

el registro de lectura queda como:

```
101217
```

El valor por omisión es ADVANCE='YES'. La especificación ADVANCE es exclusiva de FORTRAN 90, esto es, no es permitida en FORTRAN 77. Con la cláusula ADVANCE no es posible emplear un * en lugar de un formato.
Si se detecta una marca de fin de registro; la cláusula EOR=etiqueta transfiere el control hacia la instrucción indicada por la etiqueta. Si en el siguiente ejemplo

```
integer a1,a2,a3,a4
open(5,file='a.dat')
! el formato es un cadena de caracteres
read(5,'(2i5)',advance='no',eor=999)a1,a2
read(5,'(2i5)',advance='no',eor=999)a3,a4
print *,a1**2,a2**2,a3**2,a4**2
stop
999 stop 'Se detecto una marca de fin de registro'
end
```

se supone que el contenido de a.dat es:

```
100
200  300  400
```

entonces el segundo READ no será ejecutado porque no fue posible leer un valor para a2 ya que se detectó una marca de fin de registro en la primera línea de a.dat.
Con SIZE=variable entera, es posible conocer cuántos caracteres fueron leídos por la instrucción READ:

```
integer numero_de_caracteres
read(5,'(2f10.2)',advance='no',size=numero_de_caracteres)a1,a2
print *,'caracteres transferidos=', numero_de_caracteres
stop
end
```

Para poder usar EOR=etiqueta y SIZE=variable entera, ADVANCE debe ser igual a 'NO'. FORTRAN 77 no reconoce a EOR=etiqueta ni a SIZE=variable entera.
La instrucción WRITE transfiere valores a una unidad de escritura desde las entidades especificadas en una lista de escritura o en un grupo NAMELIST. PRINT es equivalente a un WRITE dirigido a la unidad estándar de salida (la pantalla de la computadora).

```
! 100 es la etiqueta de la instrucción FORMAT
write(unit=6,fmt=100)a,b,nombre

! equivalente a la instrucción anterior
write(6,100)a,b,nombre

! la lista de escritura puede contener constantes, variables o expresiones
write(fmt=1000,unit=315)5,a+b,xyz
```

Un * en vez del número de la unidad indica que se trata de la unidad estándar de salida. Otro * en el lugar de la etiqueta de la instrucción FORMAT, ocasiona que la información sea escrita de acuerdo a un formato predefinido por el compilador.

```
write(*,*)a,f,'constante'

! 195 es la etiqueta del formato y la unidad de escritura es la estándar
print 195,a,j+k,l

! el formato es predefinido y la unidad de escritura es la estándar (escritura
! dirigida por lista)
print *,a,j+k,l
```

La cláusulas ERR, IOSTAT y ADVANCE también son aplicables a un WRITE:

```
write(6,125,err=17)x,y,z
w=(x+y+z)/70.0
print *,w
stop
```

```
17 print *,'error en la escritura de las variables x,y y z'

a=17.5e-3;b=7.8 4;mm=17
! (3f10.4) es el formato de escritura; cada variable será escrita en un campo de 10
! columnas incluyendo 4 de ellas para los decimales
write(*,'(3f10.4)',iostat=ierror)a,b,mm
select case(ierror)
case(:-1)
        print *,'condición de fin de archivo o fin de registro'
case(1:)
        print *,'error en la escritura'
end select

a=23.5
b=6.7
c=-12,8
! impresión con avance de registro (default)
write(6,'(f9.4)',advance='yes')a
write(6,'(f9.4)',advance='yes')b
write(6,'(f9.4)',advance='yes')c
```

En el último ejemplo, los valores de a,b y c serían impresos de la forma:

```
  23.5000
   6.7000
 -12.8000
```

mientras que las instrucciones

```
a=23.5
b=6.7
c=-12,8
!impresión sin avance de registro
write(6,'(f9.4)',advance='no')a
write(6,'(f9.4)',advance='no')b
write(6,'(f9.4)',advance='no')c
```

producen la impresión

```
 23.5000   6.7000 -12.8000
```

Control de carro

Generalmente, cada par WRITE-FORMAT genera un registro

formateado (formado por caracteres). El primer carácter de un registro formateado cuando es enviado a un dispositivo como la pantalla o la impresora, es usado para control de carro y no es impreso. Los demás caracteres restantes son impresos en una línea, comenzando en el margen izquierdo. El carácter de control de carro es interpretado como sigue:

Caracter	Espaciamiento vertical antes de la impresión
0	dos líneas
1	a la primera línea de la siguiente página
+	ninguno
cualquier otro	una línea

Instrucción FORMAT

Provee información explícita para dirigir la edición entre la representación interna de los datos y los caracteres que son leídos o escritos. Ejemplo:

```
read(5,89)a,j
89 format(f10.2,i4)
```

donde el formato (f10.2,i4) especifica que el valor de a será leído en las primeras 10 columnas del registro y que el valor de b será leído en las siguientes 4 columnas de ese mismo registro. La siguiente es una muestra de una instrucción FORMAT aplicada en una operación de escritura

```
write(6,127)a,j
127 format(1x,f10.2,i4)
```

en la cual se utiliza al trazador de control de edición nX para avanzar n espacios dentro de un registro. Por lo tanto, en el último ejemplo, el registro de impresión consistirá de una columna en blanco, del valor de la variable a escrito en las siguientes 10 columnas y del valor de la variable b impreso en las siguientes 4 columnas. Una instrucción FORMAT puede ser proporcionada a través de una cadena de caracteres:

```
read(5,'(f10.2,i4)')a,j
```

```
write(6,'(1x,f10.2,i4)')a,j

character *10 formato
read *,formato
read(5,formato)a,j

character *12 formato
read *,formato
write(6,formato)a,j
```

Edición de la lectura / escritura de datos

El trazador Fw.d permite la edición de datos reales, donde w es el ancho del campo en columnas y d son los dígitos después del punto decimal. La edición de datos de tipo entero es con el trazador Iw, donde w también es el ancho del campo en columnas. A manera de ilustración; el par READ-FORMAT

```
read(unit=5,fmt=100)a,j
      .
      .
100 format(f10.2,i3)
```

implica que la variable a será leída en las 10 primeras columnas del registro de lectura, de las cuales las 2 últimas se consideran reservadas para los decimales. Las siguientes 3 columnas serán usadas para leer el valor de la variable j. Por lo tanto, si los datos contenidos en el registro de lectura son:

```
     17438  2
12345678901234567890123456789012345678901234567890
```

entonces a tendrá el valor 174.38 y j el valor entero 2. Obviamente, la segunda línea es colocada para ayudar al lector en el momento de contar las columnas. En caso de que el valor correspondiente a la variable a se represente con el punto decimal, la especificación de que la últimas 2 columnas son decimales será ignorada:

```
  174.38    2
12345678901234567890123456789012345678901234567890
```

El trazador de edición Ew.d es empleado para leer o escribir datos reales con un formato de notación normalizada:

```
read(5,717)z,x
717 format(e10.2,e12.4)

  -317E+02  +6.03E23
12345678901234567890123456789012345678901234567890
```

Aquí, para la variable z son utilizadas las primeras 10 columnas, de las cuales, las 2 anteriores a la letra E están reservadas para los decimales. Las siguientes 12 columnas son para la variable x. Entonces, z tendrá el valor $-3.17x10^{2}$ y x será igual a $6.03X10^{23}$.
El trazador Dw.d es empleado para leer o escribir datos de doble precisión:

```
double precision x
read(5,10)x,j,k
      .
      .
10 format(d11.5,2i3)

    77.8D2   5-20
12345678901234567890123456789012345678901234567890
```

El número 2 que antecede al trazador i3 es un factor de repetición, es decir, 2i3 es equivalente a i3,i3. En este ejemplo, x será $77.8X10^{2}$, j 5 y k -20. Sin embargo, los datos REAL (KIND=8) o DOUBLE PRECISION también pueden ser leídos o escritos con el trazador Fw.d o Ew.d.
Los datos de tipo COMPLEX son manejados con un doble trazador Fw.d, Ew.d o Dw.d; el primero es empleado para la parte real y el segundo para la parte imaginaria.

```
complex w10
      .
      .
read(9,1100)w10
1100 format(2f8.1)   ! 2 es un factor de repetición

    -3.1     .74
12345678901234567890123456789012345678901234567890
```

En el caso anterior, w10 es igual a -3.1+.74i.
El trazador de edición Lw es empleado para leer o escribir datos de tipo LOGICAL:

```
logical u,p,z
      .
```

```
      .
read(5,1100)u,p,z
1100 format(l7,l2,l4)

 .TRUE. FFBCD
12345678901234567890123456789012345678901234567890
```

Si el registro de lectura contiene datos diferentes a .TRUE. o .FALSE., entonces se analiza el primer carácter en el campo; debe ser una T o una F (verdadero o falso). Se pueden emplear tanto letras mayúsculas como minúsculas.
Para leer o imprimir cadenas de caracteres, se usa el trazador Aw:

```
character uno
character *5 dos,tres
      .
      .
read '(a1,a7,a4)',uno,dos,tres

POLITECNICO
12345678901
```

A la variable uno se le asigna la cadena 'P'. Como se especifican 7 columnas para la variable dos que puede almacenar un máximo de 5 caracteres, entonces le son asignados los 5 últimos caracteres del campo, esto es,'ITECN'. La variable tres contendrá la cadena 'ICO'.
El trazador de control de edición nX también es aplicable en formatos de lectura:

```
read 999,c
      .
      .
999 format(3x,f7.2)

    -14.77
1234 5678901234567890123456789012345678901234567890
```

donde c tendrá el valor -14.77.
Los datos enteros pueden ser manejados en sistema binario o hexadecimal, a través de los trazadores de edición Bw,Ow y Zw, respectivamente:

```
integer::var_1,var_2
read(5,"(2b5)")var_1,var_2
```

```
0101000111
12345678901234567890123456789012345678901234567890

var_1 es 10 (10 es igual a 01010 en sistema binario) y var_2
es 7 (7 es 00111 en sistema binario).

integer prueba_1,prueba_2
read(5,8)prueba_ 1,prueba_2
      .
      .
8 format(o5,z7)

17     12a3f
12345678901234567890123456789012345678901234567890

prueba_1 es 15 (15 es 17 en octal) y prueba_2 76351 (76351
es 12A3F en sistema hexadecimal). Los trazadores de edición
Bw, Ow y Zw no son parte del estándar FORTRAN 77.
En operaciones de escritura, cada par WRITE-FORMAT genera
una cadena de caracteres que será transferida a la unidad de
salida. El primer carácter de esa cadena de salida es
interpretado como un control de carro y por lo tanto no es
impreso. Por tal motivo, el ejemplo

! el valor en memoria de f es -17.8394
write(6,1717)'PRUEBA',f
      .
      .
1717 format(a6,f10.2)

produce la cadena de salida

PRUEBA   - 17.84
12345678901234567890123456789012345678901234567890

que al ser transferida a la unidad 6 se verá impresa como

RUEBA   -17. 84
12345678901234567890123456789012345678901234567890

El primer carácter de la cadena de salida ('P') fue
interpretado como un control de carro de espacio sencillo.
El valor de f aparece redondeado con 2 decimales, ya que el
trazador de edición fue f10.2.
Siempre es una buena idea proporcionar en forma explícita el
carácter de control de carro, como en el ejemplo:

! f sigue conservando el valor de -17.8394
```

```
write(6,1717)'PRUEBA',f
      .
      .
! 1x equivale a un espacio
1717 format(1x,a6,f10.2)
```

que produce la impresión de los caracteres:

```
PRUEBA   - 17.84
12345678901234567890123456789012345678901234567890
```

El FORMAT 1717 también pudo ser escrito de cualquiera de las formas:

```
1717 format(' ',a6,f10.2)

1717 format(1h ,a6,f10.2)

! espaciamiento doble
1717 format('0',a6,f10.2)

! espaciamiento doble
1717 format(1h0,a6,f10.2)

! salto de hoja
1717 format('1',a6,f10.2)

! salto de hoja
1717 format(1h1,a6,f10.2)

! sin espaciamiento
1717 format('+',a6,f10.2)

! sin espaciamiento
1717 format(1h+,a6,f10.2)
```

El trazador de edición de cadenas de caracteres cHstring; causa la impresión de la cadena de caracteres string que tiene una longitud c. FORTRAN 90 considera al trazador cH una característica obsoleta.
Las instrucciones

```
double precision area
i=15
h=-2.0
area=3.1416
```

```
      .
      .
write(6,1200)i,h,area
      .
      .
1200 format(1x,i2,f5.1,f10.3)

producen la salida

15 -2.0    3 .142
12345678901234567890123456789012345678901234567890

mientras que

!uso del trazador nX en escritura
print 100,28,28,28,28
100 format(1x,i5,2x,b5,2x,o5,2x,z5)

imprime la línea

   28  11100     34     1C
12345678901234567890123456789012345678901234567890

En el último ejemplo, es posible usar los trazadores Iw.d,
Bw.d, Ow.d y Zw.d. En estos casos, d significa que se
imprimen al menos d caracteres:

print 101,28,28,28,28
101 format(1X,i5.4,2x,b5.4,2x,o5.4,2x,z5.4)

 0028  11100   0034   001C
12345678901234567890123456789012345678901234567890

En lectura; Iw.d, Bw.d, Ew.d y Zw.d son equivalentes a Iw,
Bw, Ow y Zw, respectivamente.
La escritura de un dato de tipo COMPLEX produce la impresión
de dos cantidades; la parte real seguida por la imaginaria:

complex arco
      .
      .
arco=(7,2.5)
print 15,arco
15 format(' ',2f8.3)

   7.000   2.500
12345678901234567890123456789012345678901234567890
```

Cuando el ancho del campo es insuficiente para escribir el dato tal como lo especifica el trazador de edición, el campo completo es llenado con caracteres *:

```
! epsi=-7878.15
write(6,890)epsi
      .
      .
890 format(lh ,f7.5)

*******
12345678901234567890123456789012345678901234567890
```

El trazador Ew.d escribe el dato en notación normalizada. Ew.dEe funciona de la misma manera, pero imprime un exponente de e dígitos:

```
pivote=-52.60779
write(6,1000)pivote,pivote
1000 format(3x,e12.3,1X,e12.4e3)

   -0.5261E+02 -0.5261E+002
12345678901234567890123456789012345678901234567890
```

ENw.d imprime un dato real o complex en notación de Ingeniería (notación exponencial con un exponente múltiplo de 3):

```
uno=.00379;dos=57827.14
write(6,"(1x,en10.3,2x,en12.3)")uno,dos

 3.790E-03    57.827E+03
12345678901234567890123456789012345678901234567890
```

Para imprimir un dato en notación científica, se emplea el trazador de edición ESw.d.

```
uno=.0157;dos=57827.14
write(6,"(1x,es10.3,2x,es12.3)")uno,dos

 1.570E-02     5.783E+03
12345678901234567890123456789012345678901234567890
```

En operaciones de lectura, ENw.d y ESw.d son equivalentes a Fw.d. Ambos trazadores no son parte del estándar FORTRAN 77. Por supuesto, también es factible usar los trazadores ENw.dEe y ESw.dEe:

```
write(6,'(1x,en12.3e3,2x,es12.3e3)')147.20,1472.0

147.200E+000    1.472E+003
12345678901234567890123456789012345678901234567890
```

En instrucciones de escritura con formato exponencial, el factor de escala sP imprime la mantisa del número multiplicada por 10^s y un exponente reducido en s unidades:

```
a=5.781e+02
b=-3.794e+03
! impresión con notación normalizada
write(6,300)a
! factor de escala 1p
write(6,400)a
write(6,300)b
      .
      .
300 format(1x,e12.4)
400 format(1x,1pe12.4)

  0.5781E+03
  5.7810E+02
 -0.3794E+04
12345678901234567890123456789012345678901234567890
```

Un factor de escala continua vigente en todo el registro:

```
! los valores de a y b son los mismos que los del ejemplo anterior
write(6,800)a
write(6,900)a,b
      .
      .
800 format(1x,e12.4);900 format('0',1pe12.4,e12.4)

  0.5781e+03
  5.7810E+02 -3.7940E+03
12345678901234567890123456789012345678901234567890
```

El uso del trazador de edición de cadenas de caracteres cH hace más legible la impresión:

```
write(6,10000)pre,temp
10000 format(1h1,4hPRE=,f7.3,3x,5hTEMP=,F7.3)

PRE= 14.700   TEMP=212.000
12345678901234567890123456789012345678901234567890
```

Para facilitar aún más la impresión de cadenas de caracteres, se permite especificar a estas entre ' o ", dentro del FORMAT:

```
write(6,10000)pre,temp
10000 format('1','PRE=',f7.3,3x,"TEMP",f7.3)
```

Los siguiente segmentos de código muestran la impresión de datos de tipo LOGICAL y CHARACTER:

```
logical a,b,c
a=.true.;b=.false.;c=.true.
print ,'(1x,l4,l2,l7)',a,b,c

   T F      T
12345678901234567890123456789012345678901234567890
```

```
character uno
character *5 dos,tres
uno='A';dos='BCDE';tres='FGHI'
      .
      .
print 19,uno,dos,tres
      .
      .
19 format(' ',a,a2,a7)   !a1 y a son equivalentes

ABC  FGHI
12345678901234567890123456789012345678901234567890
```

En caso de que en número de caracteres por imprimir sea menor que el ancho del campo, estos se escriben en las últimas columnas.
El trazador de control de edición / implica el fin de un registro y la apertura de uno nuevo:

```
real::var=-17.1
integer::lin=3276      write(6,1300)var,lin
1300 format(5x,f7.2,/,5x,i5)

     -17.10
     3276
12345678901234567890123456789012345678901234567890
```

La instrucción FORMAT es no ejecutable y puede colocarse en cualquier parte del programa

```
8 format('1',///,6x,30('*'),//,9x,'ESIQIE'//9x,&
        'PROGRAMACION FORTRAN 90',&
        2(/),6x,30(1h*))
        .
        .
write(6,8)
```

Por supuesto, 2(/) es lo mismo que // y 30(1h*) o 30('*') imprimirá 30 caracteres *:

```
      ******************************

         ESIQIE

         PROGRAMACION FORTRAN 90

      ******************************
12345678901234567890123456789012345678901234567890
```

Gw.d (trazador generalizado de edición) lee o escribe datos de tipo real:

```
read(5,1100)a,f
      .
      .
! este formato es equivalente a (f5.3,f6.1)
1100 format(g5.3,g6.1)

a=7.0
b=.007
c=700.
d=.00000007
write(6'(1x,2g11.3)')a,b
write(6,'(1x,2g11.4)')c,d

    7.00       0.700E-02
   700.0      0.7000E-07
12345678901234567890123456789012345678901234567890
```

Con el trazador generalizado y en escritura, el método de representar la información numérica depende de la magnitud

de ésta. Si en $\pm 0.d_1d_2d_3..d_n x 10^s$; $0 \leq s \leq d$, el trazador F es empleado y se escribe el número en w-4 columnas, dejando las cuatro últimas en blanco. Si s es negativo o mayor que d, el descriptor E es empleado. En Gw.d, d es el número de cifras significativas.

```
125 format(1x,g12.4e4);write(6,125)6.023E+23

0.6023E+0024
12345678901234567890123456789012345678901234567890

Si en un formato son dadas más especificaciones de campo que
las necesarias, las sobrantes son ignoradas:

character *20 ff
ff='(5f8.2)'
      .
      .
! 5 especificaciones de campo, 2 variables por leer; son ignoradas las 3 últimas
read(5,ff)a,g

Cuando se tienen más entidades por leer o escribir que
especificaciones de campo, entonces se aplica la regla del
paréntesis izquierdo: para leer o escribir las variables
restantes, el control se regresa al primer paréntesis
izquierdo dentro del FORMAT, pero se abre un nuevo registro.
Por ejemplo:

character *20 f10
      .
      .
! el formato es una variable de caracteres cuyo valor es leído desde la unidad 5
! (formato en tiempo de ejecución)
read(5,'(a20)')f10
read(5,f10)v_1,v_2,v_3

y si el registro de lectura es:

(2f10.4)
  3.142   -.0093
 4.7E3
12345678901234567890123456789012345678901234567890

entonces f10='(2f10.4)',v_1=3.142, v_2=-.0093 y v_3=4.7X10^3.
Las entidades de tipo derivado también pueden ser leídas o
escritas empleando instrucciones FORMAT:

type prueba
```

```
        integer (kind=4)::i
        real (kind=8)::x,y
end type prueba
type(prueba)::variable
        .
        .
read(5,1810)variable
1810 format(i5,2f10.3)
        .
        .
write(6,1821)variable
1821 format(1x,'variable%i=',i6,&
                        'variable%x=',f12.4,&
                        'variable%y=',f12.4)
```

Lectura/Escritura dirigida por lista

Una lectura o escritura dirigida por lista es una operación realizada en la unidad estándar con un formato predefinido.

```
logical a
character *5 cpostal
complex w
real *8 min
        .
        .
read *,a,cpostal,w,min,f,k
        .
        .
print *,a,cpostal,w,min,f,k
```

Para este ejemplo, un posible registro de lectura sería:

```
.TRUE.,'07770',(5.1,3.7) 3d7 0.9, 15
```

mientras que print * produciría una salida similar a:

```
T    07770     (5.10000,3.70000)   0.30000000000000d+07
0.90000   15
```

Los trazadores de edición empleados en el formato predefinido son:

Lectura dirigida por lista

Tipo	Edición
INTEGER	i
REAL	f
COMPLEX	como una literal COMPLEX
LOGICAl	l
CHARACTER	como una cadena de caracteres

Escritura dirigida por lista

Tipo	Edición
INTEGER	i
REAL	Gw.d
COMPLEX	(Gw.d,Gw.d)
LOGICAL	T o F
CHARACTER	como una cadena de caracteres

Bloques NAMELIST

Un bloque NAMELIST se emplea para definir un conjunto de variables y/o arreglos bajo un nombre único.

```
program muestra
! uno es el nombre del bloque NAMELIST
namelist /uno/i,j,x,y
        .
        .
read *,w
! lectura del NAMELIST uno
read(*,nml=uno)
        .
        .
print *,w
! escritura del NAMELIST uno
write(*,nml=uno)
        .
        .
```

Los registros de lectura correspondientes a las instrucciones anteriores son:

```
17.5
&uno i=3,x=2.3E10 y=-.007,j=6/
```

donde los datos del NAMELIST son separados por comas o por espacios. El inicio del NAMELIST es marcado con un & seguido de su nombre, el final es indicado con un /. Los resultados escritos son:

```
17.5000
&UNO I=3,J=6,X=0.23000E+11,Y=-.70000E-2/
```

En el ejemplo anterior, read(*,nml=uno) pudo haberse escrito como read uno. De manera semejante, write(*,nml=uno) equivale a print uno

Un bloque NAMELIST puede ser usado para leer o escribir en archivos:

```
namelist/datos/i,j,z
open(11,file='xyz.dat')
      .
      .
! lectura del NAMELIST datos en la unidad 11
read(11,datos)
print datos

read(unit=12,nml=propiedades)
! propiedades es un bloque NAMELIST
read(18,propiedades)
      .
      .
! salida es otro bloque NAMELIST
write(6,salida)
      .
      .
! opciones también es otro NAMELIST
write(unit=21,nml=opciones)
```

Un bloque NAMELIST puede contener arreglos, como en el ejemplo:

```
program arreglos
real n(2,2)
character *5 nombre
```

```
logical variable
namelist/datos/n,nombres,variable,k
      .
      .
read(5,datos)
write(6,datos)
      .
      .
end program arreglos
```

cuyo registro de datos es:

```
&datos n=4*10.0 nombre='RAUL',variable=.false./
```

en el cual aparece el factor de repetición 4: 4*10.0 equivale a 10.0, 10.0, 10.0, 10.0. La línea impresa queda como

```
&DATOS N=10.000,10.000,10.000,10.000,NOMBRE='RAUL',
VARIABLE=F,K=0/
```

Otro ejemplo con arreglos es:

```
integer,dimension(6)::j
namelist/prueba/j
      .
      .
read(*,prueba)
write(*,prueba)
```

con un registro de lectura

```
&prueba j(1:3)=3*12,j(4:6)=100,110,120/
```

y salida impresa

```
&PRUEBA J=12,12,12,100,110,120/
```

El bloque NAMELIST no forma parte de FORTRAN 77.

SUBPROGRAMAS

Unidades de Programa

Las unidades de programa son los elementos más pequeños que pueden ser compilados en forma separada. Hay cinco clases de unidades de programa:

1) Programas principales
2) Subprogramas externos FUNCTION
3) Subprogramas externos SUBROUTINE
4) BLOCK DATA
5) Módulos (no soportados en FORTRAN 77)

Clasificación de las funciones

Una función es un procedimiento que es invocado en una expresión. Una función produce un sólo resultado (ya sea un vector o un arreglo) y es llamada a través de su nombre. En FORTRAN las funciones se clasifican en: intrínsecas, externas, internas y de proposición. Las funciones intrínsecas son procedimientos provistos por el compilador (también se conocen como funciones de biblioteca). Una función externa es una unidad de programa que se compila por separado (subprograma externo FUNCTION). Una función interna está definida dentro de una unidad de programa y solo es reconocida en esa unidad. FORTRAN 77 no incluye el uso de funciones internas. Una función de proposición se define en una sola expresión y puede ser empleada exclusivamente en la unidad de programa que la contiene.

Funciones intrínsecas

Las funciones intrínsecas pueden ser específicas o genéricas. La versión genérica es usada para identificar dos o más funciones, de las cuales sola una será invocada por el compilador, según el tipo y clase de los argumentos empleados. El nombre específico por usar es determinado por el tipo, la clase y el rango del argumento.
El siguiente es un ejemplo con funciones intrínsecas:

```
real (kind=4)::x=3.78; real (kind=8)::y=.951
complex::z=(3.1,-2.7)
! sqrt es el nombre genérico de la función intrínseca raíz cuadrada; puede ser
! empleado para varios tipos y varias clases de argumentos
```

```
print *,sqrt(y)
print *,sqrt(z)
print *,sqrt(x)
! sqrt, dsrqt y csqrt son nombres específicos de la función raíz cuadrada; cada uno de ellos
! corresponde a un tipo y clase
print *,dsqrt(y)
print *,csqrt(z)
```

Las funciones intrínsecas pueden ser elementales o no elementales. Una función intrínseca elemental acepta como argumentos indistintamente tanto a los escalares como a los arreglos:

```
real::escalar=1.19,result_escalar
real,dimension(5)::vector=(/11.3,.7,-.19,1.7,2.01/),&
                                        result_ vector
!tanh es una función elemental
result_escalar=tanh(escalar)
result_vector=tanh(vector)

real,dimension(4)::a,b,c
read *,a;read *,b
! c(1)=sqrt(a(1)**2+b(1)**2)
! c(2)=sqrt(a(2)**2+b(2)**2)
! c(3)=sqrt(a(3)**2+b(3)**2)
! c(4)=sqrt(a(4)**2+b(3)**2)
c=sqrt(a**2+b**2)
print *,c
```

FORTRAN 77 no contempla el uso de funciones elementales. En el apéndice se encuentra una lista completa de las funciones intrínsecas de FORTRAN 90.

Funciones externas

Las funciones externas son unidades de programa diseñadas por el programador y que se compilan independientemente de cualquier otra unidad de programa. Las constantes, variables o expresiones que se utilizan en el llamado de la función se conocen como argumentos actuales. Las variables empleadas en la definición de la función son los argumentos fingidos (dummy arguments):

```
program principal_prueba
implicit none
```

```
integer j,ifact
read *,j
! ifact es una función externa y j es el argumento actual
print *,'El factorial de',j," es",ifact(j)
end program principal_prueba
function ifact(k)
implicit none
integer k,ifact,i
! Cálculo del factorial de k
! k es el argumento fingido
ifact=1
do i=2,k
        ifact=ifact*i
end do
! el resultado devuelto es el valor almacenado en la variable ifact
return
end function ifact
```

En el ejemplo arriba mostrado, la declaración IMPLICT NONE aparece en ambas unidades de programa puesto que son compiladas por separado. De hecho, tanto la función como el programa principal manejan espacios de direcciones diferentes, es decir, su propio conjunto de variables. La instrucción RETURN implica el fin de la ejecución de la función y el regreso al punto donde fue llamada. FORTRAN 77 sólo permite la instrucción END para indicar el final de la función externa; es decir, el uso de END FUNCTION nombre no es aceptado.

```
read *,t,a,x
        .
        .
! los argumentos actuales (actual arguments) pueden ser constantes, expresiones o variables
y=func(t,2.0)
        .
        .
print *,func(x,a+y)
        .
        .
end
function func(g,h)
! los argumentos fingidos(dummy arguments) necesariamente deben ser variables
        .
        .
return
```

```
end
```

Los argumentos actuales y los argumentos fingidos deben de coincidir en tipo y en clase. El siguiente es un programa con conflictos en la correspondencia en clase y tipo entre los argumentos actuales y los fingidos:

```
program correspondencia
implicit none
integer k
real (kind=4) x
read *,x,k
! El primer argumento actual (x) no corresponde en clase con el  primer argumento
! fingido (arg1).
! El segundo argumento actual (k) no corresponde en tipo con el segundo argumento
! fingido (arg2).
result=fun(x,k)
print *,result
end program correspondencia
function fun(arg1,arg2)
implicit none
real (kind=8) arg1,arg2,fun
fun=arg1/arg2
end function fun
```

En casos como el anterior, generalmente el resultado es impredecible.
El tipo y clase del resultado de una función externa puede ser especificado en la misma instrucción FUNCTION:

```
program principal_prueba
implicit none
real (kind=8)::ifact
integer j
read *,j
print *,'El factorial de',j," es",ifact(j)
end program principal_prueba
! ifact es una función REAL (KIND=8) de argumento entero
real (kind=8) function ifact(k)
implicit none
integer i,k
ifact=1
do i=1,k
        ifact=ifact*i
end do
return
```

```
end function ifact
```

Ejemplos adicionales de la instrucción FUNCTION:

```
double precision function solve_ecuacion(x)

character *20 function carac(linea)

complex function suma(z1,z2)
```

Por omisión, el resultado de un FUNCTION es devuelto en una variable con nombre idéntico al del FUNCTION. Sin embargo, con la opción RESULT, el resultado devuelto puede ser el valor almacenado en otra variable. La opción RESULT no es permitida en FORTRAN 77.

```
read *,radio
area_circulo=area(radio)
print *,radio,area_circulo
end
function area(r) result(superficie)
!el resultado devuelto es el valor de la variable superficie
implicit none
real (kind=4) r,superficie,pi
pi=4.0*atan(1.0_4)
superficie=pi*r*r
! el return colocado antes del END FUNCTION es opcional
end function area
```

Es válido que un FUNCTION no tenga argumentos:

```
function f10()
f10=375.128;return;end
```

Los argumentos de un subprograma FUNCTION pueden ser arreglos:

```
function norma_euclid(m,x)
implicit none
integer m
! el vector x es un argumento fingido
real norma_euclid
real ,dimension(10)::x
! cálculo de la norma euclidiana del vector x
norma_euclid=sqrt(sum(x(1:m)**2))
return
```

```
end function norma_euclid
program principal
implicit none
integer n
real norma_euclid
dimension a(10)
read *,n
read *,a(1:n)
print *,a(1:n)
! el vector a es un argumento actual
print *,"La norma euclidiana es:",norma_euclid(n,a)
end program principal
```

A continuación; ejemplos de programas y funciones que manejan tipos derivados:

```
program tipos
! SEQUENCE establece que el orden en que aparecen los componentes es el mismo
! en que serán almacenados
implicit none
type datos
        sequence
        real (kind=4)::a,b,c
end type datos
real (kind=4) temp,funcion_datos,y
type (datos)::x
read(*,*)x,temp                 ! x es un argumento actual de tipo datos
y=funcion_datos(x,temp)
write(*,*)temp,y
end program tipos
function funcion_datos(parametro,t)
implicit none
type datos
        sequence
        real (kind=4)::a,b,c
end type datos
real (kind=4) t,funcion_datos
! parametro es un argumento fingido de tipo datos
type (datos) parametro
funcion_datos=parametro%a+parametro%b*t+parametro%c*t**2
end function funcion_datos

type uno
        sequence
        real x,y
```

```
end type uno
type (uno) fun
! fun es un FUNCTION que devuelve un resultado del tipo uno
print *,fun(3.0)
end
function fun(f)
type uno
        sequence
        real x,y
end type uno
type(uno) fun
fun%x=10.0*f
fun%y=100.0*f
end function fun
```

Ejercicio

Escriba un subprograma FUNCTION que aproxime a la función exponencial e^x mediante

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \ldots$$

Pseudocódigo

```
!Este programa aproxima a la función exponencial
real fexp,x
lee x
!la función fexp aproxima a la exponencial del argumento x
escribe 'La exponencial de',x,' es',fexp(x)
fin
func fexp(x)
real fexp,fact,e
!Aproximación de la función exponencial de x
entero i
fexp=0
i=1
e=x**i/fact(i)      !se manda llamar a la función fact
mientras (ABS(e)≥1x10^-14)
empieza
```

```
      fexp=fexp+e
      i=i+1
      e=x**i/fact(i)
termina
regresa
fin
func fact(n)
real fact            !Cálculo del factorial de n
entero n
fact=1
ejecuta i=1,n
      fact=fact*i
regresa
fin
```

Codificación de FORTRAN 77

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486)         Page:     1
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

PROGRAM FEXPO , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

         1         program fexpo
         2         double precision fexp
         3   c
         4   *       Este programa aproxima a la funcion exponencial
         5   c
         6         read(5,'(f10.2)')x
         7   c
         8   c       el function fexp aproxima a la exponencial del
         9   c       argumento x
        10   c
        11         write(6,"(6x,'la exponencial de',f10.4,' es',g14.7)")x,fexp(x)
        12         end

F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486)         Page:     2
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

FUNCTION FEXP , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

        13         double precision function fexp(x)
        14         double precision fact,e
        15   c
        16   c       Aproximacion de la funcion exponencial de x
        17   c
        18         fexp=1.0
        19         i=1
        20   c
        21   c       se manda llamar al function fact
        22   c
        23         e=x**i/fact(i)
        24      10 if(abs(e).lt.1e-14)go to 20
        25           fexp=fexp+e
        26           i=i+1
        27           e=x**i/fact(i)
        28         go to 10
        29      20 return
        30         end

F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486)         Page:     3
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

FUNCTION FACT , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente
```

```
        31         function fact(n)
        32         real *8 fact
        33   c
        34   c       Calculo del factorial de n
        35   c
        36         fact=1.0
        37         do 10 i=1,n
        38      10 fact=fact*i
        39         end
```

Codificación en FORTRAN 90

```
Lahey Fortran 90 Compiler Release 3.00c                                        Fri Jul 28
15:43:28 1961  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

PROGRAM FUNCION_EXPONENCIAL    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed
-nin    -inln
                                      -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo    -nxref

------------------------------------------------  Source Listing
------------------------------------------------------------
      1  program funcion_exponencial
      2  real (kind=8)::fexp
      3  !
      4  !  Este programa aproxima a la funcion exponencial
      5  !
      6  read(5,'(f10.2)')x  !el function fexp aproxima a la funcion exponencial
      7  write(6,"(6x,'la exponencial de',f10.4,' es',g14.7)")x,fexp(x)
      8  end program funcion_exponencial
Bytes of stack required for this program unit: 36.

Lahey Fortran 90 Compiler Release 3.00c                                        Fri Jul 28
15:43:28 1961  Page: 2
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

FUNCTION FEXP    Compiling Options:    -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin    -inln

                                      -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo    -nxref

------------------------------------------------  Source Listing
------------------------------------------------------------
      9  real (kind=8) function fexp(x)
     10  real (kind=8)::factorial_de_n,elemento
     11  fexp=1.0      !Aproximacion de la funcion exponencial de x
     12  i=1
     13  elemento=x**i/factorial_de_n(i)  !se manda llamar al function factorial_de_n
     14  do while(abs(elemento)>=1e-14_8)
     15    fexp=fexp+elemento
     16    i=i+1
     17    elemento=x**i/factorial_de_n(i)
     18  end do
     19  end function fexp
Bytes of stack required for this program unit: 32.

Lahey Fortran 90 Compiler Release 3.00c                                        Fri Jul 28
15:43:28 1961  Page: 3
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

FUNCTION FACTORIAL_DE_N    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin
-inln
                                      -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo    -nxref

------------------------------------------------  Source Listing
------------------------------------------------------------
     20  function factorial_de_n(n) result(fact)
     21  real (kind=8) fact
     22  fact=1.0    !Calculo del factorial de n
     23  do i=1,n
     24    fact=fact*i
     25  end do
     26  end function factorial_de_n
Bytes of stack required for this program unit: 28.
```

Subrutinas

Una subrutina externa es una unidad de programa completa e independiente de cualquier otra y que devuelve cualquier número de resultados.

```
program uno
implicit none
real a,b,c,x1,x2
logical complejas
! Cálculo de las raíces reales de una ecuación cuadrática
read *,a,b,c
print *,'A=',a,' B=',b,' C=',c
! a, b, c, x1, x2, complejas son los argumentos actuales
call cuadratica(a,b,c,x1,x2,complejas)
if(complejas)then
        print *,'Raices complejas'
else
        print *,'x1=',x1," x2=",x2
end if
end program uno
subroutine cuadratica(a,b,c,x1,x2,raices_complejas)
implicit none
real a,b,c,x1,x2,disc
logical raices_complejas
!
! Esta subrutina calcula las raíces de una ecuación cuadrática de coeficientes a,b y c.
! Si se detectan raices imaginarias, la subrutina asigna el valor .true. a raices_complejas
!
disc=b**2-4*a*c        ! disc es una variable local
if(disc.lt.0.0) then
        raices_complejas=.true.
else
        x1=(-b+sqrt(disc))/(2*a)
        x2=(-b-sqrt(disc))/(2*a)
        raices_complejas=.false.
end if
return
end subroutine cuadratica
```

La instrucción CALL se emplea para llamar a una subrutina. El nombre de la subrutina sirve solo para propósitos de identificación y no tiene que ver con el tipo y número de sus resultados. Al igual que en los subprogramas FUNCTION; los argumentos actuales pueden ser constantes, variables o expresiones y los argumentos fingidos deben ser variables:

```
read *,x,y,z
```

```
        .
        .
call prueba(x,y+z,25E4,result)
        .
        .
end
subroutine prueba(a,b,c,resultado)
        .
        .
end subroutine prueba
```

La instrucción END SUBROUTINE opcionalmente seguida del nombre de la subrutina indica el final de la misma. Sin embargo, en FORTRAN 77 sólo es posible emplear un simple END.
Los argumentos actuales y los argumentos fingidos deben de coincidir en tipo y clase. Un ejemplo de no concordancia en este aspecto es:

```
real (kind=8)::var1
integer var2
        .
        .
! el argumento actual var1 es real (kind=8)
! el argumento actual var2 es entero
call sub(var1,var2,suma,resta)
        .
        .
end
subroutine sub(x1,x2,suma_de_x1_y_x2,resta_de_x1_y_x2)
! el argumento fingido x1 es REAL (KIND=4)
! el argumento fingido x2 es REAL (KIND=4)
suma_de_x1_y_x2=x1+x2
resta_de_x1_y_x2=x1-x2
return
end
```

en el cual, muy probablemente, los resultados serán impredecibles.
Los argumentos de una subrutina pueden también ser arreglos:

```
real,dimension(10,10)::a,a_pot
read *,n;read *,a(1:n,1:n);read *,m
call potencia_de_matrices(m,n,a,a_pot)
do i=1,n
```

```
        print *,a_pot(i,1:n)
end do
end
subroutine potencia_de_matrices(m,n,mat,pot)
! Cálculo de la m-ésima potencia de una matriz real mat
dimension mat(10,10),pot(10,10)
pot=0.0;do i=1,n;pot(i,i)=1.0;end do;if(m==0)return
do i=1,m
        pot(1:n,1:n)=matmul(pot(1:n,1:n),mat(1:n,1:n))
end do
end                     ! el return antes del end es opcional
```

Una subrutina también es capaz de manejar tipos derivados:

```
program princip
type estruct
        sequence                ! definiciones de los tipos derivados
        integer k
        real a,b
end type estruct
type salida
        sequence
        real result1,result2
end type salida
type(estruct) datos
type(salida) resultado
read *,datos
call sub_datos(datos,resultado)
print *,resultado
end program princip
subroutine sub_datos(datos,res)
! definición de los tipos derivados estruct y salida
type estruct
        sequence
        integer k
        real a,b
end type estruct
type salida
        sequence
        real uno,dos
end type salida
type(estruct) datos
type(salida) res
res%uno=sin(datos%a)-cos(datos%b)
res%dos=datos%k**2
```

```
end subroutine sub_datos
```

Atributos INTRINSIC y EXTERNAL. Proposiciones INTRINSIC y EXTERNAL

El atributo EXTERNAL indica el nombre de un procedimiento externo. Al declarar a una función o a una subrutina como externa, entonces es posible usarla como un argumento actual:

```
program muestra_uno
real,external::hipotenusa
read *,a,b
call proc(a,b,result,hipotenusa)                ! hipotenusa es una función externa
print *,result
end program muestra_uno
subroutine proc(x,y,res,funci)
real,external::funci
res=x*y-funci(x,y)
end subroutine proc
function hipotenusa(a,b)
hipotenusa=sqrt(a**2+b**2)
end function hipotenusa
```

El atributo INTRINSIC especifica que una función es intrínseca. El declarar a una función como intrínseca, hace posible usarla como un argumento actual:

```
program muestra_dos
real, intrinsic::alog                   ! la función intrínseca debe ser específica
read *,a,b,
!
! se pasa la función intrínseca alog como un argumento actual
call proc(a,b,result,alog)
print *, result
end program muestra_dos
subroutine proc(x,y,res,funci)
res=x*y-funci(x)
end subroutine proc
```

En FORTRAN 77 no se permiten los atributos. Sin embargo, existen las proposiciones EXTERNAL e INTRINSIC que si son parte del estándar FORTRAN 77:

```
program tres
external hipot
```

```
read *,a,b
call proc(a,b,result,hipot)
print *,result
end
subroutine proc(x,y,res,funci)
external funci
res=x*y-funci(x,y)
end
function hipot(a,b)
hipot=sqrt(a**2+b**2)
end

program cuatro
intrinsic alog
read *,a,b
call proc(a,b,result,alog)
print *,result
end
subroutine proc(x,y,res,funci)
res=x*y-funci(x)
end
```

Con la proposición EXTERNAL también es posible pasar subrutinas o funciones externas como argumentos actuales.

```
program muestra_sub
external subr   !la subrutina subr es declarada como EXTERNAL
read *,a,b
call proc(a,b,subr,c,d)
print *,c,d
end program muestra_sub
subroutine proc(x,y,sub,res1,res2)
res1=x**2-y**3
call sub(x,y,res2)
end subroutine proc
subroutine subr(arg1,arg2,result)
cond1:if(arg1>arg2)then
                result=log(arg1-arg2)
        else cond1
                result=arg2-arg1
    end if cond1
return
end subroutine subr
```

Atributo INTENT. Proposición INTENT

El atributo INTENT especifica el uso que tendrá un argumento fingido. Un argumento declarado con INTENT(IN) no puede ser definido o redefinido en el subprograma. Un argumento declarado con INTENT(OUT) debe ser definido antes de ser usado. INTENT(INOUT)implica que un argumento puede ser usado tanto para recibir datos como para devolverlos. El atributo por omisión es INTENT(INOUT).

```
program argumentos
implicit none
real a,b,c,d
read *,a,b,d,
call subarg(a,b,c,d)
print *,a,b,c,d
end
subroutine subarg(arg1,arg2,arg3,arg4)
implicit none
real,intent(in)::arg1,arg2              ! arg1 y arg2 sólo reciben datos
real,intent(out)::arg3                  ! arg3 sólo devuelve datos
real,intent(inout)::arg4
arg3=arg1/arg2-arg4
arg4=amax1(arg1,arg2)-amin1(arg1,arg2)
end subroutine subarg
```

El siguiente programa muestra un uso incorrecto del atributo INTENT:

```
program errores
integer::m=12,n=3
call ejemplo_incorrecto(m,n)
end program errores
subroutine ejemplo_incorrecto(i,j)
integer,intent(in)::i
integer,intent(out)::j
i=456                                   ! no es posible redefinir a i
k=j                                     ! j no está definida
print *,k
end subroutine ejemplo_incorrecto
```

Es posible también emplear la proposición INTENT en vez del atributo INTENT:

```
subroutine subarg(arg1,arg2,arg3,arg4)
```

```
implicit none
intent(in) arg1,arg2
intent(out) arg3
intent(inout) arg4
arg3=arg1/arg3-arg4
arg4=amax1(arg1,arg2)-amin1(arg1,arg2)
end subroutine subarg
```

En FORTRAN 77 no se permiten el atributo ni la proposición INTENT; todos los argumentos fingidos son tanto de entrada como de salida.

Ejercicio

Cálculo de una raíz de f(x)=0 con el método de Newton Raphson.

$$x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)})}{f'(x^{(n)})}$$

Pseudocódigo

```
!Cálculo de una raíz de fun(x)=0 con el método de
!Newton-Raphson
real x0,x,tol,fun,dfun
entero maxit        !maxit es el número máximo de
                    !iteraciones
lógico conv         !tol es el criterio de convergencia
                    !x0 es la aproximación inicial
lee x0,tol,maxit
llama newton(xo,tol,maxit,x,conv,func fun,func dfun)
si conv entonces    !se checa la convergencia
     escribe 'la raíz es x=',x
otro
     escribe 'no hubo convergencia después de',maxit,
             'iteraciones'
fin
proc newton(x0,eps,max,x,conv,func f,func fprima)
!Esta subrutina calcula una raíz aproximada de f(x)=0
!con el método de Newton-Raphson
real x0,eps,x,f,fprima
```

```
entero max,iter;lógico conv
conv=VERDADERO
ejecuta iter=1,max
comienza
   x=x0-f(x0)/fprima(x0)
   si ABS((x-x0)/x)<eps entonces regresa
   x0=x
termina
conv=FALSO
regresa
fin
func fun(x)
real x,fun          !función por resolver
fun=x↑3-2*x+sin(x)
regresa
fin
Función fprima(y)
real fprima,y       !primera derivada de fun(x)
fprima=3*x↑2-2+cos(x)
regresa
fin
```

Codificación en FORTRAN 77

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486)         Page:     1
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

PROGRAM NEWRAP , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

        1   c
        2   c       Calculo de una raiz de fun(x)=0 con el metodo de
        3   c       Newton Raphson
        4   c
        5         program newrap
        6   c
        7   c       maxit es el numero maximo de iteraciones
        8   c       tol es el criterio de convergencia
        9   c       x0 es la aproximacion inicial
       10   c
       11         logical conv
       12         external fun,dfun
       13         read *,x0,tol,maxit
       14         call newton(x0,tol,maxit,x,conv,fun,dfun)
       15   c
       16   c       se checa la convergencia
       17   c
       18         if(conv)then
       19           print 1000,x
       20         else
       21           print 1100,maxit
       22         end if
       23         stop
       24   1000  format(//,11x,'una raiz es x=',g14.6)
       25   1100  format(//,11x,'no hubo convergencia despues de',i4,' iteraciones')
       26         end

F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486)         Page:     2
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

SUBROUTINE NEWTON , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente
```

```
        27         subroutine newton(x0,eps,max,x,conv,f,fprima)
        28   c
        29   c       Esta subrutina calcula una raiz aproximada de f(x)=0 con
        30   c       el metodo de Newton Raphson
        31   c
        32         logical conv
        33         external f,fprima
        34         conv=.true.
        35         do 10 iter=1,max
        36           x=x0-f(x0)/fprima(x0)
        37           if(abs((x-x0)/x).lt.eps)return
        38        10 x0=x
        39           conv=.false.
        40         return
        41         end
```

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486)         Page:     3
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

FUNCTION FUN , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

        42         function fun(x)
        43   c
        44   c       Funcion por resolver
        45   c
        46         fun=x**3-2*x+sin(x)
        47         return
        48         end
```

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486)         Page:     4
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

FUNCTION DFUN , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

        49         function dfun(x)
        50   c
        51   c      Primera derivada de fun(x)
        52   c
        53         dfun=3*x**2-2+cos(x)
        54         return
        55         end
```

Codificación en FORTRAN 90

```
Lahey Fortran 90 Compiler Release 3.00c                                                Thu Jul 27
17:49:12 1961  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

PROGRAM NEWTON_RAPHSON    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin
-inln
                                     -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
------------------------------------------------------------
      1  program newton_raphson
      2  !
      3  ! Calculo de una raiz de fun(x)=0 con el metodo de
      4  ! Newton Raphson
      5  !
      6  logical conv          !maxit es el numero maximo de iteraciones
      7  external fun,dfun     !tol es el criterio de convergencia
      8  read *,x0,tol,maxit   !x0 es la aproximacion inicial
      9  call newton(x0,tol,maxit,x,conv,fun,dfun)
     10  if(conv)then          !se checa la convergencia
     11    print 1000,x
     12  else
     13    print 1100,maxit
     14  end if
     15  stop
     16  1000  format(//,11x,'una raiz es x=',g14.6)
     17  1100  format(//,11x,'no hubo convergencia despues de',i4,' iteraciones')
     18  end program newton_raphson
Bytes of stack required for this program unit: 36.

Lahey Fortran 90 Compiler Release 3.00c                                                Thu Jul 27
```

```
17:49:12 1961  Page: 2
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

SUBROUTINE NEWTON    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin
-inln
                                        -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
-------------------------------------------------------------
     19  subroutine newton(x0,eps,max,x,conv,f,fprima)
     20  real,intent(in)::eps
     21  integer,intent(in)::max
     22  real,intent(out)::x
     23  logical,intent(out)::conv
     24  real,intent(inout)::x0
     25  real,external::f,fprima
     26  !
     27  ! Esta subrutina calcula una raiz aproximada de f(x)=0 con
     28  ! el metodo de Newton Raphson
     29  !
     30  conv=.true.
     31  iteracion:do iter=1,max
     32              x=x0-f(x0)/fprima(x0)
     33              if(abs((x-x0)/x)<eps)return
     34              x0=x
     35            end do iteracion
     36  conv=.false.
     37  return
     38  end subroutine newton
Bytes of stack required for this program unit: 28.

Lahey Fortran 90 Compiler Release 3.00c                                                  Thu Jul 27
17:49:12 1961  Page: 3
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

FUNCTION FUN    Compiling Options:     -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin    -inln

                                       -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
-------------------------------------------------------------
     39  function fun(x)
     40  fun=x**3-2*x+sin(x)    !Funcion por resolver
     41  return
     42  end function fun
Bytes of stack required for this program unit: 16.

Lahey Fortran 90 Compiler Release 3.00c                                                  Thu Jul 27
17:49:12 1961  Page: 4
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

FUNCTION DFUN    Compiling Options:    -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin    -inln

                                       -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
-------------------------------------------------------------
     43  function dfun(x)
     44  dfun=3*x**2-2+cos(x)   !Primera derivada de fun
     45  return
     46  end function dfun
Bytes of stack required for this program unit: 16.
```

Interfase de un procedimiento

La interfase de un procedimiento es el conjunto de todas las características de éste que son de interés cuando es invocado. Estas características incluyen, además del nombre del procedimiento, el número, orden, tipo, forma, intent de los argumentos, etc.
Una interfase puede ser explícita o implícita. En una interfase explícita; el compilador tiene acceso a todas las

características del procedimiento, mientras que en la implícita, el compilador tiene que deducirlas.

```
program ecuacion_cuadratica
implicit none
! Bloque de interfase (interfase explícita)
interface
        subroutine cuadratica(a,b,c,x1,x2)
        implicit none
        real,intent(in)::a,b,c
        complex,intent(out)::x1,x2
        end subroutine cuadratica
end interface
real a,b,c
complex x_uno,x_dos
read *,a,b,c
print *,'a=',a,' b=',' c=',c
call cuadratica(a,b,c,x_uno,x_dos)
print *,'x1=',x_uno,' x2=',x_dos
end program ecuacion_cuadratica
subroutine cuadratica(a,b,c,x1,x2)
implicit none
real,intent(in)::a,b,c
complex,intent(out)::x1,x2
complex disc
disc=cmplx(b**2-4*a*c,4,kind=4)          ! conversión a tipo complex
x1=(-b+sqrt(disc))/(2*a)
x2=(-b-sqrt(disc))/(2*a)
end subroutine cuadratica
```

Es deseable, para evitar errores y facilitarle el trabajo al compilador, siempre crear interfases explícitas.
En los siguientes casos, una interfase explícita es obligatoria:

1) Si en una referencia al procedimiento aparece
- un argumento asociado a una palabra clave
- como extensión del significado de un operador estándar
- en una expresión como un operador definido
- como una referencia por su nombre genérico

2) Si el procedimiento tiene
- un argumento fingido opcional
- un arreglo como resultado en el caso de una función
- un argumento fingido que es un arreglo de forma supuesta, un apuntador o un destino
- un resultado tipo CHARACTER cuya longitud no es supuesta ni constante

- un resultado que es un apuntador en el caso de una función

Una interfase siempre es explícita para procedimientos intrínsecos, procedimientos internos y procedimientos de módulo. La interfase de una proposición de función siempre es implícita. En cualquier otro caso, las interfases explicitas son establecidas a través de un bloque de interfase.
El estándar FORTRAN 77 no incluye las interfases explícitas; la única interfase que maneja es la implícita.

Atributo OPTIONAL. Proposición OPTIONAL. Argumentos con palabra clave

Un argumento actual no necesita ser provisto para corresponder a un argumento fingido, si es que se especifica el atributo OPTIONAL:

```
implicit none
interface
        real function funcion_opcional(arg1,arg2,arg3) &
                result (suma)
        real,optional::arg3      !arg3 es un argumento opcional
        end function funcion_opcional
end interface
real a,b,c,resultado;read *,a,b,c
! se manda llamar a la función con tres argumentos actuales (dos obligatorios y uno
! opcional)
resultado=funcion_opcional(a,b,c)
print *,resultado
! ahora se usan solamente dos argumentos
resultado=funcion_opcional(a,c/d)
print *, resultado
end
real function funcion_opcional(arg1,arg2,arg3) result (suma)
real,optional::arg3
suma=arg1+arg2
! la función PRESENT  pregunta si fue pasado un argumento opcional
if(present(arg3))suma=suma+sin(arg3)
end function funcion_opcional
```

Un argumento opcional al final de la lista de argumentos fingidos puede ser omitido de la lista de los correspondientes argumentos actuales.
Los argumentos con palabra clave (keyword arguments)

palabra clave=argumento actual

donde palabra clave es el nombre del argumento fingido, son empleados para omitir otros argumentos opcionales:

```
program principal
interface
        subroutine test(x,y,z)
        real y          !y es obligatorio
        optional x,z            !uso de la proposición OPTIONAL
        end subroutine test
end interface
        .
        .
! x y z son omitidos, la palabra clave es necesaria
call test(y=3.0)
        .
        .
! z es omitido
call test(2.15,17.4)
        .
        .
! x es omitido, la palabra clave es necesaria
call test(z=3e2,y=.07)
        .
        .
end program principal
subroutine test (x,y,z)
real y
optional x,z
        .
        .
end subroutine test
```

A continuación, una nueva versión del programa desarrollado en el último ejercicio, que incluye una interfase con argumentos opcionales:

```
program solucion
interface
        subroutine newton(xinic,eps,maxit,x,converg,iter,&
                        impre,fun,derivada_de_fun)
        real,intent(inout)::xinic
        real,intent(in)::eps;integer,intent(in)::maxit
        real,intent(out)::x
```

```
        logical,intent(out):: converg
        integer,intent(out),optional::iter
        integer,intent(in),optional::impre
        real,external::fun, derivada_de_fun
        end subroutine newton
end interface
logical convergencia
external funci,dfunci
read *,xinic,tolerancia
call newton(xinic,tolerancia,50,x,convergencia,impre=5,&
                        fun=funci,derivada_de_fun=dfunci)
if(convergencia)then
        print *,'una raíz es x=',x
else
        print *,"Sin convergencia"
end if
end program solucion
subroutine newton(x0,eps,maxit,x,conver,iter,impre,fun,&
        derivada_de_fun)
real,intent(inout)::x0
real,intent(in)::eps;integer,intent(in)::maxit
real,intent(out)::x;logical,intent(out)::conver
real,intent(out),optional::iter
integer,intent(in),optional::impre
real,external::fun,derivada_de_fun
! Solución de fun(x)=0 con el método de Newton Raphson
do k=1,maxit
        x=x0-fun(x0)/derivada_de_fun(x0)
        if(present(impre))then
          if(mod(k,impre)==0)print *,'iter=',k,'x=',x
        end if   !impresión cada impre iteraciones
        if(abs((x-x0)/x)<eps)exit
        x0=x
end do
if(present(iter))iter=k
if(k>maxit)then
        conver=.false.
else
        conver=.true.
end if
end subroutine newton
function funci(x)
funci=x**3-2*x+sin(x);end
function dfunci(x)
dfunci=3*x**2-2+cos(x)
```

```
end
```

FORTRAN 77 no reconoce los argumentos opcionales ni los argumentos con palabra clave.

Paso de arreglos de tamaño o forma desconocidos

Un arreglo de tamaño supuesto (assumed size array) es un argumento fingido cuyo tamaño es desconocido. Todos los límites de las dimensiones, excepto el límite superior de la última dimensión, son especificados en la declaración del arreglo fingido. En la declaración, el límite superior de la última dimensión es indicado mediante un *. Los dos arreglos (el actual y el fingido) tienen el mismo elemento inicial.

```
program paso_de_arreglos
real,dimension(10,10)::matriz
!
! El número de filas de matriz real,dimension (10)::vector es su dimensión
! principal  (leading dimension)
!
read *,m,n
read *,matriz(1:m,1:n)
read *,vector(1:n)
call impre(m,n,matriz,vector)
end program paso_de_arreglos
subroutine impre(nf,nc,x,y)
!x y y son arreglos de tamaño supuesto
!x debe declararse como un arreglo de 10 filas
real,dimension(10,*)::x
real,dimension(*)::y
do i=1,nf
        print *,x(i,1:nc)
end do
print *
print *,y(1:nc)
end subroutine impre
```

En el programa

```
program prueba
dimension x(3,2,2)
read *,x
call sub(x)
```

```
end program prueba
subroutine sub(array)
dimension array(-1:1,2,*)
do i=-1,1
        do j=1,2
                do k=1,2
                        write(*,*,advance='no')array(i,j,k)
                end do
        end do
end do
end subroutine sub
```

suponiendo un registro de lectura

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

entonces el arreglo x queda almacenado de la siguiente forma:

x(1,1,1) en prueba=.1000000E+01=array(-1,1,1) en sub
x(2,1,1) en prueba=.2000000E+01=array(0,1,1) en sub
x(3,1,1) en prueba=.3000000e+01=array(1,1,1) en sub
x(1,2,1) en prueba=.4000000e+01=array(-1,2,1) en sub
x(2,2,1) en prueba=.5000000e+01=array(0,2,1) en sub
x(3,2,1) en prueba=.6000000e+01=array(1,2,1) en sub
x(1,1,2) en prueba=.7000000e+01=array(-1,1,2) en sub
x(2,1,2) en prueba=.8000000e+01=array(0,1,2) en sub
x(3,1,2) en prueba=.9000000e+01=array(1,1,2) en sub
x(1,2,2) en prueba=.1000000e+02=array(-1,2,2) en sub
x(2,2,2) en prueba=.11000000e+02=array(0,2,2) en sub
x(3,2,2) en prueba=.1200000e+02=array(1,2,2) en sub

cabe recordar que el arreglo x, dentro de la subrutina sub, es conocido con el nombre array.
Es posible establecer la forma de un arreglo, basándose en los valores de los argumentos fingidos. Si el arreglo es un argumento fingido, es llamado un arreglo de dimensiones ajustables. Si el arreglo no es un argumento fingido,

entonces es un arreglo automático. Ejemplos:

```
program paso_de_arreglos
real,dimension(10,10)::matriz
real,dimension(10)::vector
read *,m,n
read *,matriz(1:m,1:n)
read *,vector(1:n)
call impre(m,n,matriz,vector)
end program paso_de_arreglos
subroutine impre(nf,nc,x,y)
! x y y son arreglos de dimensiones ajustables
real,dimension(10,nc)::x
real,dimension(nc)::y
do i=1,nf
        print*,x(i,1:nc)
end do
print *
print *,y(1:nc)
end subroutine impre
```

```
program uno
real muestra
real,dimension(3,2)::matriz_uno
read *,matriz_uno
result=muestra(2,matriz_uno)
print *,result
end program uno
real function muestra(n,x)
! x es un arreglo de dimensiones ajustables
real,dimension(3,n)::x
! multiplica es un arreglo automático
real,dimension(n)::multiplica
multiplica=.125
muestra=dot_product(x(1,1:n),multiplica)
end function muestra
```

Un arreglo de forma supuesta (assumed shape array) es un arreglo fingido para el que se supone la forma del arreglo actual correspondiente. Para usar arreglos de forma supuesta, es obligatorio el uso de un bloque de interfase:

```
program paso_de_arreglos
implicit none
interface
```

```
        subroutine impre(nf,nc,x,y)
        implicit none
        integer nf,nc
        real,dimension(:,:)::x
        real,dimension(:)::y
        end subroutine impre
end interface
integer m,n
real,dimension(10,10)::matriz
real,dimension(10)::vector
read *,m,n
read *,matriz(1:m,1:n)
read *,vector(1:n)
call impre(m,n,matriz,vector)
end program paso_de_arreglos
subroutine impre(nf,nc,x,y)
implicit none
integer nf,nc
integer i
! x y y son arreglos de forma supuesta
real,dimension(:,:)::x
real,dimension(:)::y
do i=1,nf
        print *,x(i,1:nc)
end do
print *
print *,y(1:nc)
end subroutine impre

program xyz
interface
        subroutine forma_supuesta(x)
        integer,dimension(-1,:)::x
        end subroutine forma_supuesta
end interface
integer a
dimension a(3,4)
        .
        .
end
subroutine forma_supuesta(x)
! x se supone con la forma del argumento actual x, pero con un nuevo límite inferior
! para  la dimensión 1
integer,dimension(-1:,:)::x
        .
```

.

end

En FORTRAN 77 son permitidos los arreglos de tamaño supuesto y los arreglos de dimensiones ajustables, pero no los arreglos automáticos ni los de forma supuesta.

Ejercicio

Intercambio de dos filas de una matriz de m x n.

Pseudocódigo

```
!Este algoritmo intercambia las filas p y q de una
!matriz  A de mxn
real A[10,10]
entero m,n,p,q,i,j
lee m,n,p,q
ejecuta j=1,n
comienza
   ejecuta i=1,m    !lectura e impresión de la matriz A
     lee A[i,j]
termina
llama impre_matriz(m,n,A)
llama cambio_de_filas(n,p,q,A) !cambio de la fila p con la q
llama impre_matriz(m,n,B)
!impresión de la matriz con las filas ya cambiadas
fin
proc impre_matriz(m,n,B)
real B[10,10]
entero i,j,m,n
ejecuta i=1,m  !escritura de una matriz de m x n
comienza
   ejecuta j=1,n
     escribe B[i,j]
termina
regresa
fin
proc cambio_de_filas(ncol,p,q,x)
real x[10,10],aux
entero k,p,q,ncol    !intercambio de la fila p con la q, en
ejecuta k=1,ncol     !matriz x
comienza
   aux=x[p,k]
   x[p,k]=x[q,k]
   x[q,k]=aux
termina
regresa
```

fin

Codificación en FORTRAN 77

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486)         Page:     1
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

PROGRAM CAMBIAF , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

         1         program cambiaf
         2   c
         3   c       Este programa intercambia las filas p y q dde una matriz
         4   c       a de mxn
         5   c
         6         dimension a(10,10)
         7         integer p,q
         8         read *,m,n,p,q
         9   c
        10   c       lectura e impresion de a
        11   c
        12         read *,((a(i,j),i=1,m),j=1,n)
        13         call imprem(m,n,a)
        14   c
        15   c       cambio de la fila p con la q e impresion de la matriz
        16   c       con las filas ya cambiadas
        17   c
        18         call cambf(n,p,q,a)
        19         print *
        20         call imprem(m,n,a)
        21         stop
        22         end

F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486)         Page:     2
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

SUBROUTINE IMPREM , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

        23         subroutine imprem(m,n,b)
        24   c
        25   c       escritura de una matriz mxn
        26   c
        27         dimension b(10,*)
        28         do 10 i=1,m
        29      10 print '(1x,10(f8.2,2x))',(b(i,j),j=1,n)
        30         return
        31         end

F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486)         Page:     3
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

SUBROUTINE CAMBF , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

        32         subroutine cambf(ncol,p,q,x)
        33         integer p,q
        34         dimension x(10,ncol)
        35   c
        36   c       intercambio de la fila p con la q, en la matriz x
        37   c
        38         do 10 k=1,ncol
        39           aux=x(p,k)
        40           x(p,k)=x(q,k)
        41           x(q,k)=aux
        42      10 continue
        43         return
        44         end
```

Codificación en FORTRAN 90

```
Lahey Fortran 90 Compiler Release 3.00c                                  Sun Jul 30
16:51:37 1961  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.
```

```
PROGRAM CAMBIO_DE_FILAS    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin
-inln
                                    -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
-----------------------------------------------  Source Listing
-------------------------------------------------------------
      1  program cambio_de_filas
      2  !
      3  ! Este programa intercambia las filas p y q de una matriz
      4  ! a de mxn
      5  !
      6  interface
      7    subroutine impre_matriz(m,n,b)
      8    real,dimension(:,:)::b
      9    end subroutine impre_matriz
     10    subroutine cambia_filas(n_columnas,p,q,x)
     11    integer p,q
     12    real,dimension(:,:)::x
     13    end subroutine cambia_filas
     14  end interface
     15  real,dimension(10,10)::a
     16  integer p,q
     17  read *,m,n,p,q
     18  read *,a(1:m,1:n)   !lectura e impresion de a
     19  call impre_matriz(m,n,a)
     20  call cambia_filas(n,p,q,a) !cambio de la fila p con la q e impresion de la
     21  print *                    !matriz con las filas ya cambiadas
     22  call impre_matriz(m,n,a)
     23  stop
     24  end program cambio_de_filas
Bytes of stack required for this program unit: 164.

Lahey Fortran 90 Compiler Release 3.00c                                                  Sun Jul 30
16:51:37 1961  Page: 2
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

SUBROUTINE IMPRE_MATRIZ    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin
-inln
                                    -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
-----------------------------------------------  Source Listing
-------------------------------------------------------------
     25  subroutine impre_matriz(m,n,b)
     26  !
     27  ! Escritura de una matriz de mxn
     28  !
     29  real,dimension(:,:)::b
     30  do i=1,m
     31    print '(1x,10(f8.2,2x))',b(i,1:n)
     32  end do
     33  return
     34  end subroutine impre_matriz
Bytes of stack required for this program unit: 36.

Lahey Fortran 90 Compiler Release 3.00c                                                  Sun Jul 30
16:51:37 1961  Page: 3
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

SUBROUTINE CAMBIA_FILAS    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin
-inln
                                    -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
-----------------------------------------------  Source Listing
-------------------------------------------------------------
     35  subroutine cambia_filas(n_columnas,p,q,x)
     36  integer p,q
     37  real,dimension(:,:)::x
     38  real,dimension(n_columnas)::aux
     39  !
     40  ! Intercambio de la fila p con la q, en la matriz x
     41  !
     42  aux=x(p,1:n_columnas)
     43  x(p,1:n_columnas)=x(q,1:n_columnas)
     44  x(q,1:n_columnas)=aux
     45  end subroutine cambia_filas
Bytes of stack required for this program unit: 68.
```

Procedimientos internos

Un programa principal, subprograma o módulo puede contener otros procedimientos o módulos, los cuales pueden ser usados solamente dentro del programa, subprograma o módulo anfitrión. Tales procedimientos o módulos se dice que son internos y son especificados después de una proposición CONTAINS, la cual debe aparecer después de todas las instrucciones ejecutables del programa, procedimiento o módulo anfitrión. Ejemplos que muestran el uso de procedimientos internos:

```
program prueba
implicit none
real x,fexp
read *,x
print *,'La función exponencial de',x,' es:',fexp(x)
end program prueba
function fexp(x)
fexp=1.0
i=1
elemento=x**i/fact(i)
do while(abs(elemento)>=1e-10)
        fexp=fexp+elemento
        i=i+1
        elemento=x**i/fact(i)
end do
return
contains
        function fact(n)
        !
        ! El tipo del resultado que devuelve la función interna debe declararse
        ! dentro de ella
        !
        real fact;integer n,j                   ! fact es un subprograma interno, el cual
        fact=1.0                                ! sólo está disponible dentro del function
        do j=1,n                                ! fexp
                fact=fact*j
        end do
        return
        end function fact
end function fexp

program prueba_norma
implicit none
integer n
real longitud
real,dimension(20)::vector
```

```
read *,n,vector(1:n)
! norma_maxima es un function real
longitud=norma_maxima(n,vector)
print *,longitud
contains
        real function norma_maxima(m,vect)
        real,dimension(:)::vect
        integer m
        norma_maxima=maxval(abs(vect(1:m)))
        end function norma_maxima
end program prueba_norma

program flash
implicit none
integer nc
real phi,cociente
real,dimension(10)::k,z
read *,nc
read *,k(1:nc)
read *,z(1:nc)
read *,phi
cociente=rashford_y_rice(nc,k,z,phi)/&
                                derivada_rashford_y_rice(nc,k,z,phi)
print *,cociente
contains
        function rashford_y_rice(n,k,z,v_en_f)
        real,dimension(:)::k,z
        real rashford_y_rice,v_en_f
        integer n
        rashford_y_rice=sum((z(1:n)*(1-k(1:n)))/&
                                (1+v_en_f*(k(1:n)-1)))
        end function rashford_y_rice
        function derivada_rashford_y_rice(n,k,z,v_en_f)
        real,dimension(n)::k,z
        real derivada_rashford_y_rice,v_en_f
        integer n
        derivada_rashford_y_rice=sum((z(1:n)*(1-k(1:n))**2)/&
                                ((1+v_en_f*(k(1:n)-1))**2))
        end function derivada_rashford_y_rice
end program flash
```

Los procedimientos internos no pueden estar anidados. El siguiente es un ejemplo de código ilegal.

```
read *,a
```

```
b=func1(a)
print*,a,b
contains
        function func1(x)
        func1=x**2+func2(x)
        contains                                ! los procedimientos internos no
                function func2(y)               ! pueden estar anidados
                func2=y/10.0
                end function func2
        end func1
end
```

Un procedimiento interno tiene acceso a las entidades de su unidad anfitriona, excepto cuando estas tienen el mismo nombre que alguna de las entidades del procedimiento. Esto se conoce como asociación con el anfitrión (host association). Para ilustrar este concepto, considere el programa

```
program xxx
real::dato=3.5
z=23.5
call interno(dato,resultado)
print *,dato,resultado
! la variable w, definida como local en el subprograma interno, está
! disponible en xxx
print *,w
contains
        subroutine interno(x,y)
        w=100.0
        y=x+z+w                         !  z está disponible dentro de interno
        end subroutine                  ! (asociación con el anfitrión)
end program xxx
```

que imprime los registros

```
  3.5000  127.0000
100.0000
```

mientras que

```
program xxx
real::dato=3.5
z=23.5
call interno(dato,resultado)
print*,dato,resultado
```

```
print *,z
contains
        subroutine interno(x,y)
        real z
        z=50.0                  ! z es una variable local de la
        w=100.0                 ! subrutina interno
        y=x+z+w
        end subroutine interno
end program xxx
```

```
genera los resultados

  3.5000  153.5000
 23.5000
```

Obviamente, un procedimiento interno no puede ser pasado como argumento a otro procedimiento.
Los procedimientos internos no forman parte del estándar FORTRAN 77.

Funciones de proposición

Una función de proposición es una función definida por el usuario, en una sola línea y con una sola expresión. La función de proposición solamente puede ser llamada dentro de la unidad de programa en la cual está definida:

```
program funcion_de_proposicion
implicit none
real a,b,hipot
real var1,var2,y,z
! La función de proposición es hipot
! Los argumentos fingidos deben ser variables
hipot(a,b)=sqrt(a**2+b**2)
read *,var1,var2
! Los argumentos actuales pueden ser variables, constantes o expresiones
y=hipot(var1,var2)
z=hipot(3.5,var1-var2)
print *,var1,var2,y,z;end
end program funcion_de_proposicion
```

```
program circulo
implicit none
real (kind=8)::superficie,radio,r,area
superficie(radio)=4*atan(1.0_8)*radio**2
```

```
read *,r
area=superficie(r)
print *,r,area
end program circulo
```

Los argumentos fingidos no pueden ser arreglos. El siguiente es un programa no válido:

```
program uno
dimension vector(10),x(10)
fun(a,vector)=sum(vector)/a                    ! error
read *,n
read*,x(1:n)
s=fun(2.5,x)
print *,s
end
```

Las variables empleadas como argumentos fingidos tienen como alcance o ámbito (scope) a la función de proposición, mientras que las entidades usadas en la unidad de programa que manda llamar a la función de proposición tienen como alcance a esa misma unidad. Por ejemplo, en el código

```
program muestra
! el argumento fingido x tiene como alcance la función de proposición f
f(x,y)=x**2+y**2
      .
      .
read *,t
print *,f(2.30,t)
x=t**2/958.1
! la variable x tiene como ámbito a todo el programa muestra
print*,f(x,.002)
      .
      .
end
```

por lo tanto, la variable x en la definición de f es diferente a la x usada en el programa muestra.
En una función de proposición se puede hacer referencia a otra función de proposición previamente declarada.

```
program funciones_de_proposicion
h(x,y)=2+atan(y/x)
g(s,t)=10.15+cos(s*t)-h(s,t)
      .
```

```
        .
read *,f,g
print*,h(f,g)
        .
        .
end
```

Los argumentos pueden ser tipos derivados, como en:

```
program tipos
implicit none
type data
        integer i
        real x
end type data
type (data) f,arg
real fun
fun(f)=f%i-f%x**2
read *,arg
print *,fun(arg)
end program tipos
```

En FORTRAN 90, una función de proposición puede ser sustituida ventajosamente por una función interna. Así, el programa

```
program muestra_funcion_interna
implicit none
real var_1,var_2,y,z
read *,var_1,var_2
y=hipotenusa(var_1,var_2)
z=hipotenusa(3.5,var_1-var_2)
print *,var_1,var_2,y,z
contains
        function hipotenusa(a,b)
        real hipotenusa
        real a,b
        hipot=sqrt(a**2+b**2)
        end function hipot
end program muestra_funcion_interna
```

sustituye a

```
program muestra
implicit none
real var1,var2,hipot,y,z
```

```
hipot(a,b)=sqrt(a**2+b**2)
read *,var1,var2
y=hipot(var1,var2)
z=hipot(3.5,var1-var2)
print *,var1,var2,y,z
end
```

Funciones que devuelven como resultado un arreglo

Es posible diseñar un subprograma FUNCTION, interno o externo, que devuelva como resultado un arreglo. Si la función es externa, se requiere emplear una interfase explícita. En FORTRAN 77 una función no puede devolver un arreglo como resultado. Los siguientes dos programas ejemplifican lo anterior:

```
program arreglo
implicit none
interface
        function vector_unitario(m,vector)
        implicit none
        integer m
        real,dimension(:)::vector
        real,dimension(size(vector))::vector_unitario
        end function vector_unitario
end interface
real,dimension(10)::x,x_unitario
integer n
read *,n
read *,x(1:n)
x_unitario(1:n)=vector_unitario(n,x(1:n))
print *,x(1:n);print *,x_unitario(l:n)
end program arreglo
function vector_unitario(m,vector)
integer m
real,dimension(:)::vector
real,dimension(size(vector))::vector_unitario
vector_unitario(1:m)=vector/sqrt(sum(vector(1:m)**2))
return
end function vector_unitario

program arreglo_uno
implicit  none
integer n
```

```
real,dimension(10)::x,x_unitario
read *,n
read *,x(1:n)
x_unitario(1:n)=vector_unitario(n,x(1:n))
print *,x(1:n);print *,x_unitario(1:n)
contains
        function vector_unitario(m,vector) result(unit)        ! vector_unitario es una
                                                               ! función interna
        integer m
        real,dimension(:)::vector
        real,dimension(size(vector))::unit
        unit(1:m)=vector/sqrt(sum(vector(1:m)**2))
        return
        end function vector_unitario
end program arreglo_uno
```

Procedimientos genéricos

Una proposición INTERFACE con un nombre especifica una interfase genérica para cada uno de los procedimientos en el bloque de interfase. De esta manera se pueden crear procedimientos genéricos, análogos a los procedimientos intrínsecos genéricos:

```
program principal
implicit none
interface volumen      ! el nombre genérico es volumen
        real function real_vol(largo,ancho,alto)
        implicit none
        real largo,ancho,alto
        end function real_vol
        integer function entero_vol(largo,ancho,alto)
        implicit none
        integer largo, ancho,alto
        end function entero_vol
end interface
real x,y,z
integer i,j,k
read *,x,y,z
read *,i,j,k
! se emplea el nombre genérico volumen, con argumentos reales
print *,'largo=',x,' ancho=',y,' alto=',z,' volumen=',&
        volumen(x,y,z)
print *
```

```
! ahora, los argumentos son enteros
print *,'largo=',i,'ancho=',j,'alto=k,'volumen=',&
        volumen(i,j,k)
end program principal
real function real_vol(largo,ancho,alto)
implicit none
real largo,ancho,alto
real_vol=largo*ancho*alto
end function real_vol
integer function entero_vol(largo,ancho,alto)
implicit none
integer largo,ancho,alto
entero_vol=largo*ancho*alto
end function entero_vol

program funciones_genericas
implicit none
interface calculo_de_area
        subroutine area_de_un_rectangulo(algun_rectangulo,area)
        implicit none
        type rectangulo
                real base
                real altura
        end type rectangulo
        type(rectagulo)::algun_rectangulo
        real area
        end subroutine area_de_un_rectangulo
        subroutine area_de_un_circulo(algun_circulo,area)
        implicit none
        type circulo
                real radio
        end type circulo
        type(circulo) algun_circulo
        real area
        end subroutine area_de_un_circulo
end interface
type rectangulo
        real base
        real altura
end type rectangulo
type circulo
        real radio
end type circulo
type(rectangulo)::caja
type(circulo)::rueda
```

```
real superficie
caja=rectangulo(3.8,5.2)
rueda=circulo(1.67)
! cálculo del área del rectángulo caja
call calculo_de_area(caja,superficie)
print *,caja,superficie
! cálculo del área del círculo rueda
call calculo_de_area(rueda,superficie)
print *,rueda,superficie
end
subroutine area_de_un_rectangulo(algun_rectangulo,area)
implicit none
type rectangulo
        real base
        real altura
end type rectangulo
type(rectangulo) algun_rectangulo
real area
area=algun_rectangulo%base*algun_rectangulo%altura
end subroutine area_de_un_rectangulo
subroutine area_de_un_circulo(algun_circulo,area)
implicit none
type circulo
        real radio
end type circulo
type(circulo)::algun_circulo
real area
area=4*atan(1.0)*algun_circulo%radio**2
end subroutine area_de_un_circulo
```

Los procedimientos genéricos no forman parte del estándar FORTRAN 77.

Recursión

La recursión es la capacidad que tiene un procedimiento de llamarse a sí mismo. La recursión puede ser directa o indirecta. La recursión directa o simple es cuando un procedimiento se llama así mismo. La recursión indirecta sucede cuando un procedimiento manda llamar a otro que a su vez llama al primero de ellos. La palabra clave RECURSIVE debe anteceder a FUNCTION o SUBROUTINE para indicar que el procedimiento puede llamarse a sí mismo. Ejemplos:

```
program principal
```

```
implicit none
real integral,z,fun_ext
external fun_ext
! aproximación de la integral ∫₀¹ ∫₀² x²ydydx
z=integral(0.0,1.0,20,fun_ext)
print*, 'La integral doble es:',z
end program principal
recursive real function integral(a,b,n_int,f)
implicit none
! es necesario emplear la palabra clave recursive
real,intent(in)::a,b
integer,intent(in)::n_int
real,external::f
real h
integer i
! aproximación de la integral de f, entre límites a y b con la fórmula compuesta
! de los trapecios
integral=f(a)+f(b)
h=(b-a)/n_int
do i=1,n_int-1
        integral=integral+2*f(a+i*h)
end do
integral=integral*h/2.0
end function integral
function fun_ext(x)
implicit none
real integral,fun_ext,fun_int,x
external fun_int
fun_ext=x**2*integral(0.,2.0,20,fun_int)                ! recursividad
end function fun_ext
function fun_int(y)
implicit none
real fun_int,y
fun_int=y
end function fun_int

program ortogonal
implicit none
integer n
real x,P,Polinomio_Legendre
read *,n,x
Polinomio_Legendre=P(n,x)
print *,'El polinomio de Legendre de orden',n,&
        ' evaluado en x=',x," es:",Polinomio_Legendre
end program ortogonal
```

```
recursive function P(n,x) result(polinomio)          ! En el caso de la recursión directa, es
                                                     ! necesario usar la opción RESULT
implicit none
integer n
real x,polinomio
select case(n)
case(0)
        polinomio=1.0
case(1)
        polinomio=x
case(2)
        polinomio=((2*n-1)*x*P(n-1,x)-(n-1)*P(n-2,x))/n
end select                                           ! recursión directa
end function P
```

El encabezado

```
real (kind=8) recursive function prueba(angulo,terminos)
```

es equivalente a

```
recursive real (kind=8) function prueba(angulo,terminos)
```

La recursión también puede emplearse en subrutinas

```
program principal
        .
        .
call muestra(a,b,x)
        .
        .
end program principal
recursive subroutine muestra(a,b,c)
        .
        .
end subroutine muestra

program sub_recursiva
implicit none
integer::n,suma,producto;read *,n
call sub(n,suma,producto)
print *n,suma,producto
end program sub_recursiva
recursive subroutine sub(m,isuma,imult)              ! cálculo de isuma=1+2+3+...+m y de
                                                     ! imult=(1)(2)(3)...(m)
implicit none
```

```
integer isuma,imult,iresult_suma,iresult_mult
if(m=<0)then
        imult=1;isuma=0
else
        call sub(m-1,iresult_suma,iresult_mult)
        isuma=m+iresult_suma
        imult=m*iresult_mult
endif
end subroutine sub
```

MÓDULOS

Subprograma MODULE

Un módulo es una unidad de programa que permite "empaquetar" información como:
1)datos globales (reemplazando al COMMON y al BLOCK DATA de FORTRAN 77)
2)definiciones de tipos
3)subprogramas
4)bloques de interfase
5)grupos de NAMELIST.
Los módulos deben aparecer antes de cualquier otra unidad de programa en un archivo fuente, o en su defecto, ser compilados antes que las demás unidades de programa.
Para que la información contenida en un módulo esté disponible en una unidad de programa, se emplea la instrucción USE seguida por el nombre del módulo. El proceso de tener acceso a la información de un módulo se conoce como asociación por uso (use association).
El programa mostrado a continuación usa un módulo denominado ejemplo

```
module ejemplo                          ! módulo ejemplo
        implicit none
        integer,parameter::clase_real=selected_real_kind(6,35)
        real (kind=clase_real),parameter::e=2.718282,&
                pi=3.141592
        real (kind=clase_real),dimension(5)::&
                vector=(/(k,k=1,9,2)/)
end
program prueba modulo_ejemplo
use ejemplo                             ! se indica el uso del módulo ejemplo
print *,e,pi
print *,10*vector
end
```

genera los resultados:

```
 2.7182     3.1415
10.0000    30.0000  50.0000   70.0000   90.0000
```

Ejemplos adicionales de programas que emplean módulos:

```
module capacidad_calorifica
        type Cp
```

```
                real (kind=8)::a,b,c            ! definición del tipo Cp
        end type Cp
        ! constantes_de_Cp  es un vector de tipo Cp
        type (Cp),dimension(5)::constantes_de_Cp
end module capacidad calorifica
! con use capacidad_calorifica será posible emeplear la información contenida en el
! módulo capacidad_calorifica
use capacidad_calorifica
read *,T
do i=1,5
        read(*,*)constantes_de_Cp(i)
        Cp_gas_ideal=constantes_de_Cp(i)%a+&
                        constantes_de_Cp(i)%b*T+&
                constantes_de_Cp(i)%c*T**2
        print *,i,Cp_gas_ideal
end do
end

module evaluacion_de_polimonio
        interface
                subroutine horner(n,coef,x,P,derivada_de_P)
                implicit none
                integer,intent(in)::n
                ! coef es un arreglo de forma supuesta
                real,dimension(0:),intent(in)::coef
                real,intent(in)::x
                real,intent(out)::P
                real,intent(out),optional::derivada_de_P
                end subroutine horner
        end interface
end module evaluacion_de_polinomio
program prueba_ de_modulo
! Evaluación de un polinomio con la regla de Horner.
! La interfase de horner está definida en el módulo evaluacion_de_polinomio.
use evaluacion_de_polinomio
implicit none                   ! el implicit se coloca después del use
integer n
real,dimension(0:10)::a
real x0,pol,pol_der
read *,n
read *,a(n:0:-1)
print *,a(n:0:-1)
read *,x0
call horner(n,a,x0,pol,der_ pol)
print *,pol,der_pol
```

```
end program prueba_de_modulo
subroutine horner(n,coef,x,P,derivada_de_P)
implicit none
integer,intent(in)::n
real,dimension(0:),intent(in)::coef
real,intent(in)::x
real,intent(out)::P
real,intent(out),optional::derivada_de_ P       ! derivada_de_P es un resultado opcional
integer j
P=coef(n)
if(present(derivada_de_P))derivada_de_P=coef(n)
do j=n-1,1,-1
        P=x*P+coef(j)
        if(present(derivada_de_P))derivada_de_P=&
                        x*derivada_de_P+P
end do
P=x*P+coef(0)
end subroutine horner

module bloque
        implicit none
        real a,b
        integer j
        namelist /datos/a,b,j
end module bloque
program uno
use bloque                                      ! bloque contiene el namelist datos
read datos
print datos
print *,a*b,12.3+j
end program uno
```

Un módulo puede emplear a otro previamente definido:

```
module uno
        implicit none
        integer,parameter::clase=selected_real_kind(15,300)
        real (kind=clase),parameter::a=12.5_clase,&
                                b=25.8_clase,c=48.4_clase
end module uno
module dos
        use uno                                 ! el módulo dos usa al módulo uno
        real (kind=clase),dimension(3)::vector=(/a,b,c/)
end module dos
program principal
```

```
use dos                                    ! el programa principal usa al módulo dos
print *,vector
end program principal
```

FORTRAN 77 no permite el uso de módulos.

Procedimientos de módulo

Los procedimientos que son definidos dentro de un módulo se conocen como procedimientos de módulo (module procedures) y son escritos después de un CONTAINS:

```
module conversiones
        real(kind=8)::pi=3.1415926535_8
        contains
        ! cartesianas_cilindricas y cartesianas_esfericas son procedimientos
        ! de módulo
        subroutine cartesianas_cilindricas(x,y,z,r,teta,z_cil)
        implicit none
        real(kind=8)::x,y,z,r,teta,z_cil
        r=sqrt(x**2+y**2)
        teta=atan(y/x)
        z_cil=z
        end subroutine cartesianas_cilindricas
        subroutine cartesianas_esfericas(x,y,z,r,phi,teta)
        implicit none
        real (kind=8)::x,y,z,r,phi,teta
        r=sqrt(x**2+y**2+z**2)
        phi=atan(y/x)
        teta=acos(z/sqrt(x**2+y**2+z**2))
        end subroutine cartesianas_esfericas
end module conversiones
program convierte_coordenadas
use conversiones
implicit none
real(kind=8)::x,y,z,r,teta,zc,phi
print *,pi
read *,x,y,z
call cartesianas_cilindricas(x,y,z,r,teta,zc)
print '(6x,3(g12.5))',r,teta,zc
call cartesianas_esfericas(x,y,z,r,phi,teta)
print '(6x,3(g12.5))',r,phi,teta
end program convierte_coordenadas
```

Los procedimientos de módulo siguen las mismas reglas que

los subprogramas externos. Sin embargo, también tienen acceso a la información del módulo que los contiene (asociación con el anfitrión). Las unidades de programa que usan al módulo anfitrión de los procedimientos de módulo son las únicas que tienen acceso a estos.

Módulos e interfases genéricas

Un módulo puede contener también interfases genéricas. En este caso; será necesario emplear la proposición MODULE PROCEDURE para indicar los nombres que son parte de dicha interfase genérica y que corresponden a procedimientos de módulo especificados en el mismo módulo.

```
module ecuacion_cuadratica
        interface cuadratica
                module procedure cuadratica_4,cuadratica_8
        end interface
        contains
        subroutine cuadratica_4(a,b,c,raices_complejas,x1,x2)
        implicti none
        real(kind=4)::a,b,c,x1,x2,disc
        logical raices_complejas
        disc=b**2-4*a*c
        if(disc<0.0_4)then
                raices_complejas=.true.
        else
                raices_complejas=.false.
                x1=(-b+sqrt(disc))/(2*a)
                x2=(-b-sqrt(disc))/(2*a)
        end if
        end subroutine cuadratica_4
        subroutine cuadratica_8(a,b,c,raices complejas,x1,x2)
        implicit none
        real(kind=8)::a,b,c,x1,x2,disc
        logical raices_complejas
        disc=b**2-4*a*c
        if(disc<0.0_8)then
                raices complejas=.true.
        else
                raices complejas=.false.
                x1=(-b+sqrt(disc))/(2*a)
                x2=(-b-sqrt(disc))/(2*a)
        endif
        end subroutine cuadratica_8
```

```
end module ecuacion_cuadratica
program princip
use ecuacion_cuadratica                    ! uso del módulo ecuacion_cuadratica
implicit none
real a,b,c,x1,x2
double precision x1doble,x2doble
logical complejas
read *,a,b,c,
print *,'a=',a,' b=',b,' c=',c
! aritmética en precisión simple (real(kind=4))
call cuadratica(a,b,c,complejas,x1,x2)
if(complejas)then
        print *,'Raices complejas'
else
        print *,'x1=',x1,' x2=',x2
endif
print *,'a=',3.0_8,' b=',12.0_8,' c=',-1.0_8
! aritmética en precisión doble (real (kind=8))
call cuadratica(3.0_8,12.0_8,-1.0_8,complejas,x1doble,x2doble)
if(complejas)then
        print *,'Raices complejas'
else
        print *,'x1=,x1doble,' x2=',x2doble
endif
end program princip
```

Opciones avanzadas de la instrucción USE

Los nombres de las entidades de un módulo pueden ser diferentes en las unidades de programa que lo usan:

use nombre del módulo,nombre en la unidad de programa=>nombre en el módulo

Loa siguientes son ejemplos del uso de nombres diferentes a los especificados en el módulo:

```
module datos
        real::a=12.3,b=-.001
        character (len=10)::escuela='ESIQIE'
end module datos
program abc
! la variable a, contenida en el módulo datos, será conocida como x en el programa abc
! la variable escuela, definida en el módulo datos, será conocida como nombre en abc
use datos,x=>a,nombre=>escuela
```

```
a=100.0
print *,x,b
print *,nombre
end program abc

module ejemplo
        real,parameter::pi=3.141592
        contains
        subroutine cilindro(radio,altura,area,volumen)
        area_base=pi*radio**2
        area=2*area_base+2*pi*radio*altura
        volumen=area_base*altura
        end subroutine cilindro
end module ejemplo
program prueba
! en prueba, la subrutina cilindro es conocida como sub
use ejemplo,sub=>cilindro
read *,r,h
call sub(r,h,area,vol)
print *,r,h,area,vol
end
```

Acceso restringido al contenido de un módulo

En una unidad de programa, se puede indicar con ONLY que solamente se tendrá acceso a cierta información del módulo:

```
module datos
        real::a=12.3,b=-0.001
        character (len=10)::escuela='ESIQIE'
end module datos
program abc
!
! abc solamente tiene acceso a las variables a y b del módulo datos
use datos,only:a,g=>b
print *,a,g
end program abc
```

La proposición PUBLIC especifica que los nombres de las entidades de un módulo están disponibles en las unidades de programa donde es empleado dicho módulo. PUBLIC es la condición por "default" para un módulo. Por otra parte, la proposición PRIVATE indica que los nombres de las entidades de un módulo sólo están disponibles dentro de ese mismo módulo.

```
module prueba de acceso
        ! la proposición PUBLIC, sin una lista de variables especifica la condición
        ! por "default" para el módulo
        public
        real::a=3.0
        integer::l=100
        contains
        function fun(x)
        fun=x**2+a+l                    ! asociación con el anfitrión
        end function fun
end module prueba_de_acceso
program uno
use prueba_de_acceso
read *,w
print *,fun(w)
print *,a,l
end program uno

module prueba_de_acceso
        ! ahora, la condición por default es PRIVATE
        private
        real::a=3.0
        integer::len=100
        ! la función fun es declarada como de acceso público
        public::fun
        ! la variable a es especificada como de acceso público
        public a
        ! len continúa con acceso privado
        contains
        function fun(x)
        fun=x**2+a+l
        end function fun
end module prueba_de_acceso
program uno
use prueba_de_acceso
read *,w
print *,fun(w)
print *,a                               ! len no es conocida dentro del programa uno
end program uno
```

Tanto PUBLIC como PRIVATE también son atributos en las declaraciones de tipo empleadas en el módulo.

```
module prueba_de_acceso
        private
```

```
        real,public::a=3.0
        public fun
        integer::l=100
        contains fun(x)
        fun=x**2+a+l
        end function fun
end module prueba_de_acceso
program uno
use prueba_de_acceso
read *,w
print *,fun(w)
print *,a
end program uno
```

Los componentes de un tipo derivado declarados en un módulo quedan inaccesibles para otras unidades de programa fuera del módulo al incluir la proposición PRIVATE en la definición del tipo derivado:

```
module prueba
        type estruc
                private
                real::x,y
        end type estruct
                .
                .
end module prueba
program abc
use prueba
type (estruct)::variable_1,variable_2
! no se puede tener acceso a los componentes del tipo derivado estruct, aunque
! si es posible definir variables de ese tipo derivado
!
        .
        .
end program abc
```

Un tipo derivado completo puede ser ocultado en cualquier otra unidad de programa que use al módulo:

```
module prueba
        type,private::estruct
                real:.x,y
        end type estruct
                .
                .
```

```
end module prueba
program abc
use prueba
!el tipo estruct no está disponible en abc
!
        .
        .
end program abc
```

En el último caso, el tipo derivado sólo puede ser empleado en operaciones dentro del módulo.

Proposición COMMON

La instrucción COMMON (declarativa) permite que diferentes variables en distintas unidades de programa compartan físicamente las mismas localidades de memoria. El uso de la instrucción COMMON permite definir una área de memoria común, llamada bloque COMMON, que es compartida por cualquier unidad de programa que contenga la instrucción COMMON adecuada:
Por ejemplo, en el programa

```
program uno
! el siguiente COMMON indica que tanto las variables k y a !como el arreglo y
! forman parte de la área común entre unidades de programa
common k,a,y(2,2)
        .
        .
call test
abc=fun(w)
        .
        .
end
subroutine test
common j,a,s(2,2)
        .
        .
end
function fun (h)
common n,b,w(2,2)
        .
        .
```

end

el área de memoria común es organizada como lo indica el siguiente esquema:

programa uno	subrutina test	función fun
k	j	n
a	a	b
y(1,1)	s(1,1)	w(1,1)
y(2,1)	s(2,1)	w(2,1)
y(1,2)	s(1,2)	w(1,2)
y(2,2)	s(2,2)	w(2,2)

En el programa mostrado a continuación:

```
program principal
common x,y,a(3)
      .
      .
call uno
      .
      .
end
subroutine uno
common a,b
      .
      .
end
```

en la subrutina uno solamente están disponibles las dos primeras variables del bloque COMMON:

programa principal	subrutina uno
x	a
y	b
a(1)	
a(2)	
a(3)	

mientras que en el siguiente ejemplo, la variable z en la subrutina uno tiene un valor indefinido:

```
program principal
common x,y,a(3)
      .
      .
call uno
      .
      .
end
subroutine uno
common a,b,w(3),z
      .
      .
end
```

programa principal	subrutina uno
x	a
y	b
a(1)	w(1)
a(2)	w(2)
a(3)	w(3)
	z?

Las declaraciones

```
dimension array(20,10)
common array
```

equivalen a

```
common array(20,10)
```

Aunque se escriban varias declaraciones COMMON, el área común es única, y por lo tanto las instrucciones

```
common a,b,c,min,max
common nombre,s,x,y
```

tienen el mismo efecto que

```
common a,b,c,min,max,s,x,y
```

Además de la área común antes descrita, es posible declarar otras áreas comunes diferentes. A cada una de estas áreas comunes adicionales le es asignado un nombre (COMMON etiquetado):

```
program main
common x,w,presion                    ! common en blanco o sin etiqueta
common /uno/z,h,arco
common /dos/j,k,solve                 ! uno y dos son los nombres de los
      .                               ! bloques COMMON
      .
end
subroutine t1
common a,b,t
common /uno/z,h,f
      .
      .
end
subroutine t2
common /dos/l,m
common /uno/x,y,z
      .
      .
end
```

área común sin nombre

programa main	subrutina t1
x	a
w	b
presion	t

área común uno

programa main	subrutina t1	subrutina t2
z	z	x
h	h	y
arco	f	z

área común dos

program main	subrutina t2
j	l
k	m
solve	

En una instrucción COMMON se pueden indicar más de un bloque COMMON:

```
! dos bloques etiquetados: abc y bloque
common /abc/f,g,x(10)/bloque/w,temp h

! equivale a common x,y,z
common //x,y,z
```

Las variables en el programa principal y en los subprogramas están relacionadas por su posición relativa en el bloque COMMON. Esta técnica se conoce como asociación por memoria (storage association) debido a que las variables son asociadas al compartir una localidad de memoria.
En FORTRAN 90, una alternativa a los bloques COMMON es el uso de módulos. La ventaja está en que el acceso a la información contenida en un módulo es más selectiva y no es posicional, como en el caso de los bloques COMMON. A manera de ilustración, el programa

```
module datos_compartidos
        real::a,b,c
        integer::n
        real,dimension(2,2)::arreglo
end module datos_compartidos
program principal
use datos_compartidos,only:x=>arreglo,a,n
x=reshape((/1,6,8,-5/),(/2,2/))
a=45.6;n=10
print *,x
call sub
end program principal
use datos_compartidos,only:a,m=>n
subroutine sub
print *,a/m
end subroutine sub
```

puede sustituir a este otro:

```
program principal
dimension x(2,2)
common a,b,c,x,n
a=45.6;n=10
x=reshape((/1,6,8,-5/),(/2,2/))
print *,x
call sub
end program principal
subroutine sub
dimension arreglo(2,2)
common a,b,c,arreglo,m
print *,a/m
end subroutine sub
```

Los arreglos ALLOCATABLE no son permitidos en un bloque COMMON. Sin embargo, en FORTRAN 90 es posible darle la vuelta a esta restricción con el uso de módulos

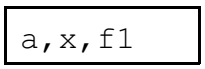
```
module arreglo_dinamico
        integer,allocatable::matriz(:,:)        ! matriz es un arreglo con atrubuto
end module arreglo_dinamico                     ! con atributo ALLOCATABLE
program xxx
use arreglo_dinamico                            ! matriz está disponible en xxx
read *,n_filas,n_columnas
allocate(matriz(n_filas,n_columnas))
read *,matriz
do i=1,n filas
        print *,matriz(i,1:n_columnas)
end do
deallocate(matriz)
end program xxx
```

Proposición EQUIVALENCE

La proposición EQUIVALENCE (declarativa) permite que diferentes variables compartan las mismas localidades de memoria dentro de una unidad de programa. Por ejemplo, en

```
equivalence (a,x,f1)
```

se declara que una localidad de memoria es conocida como a,x o f1.



De manera similar, la instrucción

```
equivalence (i,j,k),(w,h,uno)
```

indica que los nombres i,j o k apuntan a un misma localidad de memoria y que w,h o uno hacen referencia a otra localidad de memoria distinta de la primera.
Es posible emplear elementos de un arreglo dentro de un EQUIVALENCE:

```
dimension x(3)
equivalence (a,b,x(1)),(z,x(2))
```

a,b,x(1)
z,x(2)
x(3)

En este caso es conveniente recordar que los elementos de un arreglo se almacenan en localidades de memoria contiguas. Las variables señaladas como equivalentes deben ser del mismo tipo y clase:

```
equivalence (a,n)
a=17.5
print *,n                    ! n tendrá un valor imposible de predecir
      .
      .
end
```

Las variables incluidas en un EQUIVALENCE puede ser argumentos actuales:

```
program equivalencias
equivalence (a,a1),(b,b1)
      .
      .
call sub(a,b1)
      .
      .
end program equivalencias
subroutine sub(x,z)
      .
      .
end subroutine sub
```

pero no argumentos fingidos:

```
subroutine eqp(a,b,c)  !error:
equivalence (a,x),(b,f)                    ! los argumentos fingidos no pueden
:                                          ! aparecer en un EQUIVALENCE
end
```

El siguiente programa muestra como el abuso del EQUIVALENCE puede producir un código poco claro:

```
dimension a(8),b(3,2),c(2,2)
equivalence(a(3),b(1,2),c(2,1))
de i=1,10
        a(i)=10*i
end do
print *,c(2,2),a(7),b(2,1)
end
```

b(1,1)
b(2,1),a(1)
c(1,1),b(3,1),a(2)
c(2,1),b(1,2),a(3)
c(1,2),b(2,2),a(4)
c(2,2),b(3,2),a(5)
a(6)
a(7)
a(8)

Los resultados impresos por el programa anterior son:

```
50.0000   70.0000   10.0000
```

Una variables no puede aparecer más de una vez en un EQIVALENCE:

```
dimension a(3)
equivalence (j,k,n),(a(1),f),(a(2),f)                    ! error
```

Una proposición EQUIVALENCE establece un sinónimo para una variable. Ambos nombres, pueden ser usados indistintamente.

```
program muestra
```

```
equivalence (uno,dos)
read *,uno
y=17.5+dos
print *,uno,y
end program muestra
```

En las versiones anteriores a FORTRAN 90, el EQUIVALENCE es una forma de reutilizar la memoria asignada a los arreglos. Dado que FORTRAN 90 ya permite un manejo dinámico de la memoria, entonces ahora se recomienda emplear variables dinámicas en vez del EQUIVALENCE.

Proposición DATA

La proposición DATA proporciona valores iniciales para variables y arreglos dentro de una unidad de programa. DATA es una instrucción no ejecutable y puede colocarse en cualquier parte del programa, después de las declaraciones de tipo correspondientes a las variables que intervienen en ella. Ejemplos de instrucciones DATA:

```
integer f
dimension x(2)
y=50
print *,y
data f,x(1)/20.,30.15/
print *,f,x(1)
end

real x;integer min
data x,min/50,14.83/
! x tendrá el valor 50.0 y min 14 (truncamiento de decimales)

complex modulo_a
data modulo_a/17.0/
! modulo_a valdrá 17+0i

! un DATA puede dar valores a un arreglo
dimension vec(4)
data vec/3.18,-4.78,1.0e-3,5.7/

! vec(4) queda indefinido
dimension vec(4)
data vec/3,18,-4.78,1e-3/
```

```
dimension vec(4)
data vec/3.18,-4.18,2.13e-3,5.7,45.9/            ! error:sobra un valor numérico

real,dimension(5)::a
integer,dimension(3,3)::mat
data(a(i),i=1,5)/10.0,20.0,30.0,40.0,50.0/
!asignación de valores, columna por columna
data((mat(i,k),i=1,2),k=1,2)/15,7,-3,4/

dimension vector(10)
data vector/5*3.0,7.8,4*.01/                     ! 5*3.0 equivale a 3.0,3.0,3.0,3.0,3.0

! a los 100 elementos de y_nuevo se les asigna el valor 0.0,columna por columna
dimension y_nuevo(10,10)
data y_nuevo/100*0.0/

! variables de distintos tipos
character * 50 direccion
logical test
complex abc
data direccion,test,abc/'EDIFICIO 8 TERCER PISO',& .true.,(3.5,-4.1)/

! DATA con tipos derivados
type estructura
        integer::n
        real dimension(3)::w
end type estructura
type (estructura)::variable
data variable/estructura(12,(/1.2,-6.7,4.9/))/

! la única forma de asignar valores binarios, octales o hexadecimales a
! variables enteras, es a través de un DATA
data i,j,k/b'010101',o'77',z'ff'/
print *,i
print *,j
print *,k

! otra forma de asignar valores a variables
data x,g/3.7,-.003/,m,f/50,3.1415/

Los siguientes son ejemplos de instrucciones DATA no
permitidas:

! no es posible dar valores iniciales a un arreglo ALLOCATABLE con un DATA
integer,parameter::n=30
```

```
real (kind=8),allocatable,dimension(:)::array
allocate(array(n))                          ! error
data array/n*15.0/

! los argumentos fingidos no pueden ser incluidos en un DATA
subroutine sub(a,b,c,d)
data a,b/3.7,-100.7/                        ! error
        .
        .
end subroutine sub
```

No se permite proporcionar valores iniciales con un DATA a las variables de un bloque COMMON :

```
program uno
common /global/l,var(10)
data l,var/50,10*3.141592/                  ! error
        .
        .
call sub1(x)
        .
        .
end program uno
subroutine sub1(y)
common /global/j,var(10)
        .
        .
end
```

En FORTRAN 90, una alternativa al uso del DATA es dar valores iniciales directamente en las declaraciones de tipo:

```
integer::variable_uno=34,k=-56
real (kind=8),dimension(10,10)::f=0.0_8
complex::argand=(2.0,1.0)
real,dimension(2,2)::mat=reshape((/1.0,-4.1,0.9,2.7/),&
        (/2,2/))
```

Subprograma BLOCK DATA

Es una unidad de programa que se emplea para proporcionar valores iniciales a las variables y/o arreglos de un bloque COMMON etiquetado, a través de una proposición DATA. A las variables en un COMMON blanco no es factible darles valores

con la proposición DATA, ni siquiera en un BLOCK DATA.

```
program uno
common /abc/x,y,j,d(2,3)
      .
      .
call test
      .
      .
end
subroutine test
common /abc/f,g,n,w(2,3)
      .
      .
end
block data
common /abc/e,g,m,x(2,3)
! se proporcionan valores iniciales a variables y arreglos incluidos en un
! bloque COMMON etiquetado
data e,g/15.17,-4.87e-5/
data x/6*100.0/
end
```

Un subprograma BLOCK DATA puede tener un nombre:

```
block data base_de_datos
      .
      .
end block data base_de_datos
```

Un BLOCK DATA está constituida solamente por instrucciones de especificación, diferentes a un ALLOCATABLE, INTENT, PUBLIC, PRIVATE, OPTIONAL y SEQUENCE.
Dentro de un BLOCK DATA solo se permite dar valores iniciales a variables y/o arreglos incluidos en bloques COMMON etiquetados:

```
block data uno
integer x
dimensión array(10,0:2)
common /b/x,array
data x,array/50,30*800.0_4/   !error: abc no está especificada en algún COMMON
data abc/17.8/
end
```

En FORTRAN 90, un reemplazo adecuado para el BLOCK DATA es el subprograma MODULE. Por ejemplo, el programa

```
module datos
        real::a=24.5,b=67.8
        real,dimension(5)::vector=(/1,4,0,8,3/)
        real,dimension(2,2)::matriz=&
                        reshape((/3,8,-12,7/),(/2,2/))
end module datos
program prueba_datos
use datos               !uso del módulo datos
real,dimension(2)::z
print *,a+b
z=matmul(matriz,vector(1:2))
print *,z
end
```

sustituye ventajosamente a:

```
program prueba_datos
real vector(5), matriz(2,2)
dimension z(2)
common /datos/a,b,vector,matriz
print *,a+b
z=matmul(matriz,vector(1:2))
print *,z
end
block data banco_de_datos
! el block data es necesario para proporcionar valores iniciales a las variables y arreglos
! del COMMON /datos/
real vector(5),matriz(2,2)
common /datos/a,b,vector,matriz
data a,b/24.5,-67.8/
data vector/1,4,0,8,3/
data matriz/3,8,-12,7/
end block data banco_de_datos
```

Ejercicio

Cálculo de la presión de burbuja que ejerce una solución líquida de nc componentes a una temperatura Temp.

$$Pt = \sum_{i=1}^{nc} x_i \, P_i^{sat} = \sum_{i=1}^{nc} x_i \, e^{A_i + B_i/(Temp + C_i)}$$

Pseudocódigo

```
!Cálculo de la presión total de una mezcla gaseosa en
!equilibrio con una solución líquida
real Pt,x[5],Temp
entero i,nc
global {entero indices[5]}
lee nc
ejecuta i=1,nc
     lee x[i]   !lectura de las composiciones x de la
                !fase líquida
lee Temp        !lectura de la temperatura absoluta Temp
ejecuta i=1,nc
lee indices(i)
Pt=Raoult(nc,x,Temp)
escribe 'La presión total es:',Pt
fin
función Raoult(nc,x,Temp)
!Cálculo de la presión total, de acuerdo a la
!ley de Raoult
real Raoult,Temp,x[5],Psat
entero j,nc
global {real A[5],real B[5],real C[5],entero ind[5]}
Raoult=0            !ley de Raoult
ejecuta j=1,nc
empieza
     !A,B y C son arreglos contenidos en un banco de datos
     Psat=exp(A[ind[j]]+B[ind[j]]/(Temp+C[ind(j]])
     Raoult=x(j)*Psat
termina
regresa
fin
```

Codificación en FORTRAN 77

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486) Wed Apr 22 23:36:55 1998       Page:
1
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

PROGRAM PTOTAL , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

        1       program ptotal
        2   c
        3   c       Calculo de la presion total de una mezcla gaseosa en
        4   c       equilibrio con una solucion liquida
        5   c
```

```
         6         dimension x(5)
         7         common indics(5)
         8         read *,nc
         9   c
        10   c       lectura de las composiciones x de la fase liquida
        11   c       lectura de la temperatura absoluta Temp
        12   c
        13         read *,(x(i),i=1,nc)
        14         read *,Temp
        15         read *,(indics(i),i=1,nc)
        16         Pt=Raoult(nc,x,Temp)
        17         print *,'La presion total es:',Pt
        18         end

F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486) Wed Apr 22 23:36:55 1998        Page:
2
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

FUNCTION RAOULT , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

        19         function Raoult(nc,x,Temp)
        20   c
        21   c       calculo de la presion total, de acuerdo a la ley de Raoult
        22   c
        23         dimension x(5)
        24         common ind(5)
        25         common /prop/A(10),B(10),C(10)
        26         Raoult=0.0
        27         do 10 j=1,nc
        28          Psat=exp(A(ind(j))+B(ind(j))/(Temp+c(ind(j))))
        29        10 Raoult=Raoult+x(j)*Psat
        30         return
        31         end

F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486) Wed Apr 22 23:36:55 1998        Page:
3
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

BLOCKDATA _blcdat_ , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

        32         block data
        33   c
        34   c       Banco de datos con losvalores de las constantes A,B y C
        35   c       para 10 compuestos
        36   c
        37         common /prop/A(10),B(10),C(10)
        38         data A/3.15,2.78,1.97,2.11,3.45,2.17,1.11,1.87,2.97,3.02/
        39         data B/1217.,1315.,1198.,1023.1,1134.01,1227.82,1297.5,1183.4,
        40        1       1207.3,1150.9/
        41         data C/60.1,52.78,14.27,33.94,45.1,17.82,14.95,30.1,25.3,48.2/
        42         end
```

Codificación en FORTRAN 90

```
Lahey Fortran 90 Compiler Release 3.00c                                                  Wed Apr 22
23:53:21 1998  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

MODULE PROPIEDADES    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin
-inln
                                    -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
-------------------------------------------------------------
      1  module propiedades
      2  !
      3  !  Banco de datos con losvalores de las constantes A,B y C
      4  !  para 10 compuestos
      5  !
      6    real,dimension(10)::A=(/3.15,2.78,1.97,2.11,3.45,2.17,1.11,1.87,&
      7                            2.97,3.02/)
      8    real,dimension(10)::B=(/1217.,1315.,1198.,1023.1,1134.01,1227.82,&
      9                            1297.5,1183.4,1207.3,1150.9/)
     10    real,dimension(10)::C=(/60.1,52.78,14.27,33.94,45.1,17.82,14.95,&
     11                            30.1,25.3,48.2/)
     12  end module propiedades
```

```
Lahey Fortran 90 Compiler Release 3.00c                                    Wed Apr 22
23:53:21 1998  Page: 2
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

MODULE INDICES_DE_COMPUESTOS    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed
-nin    -inln
                                   -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
-------------------------------------------------------------
     13  module indices_de_compuestos
     14    integer,dimension(5)::indices
     15  end module indices_de_compuestos

Lahey Fortran 90 Compiler Release 3.00c                                    Wed Apr 22
23:53:21 1998  Page: 3
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

PROGRAM PRESION_TOTAL_MEZCLA_GAS    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed
 -nin    -inln
                                   -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
-------------------------------------------------------------
     16  program presion_total_mezcla_gas
     17  !
     18  !  Calculo de la presion total de una mezcla gaseosa en
     19  !  equilibrio con una solucion liquida
     20  !
     21  use indices_de_compuestos
     22  dimension x(5)
     23  read *,nc
     24  read *,x(1:nc)    !lectura de las composiciones x de la fase liquida
     25  read *,Temp       !lectura de la temrperaura absoluta temp
     26  read *,(indices(i),i=1,nc)
     27  Presion_total=Raoult(nc,x,Temp)
     28  print *,'La presion total es:',Presion_total
     29  end program presion_total_mezcla_gas
Bytes of stack required for this program unit: 28.

Lahey Fortran 90 Compiler Release 3.00c                                    Wed Apr 22
23:53:21 1998  Page: 4
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

FUNCTION RAOULT    Compiling Options:  -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin    -inln

                                   -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
-------------------------------------------------------------
     30  function Raoult(nc,x,Temp)
     31  !
     32  !  Calculo de la presion total, de acuerdo a la ley de Raoult
     33  !
     34  use indices_de_compuestos,ind=>indices
     35  use propiedades
     36  dimension x(nc)
     37  Raoult=0.0
     38  do j=1,nc
     39    Psat=exp(A(ind(j))+B(ind(j))/(Temp+c(ind(j))))
     40    Raoult=Raoult+x(j)*Psat
     41  end do
     42  return
     43  end function Raoult
Bytes of stack required for this program unit: 24.
```

ARCHIVOS

Archivos internos

Un archivo interno es reconocido solamente dentro de una unidad de programa y su existencia está limitada por el tiempo de ejecución de la misma. Por ejemplo, el programa dado a continuación, lee un archivo interno representado por la variable de caracteres linea:

```
program ejemplo
character (len=80)::linea
data linea/'  1  2  3'/
read (linea,67)i,j,k
67 format (3i3)
print *,i,j,k
end
```

y escribe el registro

```
1    2    3
```

El programa ejercicio muestra operaciones de lectura y/o escritura con archivos internos:

```
program ejercicio
character uno,dos
data uno/1h0/
read(uno,'(i1)')n                ! lectura del archivo interno representado
n=n+1                            ! por la variable uno.
write(dos,'(i1)')n               ! escritura en el archivo interno
print *,dos                      ! representado por la variable dos.
end
```

En este caso, la información escrita en la unidad estándar de salida es:

```
1
```

Archivos externos

Los archivos externos se clasifican, en base al modo de acceso, en secuenciales y directos. Si el criterio de clasificación es el formato de representación de la

información, entonces los archivos serán formateados (ASCII) o no formateados (binarios).
OPEN "conecta" o "reconecta" a un archivo externo con una unidad de lectura/escritura, como en:

```
open(unit=10,file='datos.dat',access='sequential',status=old')
```

La cláusula ACCESS indica el tipo de acceso: secuencial ('SEQUENTIAL') o directo ('DIRECT'). El valor por omisión es 'SEQUENTIAL'. STATUS='NEW' se emplea cuando el archivo no existe y entonces será creado al ejecutarse el OPEN. STATUS='REPLACE' reemplaza a un archivo del mismo nombre o crea uno nuevo en caso de que no exista. Con STATUS='SCRATCH', el archivo será borrado al terminar el programa o al ser ejecutada la instrucción CLOSE correspondiente. STATUS='UNKNOWN' se utiliza cuando no se aplica alguna de las opciones anteriores para STATUS. La opción por "default" para STATUS es 'UNKNOWN':

```
character *20 nombre;read *,nombre
open(11,file=nombre,status='new')

! el default es access='sequential'
open(5,file='datos.dat')

open(19,status='scratch')
! especifica un archivo temporal disponible solamente
! durante la ejecución del programa

open(497,'abc.out', status='replace')
```

La cláusula STATUS='REPLACE' no es soportada por FORTRAN 77. FORM especifica el tipo del archivo que va a ser conectado a una unidad. FORM='FORMATTED' indica un archivo formateado (código de caracteres) mientras FORM='UNFORMATTED' especifica un archivo no formateado (código binario):

```
open(45,'archivo.dat',form='formatted')

open(319,'arch.xyz',form='unformatted')
```

El valor por default es FORMATTED para ACCESS='SEQUENTIAL' y 'UNFORMATTED' para ACCESS='DIRECT'. RECL=expresión entera, indica la longitud del registro de un archivo conectado para acceso directo. Para acceso secuencial RECL especifica la longitud máxima del registro. Si ACCESS='DIRECT', entonces es obligatorio especificar RECL. FORTRAN 77 sólo permite

usar RECL con archivos de acceso directo:

```
open(15,file='uno.dat',access='direct',recl=75)
```

BLANK='NULL' causa que los espacios en campos numéricos sean ignorados, BLANK='ZERO' ocasiona que los espacios en campos numéricos sean interpretados como ceros. El valor por omisión es 'NULL'. Esta cláusula BLANK sólo es permitida cuando el archivo está conectado con FORM='FORMATTED':

```
open(39,file='muestra',blank='null')
```

```
! 'con' es el nombre de la unidad estándar de lectura en MS-DOS
open(5,file='con',blank='zero')
!los espacios en el campo i10 son tomados como ceros
read(5,'(i10)')k
print *,k
```

Con POSITION='REWIND', un archivo con acceso secuencial recientemente abierto es posicionado en su punto inicial; con POSITION='APPEND', el posicionamiento será antes de la marca de fin de archivo y con POSITION='ASIS' la posición no será cambiada. El "default" es POSITION='ASIS':

```
open(78,file='nuevo',position='append')
```

```
open(78,file='nuevo',position='rewind')
```

```
open(78,file='nuevo',position='asis')
```

ACTION='READ' implica que el archivo es conectado para solo lectura, ACTION='WRITE' indica que el archivo es conectado para solo escritura. ACTION='READWRITE' especifica una conexión de lectura y escritura. El "default" es 'READWRITE'.

```
open(111,file='archivo.nw',action='read')
```

```
open(111,file='archivo.nw',action='write')
```

```
open(111,file='archivo.nw',action='readwrite')
```

DELIM='APOSTROPHE' especifica que el apóstrofo(') será usado como carácter delimitador para las constantes de caracteres impresas con NAMELIST o con escritura dirigida por lista. Con DELIM='QUOTE', entonces el carácter delimitador será las comillas(") y con DELIM='NONE' ninguno de los delimitadores

anteriores será usado. El valor por omisión es DELIM='NONE'. Esta cláusula solo es permitida en archivos formateados e ignorada en operaciones de lectura. Dado lo anterior, el programa

```
character(len=6)::palabra_uno,palabra_dos
palabra_uno='ESIQIE'
palabra_dos='IPN'
open(76,file='abc.n',delim='quote')
write(76,*)palabra_uno,palabra_dos
```

escribirá:

```
"E S I Q I E "          I P N "
```

La cláusula PAD='YES' indica que serán usados espacios cuando la lista de variables y las especificaciones de campo en el formato requieran más datos de los que contiene el registro. La especificación PAD sólo es permitida en un archivo conectado con FORM='FORMATTED' y es ignorada en instrucciones de escritura. Por lo tanto, el programa

```
open(14,file='prueba.in',pad='yes')
read(14,'(3i2)')i,j,k
print *,i,j,k
```

con un registro de lectura en el archivo prueba.in

```
1020
```

imprimirá la línea:

```
10        20        0
```

mientras que

```
open(14,file='prueba.in',pad='no')
read(14,'(3i2)')i,j,k
print *,i,j,k
```

con el mismo registro de lectura produciría un error en tiempos de ejecución. El valor por omisión para la cláusula PAD, es PAD='YES'.
Las cláusulas ACTION, DELIM, PAD y POSITION no están incluidas en el FORTRAN 77.
En caso de ocurrir un error al ejecutarse un OPEN, la cláusula ERR permite todavía conservar el control sobre el programa:

```
! si ocurre un error al ejecutarse el open, habrá una transferencia de control a la
! instrucción etiquetada con el 67
open(34,file='ninguno.dat',status='old',err=67)
read(34,*)a,b,j
print *,a,b,j
stop
67 print *,'error en el open'
end
```

IOSTAT permite también manejar los errores que surgen durante la ejecución de un OPEN, de manera similar a la cláusula IOSTAT de la instrucción READ:

```
integer error_open
open(34,file='ninguno.dat',status='old',iostat=error_open)
select case(error_open)
case(:-1)
        print *,'fin de archivo o fin de registro'
case(0)
        read(34,*)a,b,j                                  ! sin error en el open
case(1:)
        print *,'error en la instrucción open'
end select
end
```

Instrucción CLOSE

Termina la conexión de la unidad especificada con un archivo externo:

```
!STATUS='KEEP' permite conservar un archivo después de ejecutarse el
! CLOSE (default)
close(unit=17,status='keep')

! STATUS='DELETE' borra el archivo después de la ejecución del CLOSE
close(17,status='delete')
```

CLOSE también soporta la cláusula ERR:

```
! en caso de error, la transferencia de control será hacia la instrucción etiquetada
! con 200
close(unit=451,err=200)

! instrucción equivalente a la anterior
close(err=200,unit=451)
```

Con IOSTAT=variable entera, al ocurrir un error en la ejecución de CLOSE, un valor diferente de cero será asignado a la variable entera:

```
integer::var
close(38,iostat=var)
```

Al terminar la ejecución de una unidad de programa, los archivos todavía abiertos, son cerrados automáticamente.

Instrucciones BACKSPACE, ENDFILE y REWIND

La instrucción BACKSPACE posiciona el archivo antes del registro actual, si es que lo hay. En cualquier otro caso, el posicionamiento será antes del registro inmediatamente anterior:

```
backspace(unit=21,err=103)

backspace 10                              ! la unidad es la número 10

backspace n+1
```

Nuevamente, la cláusula ERR permite realizar una transferencia de control al detectarse un error en la ejecución de un BACKSPACE:

```
backspace(unit=21,err=103)
```

IOSTAT=variable entera, devuelve un valor positivo en caso de error, cero cuando no hay error alguno y un valor negativo en una condición de fin de archivo o de fin de registro:

```
integer::error
backspace(78,iostast=error)
if(ierr/=0)stop "Error en backspace"
```

ENDFILE escribe una marca de fin de archivo como el siguiente registro de un archivo. El archivo es entonces posicionado después del fin de archivo, el cual queda como el último registro del archivo. Ejemplos de la instrucción ENDFILE:

```
endfile(unit=2,err=10)
```

```
endfile 17                                  ! la unidad es la número 17

! iostat=variable entera, con una interpretación similar que en BACKSPACE
endfile(19,iostat=k)
```

La instrucción REWIND posiciona el archivo especificado en su posición inicial:

```
rewind(unit=13)
```

Las cláusulas IOSTAT y ERR tienen en REWIND el mismo significado que en BACKSPACE y ENDFILE

```
rewind(13,err=25)

rewind(err=38,iostat=j,unit=178)
```

Manejo de archivos

A continuación; varios programas que muestran el manejo de archivos externos:

```
program creacion_de_archivo
! Este programa crea un archivo con datos leidos del teclado
character (len=40)::nombre
open(8,file='empleado.dat')                 ! acceso secuencial y archivo formateado
rewind 8
read(5,*,iostat=ierror)nombre,sueldo
! 5 es una unidad preconectada con el teclado
lectura_datos:do while(ierror==0)
        ! escritura en el archivo empleado.dat
         write(8,'(a40,f12.2)')nombre,sueldo
        read(5,*,iostat=ierror)nombre,sueldo
end do lectura_datos
stop
end program creacion_de_archivo

program lectura_de_archivo
! Este programa lee el archivo creado con el programa creacion_de_archivo
character(len=40)::nombre;integer error
open(37,file='empleado.dat')
! acceso secuencial y archivo formateado
rewind 37
read(37,'(a40,f12.2)',iostat=error)nombre,sueldo
while(error==0)
```

```
        ! escribe los nombres con sueldo>5000
        if(sueldo>5000)print *,nombre,sueldo
        read(37,'(a40,f12.2)',iostat=error)nombre,sueldo
end do
end program lectura_de_archivo

program creacion_binario
! Este programa crea un archivo binario con datos leídos a través del teclado
open(349,'binario.dat',status='replace',access=direct,&
        recl=12)
! acceso directo, archivo no formateado
do i=1,10
        read *,x,y,z
        !el número de registro indicado en REC= debe ser mayor que cero
        write(345,rec=i)x,y,z
end do
close(345,status='keep')
stop
end

program lectura_binario
! Este programa lee el archivo creado por el programa creacion_binario
integer registro
open(90,file='binario.dat',access='direct',recl=12)
read *,registro
do while(registro>0.and.registro<=6)
        read(90,rec=registro)a,b,c
        print *,a,b,c
        read *,registro
end do
stop
end program lectura_binario
```

Proposición INQUIRE

La proposición INQUIRE hace posible que un programa "interrogue" acerca de la existencia de un archivo, su conexión, método de acceso y otras propiedades:

```
character *20 acceso,formato,acción,relleno,delimitador
        .
        .
inquire(unit=89,access=acceso,formatted=formato,&
        action=accion,pad=relleno,delim=delimitador)
```

```
print *,acceso,formato,accion,relleno,delimitador

logical existencia
! existencia='yes', en caso de que el archivo exista
inquire(file='uno.dat',exist=existencia)
print *,existencia

character *25 archivo
        .
        .
! Se pregunta por el nombre del archivo conectado con la unidad 35.
! La cláusula ERR permite ejecutar una transferencia de control en caso  de error
! en el INQUIRE
inquire(unit=35,name=archivo,err=300)
print *,archivo
```

La cláusula IOSTAT tiene la interpretación acostumbrada:

```
integer longitud_del_registro,clave
inquire(17,recl=longitud_del_registro,iostat=clave)

logical apertura_de_archivo
integer unidad
character *30 acceso_directo
        .
        .
read *,unidad
inquire(unidad,opened=apertura_de_archivo,&
        direct=acceso_directo)
! apertura_de_archivo será .true. si el archivo está abierto
print *,unidad,apertura_de_archivo
! acceso_directo puede ser 'YES', 'NO' o 'UNKNOWN'
print *, acceso_directo

integer::numero de unidad
        .

        .
inquire(file='test.xyz',number=numero_de_unidad)
! se pregunta por el número de unidad conectada con el archivo test.xyz
print *,numero_de_unidad

character(len=30)::cadena1,cadena2
        .
        .
```

```
inquire(78,form=cadena1,sequential=cadena2)
! cadena1 puede ser 'FORMATTED', 'UNFORMATTED' o 'UNDEFINED'.
! cadena2 puede ser 'YES', 'NO' o 'UNKNOWN'.
print *,cadena1
print *,cadena2

character (len=20)::posicion
        .
        .
inquire(371,position=posicion)
! posición tendría cualquiera de los valores 'REWIND', 'APPEND' o 'ASIS'
```

Para un archivo de acceso directo, NEXTREC=variable entera regresa el número del último registro procesado + 1. Si el archivo es de acceso secuencial, entonces se devuelve un valor indefinido.

```
integer k
open(35,file='abc',access='direct',recl=40)
inquire(35,nextrec=k)
```

Con IOLENGTH= se calcula la longitud de registro necesaria para leer la lista de variables, en un archivo no formateado

```
inquire(iolength=longitud_del_registro)a,b,i,j
print *,longitud_del_registro
open(17,file='prueba',access='direct',&
        recl=longitud_del_registro)
```

La cláusula IOLENGTH no es parte del estándar FORTRAN 77.

Ejercicio

Elabore un programa que construya una base de datos simple, representada por un archivo no formateado y de acceso directo. Cada registro deberá contener el nombre, la temperatura crítica, la presión crítica y el factor acéntrico de un compuesto químico. El mismo programa deberá hacer una consulta al archivo a través del número de registro.

Pseudocódigo

```
!Este programa crea una base de datos, representada por
!un archivo no formateado y de acceso directo, y realiza
!una consulta en ella
```

```
hilera de caracteres nombre
entero regr,nregr,lregr
real tc,pc,w
!
!cada registro consiste en el nombre, la temperatura
!crítica y el factor acéntrico de un compuesto químico
!
llama creabd(nregr,lregr)
lee regr  !lectura del número de registro por consultar
llama consbd(nregr,regr,nombre,tc,pc,w)
escribe nombre,tc,pc,w
fin
Proc creabd(nregr,lregr)
!Este procedimiento crea un archivo formateado para
!acceso directo
hilera de caracteres nombre
real tc,pc,w
entero error,nregr,lregr
lregr=42
abre ('compuest.dat',acceso=directo)
!se leen los datos a través del teclado
lee (estado=error)nombre,tc,pc,w
nregr=0
mientras error=0
empieza
     nregr=nregr+1
     escribe ('compuesto.dat',registro=nregr)nombre,tc,pc,w
     lee (estado=error)nombre,tc,pc,w
termina
regresa
fin
Proc consbd(nregr,regr,nombre,tc,pc,w)
!Este procedimiento realiza una consulta en el archivo
!compuest.dat
hilera de caracteres nombre
real tc,pc,w
entero nregr,regr
si regr≤0 o regr>nregr entonces regresa
!el archivo ya fue abierto por creabd
lee ('compuest.dat',registro=regr)nombre,tc,pc,w
regresa
fin
```

Codificación en FORTRAN 77

```
F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486) Mon Apr 27 18:31:59 1998       Page:
1
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

PROGRAM BDATOS , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente
```

```
 1         program bdatos
 2   c
 3   c       Este programa crea una base de datos,representada por un archivo
 4   c       no formateado y de acceso directo,y realiza una consulta en ella
 5   c
 6         character *30 nombre
 7         integer regr
 8   c
 9   c       cada registro consiste en el nombre,la temperatura critica,la presion
10   c       critica y el factor acentrico de un compuesto quimico
11   c
12         call creabd(nregr,lregr)
13         open(15,file='con')
14   c
15   c       lectura del numero de registro por consultar
16   c
17         read(15,*)regr
18         call consbd(nregr,regr,nombre,tc,pc,w)
19         print *,nombre,tc,pc,w
20         stop
21         end
```

F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486) Mon Apr 27 18:31:59 1998 Page: 2
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

SUBROUTINE CREABD , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

```
22         subroutine creabd(nregr,lregr)
23   c
24   c       Esta subrutina crea un archivo no formateado para acceso directo
25   c
26         character *30 nombre
27         integer error
28         lregr=42
29   c
30   c       el archivo es compuest.dat
31   c
32         open(100,file='compuest.dat',access='direct',recl=lregr)
33   c
34   c       se leen los datos a traves del teclado
35   c
36         read(5,*,iostat=error)nombre,tc,pc,w
37         nregr=0
38      10 if(error.ne.0)go to 20
39          nregr=nregr+1
40   c
41   c         nregr es el numero de registros creados
42   c
43          write(100,rec=nregr)nombre,tc,pc,w
44          read(5,*,iostat=error)nombre,tc,pc,w
45         go to 10
46      20 return
47         end
```

F77L-EM/32 FORTRAN 77 Version 5.11 (compiling for the 80386/80486) Mon Apr 27 18:31:59 1998 Page: 3
Copyright(c) 1988-1992, Lahey Computer Systems, Inc. ALL RIGHTS RESERVED

SUBROUTINE CONSBD , Opciones de Compilación:
/N0/N2/N4/N7/NA2/NB/NC/NC1/ND/NF/H/NI/NK/NL/P/NQ1/NQ2/NQ3/R/S/NT/NV/W/NX/NZ1
Listado de fichero fuente

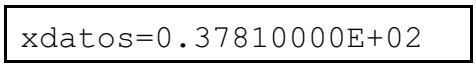
```
48         subroutine consbd(nregr,regr,nombre,tc,pc,w)
49   c
50   c       Esta subrutina realiza una consulta en el archivo compuest.dat
51   c
52
53         character *30 nombre
54         integer regr
55         if(regr.le.0.or.regr.gt.nregr)return
56   c
57   c       el archivo ya fue abierto por creabd
58   c
59         read(100,rec=regr)nombre,tc,pc,w
60         return
61         end
```

```
Lahey Fortran 90 Compiler Release 3.00c                                    Mon Apr 27
19:10:30 1998  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

MODULE DATOS_COMUNES    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin
-inln
                                   -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
-------------------------------------------------  Source Listing
-------------------------------------------------------------
     1   module datos_comunes
     2   type compuesto
     3     sequence
     4     character (len=30)::nombre
     5     real::temperatura_critica,presion_critica,factor_acentrico
     6   end type compuesto
     7   end module datos_comunes

Lahey Fortran 90 Compiler Release 3.00c                                    Mon Apr 27
19:10:30 1998  Page: 2
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

PROGRAM BASE_DE_DATOS    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin
-inln
                                   -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
-------------------------------------------------  Source Listing
-------------------------------------------------------------
      8  program base_de_datos
      9  use datos_comunes
     10  !
     11  ! Este programa crea una base de datos,representada por un archivo
     12  ! no formateado y de acceso directo,y realiza una consulta en ella
     13  !
     14  integer registro
     15  !
     16  ! cada registro consiste en el nombre,la temperatura critica,la presion
     17  ! critica y el factor acentrico de un compuesto quimico
     18  !
     19  type (compuesto)::especie_quimica
     20  call creacion_base_de_datos(n_registros,long_registro)
     21  open(15,file='con')
     22  read(15,*)registro    !lectura del numero de registro por consultar
     23  call consulta_base_de_datos(n_registros,registro,especie_quimica)
     24  print *,especie_quimica
     25  stop 'fin de programa'
     26  end program base_de_datos
Bytes of stack required for this program unit: 100.

Lahey Fortran 90 Compiler Release 3.00c                                    Mon Apr 27
19:10:30 1998  Page: 3
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

SUBROUTINE CREACION_BASE_DE_DATOS    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed
  -nin    -inln

                                   -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
-------------------------------------------------  Source Listing
-------------------------------------------------------------
     27  subroutine creacion_base_de_datos(n_registros,long_registro)
     28  !
     29  ! Esta subrutina crea un archivo no formateado para acceso directo
     30  !
     31  use datos_comunes
     32  integer error
     33  type (compuesto)::componente
     34  inquire(iolength=long_registro)componente
     35  !
     36  ! el archivo es compuest.dat
     37  !
     38  open(100,file='compuest.dat',status='replace',access='direct',&
     39       recl=long_registro)
     40  read(5,*,iostat=error)componente   !se leen los datos a traves del teclado
     41  n_registros=0
     42  do while(error==0)
     43    n_registros=n_registros+1              !n_registros es el numero de registros
     44    write(100,rec=n_registros)componente   !creados
```

```
     45    read(5,*,iostat=error)componente
     46  end do
     47  end subroutine creacion_base_de_datos
Bytes of stack required for this program unit: 116.

Lahey Fortran 90 Compiler Release 3.00c                                                Mon Apr 27
19:10:30 1998  Page: 4
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

SUBROUTINE CONSULTA_BASE_DE_DATOS    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed
  -nin    -inln

                                     -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
-------------------------------------------------  Source Listing
------------------------------------------------------------
     48  subroutine consulta_base_de_datos(n_registros,registro,componente)
     49  !
     50  ! Esta subrutina realiza una consulta en el archivo compuest.dat
     51  !
     52  use datos_comunes
     53  integer registro
     54  type (compuesto)::componente
     55  if(registro<=0.or.registro>n_registros)return  !numero de registro no valido
     56  read(100,rec=registro)componente   !el archivo ya fue abierto por
     57  return                             !creacion_base_de_datos
     58  end subroutine consulta_base_de_datos
Bytes of stack required for this program unit: 16.
```

APUNTADORES

Las variables de los cinco tipos intrínsecos de FORTRAN anteriormente tratados y las de los tipos derivados formados a partir de ellos, comparten una característica en común: almacenan algún tipo de dato. Pero en FORTRAN 90 existe un tipo intrínseco adicional: el apuntador (pointer), el cual contiene la dirección de una localidad de memoria en lugar del valor de un dato. Por ejemplo, en el siguiente esquema; xdato es una variable de tipo real mientras que apunt1 es un apuntador:



El uso de apuntadores no está permitido en FORTRAN 77.

Apuntadores y destinos

Una variable es declarada como apuntador a través del atributo POINTER, como en:

```
real,pointer::apunt1
```

o con una proposición POINTER

```
real apunt1
pointer apunt1
```

En este caso, la variable apunt1 es declarada un apuntador y su contenido es la dirección de una variable de tipo real. Es posible declarar apuntadores de variables de tipos derivados

```
type estructura
        character (len=30)::nombre
        real (kind=4)::x,y,f
end type estructura
type (estructura),pointer::apunt_estructura
```

y apuntadores de arreglos

```
! matriz apunta a un arreglo real de rango 2
```

```
real,dimension(:,:),pointer::matriz
```

La variable o arreglo que es el "blanco" del apuntador se conoce como destino (target) es declarada como tal con el atributo TARGET:

```
real,pointer::a
real,target::b
```

o en una proposición TARGET

```
real::b
target::b
```

Asignación de apuntadores

Un apuntador puede ser asociado con un destino mediante una instrucción de asignación de apuntadores:

```
apuntador=>destino
```

Cuando esta instrucción es ejecutada, la dirección de memoria del destino es almacenada en el apuntador. Después de la asignación del apuntador, cualquier referencia al apuntador es tratada como si fuera al contenido del destino (deferencia del apuntador):

```
real,pointer::var_1
real,target::x_1
var_1=>x_1
read *,var_1
c=x_1+var_1/12.5      ! uso indistinto de x_1 o var_1
print *,x_1,c
```

Por lo anterior, se dice que un apuntador es un "alias". Si un apuntador ya está asociado con un destino y otra asignación de apuntador es ejecutada usando el mismo apuntador, entonces la asociación con el primer destino es cancelada y el apuntador ahora apunta al segundo destino. Por ejemplo, el programa:

```
program prueba_apuntadores_1
real,pointer::p
real,target::t=-17.8,s=33.5
p=>t
print *,p,t,s
```

```
p=>s
print *,p,t,s
stop
end
```

imprimiría los resultados

```
-17.8000  -17.8000   33.5000
 33.5000  -17.8000   33.5000
```

Es posible asignar la dirección de un apuntador a otro apuntador:

```
apuntador_1=>apuntador_2
```

Después de una instrucción como la anterior, ambos apuntadores apuntan directa e independientemente al mismo destino. En caso de que uno de los dos apuntadores sea cambiada su asociación en una asignación posterior, el otro apuntador continuará apuntando a su destino original. Una muestra de asignación entre apuntadores es el programa:

```
program prueba_apuntadores_2
real,pointer::point1,point2
real,target::targ1=10.0_4,targ2=20.0_4
point1=>targ1
point2=>point1
print *,point1,point2,targ1,targ2
point1=>targ2
print *,point1,point2,targ1,targ2
stop
end progam prueba_de_apuntadores
```

cuyos resultados son

```
10.0000   10.0000   10.0000   20.0000
20.0000   10.0000   10.0000   20.0000
```

Un apuntador hacia un arreglo es declarado como de forma diferida, esto es, se indica el rango pero no la extensión de cada dimensión:

```
! matriz apunta a un arreglo real de rango 2
real,dimension(:,:),pointer::matriz
```

Cualquier sección de un arreglo definida mediante una tríada de subíndices, puede ser usada como destino de un apuntador. Así, el programa:

```
program seccion_arreglo
implicit none
integer::i
integer,dimension(8),target::array=(/(i,i=1,8)/)
integer,dimension(:),pointer::apunt,apunt_seccion
apunt=>array
apunt_seccion=>apunt(2:8:2)
write(*,100)apunt
write(*,100)apunt_seccion
stop
100 format(1x,8(i3))
end program
```

```
genera los registros

  1  2  3  4  5  6  7  8
  2  4  6  8
```

Estados de asociación de los apuntadores

El estado de asociación (association status) de un apuntador indica si este está asociado o no a un destino válido. Al respecto, existen tres posibilidades: indefinido, asociado y no asociado. Cuando un apuntador es indicado como tal en una declaración de tipo, su estado es indefinido. Una vez que el apuntador ha sido asociado con un destino mediante una asignación, su estado se convierte en asociado. En caso de que el apuntador quede sin asociar con algún destino, su estado queda como no asociado.
Un apuntador queda sin asociar con un destino y simultáneamente asociado con otro destino al ejecutarse una asignación de apuntadores, como en:

```
complex,pointer::w1
complex,target::var1,var2
      .
      .
w1=>var1
      .
      .
w1=>var2
```

donde al final w1 queda asociado con var2 pero ya no con var1.
Otra forma de cancelar la asociación de un apuntador con sus destinos es el uso de la instrucción NULLIFY:

```
integer,pointer::pt1,pt2
integer,target::destino_1,destino_2
pt1=>destino_1
pt2=>destino_2
      .
      .
! cancelación de la asociación de los apuntadores pt1 y pt2
nullify(pt1,pt2)
```

La función lógica intrínseca ASSOCIATE permite conocer si un apuntador está asociado con algún destino. Por ejemplo:

```
real (kind=8),pointer::uno,dos
real (kind=8),target::destino_uno,destino_dos
logical::estado
      .
      .
dos=>destino_dos                          ! dos es asociado con destino_dos
! se pregunta si uno está asociado con algún destino
estado=associated(uno)
print *,estado
estado=associated(dos)
! ahora se pregunta si dos está asociado con registro_dos
print *,estado,associated(dos,destino_dos)
```

genera los resultados

```
F
T    T
```

Asignación dinámica de memoria con apuntadores

Una de las características más poderosas de los apuntadores es la de ser usados para crear dinámicamente variables o arreglos y poder liberar el espacio que ocupan una vez que ya no son necesarios. En este contexto, la memoria es asignada mediante una instrucción ALLOCATE y liberada con DEALLOCATE. Por ejemplo, las instrucciones:

```
real (kind=8),dimension(:,:),pointer::point1
      .
      .
read *,m,n
allocate(point1(m,n))
      .
```

.
deallocate(point1)

crean un objeto de datos sin nombre, del tamaño especificado y del tipo correspondiente al apuntador. Además, el apuntador es asociado con dicho objeto de datos. Ya que el nuevo objeto de datos no tiene nombre, sólo se puede tener acceso a él, a través del apuntador. Después de que el ALLOCATE es ejecutado, el estado del apuntador queda como asociado. Si el apuntador hubiera sido asociado con otro destino antes de la ejecución del ALLOCATE, entonces la asociación original es cancelada. La siguiente es una muestra del uso de arreglos dinámicos creados con apuntadores:

```
program producto_interno
integer::error_allocate,error_deallocate
real,dimension(:),pointer::a,b
read *,ncomp
! es posible también usar la cláusula STAT= , tanto en allocate como en deallocate
allocate(a(ncomp),b(ncomp),stat=error_allocate)
if(error_allocate==0)then
        pe=dot_product(a,b)
        print *,'producto punto=',pe
        deallocate(a,b,stat=error_deallocate)
        if(error_deallocate/=0)print *,'Error en deallocate'
else
        print *,'Error en allocate'
end if
stop
end program producto_interno
```

Si todos los apuntadores que señalan a un objeto de datos como el del programa anterior son asociados con otros objetos o quedan sin asociar por la ejecución de un NULLIFY, entonces dicho objeto de datos queda inaccesible para el programa. Sin embargo, el objeto está todavía presente en la memoria, pero ya no es posible usarlo. Lo anterior se conoce como "fuga de memoria" (memory leak) y constituye una situación que siempre es conveniente evitar.

Ejercicio

Cálculo del vector residual r = A x-b correspondiente al sistema lineal A x=b. A es una matriz de nxn; r, x y b son vectores de n componentes.

Pseudocódigo

```
!Cálculo del vector residual r=Ax-b
!A es una matriz de nxn; r, x y b son vectores de n
!elementos
entero n
real A[n,n],b[n],x[n],r[n]
lee n
lee A
lee b,x
llama imprimematriz(n,n,A,'matriz A') !impresión de datos
escribe 'vector b:',b
escribe 'vector x:',x
r=MATMUL(A,x)-b   !cálculo del vector residual
escribe 'vector residual Ax-b',r
fin
Proc imprimematriz(m,n,X,titulo)
!Impresión de una matriz X de m filas y n columnas
!opcionalmente, se imprime un titulo
entero m,n
real X[:,:]
hilera de caracteres (opcional) titulo
si PRESENT(titulo) entonces
     escribe titulo
otro
     ejecuta i=1,m
          escribe X[i,1:n]
regresa
fin
```

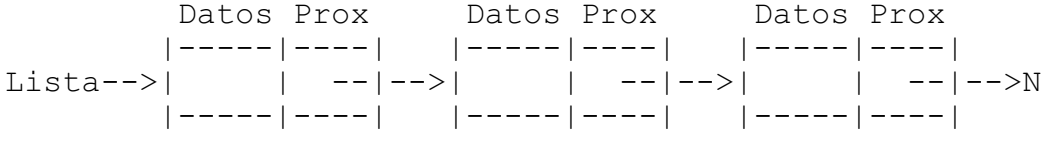
Codificación en FORTRAN 90

```
Lahey Fortran 90 Compiler Release 3.00c                                    Tue May  5
17:41:01 1998  Page: 1
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

MODULE IMPRESION    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed    -nin    -inln

                                       -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
-------------------------------------------------------------
      1  module impresion
      2  contains
      3    subroutine imprime_matriz(m,n,X,titulo)
      4    ! Impresion de una matriz X de m filas y n columnas
      5    ! opcionalmente,se imprime un titulo
      6    integer::m,n
      7    real,dimension(:,:)::x
      8    character (len=*),optional::titulo
      9    if(present(titulo))then
     10      print '(/,6x,a30,/)',titulo
     11    else
     12      print '(/)'
     13    end if
     14    do i=1,m
     15      print '(6x,6(f10.3,2x))',X(i,1:n)
     16    end do
     17    return
     18    end subroutine
Bytes of stack required for this program unit: 36.
```

```
Lahey Fortran 90 Compiler Release 3.00c                                  Tue May  5
17:41:01 1998  Page: 2
Copyright (C) 1994-1996 Lahey Computer Systems.  All rights reserved.

PROGRAM ARREGLOS_APUNTADOR    Compiling Options: -nchk   -ndal   -ndbl   -nf90   -nfix   -ng     -hed
-nin    -inln
                                    -lst    -pca    -stchk  -nsyn   -ntrap  -nvax   -w      -nwo
-nxref
------------------------------------------------  Source Listing
------------------------------------------------------------
     19  end module impresion
     20  program arreglos_apuntador
     21  !
     22  ! Calculo del vector residual r=Ax-b
     23  ! A es un matriz de nxn;r,  y b son vectores de n elementos
     24  !
     25  use impresion
     26  real,dimension(:,:),pointer::A  !arreglo apuntador
     27  real,pointer,dimension(:)::b,x,r
     28  read(*,*)n
     29  allocate(A(n,n),b(n),x(n),r(n))
     30  read(*,*)a
     31  read(*,*)b,x
     32  call imprime_matriz(n,n,A,'matriz A') !impresion de datos
     33  print '(/,6x,"vector b:",6(f10.3,2x))',b
     34  print '(/,6x,"vector x:",6(f10.3,2x))',x
     35  r=matmul(A,x)-b   !calculo del vector residual
     36  print '(/,6x,"vector residual Ax-b:",6(f10.3,2x))',r
     37  deallocate(A,b,x,r)
     38  stop 'FIN DE PROGRAMA'
     39  end program arreglos_apuntador
Bytes of stack required for this program unit: 228.
```

Listas encadenadas

Las estructuras de datos pueden ser dinámicas o estáticas. El tamaño de una estructura estática permanece fijo una vez que la memoria ocupada por ella ha sido apartada. Los arreglos son ejemplos de estructuras estáticas. Las estructuras dinámicas se caracterizan por que pueden cambiar su tamaño durante la ejecución del programa que las maneja. Una estructura dinámica es una colección de elementos, llamados nodos, que son "encadenados" entre sí. Dicho encadenamiento se logra al asociar cada nodo con un apuntador, el cual "apunta" al siguiente nodo de la estructura. Un ejemplo simple de este tipo de estructuras es la lista encadenada mostrada en el siguiente esquema:



en el cual se tiene acceso al primer elemento a través del apuntador Lista. El símbolo N indica que la parte Prox del último nodo representa un apuntador nulo, es decir, no existe un siguiente nodo.
Por lo tanto, los nodos de una lista encadenada son

representados como estructuras con dos clases de componentes; datos y un apuntador. Por ejemplo:

```
type lista_encadenada
        integer::numero
        type (lista_encadenada),pointer::proximo
end type lista_encadenada
```

donde cada nodo en esta lista es una estructura del tipo lista_encadenada y consiste de dos componentes. El primer componente, numero, es usado para guardar la información, el segundo componente, proximo, es un apuntador que apunta al próximo nodo de la lista.
Otro ejemplo de lista encadenada es:

```
type registro_alumnos
        character (len=40)::nombre
        character (len=12)::numero_de_boleta
        integer,dimension(30)::calificaciones
end type registro_alumnos
type lista_alumnos
        type (registro_alumnos)::datos
        type (lista_alumnos),pointer::proximo
end type lista_alumnos
```

Además de los nodos de una lista encadenada, se requiere de un apuntador hacia el primer nodo:

```
type (lista_encadenada),pointer::num_lista

type (lista_alumnos),pointer::alum_lista
```

Supóngase ahora que se desea construir una lista como la definida arriba con el nombre de lista_encadenada y que se declaran los apuntadores

```
type (lista_encadenada),pointer::num_lista,temporal
```

Los paso para construir la lista son:
1)Iniciar una lista vacía

```
nullify(num_lista)
```

donde num_lista es un apuntador que siempre apunta al primer nodo de la lista.
2)Con un ALLOCATE se crea un nuevo nodo temporalmente apuntado por temporal

```
allocate(temporal)
```

3)El valor numérico por almacenar es asignado a temporal%numero

```
temporal%numero=dato
```

4)El apuntador temporal%proximo es asociado con num_lista para encadenar el apuntador del nodo recientemente creado al primer nodo de la lista

```
temporal%proximo=>num_lista
```

5)Por último, el apuntador num_lista es actualizado de tal forma que ahora apunta al nuevo nodo

```
num_list=>temporal
```

El proceso anterior se puede repetir varias veces para crear una lista encadenada con varios nodos:

```
program crea_lista_encadenada
type lista_encadenada
        integer::numero
        type (lista_encadenada),pointer::proximo
end type lista_encadenada
type (lista_encadenada),pointer::num_lista,temporal
integer::dato,estado_read,estado_allocate
nullify(num_lista)
do
        read(*,*,iostat=estado_read)
        if(estado_read/=0)exit
        allocate(temporal,stat=estado_allocate)
        if(estado_allocate/=0)stop 'Memoria insuficiente'
        temporal%numero=dato
        temporal%proximo=>num_lista
        num_lista=>temporal
end do
end program crea_lista_encadenada
```

Una vez que una lista encadenada ha sido construida, es posible "recorrerla" de principio a fin. Para recorrer una lista encadenada, los pasos son:

1)Iniciar un apuntador que apunte al primer nodo

```
type(lista_encadenada),pointer::actual
actual=>num_lista
```

2)Desplegar el contenido del nodo

```
print *,actual%numero
```

3)Movimiento al siguiente nodo

```
actual=>actual%proximo
```

Los pasos anteriores pueden repetirse hasta que actual quede nulo:

```
actual=>num_lista
do
        if(.not.associated(actual))exit !fin de la lista
        print *,actual%numero
        actual=>actual%proximo
end do
```

Para insertar un nodo en una lista encadenada, primeramente se almacena el nuevo dato en un nodo creado a través de un allocate:

```
type (lista_encadenada),pointer::temporal,predecesor,actual
allocate(temporal)
read *,dato
temporal%numero=dato
```

Ahora hay dos casos por considerar: inserción del nodo al principio de la lista o después de algún elemento dado de la lista. El primer caso se maneja de manera similar al de la construcción de la lista encadenada. En el segundo, supóngase que el nuevo nodo se insertará entre los nodos apuntados por predecesor y actual; entonces el nodo es insertado asociando el apuntador del nuevo nodo con el nodo apuntado por actual:

```
temporal%proximo=>actual
```

y luego el apuntador del nodo apuntado por predecesor es asociado con el nuevo nodo

```
predecesor%proximo=>temporal
```

En la eliminación de nodos también se presentan dos casos: la cancelación del primer nodo de la lista y la eliminación de un nodo que tiene un predecesor. Para el primer caso, el apuntador actual se asocia con el primer nodo de la lista:

actual=>num_lista

seguidamente se asocia num_lista con el segundo nodo

num_lista=>actual%proximo

por último se libera la memoria ocupada por el nodo apuntado por actual

dealloacte(actual)

En el segundo caso, supóngase que el predecesor del nodo a ser borrado es apuntado por predecesor. Entonces el nodo es eliminado por asociación del apuntador del nodo apuntado por predecesor con el sucesor del nodo por eliminar:

predecesor%proximo=>actual%proximo

y luego liberando al nodo apuntado por actual

deallocate(actual)

Uso de apuntadores en procedimientos

Los apuntadores pueden ser usados como argumentos fingidos pasados como argumentos actuales en procedimientos. Además, una función puede devolver como resultado a un apuntador. Sin embargo, las siguientes restricciones son aplicables:
1)Si un procedimiento tiene argumentos fingidos con cualquiera de los atributos POINTER o TARGET, entonces el procedimiento deberá tener una interfase explícita.
2)En caso de que un argumento fingido sea un apuntador, el argumento actual pasado al procedimiento debe ser un apuntador del mismo tipo, clase o rango.
3)Un argumento fingido no puede tener el atributo INTENT.
A continuación, un programa que manda llamar dos procedimientos con argumentos fingidos que son apuntadores.

```
program apuntadores
interface                              ! interfase para los procedimientos
        subroutine crea_arreglo(m,x)
        integer,intent(in)::m
        integer::error
        real,pointer,dimension(:)::x
        end subroutine crea_arreglo
        subroutine destruye_arreglo(y)
```

```
        real,dimension(:),pointer::y
        end subroutine destruye_arreglo
end interface
real,pointer,dimension(:)::array
read *,n
call crea_arreglo(n,array)
print *,array
call destruye_arreglo(array)
stop 'FIN DE PROGRAMA'
end program apuntadores
subroutine crea_arreglo(m,x)
! creación de un arreglo x de m elementos
integer,intent(in)::m
integer::error
real,pointer,dimension(:)::x
allocate(x(m),stat=error)
if(error/=0)stop 'Error en crea_arreglo'
X(1:m)=100.0_4
return
end subroutine crea_arreglo
subroutine destruye_arreglo(y)
! Destrucción de un arreglo y
real,dimension(:),pointer::y
integer::error
deallocate(y,stat=error)
if(error/=0)stop 'Error en destruye_arreglo'
return
end subroutine destruye_arreglo
```

El uso de apuntadores implica varias ventajas sobre los arreglos ALLOCATABLE. Estos últimos no pueden ser usados en definiciones de tipos derivados, ni ser el resultado de una función, tampoco pueden ser argumentos fingidos y deben ser creados y eliminados en la misma unidad de programa.
El resultado de una función puede ser un apuntador. Para tal efecto, la opción RESULT debe ser empleada en la definición de la función y la variable que contiene el resultado tiene que declarada como un apuntador:

```
module prueba
contains
        function apuntador_impares(vector) result(p)
        real,dimension(:),pointer::vector
        real,dimension(:),pointer::p
        ! El resultado es un apuntador que apunta a los elementos impares de un arreglo
        p=>vector(1::2)
```

```
        end function apuntador_impares
end module prueba
```

TÓPICOS ADICIONALES DE FORTRAN 90

Manejo de cadenas de caracteres

Una constante o literal de caracteres es una cadena de caracteres encerrada entre ' o "

'ESIQIE' "Programas FORTRAN"

El número de caracteres encerrados entre ' o entre " es la longitud de la constante.
Las variables de caracteres son declaradas con la proposición CHARACTER.

```
character(kind=1,len=10) var                ! clase 1 y longitud 10

character(len=30)::dirección                ! kind=1 es el default

character clave                             ! len=1 es el default

character(kind=5),dimension(100)::codigo postal
```

En FORTRAN 77 no se puede especificar la clase y la longitud se declara con un *

```
character *40 alumno
```

Es posible manejar subcadenas de caracteres, como en el programa

```
program sub_cadenas
character(len=10)::var;character(len=13) suma
car='FORTRAN 90'
! car(5:6) es una subcadena con los caracteres RA
print *, car(3:6)
print *,car(1:1)
print *,car(4:10)
suma="abc"//car                             ! // es un operador de concatenación
print *,suma
end program sub_cadenas
```

cuyos resultados son:

```
RA
F
```

```
TRAN 90
abcFORTRAN
```

La operación de concatenación se ejecuta después de las sumas y restas y antes de los operadores de relación ==,/=, <,<=,> y >=.
La función de biblioteca LEN determina la longitud de una constante, variable o expresión de tipo CHARACTER. INDEX determina la posición inicial de una subcadena dentro de una cadena de caracteres. La función CHAR genera el carácter correspondiente a su argumento entero mientras que ICHAR devuelve el número asociado con el carácter dado. El código de caracteres empleado tanto por ICHAR como por CHAR es dependiente del procesador, aunque generalmente es el código ASCII. Por ejemplo, el programa

```
character(len=6)::nom='ESIQIE'
print *,len(nom)
print *,index(nom,'I')
print *,char(38)
print *,ichar("%")
stop
end
```

genera los registros

```
       6
       3
&
      37
```

El argumento opcional BACK de la función INDEX no está soportado en FORTRAN 77:

```
index('abcde','c')                          ! devuelve el valor 3
! back=.true. implica una búsqueda hacia atrás
index("Matematicas",'a',back=.true)         ! regresa el valor 10
index('Gauss','xyz')                        ! devuelve el valor 0
```

La función LGE prueba si una cadena de caracteres es léxicamente mayor o o igual que otra, de acuerdo a la secuencia de los caracteres ASCII. Con LGT se prueba si una cadena es léxicamente mayor que otra. Similarmente, LLE prueba si una cadena es léxicamente menor o igual que otra y LLT determina si una cadena es menor léxicamente que otra. Por lo tanto, el programa

```
program lexico
```

```
! ¿'raul' es mayor o igual que 'raul'?
print *,lge('raul','raul')
! ¿ 'ana' es mayor léxicamente que 'pedro'?
print *,lgt('ana','pedro')
! ¿ 'FORTRAN' es menor o igual que 'fortran'?
print *,lle('FORTRAN','fortran')
! ¿ 'MARIA' es léxicamente menor que 'JULIA'?
print *,llt('MARIA','JULIA')
end program lexico
```

arroja los resultados

```
T
F
T
F
```

ACHAR es una función que devuelve el carácter de la secuencia ASCII que corresponde a su argumento entero. Dado un carácter del código ASCII, IACHAR devuelve su correspondiente clave numérica. LEN_TRIM funciona de manera similar a LEN; determina la longitud de una cadena, pero sin tomar en cuenta los espacios después del último carácter diferente de un espacio. La función ADJUSTL justifica una cadena a la izquierda, removiendo los espacios colocados antes del primer carácter diferente de un espacio en la cadena original e insertando los espacios necesarios para rellenar la cadena. ADJUSTR justifica la cadena a la derecha, removiendo los espacios colocados después del último carácter diferente a un espacio en la cadena original e insertando los espacios necesarios al principio de la cadena. REPEAT concatena copias de una cadena. La función TRIM omite los espacios al final de la cadena. Entonces, el programa

```
character (len=10)::muestra
character (len=30)::muestra_repetida
character (len=10)::muestra_sin_espacios
muestra="EJEMPLO"
print *,achar(60)
print *,iachar('k')
print *,len_trim(muestra)
print *,adjustl(muestra)                    ! justificación a la izquierda
print *,adjustr(muestra)                    ! justificación a la derecha
muestra_repetida=repeat(muestra,3) !concatenación de tres
print *,muestra_repetida
! omisión de los espacios al final de la cadena
```

```
muestra_sin_espacios=trim(muestra)//'123'
print *,muestra_sin_espacios
end
```

```
tendría los resultados

<
     107
       7

EJEMPLO
   EJEMPLO
EJEMPLO   EJEMPLO   EJEMPLO
EJEMPLO123
```

```
Las funciones ACHAR, IACHAR, LEN_TRIM, ADJUSTL, ADJUSTR,
REPEAT y TRIM no son parte de FORTRAN 77.
Cuando en la función SCAN  el argumento opcional BACK está
ausente o presente con el valor .false., el resultado es la
posición del carácter más a la izquierda en su primer
argumento que aparece también en el segundo argumento. Con
BACK=.true., el valor de la función es la posición del
carácter más a la derecha en el primer argumento que también
aparece en el segundo. La función VERIFY trabaja de manera
muy similar, pero con la diferencia de que el resultado es
ahora la posición del carácter del primer argumento que no
está en el segundo. SCAN y VERIFY no están incluidas en el
estándar FORTRAN 77. El siguiente segmento de programa
```

```
print *,scan('POLITECNICO',"FI")
print *,scan('POLITECNICO','I',back=.true.)
print *,verify('POLITECNICO','POB')
print *,verify('POLITECNICO',"ICONO",.true.)
end
```

```
escribe

       4
       9
       3
       6
```

```
Otros programas que muestran el uso de cadenas de caracteres
son:
```

```
character *5 nombre_corto
character (len=10) nombre_largo
real::a=10.0
```

```
! se truncan los cuatro caracteres más a la derecha
nombre_largo='abcdefghijklmn'
write(6,70)nombre_largo
nombre_corto=nombre_largo
! 227 es la clave ASCII de 'π'
write(6,80)a,achar(227),nombre_corto
70 format(1x,a15)
80 format(1x,f8.2,2x,a1,2x,a5)
stop
end

con resultados

      abcdefghij
   10.00  π  abcde
12345678901234567890

y

program principal
implicit none
character (len=20)::titulo
read *,titulo
call subrutina_uno(titulo)
end
subroutine subrutina_uno(linea)
implicit none
! la longitud del argumento fingido linea se supone igual a la del argumento actual titulo
character (len=*) linea
print '(1x,a20)',linea
return
end

program prueba_caracteres
implicit none
character(len=50),dimension(30)::direcciones
        .
        .
call proceso(direcciones)
        .
        .
end program prueba_caracteres
subroutine proceso(datos)
implivit none
! datos es un arreglo de tamaño supuesto
character (len=50),dimension(*)::datos
```

```
        .
        .
end subroutine proceso
```

Extensión del significado de los operadores estándar

Todos los operadores estándar, ya sean binarios o unitarios, pueden ser extendidos, en cuanto su significado, en FORTRAN 90. Para ilustrar esta situación, considere el siguiente programa; el operador unitario + es redefinido de tal manera que ahora convierte una cadena con caracteres en minúsculas en otra con los mismos caracteres pero en mayúsculas. En esta situación, hay necesidad de escribir una interfase INTERFACE OPERATOR. Cabe aclarar que el significado original del operador + no se pierde y puede seguir siendo usado en su contexto original.

```
program operadores
! extensión del significado del operador +
interface operator (+)
        function convierte_a_mayusculas(cadena)&
                result (cadena_nueva)
        character (len=20)::cadena,cadena_nueva
        end function convierte_a_mayusculas
end interface
character (len=20)::cadena_1
read '(a20)',cadena_1
read *,x,y
print *,cadena_1
! se emplea el nuevo significado adicional de +
print *,+cadena_1
A=+56
print *,a                                        ! significado estándar de +
print *,x+y
end program operadores
function convierte_a_mayusculas(cadena) result (cadena_nueva)
character (len=20)::cadena,cadena_nueva
cadena_nueva=cadena
do i=1,len_trim(cadena)
        select case(cadena(i:i))
        case('a':'z')
           cadena_nueva(i:i)=achar(iachar(cadena(i:i))-32)
        end select
end do
```

```
end function convierte_a_mayusculas
```

Solamente se pueden usar funciones para redefinir un operador. Al definir un operador unitario, la función empleada deberá ser de un sólo argumento fingido. En el caso de operadores binarios; el primer argumento representa la expresión que precede al operador mientras que el segundo argumento representa la expresión que sigue al operador.

```
module definiciones
        type datos                          ! el módulo definiciones incluye la
          sequence                          ! definición del tipo datos
          real::operando_1,operando_2
        end type datos
end module definiciones
program operador_binario
use definiciones                            ! uso del módulo definiciones
interface operator(*)                       ! extensión del operador *
        function producto(x,y)
        implicit none
        use definiciones
        type (datos),intent(in)::x,y
        type (datos)::producto
        end function producto
end interface
type(datos)::var_uno,var_dos,var_prod
var_uno=datos(3.4,0.98)
var_dos=datos(1.27,-2.8)
var_prod=var_uno*var_dos                    ! uso del operador * extendido
print *,var_uno;print *,var_dos;print *,var_prod
end program operador_binario
function producto(x,y)
use definiciones
implicit none
type(datos),intent(in)::x,y
type(datos)::producto
producto%operando_1=x%operando_1*y%operando_1
producto%operando_2=x%producto_2*y%operando_2
end function producto
```

La extensión del significado de un operador no puede ser usada para redefinir completamente el rol normal del operador. Por ejemplo; el operador de concatenación puede ser extendido para manejar dos arreglos reales, pero no para emplear variables de caracteres de una nueva manera. Tres restricciones limitan la redefinición de un operador:

1)Un operador existente no puede ser cambiado para operar un número diferente de argumentos. Por ejemplo; * no puede ser redefinido como un operador unitario y tampoco .not. puede ser redefinido como un operador binario.
2)Los argumentos fingidos utilizados en la función que redefine un operador deben ser declarados con el atributo INTENT(IN).
3)Los argumentos fingidos usados en la función empleada en la redefinición del operador no pueden ser declarados con el atributo OPTIONAL.
Al redefinir el operador de relación >, también se afecta al operador sinónimo .LT.. Esto es aplicable a todos los operadores de relación.
Los operadores unitarios definidos tienen un orden de precedencia (jerarquía) más alto que cualquier operador intrínseco y los operadores binarios definidos tienen un orden de precedencia más bajo que cualquier operador intrínseco.

Creación de nuevos operadores

En FORTRAN 90 es posible que el programador pueda definir sus propios operadores. El proceso implica codificar una función de argumentos fingidos con INTENT(IN) y la escritura de un bloque de interfase que mapea la función al nombre deseado. El primer argumento para un nuevo operador binario corresponde al operando de la izquierda y el segundo argumento mapea al operando de la derecha. Los nombres de los operadores toman la forma .X., donde X es una cadena 1 a 31 caracteres alfanuméricos. Por supuesto, no se permiten los nombres .true. y .false. para evitar confusiones con las constantes lógicas.

```
program operador_definido
implicit none
interface operator (.inv.)                    ! definición del operador .inv.
        function inversa(matriz)
        implicit none
        real,dimension(2,2)::matriz_inversa
        end function inversa
end interface
real,dimension(2,2)::A,Ainv
integer i
read *,A
Ainv=.inv.A                                   ! cálculo de la inversa de A, utilizando el
do i=1,2                                      ! operador .inv.
```

```
        print *,A(i,1:2)
end do
do i=1,2
        print *,Ainv(i,1:2)
end do
end program operador_definido
function inversa(B)
implicit none
! Cálculo de la inversa de una matriz de 2X2
real,dimension(2,2)::B,inversa
real det
det=B(1,1)*B(2,2)-B(2,1)*B(1,2)
if(abs(det)<=1e-10)then
        print *,'Matriz singular'
        stop
else
        inversa(1,1)=B(2,2)
        inversa(1,2)=-B(1,2)
        inversa(2,1)=-B(2,1)
        inversa(2,2)= B(1,1)
        inversa=inversa/det
end if
end function inversa
```

Extensión del operador de asignación

FORTRAN 90 permite que el operador de asignación = sea extendido en su significado. La interfase INTERFACE ASSIGNEMENT asocia una subrutina con el operador =. Esta subrutina debe tener exactamente dos argumentos no opcionales. El primero de ellos debe tener el atributo INTENT(OUT) o INTENT(INOUT) y corresponde al lado izquierdo de la asignación. El segundo argumento debe tener el atributo INTENT(IN) e incumbe al lado derecho de la asignación.

```
program asignacion
implicit none
interface assignement(=)
        subroutine convierte(texto,numero_entero)
        implicit none
        ! El operador = es asociado con la subrutina convierte
        character (len=15),intent(out)::texto
        integer (kind=4),intent(in)::número_entero
        end subroutine convierte
```

```
end interface
character (len=15)::cadena
integer j
read *,j
print *,j
cadena=j                          ! se usa el significado adicional de =
m=785                             ! significado estándar de =
print *,cadena
print *,m
end program asignacion
subroutine convierte (texto,numero_entero)
implicit none
! Esta subrutina convierte un número entero en una cadena de caracteres, justificada
! a la izquierda, con la correspondiente representación en base hexadecimal.
character(len=15),intent(out)=::texto
integer (kind=4),intent(in)::número_entero
character (len=15) caracteres
write(caracteres,'(z15)')numero_entero
texto = adjustl(caracteres)
end subroutine convierte
```

En la extensión del operador de asignación, ambos argumentos no pueden ser numéricos, LOGICAL o la misma clase del tipo CHARACTER.

Atributo PARAMETER. Proposición PARAMETER

El atributo PARAMETER se usa para asignar un nombre a una constante de determinado tipo:

```
real,parameter::pi=3.141592_4

integer,parameter::n=40,j=-500

complex,parameter::i=(0.0,1.0)

character (len=5),parameter::nombre='ABCDE'

logical,parameter::respuesta_afirmativa=.true.

integer,parameter::l=10,k=l+1
```

Dado que el identificador especificado en el atributo PARAMETER es el nombre de una constante; instrucciones como las siguientes no son permitidas:

```
real,parameter::x=-20.75,var=.02e-3
read *,x   ! x y var son constantes
var=var+10     ! error
```

Las constantes con nombre pueden ser empleadas en la declaración del tamaño de arreglos.

```
program arreglo_parameter
implicit none
integer k
integer,parameter::m=3,n=4
real,dimension(m,n)::x
real,dimension(m+1)::y
x=reshape((/(k,k=1,m*n)/),(/m,n/))
data (y(i),i=1,m+1)/.10,-.30,7.8,5.89/
        .
        .
end program arreglo_parameter
```

La proposición PARAMETER (declarativa) puede ser usada en vez del atributo PARAMETER.

```
real pi
integer n,j
parameter (pi=3.141592_4,n=40)
parameter (j=-500)
```

FORTRAN 77 no permite atributo alguno, pero si contempla el uso de la proposición PARAMETER:

Atributo SAVE. Proposición SAVE

Los valores de las variables y arreglos locales en un procedimiento se pierden cuando el control regresa a la unidad de programa que realizó la llamada; de tal forma que en subsecuentes llamadas a ese procedimiento, los valores de las variables y arreglos locales estarán indefinidos. Sin embargo, es posible conservar todos o algunos de los valores de las variables locales del procedimiento mediante el atributo SAVE. Tal situación es mostrada en el programa:

```
program salva_variables_locales
implicit none
integer i,j
real x
integer,parameter::n=3
```

```
ciclo_lectura:do i=1,n
                        read *,x
                        call sub(i,x)
                    end do ciclo_lectura
end program salva_variables_locales
subroutine sub(j,x)
real,save::sx                          ! la variable local sx tiene el atributo SAVE
if(j/=1)then
        sx=sx+x
else
        sx=x
end if
print *,'La suma acumulada es=',sx
return
end subroutine sub
```

el cual, para los datos x=10,20 y 30, imprimiría

```
La suma acumulada es= 10.0000
La suma acumulada es= 30.0000
La suma acumulada es= 60.0000
```

Cuando a variables o arreglos locales le son asignados valores iniciales en una declaración de tipo, estos son automáticamente retenidos. Por lo tanto

```
real::sum=0.0
```

es equivalente a

```
real,save::sum=0.0
```

La proposición SAVE (declarativa) puede también ser usada para retener los valores de variables y arreglos locales en un procedimiento.

```
subroutine abc(x,y,z)
dimension h(10,10)
save f,g,h
        .
        .
end subroutine abc

function muestra(var_1,var_2)
real (kind=8)::muestra
! se conservan los valores de todas las variables y  arreglos locales
```

```
save
        .
        .
end function muestra

program principal
save                                    ! un save en el programa principal es ignorado
        .
        .
call muestra(5.0,12.4,z)
        .
        .
end program principal
subroutine muestra(a,b,s)
        .
        .
end subroutine muestra
```

Para garantizar que los valores de las variables de un bloque COMMON etiquetado sean conservados, se utiliza SAVE:

```
subroutine prueba
common /xyz/f(10),a_uno,a_dos
save /xyz/              ! es lo mismo que save f,a_uno,a_dos
        .
        .
        .
end subroutine prueba
```

Instrucción ENTRY

La proposición ENTRY permite entrar a un procedimiento en un punto diferente de la primera instrucción ejecutable después de SUBROUTINE o FUNCTION. El punto de entrada es definido por un nombre y pueden usarse argumentos fingidos. De esta forma, un procedimiento puede tener más de un punto de entrada. ENTRY es no ejecutable. Para mostrar el uso de ENTRY, se puede emplear el programa

```
program p10
a=2.0
call cuadra(a)
b=2.0
call sub(b)
print *,a,b
```

```
end
subroutine sub(x)
x=x*x
entry cuadra(x)
x=x*x
return
end
```

que escribiría los registros:

```
 4.0000
16.0000
```

Otro programa que ilustra el empleo de la instrucción ENTRY es

```
program p20
implicit none
real a,g,h,fun,z
a=2.0
h=g(a)
z=fun(3.0,4.0)
print *,h,z
end
function fun(x,y)
implicit none
real x,y,fun,z,g
fun=sqrt(x**2+y**2)
z=fun
return
entry g(z)
g=z*10
return
end
```

con resultados:

```
20.0000   5.0000
```

En FORTRAN 90 no se recomienda usar la instrucción ENTRY porque en él hay herramientas que producen un código mucho más claro, como en:

```
module procedimientos
contains
        function fun(x,y)
```

```
        implicit none
        real x,y,fun
        fun=sqrt(x**2+y**2)
        return
        end function fun
        function g(z)
        implicit none
        real a,g,h,fun,z
        g=z*10
        return
        end function g
end module procedimientos
program p20_nuevo
use procedimientos
a=2.0
h=g(a)
z=fun(3.0,4.0)
print *,h,z
end program p20_nuevo
```

Regresos alternativos de una subrutina (característica obsoleta)

En una subrutina, al ejecutarse un RETURN; el control pasa a la siguiente instrucción ejecutable después del CALL que hizo la llamada a la subrutina. Esta situación puede modificarse usando regresos alternativos, como en el programa:

```
read *,x
call regresos(x,resultado,*20,*10)
 20 print *,'El seno de',x,' es negativo'
print *,resultado
stop
10 print *,'El seno de',x,' es mayor o igual que cero'
print *,resultado
stop
end
subroutine regresos(y,z,*,*,*)
z=sin(x)
if(z<0)then
        return 1
else
        return 2
end if
```

```
end subroutine regresos
```

En el CALL se indican, precedidos de un *, las etiquetas que representan los puntos de regreso de la subrutina en la unidad de programa que efectuó la llamada. El primer regreso corresponde al indicado por return 1, el segundo por return 2, y así sucesivamente.
Es recomendable no usar regresos alternativos en FORTRAN 90, pues en éste se cuenta con instrucciones que generan un código totalmente estructurado.

```
read *,x
call muestra_estructurada(x,resultado)
if(resultado<0)then
        print *,'El seno de',x,' es negativo'
else
        print *,'El seno de',x,' es mayor o igual que cero'
end if
print *,resultado
stop
end
subroutine muestra_estructurada(y,z)
z=sin(x)
end subroutine muestra_estructurada
```

Aspectos adicionales de lectura/escritura

El trazador de edición T (tabulador) permite "brincar" directamente a una columna en particular, dentro de un registro de lectura o de escritura. Por ejemplo, la instrucción

```
read 18,f,k
18 format(t10,f6.1,t20,i3)
```

requiere de la línea de datos

```
          2.78     500
12345678901234567890123456789
```

en la cual; los 6 espacios para leer el valor de la variable f comienzan en la columna 10 y a partir de la columna 20 se dispone de 3 columnas para leer el valor de la variable k.
El segmento de programa

```
a=100.0
```

```
write(6,230)a
230 format(t4,f10.2)
```

genera la cadena de salida

```
       100.00
12345678901234567890
```

donde las 10 columnas apartadas para el valor de a inician en la columna 4. Después de interpretar el control de carro, el registro impreso queda

```
      100.00
12345678901234567890
```

Con TLn y TRn el siguiente carácter es leído o impreso n columnas hacia la derecha o hacia la izquierda a partir de la posición actual, respectivamente. Así, las instrucciones

```
a=100.0_4;b=200.0_4;c=300.0_4
write(6,57)b,c
write(6,89)a,b
57 format(t21,f10.2,tl20,f10.2)
89 format(1x,f10.2,tr5,f10.2)
```

imprimen las líneas

```
             300.00    200.00
    100.00         200.00
12345678901234567890123456789
```

Nótese que TRn es equivalente a nX.
Con el trazador TLn es posible leer varias veces un mismo dato en el registro de lectura. Por ejemplo, para una línea de datos:

```
500
```

la instrucción

```
read(5,'(i3,tl3,f3.1,tl3,f3.2)')n,x_uno,x_dos
```

hace que n=500, x_uno=50.0 y x_dos=5.00.

Para terminar el control sobre la edición de datos cuando no hay más elementos en la lista de lectura/escritura, se emplea el trazador : ,como en las instrucciones

```
Write(6,10)5.0
write(6,10)5.0,6.0
10 format(1x,'cinco=',f3.1:1x,'seis=',f3.1)
```

que producen los registros

```
cinco=5.0
cinco=5.0 seis=6.0
```

SP es un trazador de edición que imprime un + delante de números positivos en todos los datos posteriores. SS o S no producen un + en cualquier dato subsecuente que sea positivo.
Tanto SP como SS o S sólo tienen efecto en instrucciones de escritura. De acuerdo a lo anterior, las instrucciones

```
x=120.0;y=890.0;z=678.0
print 5679,x,y,z
5679 format(1x,sp,2f10.2,3x,ss,f10.2)
```

entonces imprimirían

```
   +120.00   +890.00       678.00
```

Un factor de escala tiene la forma sP y se utiliza en combinación con los trazadores F, E, D y G, donde s es una constante entera que puede ser positiva, cero o negativa. En lectura (sin especificar el exponente) y en escritura con el trazador F; el efecto del factor de escala k es tal que el número representado externamente es igual al guardado en la memoria multiplicado por 10^k. Por ejemplo, la instrucción

```
read(5,1000)auxiliar
1000 format(3pf10.3)
```

con el registro de lectura

```
   3500.0
1234567890
```

implica que el valor realmente almacenado en la variable auxiliar sea 3.5000, mientras que

```
z10=-47.83
write(6,95)z10
95 format(1x,2pf10.2)
```

escribe el número

-4783.00

Los trazadores BN y BZ solamente afectan la edición con trazadores I, F, E, D y G en instrucciones de lectura. BN implica que los espacios encontrados en los siguientes campos sean ignorados (condición por "default"). BZ permite que los espacios encontrados en los campos subsecuentes sean considerados como ceros. Por lo tanto, la instrucción de lectura

```
read(5,109)i,j,k
109 format(i5,bz,i5,i4)
```

con registro

```
5     78  1
12345678901234I
```

implica que i=5, j=7800 y k=1000.

Línea INCLUDE

INCLUDE es una facilidad provista por FORTRAN 90, que permite insertar texto dentro de una unidad de programa en un punto dado. Su empleo es:

```
INCLUDE cadena de caracteres
```

donde cadena de caracteres típicamente indica el nombre de un archivo que contiene el texto por insertar. En el siguiente código

```
program muestra_include
common /datos/a,b,k
print *,a,b,k
end
include 'bldata.for'
```

suponiendo que 'bldata.for' es un archivo cuyo contenido es

```
block data
common /datos/a,f,j
data a,f,j/100.27,1.5e2,-50/
end
```

entonces el conjunto de instrucciones que procesará el

compilador son:

```
program muestra_include
common /datos/a,b,k
print *,a,b,k
end
block data
common /datos/a,f,j
data a,f,j/100.27,1.5e2,-50/
end
```

Cabe aclarar que INCLUDE no es una instrucción FORTRAN 90.

Ámbito. Unidades de ámbito

El ámbito (scope) de un objeto (una variable, una constante con nombre, un procedimiento, una etiqueta, etc.) es la porción de un programa FORTRAN en la cual está definido. Existen tres niveles de ámbito: global, local y de instrucción.
1)Ámbito global. Los objetos con ámbito global so aquellos que están definidos en todo un programa. Los nombres de estos objetos deben ser únicos dentro del programa. Los nombres de programa, procedimientos externos y módulos tienen ámbito global.
2)Ámbito local. Los objetos locales son aquellos que están definidos dentro de una unidad de ámbito. Ejemplos de unidades de ámbito son los programas principales, los procedimientos externos y los módulos. Un objeto local dentro de una unidad de ámbito debe ser único, pero puede ser empleado en alguna otra unidad de ámbito.
3)Ámbito de instrucción. El ámbito de ciertos objetos puede estar restringido a una sola instrucción. El único caso de ámbito de instrucción en FORTRAN 90 se da en el índice de un DO implícito de un constructor de arreglos; como en

```
vector=(/(i+5,i=1,10)/)
```

donde i tiene un ámbito de instrucción. Esto implica que la variable i puede ser usada en cualquier otra situación dentro de la misma unidad de programa ya que sería independiente del índice del DO implícito.
En FORTRAN 90, las unidades de ámbito pueden ser:
1)Un programa principal, un procedimiento interno o externo, un módulo (excluyendo cualquier definición de tipos derivados y a los procedimientos contenidos en él).
2)Una definición de tipos derivados.

3)Una interfase.
Dado que una definición de un tipo derivado es por si sola una unidad de ámbito, entonces es posible emplear una variable dada como componente en la definición del tipo derivado y también tener otra variable con el mismo nombre dentro del programa que contiene la definición del tipo derivado, sin causar conflicto alguno.
Si una unidad de ámbito contiene completamente a otra unidad de ámbito, se dice que la primera es la unidad anfitriona, o simplemente el anfitrión, de la segunda. La unidad de ámbito más interna automáticamente hereda las definiciones de objetos declaradas en la unidad anfitriona, a menos que en la unidad interna sen redefinidos dichos objetos. Este tipo de mecanismo se conoce asociación con el anfitrión.
Los objetos definidos dentro de un módulo normalmente tienen como ámbito ese mismo módulo. Sin embargo, ese ámbito puede ser extendido más allá del módulo mediante asociación por uso: si el nombre del módulo aparece en una instrucción USE en una unidad de programa, entonces los objetos del módulo quedan como si hubieran sido definidos en la unidad de programa. Si un objeto con un nombre dado es declarado dentro de un módulo y ese módulo es usado en otra unidad de programa, entonces ningún otro objeto en esa unidad de programa puede tener dicho nombre.

Funciones intrínsecas EPSILON, TINY, HUGE, PRECISION y RANGE

El intrínseco EPSILON(X) devuelve un número positivo, del mismo tipo y clase de X, que es casi indistinguible de 1.0. Más precisamente; EPSILON(X) es el menor número posible tal que (EPSILON(X)+1.0)>1.0. El argumento X debe ser real.
TINY(X) y HUGE(X) se emplean para conocer el número más pequeño y más grande, respectivamente, del mismo tipo y clase que X. El argumento de TINY o HUGE puede ser real o entero.
A manera de ejemplo, la siguiente función constituye una alternativa razonable para probar la igualdad de dos números reales:

```
function igualdad_a_b(a,b)
implicit none
logical::igualdad_a_b
real,intent(in)::a,b
real::eps
eps=abs(a)*epsilon(a)                          ! escalamiento de a
if(eps==0)then
        eps=tiny(a)
```

```
end if
if(abs(a-b)>eps)then
        igualdad_a_b=.false.
else
        igualdad_a_b=.true.
end if
return
end function igualdad_a_b
```

PRECISION(X) se puede usar para determinar el número de dígitos decimales significativos en valores del mismo tipo y clase que X. RANGE(X) regresa el intervalo del exponente decimal en valores del mismo tipo y clase que X. El argumento X puede ser entero, real o complejo. Cuando X es entero, RANGE devuelve el valor $\log_{10}$(HUGE(X)). Para X real o complejo, el resultado de RANGE es el mínimo entre $\log_{10}$(HUGE(X)) y -$\log_{10}$(TINY(X))). Por ejemplo, las instrucciones

```
real(kind=4)::x_simple=12.0_4
real(kind=8)::x_doble=12.0_8
print *,precision(x_simple),precision(x_doble)
print *,range(x_simple),range(x_doble)
```

compiladas con Lahey FORTRAN 90 entonces generan los resultados

```
 6     15
37    307
```

FORTRAN 77 no incluye a los intrínsecos EPSILON, HUGE, TINY, PRECISION y RANGE.

Subrutinas intrínsecas DATE_AND_TIME y SYSTEM_CLOCK

La subrutina DATE_AND_TIME determina la fecha y la hora actual:

```
DATE_AND_TIME(DATE,TIME,ZONE,VALUES)
```

donde el argumento DATE debe ser tipo CHARACTER de longitud por lo menos ocho. El resultado devuelto en DATE tiene un formato ccyymmdd; donde cc es la centuria, yy el año dentro de la centuria, mm el mes del del año y dd el día del mes. TIME también es de tipo CHARACTER, pero de longitud al menos diez. El formato para TIME es hhmmss.sss; donde hh es la

hora del día, mm son los minutos y ss.sss son los segundos y milisegundos del minuto. ZONE devuelve una cadena de caracteres en la forma ±hhmm, donde hhmm es la diferencia entre el tiempo local y el tiempo GMT(Greenwich Mean Time). Si por alguna razón no están disponibles los resultados para DATE, TIME y ZONE, las cadenas de caracteres contendrán espacios. VALUES es un arreglo entero que devuelve la siguiente información:

```
VALUES(1)      centuria y año (p.e. 1998)
VALUES(2)      mes (1-12)
VALUES(3)      día (1-31)
VALUES(4)      diferencia con respecto al GMT, en minutos
VALUES(5)      hora (0-23)
VALUES(6)      minutos (0-59)
VALUES(7)      segundos (0-59)
VALUES(8)      milisegundos (0-999)
```

Si no hay información disponible para VALUES, el valor devuelto es -HUGE(0).
Todos los argumentos son opcionales, pero deberá especificarse por lo menos uno de ellos. DATE, TIME, ZONE y VALUES tienen el atributo INTENT(OUT).
SYSTEM_CLOCK permite obtener información del reloj de tiempo real de la computadora:

```
SYSTEM_CLOCK(COUNT,COUNT_RATE,COUNT_MAX)
```

donde COUNT es el número de cuentas del reloj del sistema. La cuenta inicial es arbitraria. COUNT_RATE da el número de cuentas por segundo y COUNT_MAX representa el máximo valor posible para el número de cuentas del reloj del sistema. Todos los argumentos son enteros con atributos OPTIONAL e INTENT(OUT). Cuando COUNT_MAX es alcanzado, el valor de COUNT es restablecido en cero. A continuación; una aplicación que permite conocer el tiempo de ejecución consumido entre dos puntos de un programa:

```
program tiempo
implicit none
real tiempo_de_ejecucion
integer cuentas_1,cuentas_por_segundo,maximo_de_cuentas
integer cuentas_2
      .
      .
call sytem_clock(cuentas_1,cuentas_por_segundo,&
                                        maximo_de_cuentas)
      .
```

```
        .
call system_clock(count=cuentas_2)
! real(x,kind) convierte un dato x a real de clase kind
tiempo_de_ejecucion=real((cuentas_2-cuentas_1),4)/&
                                                    cuentas_por_segundo
print *,'Tiempo de ejecución=',tiempo_de_ejecucion,' seg'
        .
        .
end program tiempo
```

Las subrutinas DATE_AND_TIME y SYSTEM_CLOCK son exclusivas de FORTRAN 90.

Función TRANSFER

Interpreta la representación física de un número con el tipo y parámetros de tipo de un número dado. Su sintaxis es:

TRANSFER(SOURCE,MOLD,SIZE)

SOURCE y MOLD son argumentos fingidos requeridos que pueden ser de cualquier tipo. SIZE es un argumento fingido opcional y debe ser un escalar de tipo entero. El argumento actual que corresponde a SIZE no debe ser un argumento fingido opcional.
El resultado de TRANSFER es del mismo tipo y parámetros de tipo que MOLD. Si MOLD es un escalar y SIZE está ausente, el resultado es un escalar. Si MOLD es un arreglo y SIZE no está presente, el resultado es un arreglo de rango 1 del menor tamaño posible (pero no más pequeño que SOURCE). En caso de que SIZE esté presente, el resultado es un arreglo de rango 1 de tamaño SIZE. Ejemplos:

transfer(4.0,0)

devuelve el valor 1082130432 en una computadora personal porque la representación interna de la constante de punto flotante 4.0 y la del entero 1082130432 son idénticas.

transfer((/1.0,2.0,3.0/),(/(0.,0.)/))

aquí el argumento actual correspondiente a SOURCE es una arreglo real de tamaño tres y el argumento actual asociado con MOLD es un arreglo de rango 1 de solo un número complejo. Entonces, el resultado de TRANSFER será un arreglo complejo de rango 1 con primer elemento (1.0,2.0) y como segundo elemento el número complejo con parte real 3.0 y

parte imaginaria indefinida.

transfer((/1.0,2.0,3.0/),(/(0.,0.)/),1)

este caso difiere del anterior en que ahora se proporciona un argumento actual correspondiente a SIZE. Por lo tanto, el resultado es el arreglo complejo de rango 1 de un solo elemento: (1.0,2.0).
El estándar FORTRAN 77 no incluye a la función TRANSFER.

Generación de números aleatorios

La subrutina intrínseca RANDOM_NUMBER(HARVEST) genera números seudo-aleatorios uniformemente distribuidos en el intervalo $0 \leq x < 1$. HARVEST debe ser un escalar o un arreglo de tipo real y es un argumento con atributo INTENT(OUT). Por ejemplo, las instrucciones:

```
real,dimension(100)::xnum
call random_number(xnum)
```

forman un vector xnum de 100 número aleatorios. RANDOM_SEED(SIZE,PUT,GET) es una subrutina intrínseca que trabaja con el generador de números seudo-aleatorios. SIZE es un entero, PUT y GET son arreglos enteros. Todos los argumentos son opcionales y a lo más uno puede ser especificado en una llamada. SIZE y GET son INTENT(OUT) y PUT es INTENT(IN):

```
integer,dimension(100)::semilla,seed
call random_seed()                                  ! inicia el generador
! j es el número de enteros usados para guardar la semilla
call random_seed(size=j)
read *,semilla(1:j)
! se coloca una nueva semilla
call random_seed(put=semilla(1:j))
! impresión de los valores de la semilla
call random_seed(get=seed(1:j))
print *,seed(1:j)
```

FORTRAN 77 no soporta a RANDOM_NUMBER ni a RANDOM_SEED.

Orden de las instrucciones en un programa FORTRAN 90

Los programas FORTRAN consisten de una o más unidades de

programas, las cuales contienen por lo menos dos instrucciones. Cualquier número y tipo de unidades pueden ser incluidas en un programa, sin embargo una de ellas, y solamente una, debe ser un programa principal.
La siguiente tabla muestra el orden en que deben aparecer las instrucciones en una unidad de programa. Las líneas horizontales indican instrucciones que no pueden ser mezcladas; las líneas verticales denotan a las instrucciones que si pueden ser entremezcladas.

<table>
<tr><td colspan="3">PROGRAM,FUNCTION,MODULE,SUBROUTINE o BLOCK DATA</td></tr>
<tr><td colspan="3">USE</td></tr>
<tr><td colspan="3">IMPLICIT NONE</td></tr>
<tr><td rowspan="3">FORMAT y ENTRY</td><td>PARAMETER</td><td>IMPLICIT</td></tr>
<tr><td>PARAMETER Y DATA</td><td>Definiciones de tipos derivados
Bloques de interfase
Declaraciones de tipo
Instrucciones de especificación
Funciones de proposición</td></tr>
<tr><td>DATA</td><td>Instrucciones ejecutables y construcciones</td></tr>
<tr><td colspan="3">CONTAINS</td></tr>
<tr><td colspan="3">Subprogramas internos o subprogramas de módulo</td></tr>
<tr><td colspan="3">END</td></tr>
</table>

NUEVOS ASPECTOS INCLUIDOS EN FORTRAN 95

La construcción WHERE en FORTRAN 95

El nuevo estándar permite que la construcción WHERE tenga más de una clausula ELSEWHERE:

```
WHERE(máscara 1)
      .
      instrucciones de asignación (bloque 1)
      .
ELSEWHERE(máscara 2)
      .
      instrucciones de asignación (bloque 2)
      .
ELSEWHERE(máscara 3)
      .
      .
      .
ELSEWHERE(máscara n)
      .
      instrucciones de asignación (bloque n)
      .
ELSEWHERE
      .
      instrucciones de asignación
      .
END WHERE
```

donde cada máscara es un arreglo lógico de la misma forma del arreglo especificado en las instrucciones de asignación. La construcción aplica las operaciones dentro del bloque 1 a todos aquellos elementos del arreglo para los cuales la máscara 1 es cierta, ejecuta las instrucciones del bloque 2 en los elementos del arreglo para los cuales es falsa la máscara 1 y verdadera la máscara 2 y así sucesivamente. No más de un bloque será ejecutado. Por ejemplo, el programa

```
implicit none
integer i
real,dimension(3,3)::arreglo
arreglo=reshape((/-3,-2,-1,0,1,2,3,100,400/),(/3,3/),&
  order=(/2,1/))
do i=1,3                                    ! escritura del arreglo original
       print*,a(i,1:3)
```

```
end do
print *
f95:where(arreglo<0.0)
        arreglo=-10*arreglo
elsewhere(arreglo>=0.0.and.arreglo<=3.0) f95
        a=500.0
elsewhere f95
        arreglo=sqrt(arreglo)
end where f95
do i=1,3                                    ! escritura del arreglo final
        print *,a(i,1:3)
end do
end
```

produciría la salida:

```
-3.0000   -2.0000     -1.0000
 0.0000    1.0000      2.0000
 3.0000    100.00    400.0000

30.0000   20.0000     10.0000
500.000   500.000    500.0000
500.000   10.0000     20.0000
```

Una construcción WHERE puede tener tantas cláusulas ELSEWHERE como sean necesarias.

La construcción FORALL

La nueva construcción FORALL está diseñada para aplicar ciertas operaciones elemento por elemento a subconjuntos de arreglos. Dichos elementos son escogidos en base a los valores que puedan tomar sus subíndices y a una condición lógica. Las operaciones serán ejecutadas solamente en aquellos elementos que satisfagan las restricciones impuestas por los subíndices y la condición lógica. La forma de la construcción FORALL es:

FORALL (indice1=tríada,índice=tríada,....,expresión lógica)

donde cada tríada se escribe como

valor inicial:valor final:incremento

y la expresión lógica emplea arreglos con algunos o todos los subíndices indicados en las tríadas. Si en una tríada, se omite el incremento, entonces el compilador supondrá el

valor de uno. Las instrucciones que forman el cuerpo de la estructura pueden ser asignaciones o asignaciones de apuntadores que manejen los arreglos con los subíndices especificados en las tríadas, construcciones o instrucciones FORALL y construcciones o instrucciones WHERE. Ejemplos:

```
real,dimension(10,10)::x
call random_seed()
call random_number(x)
x=0.0                                   ! Generación de la matriz identidad de orden 10
forall(i=1:10)
        x(i)=1.0
end forall

forall(i=1:n-1,k=1:n,matriz(k,i)/=0.0)
        matriz(k,i)=1.0/matriz(k,i)
end forall

real,allocatable,dimension(:,:)::a,b
read *,n
allocate(a(n,n),b(n,n))
read *,a
b=0.0
forall(i=2:n-1,j=2:n-1)
        a(i,j)=(a(i,j-1)+a(i,j+1)+a(i-1,j)+a(i+1,j))/4
        b(i,j)=a(i,j)
end forall
```

También es posible darle un nombre a la construcción FORALL:

```
integer,parameter::n=25
real,dimension(n,n)::rejilla,aux
read(*,*)n
rejilla=0.0;aux=1.0
ciclo:forall(l=1:n,j=2:n-1:2)
        rejilla(l,j)=50.0*sin(l*j*.10)
        aux(i,j)=rejilla(i,j)**2+1.0
end forall ciclo
```

la instrucción FORALL es útil cuando se trata de ejecutar una sola instrucción:

```
! construcción de la matriz de Gilbert de orden n
forall(i=1;n,j=1;n)h(i,j)=1.0/real(i+j-1)
```

Las instrucciones dentro de una construcción FORALL son ejecutadas en un orden seleccionado por el procesador, a diferencia del ciclo DO en el cual los elementos del arreglo son procesados en un orden muy estricto. Esta libertad es muy útil en computadoras con más de un procesador ya que el compilador tomará las acciones más adecuadas para generar el código más eficiente para el procesamiento en paralelo.

Procedimientos puros

Las funciones puras son aquellas que no presentan "efectos laterales". Esto es, que no modifiquen sus argumentos de entrada o datos contenidos en módulos. Además, sus variables locales no pueden tener el atributo SAVE o tomar valores iniciales en declaraciones de tipo. Dada la ausencia de efectos laterales, un función pura puede ser incluida dentro de una construcción FORALL. Cada argumento de una función pura debe ser declarado con el atributo INTENT(IN) y las funciones y subrutinas invocadas deberán ser también puras. Por otra parte, las funciones puras no deben incluir operaciones de lectura y escritura de archivos externas o instrucciones STOP. En FORTRAN 95 una función es declarada como pura mediante el prefijo PURE. Por ejemplo:

```
pure function media_logaritmica(a,b)
implicit none
real (kind=8),intent(in)::a,b
real (kind=8)::media_logaritmica
media_logaritmica=(a-b)/log(a/b)
return
end function media_logaritmica
```

Las subrutinas que no tienen efectos laterales son conocidas como subrutinas puras. Sus restricciones son las mismas de las funciones puras, excepto que se permite modificar argumentos declarados con el atributo INTENT(INOUT)o INTENT(OUT). Las subrutinas puras también son declarads como tales a través del prefijo PURE:

```
pure subrutine & medias(x,y,media_aritmetica,media_geometrica)
implicit none
real (kind=4),intent)in)
real (kind=4),intent(out)::,media_aritmetica,&
        media_geometrica
```

```
media_aritmetica=(x+y)/2
media_geometrica=sqrt(x*y)
return
end subroutine medias
```

Cualquier procedimiento invocado dentro de una construcción FORALL debe ser un procedimiento puro.

Procedimientos elementales

Las funciones elementales son aquellas que si bien están escritas para argumentos escalares, pueden en un momento dado aceptar que esos mismos argumentos sean arreglos. Si los argumentos actuales de una función elemental son arreglos, entonces el resultado será un arreglo de la misma forma que la de dichos argumentos.
Desde FORTRAN 90 están disponibles funciones de biblioteca elementales, tales como SIN o EXP, pero es a partir de FORTRAN 95 en que aparecen las funciones y los procedimientos elementales definidos por el usuario.
Las funciones elementales deben también ser funciones puras y se definen mediante el prefijo ELEMENTAL. Además deben con cumplir los requisitos adicionales:
1) Todos los argumentos fingidos deben ser escalares y no poseer el atributo POINTER.
2) El resultado de función también debe ser un escalar y no tener el atributo POINTER.
3) Los argumentos fingidos no deben ser usados en expresiones de especificación excepto como argumentos de los intrínsecos BIT_SIZE, KIND y LEN o de las funciones numéricas de indagación.Por ejemplo, en:

```
elemental function pp(x,y)
      .
dimension array(n)
      .
      .
end function pp
```

no es válido declarar array como un arreglo de longitud n. Esta restricción impide el uso de arreglos automáticos en funciones elementales.
El siguiente programa es una muestra del uso de funciones elementales:

```
program prueba_elementales
implicit none
```

```
interface
        elemental function funci(v)
        real (kind=8),intent(in)::v
        real (kind=8) funci
end function funci
end interface
real (kind=8)::x,y
real (kind=8) z
integer n
read *,n
read *,x(1:n)
y=funci(x)                          ! el argumento actual es un arreglo
z=funci(3.0_8)                      ! ahora el argumento actual es un escalar
print *,y(1:n);print *,z
stop
end program prueba_elementales
elemental function funci(x)
real (kind=8),intent(in)::x
real (kind=8)::funci
funci=x**2*sin(5*x)
return
end function funci
```

Es necesario incluir la información sobre la función elemental dentro de una interfase para poder emplear adecuadamente a la función.
Las subrutinas elementales son aquellas que están diseñadas para argumentos escalares pero que también pueden emplear arreglos como argumentos. Tienen las mismas restricciones de la funciones elementales. Para definir una subrutina elemental se usa nuevamente el prefijo ELEMENTAL:

```
elemental subroutine calcula(x,y,z)
implicit none
real,intent(in)::x
real,intent(out)::y,z
y=(x-1)**2
z=sin(x)+cos(2*x)
return
end subroutine calcula
```

Subrutina intrínseca CPU_TIME

En cada llamada de la subrutina CPU_TIME(t), es devuelto en el escalar real t el tiempo actual del procesador en

segundos. Si el procesador por alguna razón no puede proporcionar el tiempo actual, un valor negativo es regresado en t. A manera de ilustración, este programa determina el tiempo de ejecución de una parte del mismo:

```
program tiempo_ejecucion
implicit none
real t_inicial,t_final
      .
      .
call cpu_time(t_inicial)
      .
      .
      .
call cpu_time(t_final)
print *,'tiempo de ejecucion=',t_final-t_inicial,' s'
      .
      .
end program tiempo_ejecucion
```

Características adicionales contempladas en FORTRAN 95

Las funciones de localización MINLOC y MAXLOC ahora permiten el argumento opcional DIM, como en el código

```
program adicional
implicit none
integer i
real (kind=4),dimension(2,3)::m=reshape(order=(/2,1/),&
source=(/8.0,9.50,0.0,4.0,-1.3,7.6/),dim=1,shape=(/2,3/))
do i=1,2
  print *,m(i,1:3)
end do
print *;print *,minloc(m,dim=1)
end program adicional
```

cuya salida impresa es:

```
8.00000    9.50000   0.00000
4.00000   -1.30000   7.60000

2    2    1
```

Los intrínsecos CEILING y FLOOR soportan ahora el argumento opcional KIND. Por ejemplo; CEILING(14.7,kind=4) devuelve

el resultado 15 (el entero más pequeño mayor o igual que 14.7) mientras que con FLOOR(14.7,kind=4) se obtiene 14 (el entero más grande menor o igual que 14.7).
La nueva función NULL puede ser usada para definir que un apuntador inicialmente no está asociado:

```
real,pointer.dimension(:)::xyz=>null()
```

Aquí el argumento no es necesario, si está presente determina las características del apuntador, en caso contrario, estas son determinadas a partir del contexto.
En FORTRAN 95 es posible especificar valores iniciales por omisión para los componentes de un tipo derivado, como en:

```
type prueba
        real::valor=2.0_4
        integer::indice
end type prueba
```

La inicialización no necesariamente debe aplicarse a todos los componentes del tipo derivado.
Si el programador no ha eliminado explícitamente a los arreglos locales de tipo ALLOCATABLE con una instrucción DEALLOCATE, automáticamente la cancelación de estos es llevada a cabo al terminar la ejecución de la unidad de ámbito correspondiente.
Con el objetivo de optimizar el uso de las columnas disponibles en operaciones de escritura, ahora es posible solamente proporcionar el número de decimales, si es que los hay, y no el ancho total del campo en los trazadores B,F,I,O y Z. Por ejemplo, el programa

```
real::a=234.9578
integer::j=45
print '(1x,i0)',j                              ! Campos mínimos en escritura
print '(1x,f0.3)',a
end
```

generaría la impresión:

```
45
234.958
```

Ahora cualquier tipo de interfase puede tener un nombre y en END INTERFACE se puede indicar este. Los siguientes códigos ilustran tal situación:

```
program principal
```

```
implicit none
interface uno                          ! La interfase tiene el nombre uno
        subroutine prueba(x,y,z)
        implicit none
        real (kind=4),intent(in)::x,y
        real (kind=4),intent(out)::z
        end subroutine prueba
end interface uno                      ! en END INTERFACE se especifica el nombre
        .
        .
end program principal
subroutine prueba(x,y,w)
implicit none
real (kind=4),intent(in)::x,y
real (kind=4),intent(out)::w
        .
        .
end subroutine prueba

program ejemplo
interface operator (+)
        function suma(a,b)     ! Ahora el operador + se usa para concatenar 2 caracteres
        implicit none
        character,intent(in):: a,b
        character (len=2) suma
        end function suma
end interface operator (+)
character uno,dos
character (len=2) tres
read '(2a1)',uno,dos
tres=uno+dos
print *,tres
end program ejemplo
function suma(a,b)
implicit none
character,intent(in):: a,b
character (len=2) suma
suma=a//b
end function suma
```

Ejercicio

Solución numérica de la ecuación de Laplace.

La ecuación de Laplace en dos dimensiones

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

sujeta a las condiciones frontera

```
T(x,0)=100+20*sin(x),     0≤x≤B
T(x,B)=0,                 0≤x≤B
T(0,y)=T(B,y)=0,          0<y<B
```

es resuelta iterativamente. La siguiente ecuación se aplica para generar las nuevas estimaciones de T en función de las anteriores:

$$T_{i,j,nueva} = \frac{T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}}{4}$$

Pseudocódigo

```
!Solución numérica de la ecuación de Laplace
entero n,iter,i,j
real T[n,n],T_anterior[n,n]
real B=1.0,T0=20.0,deltax
lee n
!
! la placa es un cuadrado de lado B
!
deltax=B/n
ejecuta i=1,n     ! condiciones frontera
     T[1,i]=f(1.0/(i*deltax))
T[n,1:n]=0.0
T[2:n-1,1]=0.0
T[2:n-1,n]=0.0
T_anterior=T
T_anterior[2:n-1,2:n-1]=T0   ! T0 es una estimación inicial
ejecuta iter=1,200
empieza
     para(i=2:n-1,j=2:n-1)
     empieza
       T[i,j]=(T_anterior[i-1,j]+T_anterior[i+1,j]+ &
                  (T_anterior[i,j-1]+T_anterior[i,j+1])/4
     termina
     si MAXVAL(ABS(T[2:n-1,2:n-1]-T_anterior[2:n-1,&
        2:n-1]))<.01 entonces salida
```

```
      T_anterior=T
termina
si iter≤200 entonces
empieza
      ejecuta i=1,n
        escribe T[i,1:n]
      escribe 'Iteraciones realizadas=',iter
termina
otro
      escribe 'Sin convergencia'
fin
func f(x)
real x,f
f=100+200*sin(x)
regresa
fin
```

Codificación en FORTRAN 95

```
Lahey/Fujitsu Fortran 95 Compiler Release 5.50a  Wed Oct 18 23:30:38 2000
Copyright (C) 1994-1999 Lahey Computer Systems.  All rights reserved.
Copyright (C) 1998-1999 FUJITSU LIMITED. All rights reserved.

Compilation information
  Current directory : C:\LF9555\Src
  Source file       : c:\Lf9555\Src\laplace.f90
  Compiler options  : -nap -nc -nchk -co -ncover -dal -ndbl -ndll -nf95 -nfix
                    : -g -nin -ninfo -lst -nlong -maxfatals 50 -nml -o0 -no
                    : -out c:\Lf9555\Src\laplace.exe -pause -nprivate -npca
                    : -nquad -nsav -nstaticlink -stchk -nswm -tp -trace -ntrap
                    : -nvsw -w -winconsole -nwisk -nwo -nxref -zero

Main program "ecuacion_de_laplace"
 (line-no.)(nest)
        1        !     Last change:  AJL  18 Oct 2000   11:30 pm
        2        program ecuacion_de_laplace
        3        implicit none
        4        INTEGER n,iter,ERROR,i,j
        5        REAL (KIND=8),DIMENSION(:,:),ALLOCATABLE::T,T_anterior
        6        REAL (KIND=8)::B=1.0_8,T0=20.0_8,deltax
        7        !
        8        !  Este programa encuentra una solucion aproximada de la
        9        !  ecuacion de Laplace d(dT/dx)/dx + d(dT/dy)/dy = 0
       10        !  sujeta a la condiciones frontera
       11        !  T(x,0)=50*sin(x)  ,0<=x<=B
       12        !  T(x,B)=0          ,0<=x<=B
       13        !  T(0,y)=T(B,y)=0   ,0<y<B
       14        !
       15        interface condicion_frontera
       16          ELEMENTAL function f(x)
       17          REAL (KIND=8),INTENT(IN)::x
       18          REAL (KIND=8) f
       19          END function f
       20        end interface condicion_frontera
       21        read (*,*)n
       22        ALLOCATE(T(n,n),T_anterior(n,n),stat=ERROR)
       23        IF(ERROR/=0)STOP 'Error en allocate'
       24        !
       25        !  la placa es un cuadrado de lado B
       26        !
       27        deltax=B/n
       28        T(1,1:n)=f((/(1.0_8/(i*deltax),i=1,n)/))  ! condiciones frontera
       29        T(n,1:n)=0.0_8
       30        T(2:n-1,1)=0.0_8
       31        T(2:n-1,n)=0.0_8
       32        T_anterior=T
       33        T_anterior(2:n-1,2:n-1)=T0  ! To es una estimacion inicial
       34     1  itera:do iter=1,200
       35     2    calcula:FORALL(i=2:n-1,j=2:n-1)
```

```
        36     2      T(i,j)=(T_anterior(i-1,j)+T_anterior(i+1,j)+T_anterior(i,j-1)+&
        37     2             T_anterior(i,j+1))/4
        38     2    END forall calcula
        39     1    IF(MAXVAL(ABS(T(2:n-1,2:n-1)-T_anterior(2:n-1,2:n-1)))<.01_8)exit
        40     1    T_anterior=T
        41     1  end do itera
        42     1  escribe:IF(iter<=200)THEN
        43     2    do i=1,n
        44     2      PRINT '(10(f8.3,2x))',T(i,1:n)
        45     2    end do
        46     1    PRINT *,'Iteraciones realizadas=',iter
        47     1  else escribe
        48     1    PRINT *,'Sin convergencia'
        49     1  END if escribe
        50        stop
        51        end program ecuacion_de_laplace

Procedure information
  Lines      : 51
  Statements : 38

External function subprogram "f"
 (line-no.)(nest)
        52        ELEMENTAL function f(x)
        53        REAL (KIND=8),INTENT(IN)::x
        54        REAL (KIND=8) f
        55        f=100+20*SIN(x)
        56        return
        57        END function f
        58

Procedure information
  Lines      : 7
  Statements : 6

Total information
  Procedures       : 2
  Total lines      : 58
  Total statements : 44
```

APÉNDICE

Procedimientos intrínsecos de FORTRAN 90

Las tablas mostradas a continuación ofrecen un resumen de los procedimientos intrínsecos incluidos en FORTRAN 90. Para mayores detalles, se recomienda consultar el texto de la explicación correspondiente al procedimiento, en los capítulos adecuados de esta Guía de Programación.
Los nombres específicos de una función pueden ser pasados como argumentos actuales, excepto aquellos marcados por un *.
En FORTRAN 90, los procedimientos intrínsecos se clasifican en cuatro categorías:
I)Funciones de indagación (inquiry functions). Son aquellas que devuelven información sobre algo. Sus resultados dependen más bien de las propiedades del principal argumento que del valor de este.
E)Funciones elementales. Muchas funciones intrínsecas pueden admitir como argumentos indistintamente tanto escalares como arreglos. En esta circunstancia, la función es aplicada elemento por elemento al arreglo y el resultado es también un arreglo.
T)Funciones de transformacionales. Los procedimientos transformacionales tienen ya sea un arreglo como argumento fingido (p.e. la función SUM) o un argumento actual que es un arreglo sin causar una interpretación elemental(p.e. la función SHAPE). Una función de este tipo transforma un arreglo en un resultado escalar o en otro arreglo, en vez de aplicar la operación elemento por elemento.
S)Subrutinas. El nombre de una subrutina intrínseca no puede ser empleado como un argumento actual.

Funciones Numéricas

Nombre Nombre específico	Tipo de la función	Tipo del argumento	Descripción	Clase
ABS CABS DABS IABS	Numérico REAL_4 REAL_8 INTEGER_4	Numérico COMPLEX_4 REAL_8 INTEGER_4	Valor absoluto	E
AIMAG DIMAG	REAL REAL_8	COMPLEX COMPLEX_8	Parte imaginaria de un número complejo	E
AINT DINT	REAL REAL_8	REAL REAL_8	Truncamiento	E
ANINT DNINT	REAL	REAL	Representación REAL del entero más cercano	E
CEILING	INTEGER_4	REAL	INTEGER más pequeño mayor o igual que un número	E
CMPLX DCMPLX	Numérico Numérico	COMPLEX COMPLEX_8	Conversión a COMPLEX	E
CONJG DCONJG	COMPLEX COMPLEX_8	COMPLEX COMPLEX_8	Conjugado de un número complejo	E
DBLE	REAL_8	Numérico	Conversión a REAL_8	E
DIM DDIM IDIM	INTEGER o REAL REAL_8 INTEGER_4	INTEGER o REAL REAL_8 INTEGER_4	Diferencia entre dos números si la diferencia es positiva; de otra forma, 0	E
DPROD	REAL_8	REAL_4	Producto REAL_8	E
EXPONENT	REAL	REAL	Parte del exponente en la representación de un número	E
FLOOR	INTEGER_4	REAL	El entero más grande menor o igual que un número	E

Nombre Nombre específico	Tipo de la función	Tipo del argumento	Descripción	Clase
FRACTION	REAL	REAL	Parte fraccionaria de la representación física de un número	E
INT IDINT* IFIX*	INTEGER INTEGER INTEGER	Numérico REAL_8 REAL_4	Conversión a INTEGER	E
MAX AMAX0* AMAX1* DMAX1* MAX0* MAX1*	INTEGER o REAL REAL_4 REAL_4 REAL_8 INTEGER_4 INTEGER_4	INTEGER o REAL INTEGER_4 REAL_4 REAL_8 INTEGER_4 REAL_4	Máximo valor	E
MIN AMIN0* AMIN1* DMIN1* MIN0* MIN1*	INTEGER o REAL REAL_4 REAL_4 REAL_8 INTEGER_4 INTEGER_4	INTEGER o REAL INTEGER_4 REAL_4 REAL_8 INTEGER_4 REAL_4	Mínimo valor	E
MOD AMOD DMOD	INTEGER o REAL REAL_4 REAL_8	INTEGER o REAL REAL_4 REAL_8	Residuo	E
MODULO	INTEGER o REAL	INTEGER o REAL	Módulo	E
NEAREST	REAL	REAL	Número más cercano a otro en una dirección dada por un segundo número	E
NINT IDNINT	INTEGER INTEGER_4	REAL REAL_8	Entero más cercano	E
REAL FLOAT* SNGL*	REAL REAL_4 REAL_4	Numérico INTEGER REAL_8	Conversión a REAL	E

Nombre Nombre específico	Tipo de la función	Tipo del argumento	Descripción	E
RRSPACING	REAL	REAL	Recíproco del espaciamiento relativo cercano a un número dado	E
SCALE	REAL	REAL e INTEGER	Multiplicación de un número por una potencia de dos	E
SET_EXPONENT	REAL	REAL e INTEGER	Representación de un número con un exponente que es una potencia de 2	E
SIGN DSIGN ISIGN	INTEGER o REAL REAL_8 INTEGER_4	INTEGER o REAL REAL_8 INTEGER_4	Transferencia de signo	E
SPACING	REAL	REAL	Espaciamiento absoluto cercano a un número dado	E

Funciones Matemáticas

Nombre Nombre específico	Tipo de la función	Tipo del argumento	Descripción	Clase
ACOS DACOS	REAL REAL_8	REAL REAL_8	Arco coseno	E
ASIN DASIN	REAL REAL_8	REAL REAL_8	Arco seno	E
ATAN DATAN	REAL REAL_8	REAL REAL_8	Arco tangente	E
ATAN2 DATAN2	REAL REAL_8	REAL REAL_8	Arco tangente de y/x	E
COS CCOS DCOS	REAL o COMPLEX COMPLEX_4 REAL_8	REAL o COMPLEX COMPLEX_4 REAL_8	Coseno	E
COSH DCOSH	REAL REAL_8	REAL REAL_8	Coseno hiperbólico	E
EXP CEXP DEXP	REAL o COMPLEX COMPLEX_4 REAL_8	REAL o COMPLEX COMPLEX_4 REAL_8	Exponencial	E
LOG ALOG CLOG DLOG	REAL o COMPLEX REAL_4 COMPLEX_4 REAL_8	REAL o COMPLEX REAL_4 COMPLEX_4 REAL_8	Logaritmo natural	E
LOG10 ALOG10 DLOG10	REAL REAL_4 REAL_8	REAL REAL_4 REAL_8	Logaritmo base 10	E
SIN CSIN DSIN	REAL o COMPLEX COMPLEX_4 REAL_8	REAL o COMPLEX COMPLEX_4 REAL_8	Seno	E
SINH DSINH	REAL REAL_8	REAL REAL_8	Seno hiperbólico	E

Nombre Nombre específico	Tipo de la función	Tipo del argumento	Descripción	Clase
SQRT CSQRT DSQRT	REAL o COMPLEX COMPLEX_4 REAL_8	REAL o COMPLEX COMPLEX_4 REAL_8	Raíz cuadrada	E
TAN DTAN	REAL REAL_8	REAL REAL_8	Tangente	E
TANH DTANH	REAL REAL_8	REAL REAL_8	Tangente hiperbólica	E

Funciones de caracteres

Nombre	Descripción	Clase
ACHAR	Carácter en una posición dada de la secuencia ASCII	E
ADJUSTL	Ajuste a la izquierda, removiendo espacios iniciales e insertando espacios al final	E
ADJUSTR	Ajuste a la derecha, removiendo espacios finales e insertando espacios al inicio	E
CHAR	Carácter en una posición dada de la secuencia del procesador	E
IACHAR	Posición de un carácter de la secuencia ASCII	E
ICHAR	Posición de un carácter de la secuencia del procesador	E
INDEX	Posición inicial de una subcadena dentro de una cadena	E
LEN	Longitud de un objeto de caracteres	I
LEN_TRIM	Longitud de una cadena sin considerar a los espacios finales	E
LGE	Determina si una cadena es léxicamente mayor o igual que otra, de acuerdo a la secuencia ASCII	E
LGT	Determina si una cadena es léxicamente mayor que otra, de acuerdo a la secuencia ASCII	E
LLE	Determina si una cadena es léxicamente menor o igual que otra, de acuerdo a la secuencia ASCII	E

Nombre	Descripción	Clase
LLT	Determina si una cadena es menor que otra, de acuerdo a la secuencia ASCII	E
REPEAT	Concatena copias de una cadena	T
SCAN	Busca en una cadena la primera aparición de un carácter que es parte de un conjunto dado	E
TRIM	Omisión de espacios al final de la cadena	T
VERIFY	Verifica que un conjunto de caracteres contenga a todos los caracteres en una cadena	E

Funciones de arreglos

Nombre	Descripción	Clase
ALL	Determine si todos los valores en una máscara son ciertos a lo largo de una dimensión dada	T
ALLOCATED	Indica si la memoria de un arreglo allocatable ha sido apartada	I
ANY	Indica si cualquier valor es cierto en una máscara a lo largo de un dimensión dada	T
COUNT	Cuenta el número de elementos que cumplen una condición a lo largo de una dimensión dada	T
CSHIFT	Avance circular de secciones de rango 1 en un arreglo	T
DOT_PRODUCT	Producto punto de dos vectores	T
EOSHIFT	Avance de secciones de rango 1 en un arreglo	T
LBOUND	Límites inferiores de los subíndices de un arreglo o de una dimensión de un arreglo	I
MATMUL	Multiplicación de matrices	T
MAXLOC	Localización del primer elemento de un arreglo con el máximo valor, de acuerdo a una máscara	T
MAXVAL	Máximo valor de los elementos de un arreglo, a lo largo de una dimensión, para los cuales una máscara es cierta	T
MERGE	Selección de valores alternativos en base a una máscara	E
MINLOC	Localización del primer elemento de un arreglo con mínimo valor de los elementos, de acuerdo a un máscara	T
MINVAL	Mínimo valor de los elementos de un arreglo, a lo largo de una dimensión, para los cuales una máscara es cierta	T
PACK	Empaquetamiento de un arreglo dentro de un vector bajo el control de una máscara	T

Nombre	Descripción	Clase
PRODUCT	Producto de los elementos de un arreglo, a lo largo de una dimensión, para los cuales una máscara es cierta	T
RESHAPE	Construcción de un arreglo de una forma especificada a partir de un arreglo dado	T
SHAPE	Forma de un arreglo	I
SIZE	Tamaño de un arreglo o de una dimensión de un arreglo	I
SPREAD	Agrega una dimensión a un arreglo mediante adición de copias de un objeto a lo largo de una dimensión	T
SUM	Suma de los elementos de un arreglo, a lo largo de una dimensión, para los cuales una máscara es cierta	T
TRANSPOSE	Transposición de un arreglo de rango 2	T
UBOUND	Límites superiores de los subíndices de un arreglo o de una dimensión de un arreglo	I
UNPACK	Desempaquetamiento de un arreglo de rango 1 dentro de un arreglo, bajo el control de una máscara	T

Funciones de indagación y funciones de clase

Nombre	Descripción	Clase
ALLOCATED	Indica si la memoria de un arreglo allocatable ha sido apartada	I
ASSOCIATED	Indica si un apuntador ha sido asociado con un destino	I
BIT_SIZE	Tamaño, en bits, de un objeto de tipo INTEGER	I
DIGITS	Número de dígitos binarios significativos	I
EPSILON	Valor positivo que es casi despreciable comparado con la unidad	I
HUGE	El mayor número representable de un tipo de datos	I
KIND	Parámetro de clase de un tipo de datos	I
LBOUND	Límites inferiores de un arreglo o de una dimensión de un arreglo	I
LEN	Longitud de un objeto de caracteres	I
MAXEXPONENT	Máximo exponente binario de un tipo de datos	I
MINEXPONENT	Mínimo exponente binario de un tipo de datos	I
PRECISION	Precisión decimal de un tipo de datos	I
PRESENT	Determina si un argumento opcional está presente	I
RADIX	Base de la representación física de un número	I
RANGE	Intervalo decimal de la representación de un número	I
SELECTED_INT_KIND	Parámetro de clase de un tipo INTEGER que representa a todos los valores enteros n con $-10^r<n<10^{-r}$	T

Nombre	Descripción	Clase
SELECTED_REAL_KIND	Parámetro de clase de un tipo REAL con una precisión decimal de al menos p dígitos y un intervalo para el exponente decimal de al menos r	T
SHAPE	Forma de un arreglo	I
SIZE	Tamaño de un arreglo o de la dimensión de un arreglo	I
TINY	El número positivo más pequeño representable en un tipo de datos	I
UBOUND	Límites superiores de un arreglo o de una dimensión de un arreglo	I

Procedimientos de manipulación de bits

Nombre Nombre específico	Tipo de la función	Tipo del argumento	Descripción	Clase
BTEST	LOGICAL_4	INTEGER_4	Prueba de bits	E
IAND	INTEGER	INTEGER	AND lógico bit por bit	E
IBCLR	INTEGER	INTEGER	Darle a un bit el valor uno (borrado)	E
IBITS	INTEGER	INTEGER	Extracción de una secuencia de bits	E
IBSET	INTEGER	INTEGER	Darle a un bit el valor uno	E
IEOR	INTEGER	INTEGER	OR lógico exclusivo bit por bit	E
IOR	INTEGER	INTEGER	OR lógico inclusivo bit por bit	E
ISHFT	INTEGER	INTEGER	Avance bit por bit	E
ISHFTC	INTEGER	INTEGER	Avance circular bit por bit de los bist más a la derecha	E
MVBITS		INTEGER	Copia de una secuencia de bits de un dato INTEGER a otro	SE
NOT	INTEGER	INTEGER	Complemento lógico bit por bit	E

Otras funciones intrínsecas

Nombre	Descripción	Clase
LOGICAL	Conversión entre clases de LOGICAL	E
TRANSFER	Interpretación de la representación física de un número con el tipo y los parámetros de un número dado	T

Subrutinas intrínsecas estándar

Nombre	Descrpición	Clase
DATE_AND_TIME	Fecha y tiempo	S
MVBITS	Copia de una secuencia de bits de un dato INTEGER a otro	SE
RANDOM_NUMBER	Generación de números aleatorios uniformemente distribuidos en el intervalo $0 \leq x < 1$	S
RANDOM_SEED	Colocación o indagación de la semilla del generador de números aleatorios	S
SYSTEM_CLOCK	Información del reloj de tiempo real	S

Bibliografía

- Fortran 90/95 for Scientists and Engineers
  Second Edition
  Stephen J. Chapman
  Mc Graw Hill
  2004
  ISBN 0-07-282575-8

- Fortran 90 Programming
  T. M. R. Ellis, Ivor R. Philips y Thomas M. Lahey
  Addison-Wesley Publishing Company
  1994
  ISBN 0-201-54446-6

- Lenguaje de Programación Fortran 90
  Incluye Fortran 95
  Félix García Merayo
  Paraninfo
  1999
  ISBN 84-283-2527-8

- Migrating to Fortran 90
  James F. Kerrigan
  O' Reilly & Associates, Inc.
  1994
  ISBN 1-56592-049-X

- Fortran 90 for Engineers and Scientists
  Larry R. Nyhoff y Sanford C. Leestma
  Prentice Hall, Inc.
  1997
  ISBN 0-13-519729-5