CURSO BÁSICO

DE

Fortran

ÁNGEL FELIPE ORTEGA

Departamento de ESTADÍSTICA E INVESTIGACIÓN OPERATIVA
Facultad de MATEMÁTICAS
UCM

CURSO BÁSICO

DE

Fortran

ÁNGEL FELIPE ORTEGA

Departamento de ESTADÍSTICA E INVESTIGACIÓN OPERATIVA
Facultad de MATEMÁTICAS
UCM

CURSO BÁSICO DE Fortran

ÍNDICE

PREFACIO

1. PRIMER CONTACTO CON Fortran
1.1. Calculadora elemental
1.2. Cálculos estadísticos simples
1.3. Resolución de sistemas de ecuaciones lineales
1.4. Recta de regresión
1.5. Resolución de ecuaciones no lineales
1.6. Resumen
1.7. Ejercicios

2. INTRODUCCIÓN
2.1. Historia
2.2. Estandarización
2.3. Características de Fortran 90
2.3.1. Características nuevas
2.3.2. Características mejoradas
2.3.3. Características obsoletas
2.4. Características de Fortran 95
2.5. Características de Fortran 2003
2.6. Notas

3. ESTRUCTURA DEL PROGRAMA. CÓDIGO FUENTE
3.1. Elementos del lenguaje
3.1.1. Caracteres
3.1.2. Nombres. Entidades
3.2. Formato del código fuente
3.2.1. Formato fijo
3.2.2. Formato libre
3.2.3. Compatibilidad de código en formato fijo y en formato libre
3.3. Tipos intrínsecos de datos
3.3.1. Constantes. Rangos
3.3.2. Parámetros. Variables. Declaración. Asignación
3.3.3. Arrays. Subíndices. Substrings
3.4. Operadores. Expresiones. Prioridades
3.4.1. Operadores y expresiones aritméticas
3.4.1.1. Aritmética entera
3.4.1.2. Mezcla de operandos. Conversiones
3.4.2. Operadores y expresiones de caracteres
3.4.3. Operadores y expresiones de relación
3.4.4. Operadores y expresiones lógicas

3.5. Entrada y salida estándar sin formato
3.6. Sentencias PROGRAM, END
3.7. Programa ejemplo
3.8. Ejercicios

4. SENTENCIAS DE CONTROL
4.1. Sentencia CONTINUE
4.2. Sentencia STOP
4.3. Sentencia GOTO incondicional
4.4. Sentencia IF. Bloques IF
4.4.1. IF lógico
4.4.2. Bloque IF-THEN-ENDIF
4.4.3. Bloque IF-THEN-ELSE-ENDIF
4.4.4. Bloques ELSE IF
4.5. Selector SELECT CASE
4.6. Iteraciones DO
4.6.1. DO no limitado
4.6.2. EXIT
4.6.3. CYCLE
4.7. Sentencias de control redundantes
4.7.1. DO WHILE
4.7.2. Bloques DO con etiqueta
4.7.3. Final de bloques DO en CONTINUE
4.8. Sentencias de control obsoletas
4.8.1. Sentencias obsoletas sólo en Fortran 95
4.8.1.1. GOTO calculado
4.8.2. Sentencias obsoletas en Fortran 90 y en Fortran 95
4.8.2.1. IF aritmético
4.8.2.2. Final compartido de bloques DO
4.8.3. Sentencias obsoletas en Fortran 90 y eliminadas en Fortran 95
4.8.3.1. Indices no enteros en bloques DO
4.8.3.2. GOTO asignado
4.8.3.3. Salto a la sentencia ENDIF
4.9. Programa ejemplo
4.10. Ejercicios

5. UNIDADES DE PROGRAMA. PROCEDIMIENTOS
5.1. Programa principal
5.2. Subprogramas externos
5.2.1. Uso no recursivo de subprogramas FUNCTION
5.2.1.1. Sentencia RETURN en subprogramas FUNCTION
5.2.2. Uso no recursivo de subprogramas SUBROUTINE
5.2.2.1. Sentencia RETURN en subprogramas SUBROUTINE
5.2.3. Argumentos de subprogramas externos
5.2.3.1. Dimensiones de argumentos array
5.2.3.2. Propósito de los argumentos

5.2.3.3. Sentencia EXTERNAL
5.2.3.4. Sentencia INTRINSIC
5.2.3.5. Argumentos opcionales. Palabras clave. Bloque INTERFACE
5.3. Subprogramas internos
5.4. Módulos
5.5. Sentencia DATA
5.6. Orden de las sentencias
5.7. Sentencias redundantes en unidades de programa
5.7.1. Líneas INCLUDE
5.7.2. Declaración de funciones en sentencias
5.7.3. Sentencia COMMON
5.7.4. Subprogramas BLOCK DATA
5.7.5. RETURN alternativo en subprogramas SUBROUTINE
5.7.6. Longitud asumida de argumentos de tipo carácter
5.8. Ejercicios

6. PROCEDIMIENTOS INTRÍNSECOS
6.1. Funciones numéricas
6.1.1. Funciones elementales que pueden convertir tipos
6.1.2. Funciones elementales que no convierten tipos
6.1.3. Funciones matemáticas elementales
6.1.4. Multiplicación vectorial y matricial
6.1.5. Operaciones globales en arrays numéricos
6.2. Funciones de caracteres
6.2.1. Conversiones CHARACTER←→INTEGER
6.2.2. Comparaciones léxicas
6.2.3. Manejo de caracteres
6.3. Subrutinas de fecha y hora. Números aleatorios
6.3.1. Reloj de tiempo real
6.3.2. Números aleatorios
6.4. Ejercicios

7. ENTRADA Y SALIDA DE DATOS. FICHEROS. FORMATOS
7.1. Elementos y clases de ficheros
7.2. Lectura y escritura de datos
7.3. Acceso a ficheros externos
7.3.1. Sentencia OPEN
7.3.2. Sentencia CLOSE
7.3.3. Nombres y números reservados de ficheros
7.4. Entrada y salida formateada
7.4.1. READ con número de unidad
7.4.2. WRITE con número de unidad
7.4.3. No avance en entrada/salida. ADVANCE=adv
7.5. Códigos de formato
7.5.1. Código para datos enteros: I
7.5.2. Códigos para datos reales: F, E, D, G

7.5.2.1. Código F (datos reales sin exponente)
7.5.2.2. Código E (datos reales con exponente)
7.5.2.3. Código D (datos reales con exponente d)
7.5.2.4. Código G (datos reales con rango amplio)
7.5.3. Código para datos lógicos: L
7.5.4. Código para datos carácter: A
7.5.5. Literales
7.5.6. Códigos de control
7.6. Posicionamiento de ficheros
7.6.1. Sentencia BACKSPACE
7.6.2. Sentencia REWIND
7.6.3. Sentencia ENDFILE
7.7. Ficheros internos
7.8. Programa ejemplo
7.9. Ejercicios

8. ELABORACIÓN DE PROGRAMAS
8.1. Estilo de programación
8.2. Depuración de errores
8.3. Optimización de programas
8.4. Ejercicios

APÉNDICE. CARACTERÍSTICAS AVANZADAS EN Fortran
A.1. Tipos y clases de datos
A.2. Tratamiento de arrays
A.3. Procedimientos
A.4. Ficheros
A.5. Interfaces y operadores
A.6. Compiladores
A.7. Librerías matemáticas
A.8. Fortran en internet

BIBLIOGRAFÍA

Bibliografía básica
Bibliografía complementaria

PREFACIO

Este "Curso Básico de Fortran" no pretende ser un resumen teórico del lenguaje Fortran sino un sencillo y práctico tutorial que cubre gran parte del lenguaje con una amplia colección de ejemplos y ejercicios propuestos con el objetivo de proporcionar un fácil aprendizaje de un uso básico de este lenguaje de programación.

En contra de lo que suele ser habitual en la confección de libros o manuales sobre lenguajes de programación, que comienzan explicando las sentencias más sencillas o de uso más inmediato, este Curso comienza con una amplia visión de lo que se puede hacer y cómo hacerlo en Fortran. Así, en el capítulo 1 se presenta una colección de programas cortos gradualmente más extensos, representativos de las principales características del lenguaje. El aprendizaje será más rápido y fiable después de analizarlos y entenderlos y, desde un primer momento, se estará en condición de elaborar programas para un amplio espectro de tareas.

No sólo es importante elaborar programas, a veces también es necesario leer, interpretar y modificar programas ya escritos, posiblemente con versiones antiguas del lenguaje, para adaptarlos a necesidades específicas. En los comienzos de preparación de este Curso los compiladores de FORTRAN 77 se usaban ampliamente y los de Fortran 90 llevaban poco tiempo en el mercado. Aunque actualmente los compiladores de Fortran 90/95 están muy extendidos (e incluso incorporan características de Fortran 2003), se describen algunos detalles de FORTRAN 77 estándar y otras características declaradas obsoletas en Fortran 90/95. El capítulo 2 contiene una breve introducción histórica y un resumen de estas características obsoletas.

Los capítulos 3, 4, 5, 6 y 7 tratan los principales tópicos en Fortran: elementos del lenguaje, tipos de datos, expresiones, sentencias de control, unidades de programa, procedimientos intrínsecos y lectura y escritura de datos.

El capítulo 8 muestra diversas consideraciones para la elaboración de programas, junto con una sugerente colección de programas propuestos. En el Apéndice se muestra una especie de índice de características de Fortran apenas tratadas en este Curso Básico que serían objeto de un Curso Avanzado. La bibliografía incluida permite completar el estudio del lenguaje.

Se dedica especial atención al estilo en la presentación de algunos programas. El uso del color y resalte en negrita de las palabras clave de Fortran permite leer más eficazmente el listado de un programa. Los entornos de edición de programas Fortran disponen de estas facilidades para la redacción del código fuente.

Este Curso Básico no requiere conocimientos previos de Fortran. Es bastante autocontenido y su duración será flexible según la cantidad de ejemplos y ejercicios a desarrollar, muchos de los cuales se prestan a explicaciones adicionales, variantes, análisis de errores, uso de sentencias más avanzadas del lenguaje, etc, con las que el aprovechamiento será mejor.

CAPÍTULO 1. *PRIMER CONTACTO CON Fortran*

Fortran es un lenguaje de programación especialmente adecuado para realizar cálculos científicos. Entre otros compiladores de Fortran para PC, algunos ya fuera de mercado, pueden citarse los siguientes:

- MICROSOFT. Fortran 77: Versiones 5.1 para MSDOS, POWERSTATION 1.0 para Windows 3.1. Fortran 90: POWERSTATION 4.0 para Windows 95.
- SALFORD/SILVERFROST. Fortran 90: Versiones 2.18 para MSDOS y Windows 3.1, 2.17 y 2.191 para Windows 95. Fortran 95 para Windows 98, XP, Vista: FTN95 V4.5. Fortran 95 para Windows XP y superior: FTN95 V6.2.
- COMPAQ VISUAL FORTRAN. Fortran 95
- Otros compiladores de Fortran 95: INTEL, NAGWare, LAHEY, G95.
- NDP. Fortran 77
- F. Subconjunto de Fortran 90.

Existen numerosas páginas web sobre Fortran, en las que pueden encontrarse tutoriales, manuales, programas, artículos, grupos de noticias. También pueden descargarse versiones gratuitas de algunos compiladores. La página básica es: http://www.fortran.com.

Las fases de creación y uso de un programa Fortran son: edición, compilación, carga y ejecución. Como editor puede utilizarse, además del editor que posiblemente tenga el propio compilador, cualquier otro editor de textos ASCII (Notepad, Edit, TextPad,...) o incluso procesadores de textos que exporten en ASCII.

La versión 2.18 de SALFORD FTN90 puede utilizarse en modo comando desde una ventana de MSDOS. Si ya está instalado correctamente, hay que poner el PATH adecuado y cargar el fichero residente DBOS que gestiona la memoria extendida. Una vez editado el fichero de código, la forma *usual* de compilar, cargar y ejecutar es con los ficheros FTN90 y LINK77. Su uso se limita a Windows 98 o inferior.

Los compiladores SALFORD FTN90 V2.191 y FTN95 (cualquier versión) tienen en Windows el entorno PLATO que integra edición, compilación, carga y ejecución. También pueden utilizarse en modo comando con FTN90 ó FTN95, respectivamente, y SLINK. Su uso requiere Windows 98 o superior. En cualquiera de ellos se pueden utilizar librerías comerciales y crear librerías con procedimientos propios. Dentro del entorno PLATO, además del compilador FTN95 también pueden realizarse programas en C/C++. La ayuda que viene incorporada con este software es muy útil. Es preferible utilizar la versión 2.21 de PLATO, en lugar de otras posteriores, ya que facilita el uso de librerías.

Los siguientes ejemplos introducen de forma gradual las sentencias más utilizadas del lenguaje.

1.1. CALCULADORA ELEMENTAL

La sentencia **PRINT** permite visualizar valores en pantalla. **PRINT*** indica 'sin formato específico' y los valores se visualizarán en función de su tipo, magnitud, sistema operativo y compilador utilizado. Los caracteres entre apóstrofos se llaman literales. La última sentencia de un programa debe ser **END**.

Ejemplo 1-1. *Operaciones elementales. Sentencia* PRINT **EJ1-1**

```
PRINT*, ' Lo primero que hago en Fortran'
PRINT*, 1+3+5+7+9
PRINT*, 3.2*(4.2-3.1*2.5)/6.35
END
```

Pueden utilizarse variables escalares para contener valores numéricos y combinarse para formar expresiones. Existen numerosas funciones matemáticas intrínsecas. La sentencia **PRINT** también permite visualizar valores de variables y expresiones. Una sentencia larga puede codificarse en varias líneas. (Para indicar que una línea continúa en la siguiente se pone & al final de ella).

Ejemplo 1-2*. Variables escalares. Funciones intrínsecas* **EJ1-2**

```
x = 2.3 + 5.2*4.78
y = SQRT(x**2+3.5)
epi = EXP(1.0) + 2*ASIN(1.0)
PRINT*, 'x = ', x, ' y = ', y
PRINT*, 'e + pi = ', epi
fxy = LOG(0.1*x)/(SIN(y)+1.2) - (x+1)**2.6 +   &
      5*x/(x*y+1)
PRINT*, 'f(x,y) = ', fxy
END
```

1.2. CALCULOS ESTADÍSTICOS SIMPLES

Ejemplo 1-3. *Variables escalares* **EJ1-3**

```
n = 5
x1 = 1.23
x2 = 2.34
x3 = 3.45
x4 = 4.56
x5 = 5.67
suma = x1 + x2 + x3 + x4 + x5
xmed = suma / n
varianza = (x1-xmed)**2 + (x2-xmed)**2 + (x3-xmed)**2 +   &
           (x4-xmed)**2 + (x5-xmed)**2
varianza = varianza / n
PRINT*, 'suma = ', suma
PRINT*, 'media = ', xmed
PRINT*, 'varianza = ', varianza
END
```

En la codificación de las sentencias de un programa es muy aconsejable incluir comentarios para ayudar a la interpretación del código. Lo que sigue al carácter de admiración ! se considera comentario, salvo si forma parte de un literal. Conviene dejar líneas y espacios en blanco para facilitar la lectura del código.

Hay 5 tipos intrínsecos de datos: enteros (**INTEGER**), reales (**REAL**), complejos (**COMPLEX**), literales (**CHARACTER**) y lógicos (**LOGICAL**). A partir de ellos pueden definirse estructuras más complejas de datos.

En cualquier tipo de datos pueden utilizarse vectores y es obligatorio declararlos indicando el tipo y dimensión.

Una de las sentencias más características de un lenguaje de programación son los bucles para realizar repeticiones. En Fortran se codifican con las construcciones **DO-ENDDO**.

Ejemplo 1-4. *Comentarios. Estilo. Tipo de variables. Vectores. Bucles* DO **EJ1-4**

```
! Declaracion de variables

INTEGER n, i
REAL x(100)
REAL suma, media, varianza

! Definicion de los 100 primeros numeros impares

n = 100
DO i = 1, n
  x(i) = 2*i - 1
ENDDO

! Calculo de la suma, media y varianza

suma = 0              ! Iniciacion de suma
DO i = 1, n
  suma = suma + x(i)
ENDDO

media = suma / n

varianza = 0          ! Iniciacion de varianza
DO i = 1, n
  varianza = varianza + (x(i)-media)**2
ENDDO
varianza = varianza / n

! Escritura de resultados

PRINT*, 'suma = ', suma
PRINT*, 'media = ', media
PRINT*, 'varianza = ', varianza

END
```

El ejemplo **EJ1-5** muestra cómo leer datos desde el teclado y cómo visualizarlos en la pantalla con formatos específicos.

Ejemplo 1-5. *Lectura desde el teclado. Etiquetas. Sentencias* `IF`, `GOTO`. *Escritura con formatos* `A`, `I`, `F`, `/`, `X` **EJ1-5**

```
PROGRAM ejemplo5
INTEGER n, i
REAL, DIMENSION(100):: x        ! equivalente a REAL x(100)
REAL:: suma, media, varianza

! Lectura por teclado de los numeros x(i). Para terminar poner 9999

i = 0
DO
  i = i + 1
  PRINT*, 'i = ', i
  READ*, x(i)
  IF (x(i) == 9999) EXIT
ENDDO
n = i - 1        ! el ultimo numero, 9999, no se considera

! Calculo de la suma, media y varianza

suma = 0
DO i = 1, n
  suma = suma + x(i)
ENDDO

media = suma / n

varianza = 0
DO i = 1, n
  varianza = varianza + (x(i)-media)**2
ENDDO
varianza = varianza / n

! Escritura de resultados

PRINT 9000, 'Se han leido ', n, ' numeros'
9000 FORMAT (5X,A,I4,A)
PRINT 9010, 'suma = ', suma, 'media = ', media, 'varianza = ', varianza
9010 FORMAT (//2X,A,F10.4,3X,A,F10.4,3X,A,F14.5)

END
```

La sentencia **`PROGRAM`** es opcional y permite dar un nombre al programa.

La lectura de datos desde el teclado se realiza con la sentencia **`READ`**. Si no se desea utilizar un formato de lectura específico, que es lo habitual para datos numéricos, la sintaxis es: **`READ`**`*, lista_de_variables`.

Conviene indicar antes en la pantalla (sentencia **`PRINT`**`)` los datos a ser leídos.

Una etiqueta es un número positivo de no más de 5 cifras que se pone al principio de una sentencia y puede ser referenciada por otras sentencias.

La sentencia **`IF`** `(condic) sentenc` ejecuta la sentencia `sentenc` si la condición `condic` es verdadera y no la ejecuta, esto es, continúa en la siguiente sentencia a **`IF`**, si `condic` es falsa.

En las comparaciones se utilizan los operadores de relación: **`.EQ.`** (igual), **`.NE.`** (distinto), **`.LT.`** (menor), **`.LE.`** (menor o igual), **`.GT.`** (mayor), **`.GE.`** (mayor o igual) ó sus equivalentes: `==, /=, <, <=, >, >=,` respectivamente.

La sentencia **`GOTO`** `etiq` transfiere la ejecución a la línea con la etiqueta `etiq` (se recomienda utilizar la sentencia **`GOTO`** lo menos posible).

La sentencia **`FORMAT`** establece códigos de formato específicos para leer o escribir datos. Proporciona mucha flexibilidad. Los códigos de formato más usuales son: `/` (salto de línea), `nX` (n espacios en blanco), `Iw` (datos enteros, w posiciones), `Fw.d` (datos reales, w posiciones, d decimales), `A` (caracteres), `' '` (literales).

1.3. RESOLUCIÓN DE SISTEMAS DE ECUACIONES LINEALES

Sea el sistema de dos ecuaciones con dos incógnitas $\left.\begin{array}{l} a_{11}\,x_1 + a_{12}\,x_2 = b_1 \\ a_{21}\,x_1 + a_{22}\,x_2 = b_2 \end{array}\right\}$

Sea d el determinante de la matriz de coeficientes, esto es: $d = a_{11}\,a_{22} - a_{12}\,a_{21}$. Si $d = 0$ el sistema no tiene solución única (ó no tiene solución ó tiene infinitas soluciones). No se desea realizar un estudio más detallado en este caso.

Si $d \neq 0$ la única solución es $x_1 = \frac{1}{d}\left(a_{22}\,b_1 - a_{12}\,b_2\right)$, $x_2 = \frac{1}{d}\left(a_{11}\,b_2 - a_{21}\,b_1\right)$.

Ejemplo 1-6. *Ficheros ASCII. Sentencias* `OPEN, STOP, WRITE` **EJ1-6**

```
PROGRAM sistemas_de_dos_ecuaciones
REAL a11, a12, a21, a22, b1, b2, det, x1, x2

! Lectura de los coeficientes del fichero 'datos' (que ya debe existir)

OPEN (11, FILE='datos')
READ (11,*) a11, a12, b1, a21, a22, b2

! Solucion del sistema

det = a11*a22 - a12*a21   ! determinante de la matriz a
IF (det == 0) STOP ' el sistema no tiene solucion unica'
x1 = (b1*a22 - b2*a12) / det
x2 = (b2*a11 - b1*a21) / det

! Escritura de resultados en el fichero 'result' (lo crea el programa)

OPEN (12, FILE='result')
WRITE (12,9000) x1, x2
9000 FORMAT (3X,'x1 = ',F12.4/3X,'x2 = ',F12.4)

END
```

En el ejemplo **EJ1-6** los coeficientes del sistema se leen del fichero de entrada `'datos'`, creado con cualquier editor de ficheros ASCII, y se escriben en el fichero de salida `'result'`. Para tener acceso a un fichero ASCII se debe 'conectar' al programa con la sentencia **OPEN**, en la cual se asigna un número de unidad al fichero (el nombre puede llevar una ruta ó PATH). Las sentencias **READ/WRITE** referidas a ese número de unidad transferirán datos desde/al fichero.

La sentencia **STOP** detiene la ejecución del programa y si se usa la sintaxis **STOP** `'texto'` muestra en pantalla el mensaje entre apóstrofos.

En el ejemplo **EJ1-7** los coeficientes del sistema se almacenan en una matriz y un vector. El nombre de los ficheros se lee desde el teclado y se almacena en una variable **CHARACTER** que puede contener no más de 20 caracteres.

Cada sentencia **READ** ó **WRITE** comienza la transferencia de datos en un nuevo registro. El especificador **ADVANCE='NO'** en la sentencia **WRITE** anula el salto de registro. Las sentencias de lectura de datos utilizan **DO**'s implícitos.

Ejemplo 1-7. *Matrices. Datos* CHARACTER. *Formato* A. DO *implícito* **EJ1-7**

```
PROGRAM sistemas_de_dos_ecuaciones
REAL a(2,2), b(2), x(2), det
CHARACTER(LEN=20) filedat, filesol

! Lectura de los nombres de los ficheros de datos y resultados

WRITE (*,'(A)',ADVANCE='NO') ' fichero de datos = '
READ (*,'(A)') filedat
OPEN (11, FILE=filedat)
WRITE (*,'(A)',ADVANCE='NO') ' fichero de resultados = '
READ (*,'(A)') filesol
OPEN (12, FILE=filesol)

! Lectura de los datos

READ (11,*) (a(1,j),j=1,2), b(1)
READ (11,*) (a(2,j),j=1,2), b(2)

! Solucion del sistema

det = a(1,1)*a(2,2) - a(1,2)*a(2,1)
IF (det == 0) STOP ' el sistema no tiene solucion unica'
x(1) = (b(1)*a(2,2) - b(2)*a(1,2)) / det
x(2) = (b(2)*a(1,1) - b(1)*a(2,1)) / det

! Escritura de resultados

WRITE (12,9000) (x(j),j=1,2)
9000 FORMAT (3X,'Solucion x = ',2F12.4)

ENDPROGRAM sistemas_de_dos_ecuaciones
```

1.4. RECTA DE REGRESIÓN

Dada la nube de puntos $(x_i, y_i)\ \forall i \in \{1,\cdots,m\}$, se trata de ajustar la recta de regresión $y = px + q$ por el criterio de mínimos cuadrados. Los valores de *p*, *q* son tales que minimizan $F(p,q) = \sum_{i=1}^{m} (y_i - px_i - q)^2$. Las ecuaciones a satisfacer por *p*, *q* son:

$$\left.\begin{aligned} \frac{\partial F(p,q)}{\partial p} &= -2\sum_{i=1}^{m}(y_i - px_i - q)\ \ x_i = 0 \\ \frac{\partial F(p,q)}{\partial q} &= -2\sum_{i=1}^{m}(y_i - px_i - q) = 0 \end{aligned}\right\} \text{ esto es: } \left.\begin{aligned} p\sum_{i=1}^{m} x_i^2 + q\sum_{i=1}^{m} x_i &= \sum_{i=1}^{m} x_i y_i \\ p\sum_{i=1}^{m} x_i + qm &= \sum_{i=1}^{m} y_i \end{aligned}\right\}$$

Sea $d = m\sum_{i=1}^{m} x_i^2 - \left(\sum_{i=1}^{m} x_i\right)^2$.

Si $d = 0$ todos los puntos tienen la misma abscisa x_i y no existe recta de regresión.

Si $d \neq 0$ la solución es $p = \frac{1}{d}\left(m\sum_{i=1}^{m} x_i y_i - \sum_{i=1}^{m} x_i \sum_{i=1}^{m} y_i\right)$, $q = \frac{1}{m}\left(\sum_{i=1}^{m} y_i - p\sum_{i=1}^{m} x_i\right)$.

Ejemplo 1-8. *Atributo* PARAMETER. *Bloque* IF. *Subrutinas* **EJ1-8**

```
PROGRAM regresion
INTEGER m, i, indic
INTEGER, PARAMETER:: mmax=100
REAL x(mmax), y(mmax), p, q
CHARACTER(LEN=30) filedat, filesol

! Lectura de los nombres de los ficheros de datos y de resultados

WRITE (*,'(A)',ADVANCE='NO') ' fichero de datos = '
READ (*,'(A)') filedat
OPEN (11, FILE=filedat)
WRITE (*,'(A)',ADVANCE='NO') ' fichero de resultados = '
READ (*,'(A)') filesol
OPEN (12, FILE=filesol)

! Lectura de la nube de puntos

READ (11,*) m
IF (m > mmax) STOP ' m es demasiado grande'
READ (11,'(8F10.5)') (x(i),i=1,m)
READ (11,'(8F10.5)') (y(i),i=1,m)

! Calculo y escritura de la recta de regresion

CALL regres (x, y, m, p, q, indic)
IF (indic == 0) THEN
  WRITE (12,9000) p, q
  9000 FORMAT (3X,'p = ',F12.4/3X,'q = ',F12.4)
ELSE
  WRITE (12,'(A)') ' no hay recta de regresion'
ENDIF

ENDPROGRAM regresion
```

```
! Subrutina que calcula la recta de regresion

SUBROUTINE regres (x, y, m, p, q, indic)
INTEGER m, indic
REAL x(*), y(*), p, q

! Variables locales

INTEGER i
REAL sumx2, sumx, sumxy, sumy, d

! Calculo de las sumas sumx2, sumx, sumxy, sumy

sumx2 = 0
sumx = 0
sumxy = 0
sumy = 0
DO i = 1, m
  sumx2 = sumx2 + x(i)**2
  sumx = sumx + x(i)
  sumxy = sumxy + x(i)*y(i)
  sumy = sumy + y(i)
ENDDO

! Calculo de p, q

d = m*sumx2 –sumx**2
IF (d == 0) THEN
  indic = 1
ELSE
  indic = 0
  p = (m*sumxy – sumx*sumy) / d
  q = (sumy – p*sumx) / m
ENDIF

ENDSUBROUTINE regres
```

Una constante tiene un valor que no cambia en el programa. Se declaran con el atributo **PARAMETER**. Entre otros usos, sirven para establecer cotas de dimensiones de vectores y matrices. En el ejemplo **EJ1-8**, la dimensión de los vectores `x, y` es 100 (el valor del parámetro `nmax`), mientras que el número de puntos de la nube será `m` (leído del fichero de datos) y debe ser `m <= nmax`.

El formato de lectura o escritura puede ponerse en la sentencia **READ** ó **WRITE** delimitado por apóstrofos. Puede ponerse un factor de repetición delante de un código de formato: `8F10.5` equivale a `F10.5,F10.5,...,F10.5` (8 veces).

Pueden utilizarse bloques **IF**. Dos formas usuales de estos bloques son:

(a)

```
IF (expres) THEN
  sentencias (se ejecutan si expres es verdadera)
ENDIF
```

(b)

```
IF (expres) THEN
  sentencias (se ejecutan si expres es verdadera)
ELSE
  sentencias (se ejecutan si expres es falsa)
ENDIF
```

Con el uso de subprogramas aumenta notablemente la potencia de un lenguaje de programación. Un subprograma **SUBROUTINE** tiene un nombre y una lista de argumentos ficticios. Se llama con la sentencia **CALL** indicando el nombre del subprograma y la lista de argumentos actuales, los cuales se sustituyen en los ficticios.

Las variables y etiquetas de un subprograma son independientes de las de otros subprogramas y del programa principal. Incluso pueden codificarse en ficheros distintos y compilarse por separado.

1.5. RESOLUCIÓN DE ECUACIONES NO LINEALES

Uno de los métodos más simples para resolver numéricamente una ecuación no lineal $f(x)=0$ es el método de la bisección que está basado en el siguiente resultado:

> "Sea f una función continua en $[a, b]$ tal que $f(a), f(b)$ tienen signos contrarios. El teorema de Bolzano establece que existe al menos un valor $\alpha \in [a, b]$ tal que $f(\alpha)=0$".

Una forma de aproximar uno de los valores α es calcular $c=\frac{1}{2}(a+b)$ y $f(c)$. Si $f(a), f(c)$ tienen el mismo signo a se sustituye por c (figura 1); en otro caso, b se sustituye por c (figura 2). Así, en cada etapa se obtiene un nuevo intervalo $[a, b]$ cuya longitud es la mitad del de la etapa anterior. Se itera hasta alcanzar una precisión dada.

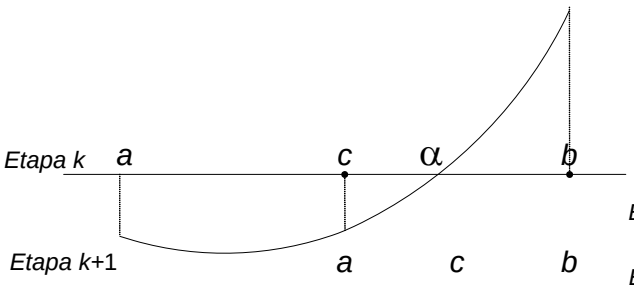


Figura 1

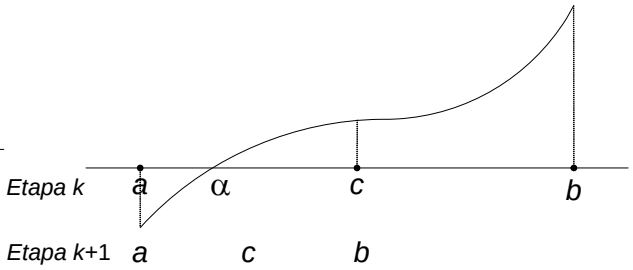


Figura 2

La codificación del ejemplo **EJ1-9** puede servir de base para la realización de programas de usuario de complejidad moderada. Es importante cuidar el estilo con comentarios, espacios, explicaciones, declaraciones de variables.

El programa pretende resolver en una única codificación y ejecución las dos ecuaciones siguientes:

(1) $x^6 - 6x^5 + 15x^4 - 24x^3 + 15x^2 - 6x + 1 = 0$
(2) $x^4 + 2x^3 - 36x^2 - 2x - 1 = 0$

El módulo `numeroecu` contiene una variable entera `iecu` para seleccionar la ecuación a resolver (bloque **`DO`** del programa principal y bloque **`SELECT CASE`** de la función `fun`). Para tener acceso al contenido de un módulo en una unidad de programa debe ponerse la sentencia **`USE`** en dicha unidad antes de la especificación de variables y argumentos.

La subrutina `bisec` implementa el algoritmo de la bisección. Contiene en comentarios una breve pero precisa descripción de sus argumentos, indicando si son de entrada **`INTENT(IN)`**, de salida **`INTENT(OUT)`** o de entrada/salida **`INTENT(INOUT)`**. Además del intervalo [*a*, *b*] requiere como datos de entrada unas tolerancias `tolx`, `tolf` y un máximo número de iteraciones `niter`. Proporciona como salida una variable `indice` que refleja la situación de convergencia en que termina el algoritmo y, salvo si $f(a)$, $f(b)$ tienen el mismo signo, la estimación final `x` y el valor `fx`$= f(x)$. Internamente la subrutina acota el valor `niter` a 50 si fuera mayor y devuelve en `niter` el número de evaluaciones de *f* realizadas (con 50 divisiones el intervalo inicial se divide por $2^{50} \approx 10^{15}$). Se observa que su codificación es *independiente* de la función *f*, es decir, sólo necesita poder evaluar *f* en los puntos requeridos por el algoritmo mediante una función externa dada por el usuario, en la cual se puede incluir las sentencias necesarias para evaluar dicha función.

En el programa principal `ecuacion` se utiliza la variable carácter `caso` para describir la forma de terminar el algoritmo de la bisección. Se piden por pantalla los extremos del intervalo inicial `a, b`, las tolerancias `tolx`, `tolf` y el máximo número de iteraciones (divisiones del intervalo a la mitad) `niter`. El especificador **`ADVANCE='NO'`** en las sentencias **`WRITE (*,...)`** hace que el cursor no salte a la línea siguiente después de volcar los datos a la pantalla. En la sentencia **`CALL`** `bisec`, como el argumento `fun` es el nombre de un subprograma debe declararse **`EXTERNAL`** en la unidad de programa que contiene a dicha sentencia **`CALL`**. Los subprogramas **`FUNCTION`** también tienen tipo. El código de formato `Gw.d` escribirá los datos reales con o sin exponente según la magnitud del dato y la anchura `w` y cifras significativas `d` especificadas.

Ejemplo 1-9. *Estilo. Módulos.* `ADVANCE='NO'`. *Formato* `G`. *Funciones. Propósito de los argumentos.* `ELSEIF`. `SELECT CASE` **EJ1-9**

```
MODULE numeroecu
  INTEGER iecu
ENDMODULE numeroecu
```

```
! ======================================================================
! PROPOSITO: Resolver la ecuacion f(x)=0 por el metodo de la biseccion

! SUBPROGRAMAS LLAMADOS:
!    DOUBLE PRECISION, EXTERNAL:: f <=> ecuacion a resolver f(x)=0

!    La funcion f(x) debe codificarse como sigue:

!           FUNCTION f(x)
!           DOUBLE PRECISION f, x
!             ...
!           f =
!           END

!    y declararse EXTERNAL en la unidad de programa de llamada.

! ARGUMENTOS DE ENTRADA:
!    DOUBLE PRECISION:: a, b <=> extremos del intervalo
!    DOUBLE PRECISION:: tolx, tolf <=> tolerancias en x, f

! ARGUMENTOS DE SALIDA:
!    DOUBLE PRECISION:: x <=> valor estimado de la raiz
!    DOUBLE PRECISION:: fx <=> f(x)
!    INTEGER:: indice <=> situacion alcanzada
!              indice=0 : f(a) y f(b) tienen el mismo signo
!              indice=1 : se han alcanzado ambas tolerancias
!              indice=2 : solo se ha alcanzado la tolerancia en x
!              indice=3 : solo se ha alcanzado la tolerancia en f
!              indice=4 : no se ha alcanzado la tolerancia en x ni en f

! ARGUMENTOS DE ENTRADA/SALIDA:
!    INTEGER:: niter
!              En entrada: maximo numero de divisiones del intervalo
!              En salida : numero de evaluaciones de f realizadas
! ======================================================================

SUBROUTINE bisec (f, a, b, tolx, tolf, niter, x, fx, indice)
IMPLICIT NONE

! Subprogramas llamados

DOUBLE PRECISION, EXTERNAL:: f

! Argumentos

DOUBLE PRECISION, INTENT(IN):: a, b, tolx, tolf
DOUBLE PRECISION, INTENT(OUT):: x, fx
INTEGER, INTENT(OUT):: indice
INTEGER, INTENT(INOUT):: niter

! Variables locales

INTEGER n
DOUBLE PRECISION x1, x2, f1, f2

! ----------------------------
! Primera sentencia ejecutable

x1 = MIN(a,b) ; f1 = f(x1)
x2 = MAX(a,b) ; f2 = f(x2)
```

```
niter = MAX(1,MIN(ABS(niter),50))

indice = 0
IF (f1*f2 > 0) THEN     ! f(a) y f(b) tienen el mismo signo
  niter = 2
  RETURN
ENDIF

! Se realizaran a lo sumo niter iteraciones

DO n = 1, niter
  x = 0.5d0*(x1+x2) ; fx = f(x)
  IF (f1*fx > 0) THEN
    x1 = x ; f1 = fx
  ELSE
    x2 = x ; f2 = fx
  ENDIF
  IF (x2-x1<tolx .AND. ABS(fx)<tolf) THEN
    niter = 2 + n
    indice = 1   ! se han alcanzado ambas tolerancias
    EXIT
  ENDIF
ENDDO

IF (indice == 0) THEN  ! No se han alcanzado las dos tolerancias
  niter = 2 + niter
  IF (x2-x1 < tolx) THEN
    indice = 2   ! solo se ha alcanzado la tolerancia en x
  ELSEIF (MIN(ABS(f1),ABS(f2)) < tolf) THEN
    indice = 3   ! solo se ha alcanzado la tolerancia en f
  ELSE
    indice = 4   ! no se ha alcanzado la tolerancia en x ni en f
  ENDIF
ENDIF

! Mejor punto obtenido

IF (ABS(f1) <= ABS(f2)) THEN
  x = x1 ; fx = f1
ELSE
  x = x2 ; fx = f2
ENDIF

ENDSUBROUTINE bisec

! ========================================================================
! Ecuacion a resolver fun(x)=0

FUNCTION fun(x)
USE numeroecu
DOUBLE PRECISION fun, x

SELECT CASE (iecu)
CASE (1)
  fun = 1 + x*(-6+x*(15+x*(-24+x*(15+x*(-6+x)))))
CASE (2)
  fun = x**4 + 2*x**3 - 36*x**2 - 2*x - 1
ENDSELECT

ENDFUNCTION fun
```

```
! ========================================================================
! Programa principal

PROGRAM ecuacion
USE numeroecu
IMPLICIT NONE

INTEGER indice, niter
CHARACTER(LEN=50) caso(0:4)
DOUBLE PRECISION a, b, tolx, tolf, x, fx
DOUBLE PRECISION, EXTERNAL:: fun

! Descripcion de la convergencia alcanzada

caso(0) = 'f(a) y f(b) tienen el mismo signo'
caso(1) = 'se ha alcanzado la precision deseada'
caso(2) = 'solo se ha alcanzado la precision en x'
caso(3) = 'solo se ha alcanzado la precision en f'
caso(4) = 'no se ha alcanzado la precision en x ni en f'

! Lectura del intervalo, tolerancias e iteraciones

DO iecu = 1, 2

  WRITE (*,'(/A,I2)') 'Ecuacion ', iecu
  WRITE (*,'(A)',ADVANCE='NO') ' Extremo a = ' ; READ*, a
  WRITE (*,'(A)',ADVANCE='NO') ' Extremo b = ' ; READ*, b
  WRITE (*,'(A)',ADVANCE='NO') ' Tolerancia para x = ' ; READ*, tolx
  WRITE (*,'(A)',ADVANCE='NO') ' Tolerancia para f = ' ; READ*, tolf
  WRITE (*,'(A)',ADVANCE='NO') ' Iteraciones = ' ; READ*, niter

  CALL bisec (fun, a, b, tolx, tolf, niter, x, fx, indice)

  IF (indice == 0) THEN
    WRITE (*,'(3X,A)') caso(0)
  ELSE
    WRITE (*,'(3X,A,G25.15/3X,A,G25.15/3X,A/3X,A,I2)') 'x = ', x,       &
              'f = ', fx, caso(indice), 'Evaluaciones de f = ', niter
  ENDIF

ENDDO

ENDPROGRAM ecuacion
```

1.6. RESUMEN

En los ejemplos anteriores se han introducido gran parte de los elementos más utilizados en Fortran, a recordar:

- Tipos de variables **INTEGER**, **REAL**, **CHARACTER**.
- Variables escalares, vectores, matrices. Parámetros.
- Comentarios, espacios en blanco, estilo del programa.
- Lectura del teclado, escritura en la pantalla.
- Lectura y escritura en ficheros ASCII.
- Formatos de escritura.

- Expresiones. Funciones matemáticas elementales.
- Sentencias de control: **`GOTO`**, bucles **`DO`**, condiciones y bloques **`IF`**, **`SELECT CASE`**.
- Subprogramas **`FUNCTION`** y **`SUBROUTINE`**. Propósito de argumentos
- Módulos

Los programas relativos a la resolución de ecuaciones no lineales, sistemas de ecuaciones y el cálculo de la recta de regresión son sólo ejemplos de introducción al lenguaje Fortran; no pretenden ser codificaciones numéricamente satisfactorias para los problemas respectivos. Asimismo, los algoritmos codificados a lo largo de este documento son ejemplos que sirven para aprender el lenguaje Fortran y no deben considerarse implementaciones eficientes desde el punto de vista numérico o computacional.

Fortran ha sido y sigue siendo el lenguaje de programación más ampliamente utilizado en aplicaciones científicas y de ingeniería. Existen grandes librerías (IMSL, NAG,...) sobre cálculo numérico y estadístico que facilitan enormemente la rápida y eficiente creación de complejos y potentes programas. Además, la cantidad de software comercial y de dominio público, tanto educativo como de investigación, escrito en Fortran es inmensa.

1.7. EJERCICIOS

1. Escribir un programa que pida por teclado el primer término de una progresión aritmética (`a1`), la diferencia de la progresión (`d`) y el número de términos (`n`) y muestre en pantalla el último término (`an`) y la suma de todos ellos (`sn`).
Fórmulas: $a_n = a_1 + (n-1)d$, $S_n = \frac{1}{2}(a_1 + a_n)n$.

2. Un coche va de un lugar A a otro B con una velocidad media `v1` y, sin pérdida de tiempo, regresa de B a A con una velocidad media `v2`. Escribir un programa que pida por teclado ambas velocidades y muestre en pantalla la velocidad media (`vm`) de su recorrido.
Fórmula: $\overline{v} = 2v_1v_2 / (v_1 + v_2)$.

3. Calcular el valor de los siguientes números ¿raros?:

$$x = \sqrt[3]{3\sqrt{21}+8} - \sqrt[3]{3\sqrt{21}-8}$$

$$y = \sqrt[5]{29\sqrt{2}+41} - \sqrt[5]{29\sqrt{2}-41}$$

$$z = \sqrt[3]{8\sqrt{37}+45} - \sqrt[3]{8\sqrt{37}-45}$$

4. Escribir un programa que pida por teclado un número entero *n* y evalúe aproximadamente *n*! mediante las expresiones siguientes:
(1) $n^n e^{-n}\sqrt{2\pi n}$

(2) $n^n e^{-n}\sqrt{2\pi n}\left(1+\frac{1}{12n}+\frac{1}{288n^2}-\frac{139}{51840n^3}-\frac{571}{2488320n^4}\right)$

Utilizar el tipo adecuado de variables. Estimar la precisión proporcionada por ambas expresiones.

5. Elaborar un programa que pida por teclado un número natural n y calcule sus factoriales $n!$, $(2n+1)!!=1\cdot 3\cdot ...\cdot(2n-1)\cdot(2n+1)$, $(2n)!!=2\cdot 4\cdot ...\cdot(2n-2)\cdot(2n)$.

6. Realizar un programa que presente un menú de opciones para diferentes casos de resolución de un triángulo. Por ejemplo, considerar los siguientes casos, descritos por los datos y fórmulas indicadas:

(1) Datos: lados *a*, *b*, *c*
$A=\arccos\left((b^2+c^2-a^2)/(2bc)\right)$; $B=\arccos\left((a^2+c^2-b^2)/(2ac)\right)$; $C=\pi-A-B$

(2) Datos: lado *a*, angulos *B*, *C*
$A=\pi-B-C$; $b=a\,\mathrm{sen}(B)/\mathrm{sen}(A)$; $c=a\,\mathrm{sen}(C)/\mathrm{sen}(A)$

(3) Datos: lados *a*, *b*, angulo *C*
$c=\sqrt{a^2+b^2-2ab\cos(C)}$; $A=\mathrm{arsen}\left(a\,\mathrm{sen}(C)/c\right)$; $B=\pi-A-C$

(4) Datos: lados *a*, *b*, angulo *A*
$B=\mathrm{arsen}\left(b\,\mathrm{sen}(A)/a\right)$; $C=\pi-A-B$; $c=a\,\mathrm{sen}(C)/\mathrm{sen}(A)$

En cualquiera de los casos calcular el área mediante $area=\frac{1}{2}a\,b\,\mathrm{sen}(C)$.

7. Escribir un programa que calcule y muestre en pantalla el valor de las expresiones

(a) $\sum_{k=0}^{n}\binom{n}{k}$

(b) $\sum_{k=0}^{n}(-1)^k\binom{n}{k}$

para $n\in\{1,\cdots,20\}$. ¿Son cantidades conocidas?.

8. Dada una cantidad *n* de términos de la serie $\sum_{k=1}^{n}\frac{1}{k^2+k}$, calcular su suma y comparar con el verdadero valor $1-1/(n+1)$. Realizar los cálculos con variables reales de precisión sencilla y repetirlos con doble precisión.

9. Obtener mediante un programa todos los números capicúas entre 10 y un millón (¿cuántos hay?). Escribirlos en un fichero. Indicar para cada uno de ellos si es un cuadrado perfecto.

10. Elaborar un programa que calcule el producto de dos polinomios cuyos grados y coeficientes se introducen por teclado.

Si $p(x)=\sum_{i=0}^{m} p_i x^i$, $q(x)=\sum_{j=0}^{n} q_j x^j$ y $r(x)=p(x)\cdot q(x)$ entonces es $r(x)=\sum_{k=0}^{m+n} r_k x^k$, siendo $r_k=\sum\left\{p_i q_j \mid i+j=k;\ 0\le i\le m;\ 0\le j\le n\right\}=\sum_{i=\max\{0,k-n\}}^{\min\{m,k\}} p_i q_{k-i}$.

Obtener el producto de los polinomios $p(x)=2-3x+8x^2+4x^3$, $q(x)=1+9x-x^2+8x^3-5x^4+7x^5$.

11. Elaborar un programa que pida por pantalla la dimensión `n` y genere la matriz $\mathbf{A}=(a_{ij})$ cuyo término general es $a_{ij}=(-1)^{n-j}\binom{n-j}{i-1}$. Calcular y escribir $\mathbf{A}^3$.

12. Escribir un subprograma **`FUNCTION`** que calcule las funciones hiperbólicas inversas definidas como:

$$\operatorname{arg\,sinh}(x)=\ln\left(x+\sqrt{x^2+1}\right), \qquad \operatorname{arg\,sinh}(x)=\ln\left(x+\sqrt{x^2+1}\right)$$

CAPÍTULO 2. *INTRODUCCIÓN*

2.1. HISTORIA

En 1953 *John Backus* presentó un proyecto para desarrollar un sistema de programación automático que convirtiera programas escritos en notación matemática a instrucciones máquina. En 1957 IBM diseña el primer compilador de FORTRAN (FORmula TRANslation).

2.2. ESTANDARIZACIÓN

En la década de los 60 proliferaron numerosos dialectos con nuevas características, no siempre compatibles entre distintos sistemas informáticos. Un proyecto de estandarización en 1966 dio lugar al FORTRAN 66, inicialmente llamado FORTRAN IV, con numerosas restricciones (no más de 63 parámetros por subrutina, nombres de hasta 6 caracteres, matrices de no más de 3 dimensiones, entre otras).

En 1978 se crea el FORTRAN 77, recogiendo diversas facilidades que ofrecían los compiladores y preprocesadores existentes. Sus novedades principales son:

- Tipo de datos `CHARACTER`
- Hasta 7 dimensiones en una matriz
- Uso ampliado de expresiones
- Bloques `IF-THEN-ELSE`
- Subprogramas genéricos
- Facilidades de I/O: Ficheros de acceso directo, sentencia `OPEN`, ...

La siguiente versión, Fortran 90, se diseñó teniendo en cuenta las sugerencias de numerosos usuarios de FORTRAN 77. Intentaba proporcionar la potencia de los nuevos lenguajes C++ y Ada y tuvo un desarrollo lento y costoso. No se limita sólo a estandarizar características existentes, sino que una revisión profunda del lenguaje.

Fortran 95 es una revisión menor de Fortran 90 con algunas pequeñas mejoras. La siguiente revisión mayor, Fortran 2003, apareció en el año 2004 y tiene una revisión menor, Fortran 2008, que la amplía.

2.3. CARACTERÍSTICAS DE Fortran 90

El Fortran 90 estándar contiene al FORTRAN 77 estándar. Algunos compiladores de FORTRAN 77 admiten, como extensiones, varias de las características nuevas o mejoradas en Fortran 90, si bien al no ser estándar en FORTRAN 77 hay notables diferencias entre ellos.

2.3.1. Características Nuevas

- Nueva sintaxis para declaración de variables
- Operadores simbólicos <, >
- Uso de `INCLUDE` (característica redundante)
- Anidación de procedimientos. Funciones y subrutinas internas
- Procedimientos recursivos
- Argumentos de procedimiento `INTENT, OPTIONAL`
- Operaciones en arrays:
  - Funciones intrínsecas operando en todos o parte de los elementos de una matriz
  - Asignación dinámica de arrays en memoria (`ALLOCATABLE, POINTER`)
  - Procedimientos intrínsecos de creación, manipulación y cálculos con matrices
- Módulos: nuevas unidades de programas para compartir datos, especificaciones,…
- Tipos derivados y operadores creados por el usuario combinando otros tipos
- Definiciones genéricas para bloques de procedimientos
- Punteros para acceder dinámicamente a la memoria
- Especificaciones `INTERFACE`, para describir características de un procedimiento externo, un nombre genérico, un nuevo operador ó un nuevo tipo de asignación

2.3.2. Características Mejoradas

- Formato del código fuente:
  - Identificadores de nombres más largos
  - Comentarios en línea
  - Formato libre sin posiciones fijas ni columnas reservadas
  - Líneas multisentencia
- Argumentos de subprogramas:
  - Argumentos opcionales
  - Palabras clave. Alteración del orden de los argumentos
  - Declaración del propósito de los argumentos
- Entrada / salida:
  - Parámetros nuevos en `OPEN`, `INQUIRE`
  - Carácter de no-avance
- Sentencias de control:
  - `SELECT CASE`
  - `DO-ENDDO` con `CYCLE` y `EXIT`
  - `DO WHILE` (ya considerada obsoleta)
  - Bucles con nombres
- Procedimientos intrínsecos: operaciones con bits, cálculos matriciales
- Precisión numérica
- Sentencias de especificación: `INTENT, OPTIONAL, POINTER, PUBLIC, PRIVATE, TARGET`

2.3.3. Características Obsoletas

Algunas sentencias o hábitos de programación se consideran obsoletos y podrían no incluirse en futuras versiones de Fortran. Se recomienda evitar tales sentencias. Se mantienen en Fortran 90 por compatibilidad.

- `RETURN` alternativo (etiquetas en la lista de argumentos)
  Alternativa: usando una variable y `GOTO` calculado ó `CASE`
- `PAUSE`. Alternativa: `READ`
- `ASSIGN` y `GOTO` asignado. Alternativa: procedimientos internos
- `FORMAT` asignado. Alternativa: expresiones de caracteres
- Descriptor Hollerith `H`. Alternativa: caracteres entre apóstrofos
- `IF` aritmético. Alternativa: bloques `IF`, `SELECT CASE`
- Variables de control `REAL` ó `DOUBLE PRECISION` en bucles `DO`
- Múltiples bucles `DO` terminando en la misma sentencia
- Final de bucle `DO` en una sentencia diferente de `ENDDO` ó `CONTINUE`
- Bucles `DO WHILE`
- Salto a un `ENDIF` desde fuera de su bloque `IF`
- Bloques `COMMON`. Alternativa: uso de Módulos
- `EQUIVALENCE`. Alternativa: uso de Módulos, memoria dinámica
- `BLOCK DATA`. Alternativa: uso de Módulos
- Funciones intrínsecas. Alternativa: funciones genéricas
- Sentencia de función. Alternativa: funciones internas
- Arrays de tamaño asumido (`*`). Alternativa: arrays de forma asumida (`:`)
- Línea `INCLUDE`. Alternativa: uso de Módulos
- `ENTRY`. Alternativa: subprogramas independientes
- Extensiones del compilador. Alternativa: lenguaje estándar
- Código fuente en formato fijo

2.4. CARACTERÍSTICAS DE Fortran 95

Su nacimiento está motivado fundamentalmente por la aparición de una versión de Fortran HPF (High Performance Fortran), basada en Fortran 90 con directivas bajo líneas de comentarios, para manejar código portable en ordenadores con procesadores en paralelo. Para evitar la proliferación de dialectos se incluyen en Fortran 95 las características del HPF.

Fortran 95 es compatible con Fortran 90, con un ligero cambio en la definición de la función `sign` y ha eliminado algunas características de FORTRAN 77 que habían sido declaradas obsoletas en Fortran 90.

Algunos complementos de Fortran 95 respecto a Fortran 90 son:

- Extensiones de construcciones `WHERE`
- Sentencia `FORALL`
- Procedimientos puros y elementales
- Inicialización de punteros. Función `NULL`
- Inicialización de componentes
- Funciones de especificación
- Función de tiempo del procesador `CPU_TIME`
- Comentarios en entrada `NAMELIST`
- Campos de salida con anchura mínima
- Argumento opcional mask en algunas funciones de manipulación de arrays
- Parámetro de tipo de clase opcional en algunas funciones elementales

Los compiladores de Fortran 95 suelen tener algunas extensiones, tales como:

- Manejo de excepciones de punto flotante IEEE
- Arrays `ALLOCATABLE` como componentes de estructuras, argumentos ficticios y resultados de funciones
- Interoperatividad con C

2.5. CARACTERÍSTICAS DE Fortran 2003

Las nuevas características principales de Fortran 2003 son:

- Mejoras en los datos de tipo derivado: parametrización, constructores, accesibilidad, finalizadores
- Programación orientada a objetos: extensión de tipos, asignación dinámica de tipos, entidades polimórficas, procedimientos en tipos
- Mejoras de manipulación de datos: componentes `ALLOCATABLE`, parámetros de tipo retardados, atributo `VOLATILE`, especificación explícita de tipo en constructores de arrays y sentencias `ALLOCATE`, expresiones de inicialización extendidas, enumeradores, `SELECT TYPE`, mejoras de procedimientos intrínsecos, mejora de las características de los módulos
- Mejoras de entrada / salida: transferencia de tipos derivados, control de redondeo, acceso a mensajes de error, reconocimiento de unidades preconectadas, sentencia `FLUSH`
- Procedimientos puntero
- Mejoras de entrada / salida: transferencia de tipos derivados, control de redondeo, acceso a mensajes de error
- Manejo de excepciones de punto flotante IEEE
- Interoperatividad con C
- Soporte para internacionalización de conjuntos de caracteres
- Integración con sistemas operativos: acceso a línea de argumentos, variables de entorno, errores del procesador

2.6. NOTAS

En este manual se mencionan algunas extensiones frecuentes de algunos compiladores de FORTRAN 77. Estas extensiones generalmente no son estándar en Fortran 90 y, por ello, pueden no ser válidas en algunos compiladores de Fortran 90. Se usará la notación Extensión FORTRAN 77. Conviene evitar tales extensiones.

Obviamente, las características propias de Fortran 90 normalmente no serán válidas en los compiladores de FORTRAN 77 y, caso de serlo, posiblemente lo serán con algunas diferencias.

Se usará la notación Nuevo en Fortran 90 para indicar algunas características importantes nuevas en Fortran 90, que pueden no ser extensiones en la mayoría de compiladores de FORTRAN 77.

Como es habitual en los textos de lenguajes de programación se denotará entre corchetes [ ] la información opcional en la sintaxis.

CAPÍTULO 3. *ESTRUCTURA DEL PROGRAMA. CÓDIGO FUENTE*

3.1. ELEMENTOS DEL LENGUAJE

3.1.1. Caracteres

Los componentes básicos del lenguaje Fortran son los *caracteres*:

- Letras `A...Z, a...z`
- Números `0...9`
- Guión underscore `_` Signo dólar `$` (Extensión Fortran 77)
- Caracteres especiales:
  `= : espacio (b) + - * / ( ) , . ' \` (antes de Fortran 90)
  `:: ! " % ; < > ? &` (Nuevo en Fortran 90)
- Resto de caracteres ASCII

Con los caracteres se construyen *tokens* (componentes simbólicos) que tienen un significado sintáctico para el compilador. Hay 6 clases de tokens:

- Etiqueta (*label*) : de 1 a 5 dígitos `123 54321`
- Constante `38 29.3d-2 'Calidad' (1.2,4.6)`
- Palabra clave (*keyword*) **`WRITE SUBROUTINE`**
- Operador
  `** * / + - //`
  `.EQ. .NE. .LT. .LE. .GT. .GE.`
  `== /= < <= > >=`
  (Nuevo en Fortran 90)
  `.NOT. .AND. .OR. .EQV. .NEQV.`
- Nombre de entidad `veloc_luz ecuacion x3j3`
- Separador `/ ( ) (/ /) , = => : :: %`

Con los *tokens* se forman sentencias. Un conjunto de sentencias forma una unidad de programa. Los espacios en blanco no pueden aparecer en un token.

3.1.2. Nombres. Entidades

Las entidades básicas en Fortran susceptibles de admitir un nombre son: variables, arrays, bloques `DO`, bloques `IF`, bloques `SELECT CASE`, bloques `COMMON`, programa, funciones, subprogramas, módulos, bloques `INTERFACE`.

- En Fortran 90 se permiten hasta 31 caracteres alfanuméricos; el primero ha de ser letra (en FORTRAN 77 estándar sólo se permiten 6 caracteres).
- El underscore `_` puede estar en un nombre pero no como su primer carácter.
- Los espacios en blanco no son significativos, salvo en los literales.
- Las minúsculas y mayúsculas son equivalentes, salvo en literales.
- No se permiten acentos ni `'ñ'` salvo en constantes literales o en comentarios.

3.2. FORMATO DEL CÓDIGO FUENTE

Puede ser libre o fijo. No deben mezclarse ambos en un fichero de código.

3.2.1. Formato Fijo

- Las sentencias de un programa se escriben en diferentes líneas.
- La posición de los caracteres dentro de las líneas es significativa.
- Columnas:

1-5	número de etiqueta (de 1 a 5 dígitos)
6	carácter de continuación de línea (ni espacio, ni 0) Se permiten al menos 19 líneas de continuación seguidas
7-72	sentencia FORTRAN
73 en adelante	se ignoran

- Comentarios:
  - Las líneas en blanco se ignoran. Hacen más legible el programa.
  - Si el primer carácter de una línea es `*`, `c` ó `C` la línea es de comentario.
  - Si aparece el carácter `!` en una línea (salvo en la columna 6), lo que le sigue es un comentario (Nuevo en Fortran 90).
- Una línea puede contener varias sentencias separadas por punto y coma (`;`), el cual no puede estar en la columna 6. Sólo la primera de estas sentencias podría llevar etiqueta.

El formato fijo para el código fuente se ha declarado obsoleto en Fortran 95.

3.2.2. Formato Libre (Nuevo en Fortran 90)

- Se pueden utilizar líneas de hasta 132 caracteres.
- La posición de los caracteres dentro de las líneas no es significativa.
- Una misma línea puede contener varias sentencias separadas por punto y coma (`;`)
- Los espacios en blanco son significativos: `IMPLICIT NONE`, `DO WHILE`, `CASE DEFAULT`. Son opcionales en:
  - palabras clave dobles que comienzan por `END` ó `ELSE`
  - `DOUBLE PRECISION`, `GO TO`, `IN OUT`, `SELECT CASE`
- El indicador de continuación de una línea es el carácter `&`.

3.2.3. Compatibilidad de código en formato fijo y en formato libre

Si se desea compatibilizar lo más posible un código para que pueda ser tratado como formato fijo o formato libre, las siguientes reglas ayudan notablemente a ello:
- Poner las etiquetas sólo en las columnas 1 a 5 y las sentencias sólo entre la 7 y 72.
- Tratar los espacios en blanco como significativos.
- Utilizar sólo `!` para indicar comentarios y no colocarlo en la columna 6.
- Para continuar sentencias poner un `&` en la columna 73 de la línea a continuar y otro `&` en la columna 6 de la línea que la continúa.

Ejemplo 3-1. *Formato fijo*

EJ3-1

```
      n = 100
      suma = 0
      DO i = 1, n
        suma = suma + i*i
      ENDDO
      media = suma/n

      dt = 0
      DO i = 1, n
        dt = dt + (i*i-media)**2
      ENDDO
      dt = SQRT(dt/n)

      IF (dt .LT. media) PRINT *, ' La desviacion tipica de la muestra f
     &ormada por los numeros 1, 4, 9, 16, ... , 10000 es menor que su me
     &dia'
      PRINT*, media, dt
      END
```

Ejemplo 3-2. *Formato libre*

EJ3-2

```
n=100; suma=0; DO i = 1, n; suma = suma + i*i; ENDDO; media=suma/n
dt=0; DO i = 1, n; dt = dt + (i*i-media)**2; ENDDO; dt=SQRT(dt/n)
IF (dt .LT. media) PRINT *, ' La desviacion tipica de la muestra&
   & formada por los numeros 1, 4, 9, 16, ... , 10000 es menor que&
   & su media'; PRINT *, media, dt; END
```

Los ejemplos **EJ3-1** y **EJ3-2** tienen las mismas sentencias. Es importante cuidar el estilo de codificación y no sacrificar la claridad por ahorrar unas líneas de programa. El abuso de líneas multisentencia puede entorpecer la legibilidad.

Ejemplo 3-3. *Líneas de continuación con formato libre*

EJ3-3

```
CHARACTER(LEN=200) nombre_largo

WRITE (*,'(A)',ADVANCE='NO') ' y = '
READ*, y

t1cosh = EXP(y) +  &   ! Linea inicial de la sentencia
         EXP(-y)       ! Continuacion

t2cosh = EXP(y) +  &   ! Linea inicial de la sentencia
       & EXP(-y)       ! Continuacion

t3cosh = EXP(y) + EX&  ! Linea inicial de la sentencia
         &P(-y)        ! Continuacion

nombre_largo = &
   'De un lugar de la Mancha &
   &cuyo nombre no se si alguien sabra &
   &se escribio un famoso libro &
   &que EL QUIJOTE vinose a llamar'

PRINT*, y, t1cosh, t2cosh, t3cosh
PRINT*, nombre_largo
END
```

***Ejemplo 3-4**. Ejemplo válido para formato libre y formato fijo* **EJ3-4**

```
Columna  1         2         3         4         5         6         7  7
1234567890123456789012345678901234567890123456789012345678901234567890123
```

```
      PRINT*, seno(1.04), SIN(1.04)
      END

! funcion de usuario seno

      FUNCTION seno(x)
          seno = x - x**3/factorial(3) + x**5/factorial(5)              &
     &               - x**7/factorial(7)
      CONTAINS
        INTEGER FUNCTION factorial(n)
          factorial = 1
          DO i = n, 1, -1
            factorial = factorial*i
          ENDDO
        END FUNCTION factorial
      END FUNCTION seno
```

3.3. TIPOS INTRÍNSECOS DE DATOS

Fortran tiene cinco tipos básicos de datos:

- Enteros (`INTEGER`)
- Reales (`REAL, DOUBLE PRECISION`)
- Complejos (`COMPLEX`)
- Lógicos (`LOGICAL`)
- Caracteres (`CHARACTER, CHARACTER(LEN=n), CHARACTER*n`)

3.3.1. Constantes. Rangos

Tipo de dato	Bytes	Rango de valores más frecuentes
INTEGER	4	-2147483648 a 2147483647
REAL	4	±1.175e-38 a ±3.402e+38
DOUBLE PRECISION	8	±2.225d-308 a ±1.797d+308
COMPLEX	8	(preal,pimag) como REAL
LOGICAL	4	.TRUE. ó .FALSE.
CHARACTER	1	1 carácter
CHARACTER(LEN=n)	n	n caracteres (n<=32767)

Extensiones FORTRAN 77

Muchos compiladores de FORTRAN 77 admiten los siguientes tipos de datos:

```
BYTE, INTEGER*1, INTEGER*2, INTEGER*4
REAL*4, REAL*8
COMPLEX*8, DOUBLE COMPLEX, COMPLEX*16
LOGICAL*1, LOGICAL*2, LOGICAL*4
```

Tipo de dato	Bytes	Rango de valores más frecuentes
INTEGER	2 ó 4	depende
INTEGER*1 (BYTE)	1	-128 a 127
INTEGER*2	2	-32768 a 32767
INTEGER*4	4	-2147483648 a 2147483647
REAL, REAL*4	4	±1.175e-38 a ±3.402e+38
DOUBLE PRECISION, REAL*8	8	±2.225d-308 a ±1.797d+308
COMPLEX, COMPLEX*8	8	(preal,pimag) como REAL*4
DOUBLE COMPLEX, COMPLEX*16	16	(preal,pimag) como REAL*8
LOGICAL	2 ó 4	.TRUE. ó .FALSE.
LOGICAL*1	1	.TRUE. ó .FALSE.
LOGICAL*2	2	.TRUE. ó .FALSE.
LOGICAL*4	4	.TRUE. ó .FALSE.
CHARACTER	1	1 carácter
CHARACTER*n	n	n caracteres (n<=32767)

Algunos compiladores de Fortran 90 admiten esta sintaxis, pero pueden cambiar las características de los tipos de datos (rango, precisión).

***Ejemplo 3-5**. Constantes válidas y no válidas* **EJ3-5**

`INTEGER` válidas

```
-12
 38563
 0
-50546874
```

`INTEGER` no válidas, erróneas

```
 5.7456
-3457356467
 2147483648
 12e2
```

`REAL` válidas

```
 5.6345
 38896.45e12
-45.67e-31
 7.45e0
```

`REAL` no válidas, erróneas

```
 1,565
-563.56e-58
 96789678968946
 3.46e2.6
```

`DOUBLE PRECISION` válidas

```
 0.56345d1
-8.23452d0
 38896.45d120
 45.67d-300
```

`DOUBLE PRECISION` no válidas, erróneas

```
 1,565
 7.8567e39
-563.56d-5879
 96789678968946
```

`COMPLEX` válidas

```
(1.3,4.5)
(-0.45e10, 7.23)
(2.d13,-5.764d-1)
(0,1)
```

`COMPLEX` no válidas, erróneas

```
(2.3e66,1.3)
4.5,3.98
(3.0,i)
(12345678912345,7.2)
```

`LOGICAL` válidas

```
.TRUE.
.false.
```

`LOGICAL` no válidas, erróneas

```
.T.
.FALSO.
```

`CHARACTER` válidas

```
'hola ¿qué tal?'
'O''DONNELL'
```

`CHARACTER` no válidas ó erróneas

```
'lunes"
'O'DONNELL'
```

3.3.2. Parámetros. Variables. Declaración. Asignación

- Un parámetro tiene un valor que no se puede cambiar (PARAMETER).
- Una variable puede cambiar su valor cuantas veces sea necesario.
- Por defecto, todas las variables que empiezan por `i, j, k, l, m` ó `n` son enteras y las demás reales. Es muy recomendable declarar todas las variables que se utilicen (la sentencia `IMPLICIT NONE` obliga a declarar todas las variables).

Ejemplo 3-6. *Parámetros. Variables. Declaración. Asignación* **EJ3-6**

```
INTEGER:: a, x, n
DOUBLE PRECISION doble
COMPLEX:: c, d
CHARACTER(LEN=10) nombre, vocales*5, comput
LOGICAL:: logico, zz
PARAMETER (n=5, r=89.34, pi=3.141592, lac=-40, zz=.FALSE.,       &
          c=(2.45e2,-1.17), comput='ordenador', vocales='aeiou')

!  daria error poner n=8 porque no se puede cambiar un parametro

x = 215
doble = 6345.700234512846d-125
logico = .TRUE.
d = (2,4.5)
a = 1200
x = -1                ! se puede cambiar el valor de una variable
nombre = 'Tarzán'     ! atencion al acento en algunos compiladores
PRINT*, pi, x, doble, logico, d, a
PRINT*, nombre, vocales
END
```

3.3.3. Arrays. Subíndinces. Substrings

- Un array se define mediante su nombre y dimensiones (cantidad y límites).
- Por defecto el primer índice es 1. En otro caso, hay que indicar el rango `i1:i2`.
- Los elementos del array se acceden por sus índices entre paréntesis.

Ejemplo 3-7. *Declaraciones de arrays y acceso a sus elementos* **EJ3-7**

```
INTEGER n(10), n1(3,5), c(4,4,4,4), hh(0:4), bb(-6:4,2:5,75:99)
REAL r1(5), r2(-2:4,6)
CHARACTER(LEN=15):: mes(12), dia(0:6)

n(3) = 4563
n1(2,4) = n1(1,3) + 89
hh(0) = -1
bb(-3,4,80) = bb(-5,2,90) + 2.3*c(3,3,3,1)
r2(-1,6) = r1(3) + r2(-2,3)
mes(2) = 'FEBRERO'
dia(0) = 'DOMINGO'
PRINT*, n(3), n1(2,4), hh(0), mes(2)
END
```

- En un dato CHARACTER, también llamado *string*, se puede acceder a sus caracteres individuales y obtener *substrings*.

Ejemplo 3-8. *Acceso a los caracteres de un string* **EJ3-8**

```
CHARACTER(LEN=15) mes(12), dia(0:6), estacion
CHARACTER(LEN=3) submes, diasem, trozo*6
mes(1) = 'ENERO'
mes(7) = 'JULIO'
dia(3) = 'miercoles'
dia(6) = 'sabado'
estacion = 'primavera'
submes = mes(1)(2:4)      !  submes <- 'NER'
diasem = dia(3)(:3)       !  diasem <- 'mie'
trozo = estacion(5:)      !  trozo  <- 'avera '
PRINT*, mes(1), submes, dia(3), diasem, estacion, trozo
END
```

3.4. OPERADORES. EXPRESIONES. PRIORIDADES

3.4.1. Operadores y Expresiones Aritméticas

- Los operadores aritméticos son `+` , `-` , `*` , `/` , `**` (potenciación).
- Orden de prioridad: `**` , `*` y `/` , `+` y `-`.
- Paréntesis: las mismas reglas del álgebra.
- Orden de evaluación: se respetan los paréntesis y las prioridades. Dos operaciones consecutivas con la misma prioridad se efectúan de izquierda a derecha, salvo la doble potenciación que se efectúa de derecha a izquierda.

Ejemplo 3-9. *Operadores y expresiones aritméticas* **EJ3-9**

```
x1 = a + b*c          ! equivalente a  x1 = a + (b*c)
x2 = a*b/c            ! equivalente a  x2 = (a*b) / c
x3 = a*b-c+d*e**f-g   ! equivalente a  x3 = (a*b) - c + d*(e**f) - g
x4 = a**b**c          ! equivalente a  x4 = a**(b**c)
x5 = -a**b            ! equivalente a  x5 = -(a**b)
```

- No puede haber dos operadores seguidos
  Es incorrecto: `a*-b` debe ponerse `a*(-b)`
  Es incorrecto: `a**-b` debe ponerse `a**(-b)`
- Si una expresión o subexpresión no tiene paréntesis el procesador puede evaluar cualquier expresión equivalente. Así, `a/b/c` podría evaluarse como `a/(b*c)`.

3.4.1.1. Aritmética Entera

- El resultado de una suma, resta ó producto de datos enteros no debe salir del rango válido para valores enteros; si lo rebasa el valor calculado carece de sentido.
  Es `123456*78901=9740801856`. Sin embargo, como excede el mayor entero representable (`2147483647`), no se obtiene el resultado correcto.

- En la división entre datos enteros se pierden los decimales:

Ejemplo 3-10. *División entera* EJ3-10

Expresión	***Resultado***	***Expresión***	***Resultado***
`7/5`	`1`	`-7/5`	`-1`
`3*10/3`	`10`	`3*(10/3)`	`9`
`5*2/5`	`2`	`2/5*5`	`0`
`1/3 + 1/3 + 1/3`	`0`	`3**(-2)`	`0 : =1/(3**2)=1/9=0`

El efecto es el mismo si en lugar de constantes intervienen variables enteras.

3.4.1.2. Mezcla de Operandos. Conversiones

- Los operandos de una expresión pueden ser de diversos tipos (enteros, reales, complejos).
  La operación `a#b`, donde # es `+, -, *` ó `/` se rige por las siguientes reglas:
  (I denota INTEGER, R denota REAL, C denota COMPLEX).

Tipo de `a`	Tipo de `b`	Valor de `a` usado	Valor de `b` usado	Tipo del resultado
I	I	`a`	`b`	I
I	R	`REAL(a)`	`b`	R
I	C	`CMPLX(a)`	`b`	C
R	I	`a`	`REAL(b)`	R
R	R	`a`	`b`	R
R	C	`CMPLX(a)`	`b`	C
C	I	`a`	`CMPLX(b)`	C
C	R	`a`	`CMPLX(b)`	C
C	C	`a`	`b`	C

Las funciones intrínsecas `REAL`, `CMPLX` convierten su argumento al tipo real ó complejo, respectivamente.

- La operación `a**b` se rige por las siguientes reglas:

Tipo de `a`	Tipo de `b`	Valor de `a` usado	Valor de `b` usado	Tipo del resultado
I	I	`a`	`b`	I
I	R	`REAL(a)`	`b`	R
I	C	`CMPLX(a)`	`b`	C
R	I	`a`	`b`	R
R	R	`a`	`b`	R
R	C	`CMPLX(a)`	`b`	C
C	I	`a`	`b`	C
C	R	`a`	`CMPLX(b)`	C
C	C	`a`	`b`	C

- Da error si se eleva un número negativo a un exponente real.
- La exponenciación compleja se define como $a^b = \exp(b(\ln|a| + i\arg a))$, donde $-\pi < \arg a \le \pi$, esto es, se toma el valor principal.
- Como idea general, si los tipos no coinciden se convierte el operando de menor precisión al tipo del operando de mayor precisión o rango, según la escala INTEGER→REAL→DOUBLE PRECISION, REAL→COMPLEX.

Ejemplo 3-11. *Conversiones de operandos* **EJ3-11**

Expresión	***Resultado***	***Expresión***	***Resultado***
`-7/5.0`	`-1.400000`	`4.2+(1,2)`	`(5.200000,2.000000)`
`7.d0/1.4d0`	`5.000000000000000`	`7.d0/1.4`	`5.000000085149494`
`3**(-2.0)`	`0.111111`	`(-3.0)**2.5`	`¡ERROR!`
`3*(1/REAL(3))`	`1.000000`	`3*(10/3.0)`	`10.00000`

3.4.2. Operadores y Expresiones de Caracteres

- Operador de concatenación (une datos CHARACTER): `//`

Ejemplo 3-12. *Concatenación de caracteres* **EJ3-12**

```
CHARACTER uno*10, dos*8, todo*18
CHARACTER(LEN=15), DIMENSION(100):: color
uno = 'PRIMERO'
dos = ' SEGUNDO'
color(23)= 'magenta'
todo = uno // dos

PRINT*,' todo = ', todo        ! todo = 'PRIMERO    SEGUNDO'
PRINT*,'abc'//'EF'//'1 2 3'    ! 'abcEF1 2 3'
PRINT*, uno(3:5)//color(23)(2:5)//dos(1:4)     ! 'IMEagen SEG'

END
```

3.4.3. Operadores y Expresiones de Relación

- Los operadores de relación de expresiones son:

Antes de Fortran 90	`.EQ.`	`.NE.`	`.LT.`	`.LE.`	`.GT.`	`.GE.`
Nuevos en Fortran 90	`==`	`/=`	`<`	`<=`	`>`	`>=`
Significado	igual	distinto	menor	menor o igual	mayor	mayor o igual

- Las operaciones de relación con datos de tipo complejo sólo pueden utilizar `.EQ. (==)` y `.NE. (/=)`.
- Entre caracteres se cumple `A<B...<Z; 0<1...<9;` {*b*`<A, Z<0`} ó {*b*`<0, 9<A`}; `a<b<...<z;` {*b*`<a, z<0`} ó {*b*`<0, 9<a`}.
  Depende del procesador si las letras van antes que los números o al revés y el orden del resto de caracteres ASCII.

3.4.4. Operadores y Expresiones Lógicas

- Los operadores lógicos son:

Operador	.NOT.	.AND.	.OR.	.EQV.	.NEQV.
Prioridad	1	2	3	4	4
Significado	Negación	Conjunción	Disy. inclusiva	Equivalencia	No equivalencia

Ejemplo 3-13. *Prioridades entre operadores lógicos* **EJ3-13**

```
LOGICAL a, b, c, d, e, f1, f2, f3, f4, f5, f6
a=.TRUE.; b=.TRUE.; c=.TRUE.; d=.FALSE.; e=.FALSE.

! Los siguientes pares de sentencias son equivalentes

f1 = a .AND. b .AND. c
f2 = (a .AND. b) .AND. c

f3 = .NOT.a .OR. b .AND. c
f4 = (.NOT.a) .OR. (b.AND.c)

f5 = .NOT.a .EQV. b .OR. c .NEQV. d .AND. e
f6 = ((.NOT.a) .EQV. (b.OR.c)) .NEQV. (d.AND.e)

PRINT*, f1, f2 ; PRINT*, f3, f4 ; PRINT*, f5, f6
END
```

- Se pueden relacionar expresiones aritméticas con expresiones lógicas y expresiones de caracteres. Si en una expresión aparecen operadores de varios tipos, el orden de prioridad es: aritméticos, de caracteres, relacionales y lógicos.
- El resultado de una expresión relacional es de tipo lógico.

Ejemplo 3-14. *Expresiones mixtas* **EJ3-14**

```
LOGICAL a, f1, f2, f3, f4
CHARACTER(LEN=4) c1, c2, c3*8

a = .true.
x = 5
y = 7
z = 8
c1 = 'abcd'
c2 = '1234'
c3 = 'abcd1234'

! Los siguientes pares de sentencias son equivalentes

f1 = .NOT.a .OR. x+y<z
f2 = (.NOT.a) .OR. (x+y<z)

f3 = c1(1:4)//c2==c3 .AND. x*y<z .OR. a
f4 = ( ((c1(1:4)//c2)==c3) .AND. (x*y<z) ) .OR. a

PRINT*, f1, f2 ; PRINT*, f3, f4
END
```

- Es recomendable utilizar paréntesis y/ó sustituir las expresiones complicadas por combinaciones de expresiones más simples.
- Si se puede determinar el valor de una expresión lógica sin evaluar todas las subexpresiones, algunas de éstas podrían no llegar a evaluarse.
  Ejemplo: Sea la expresión `i<10 .AND. x(i)+y(j)<57`
  Si el valor de `i` es 23 puede no evaluarse la subexpresión `x(i)+y(j)<57`, ya que la expresión será `.FALSE.`.

3.5. ENTRADA Y SALIDA ESTÁNDAR SIN FORMATO

Los dispositivos estándar (por defecto) de entrada y salida de datos son el teclado y la pantalla.

- Lectura de datos de teclado. Son equivalentes las siguientes sentencias:

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

- Escritura de datos en pantalla. Son equivalentes las siguientes sentencias:

```
WRITE (*,*) listavar
PRINT*, listavar
```

`listavar:` es una lista de variables o elementos de arrays separados por comas.

Ejemplo 3-15. *Lectura y escritura de datos sin formato* **EJ3-15**

```
REAL y(10), z(4,10)
CHARACTER(LEN=2) alfa
READ (*,*) x, y(1), z(2,6), alfa
WRITE (*,*) x, y(1), z(2,6)+x*y(1), alfa
WRITE (*,*) 'y1=',y(1),'   alfa=',alfa
END
```

si se introducen por teclado los datos (separados por comas o por espacios):

```
2.5 2.0 5 'ab'
```

la salida por pantalla sería:

```
   2.500000   2.000000   10.000000 ab
y1=    2.000000   alfa=ab
```

3.6. SENTENCIAS `PROGRAM, END`

- Un programa puede comenzar con la sentencia `PROGRAM nombprog`.
- Un programa debe terminar con la sentencia `END [PROGRAM [nombprog]]`
  `nombprog` es el nombre del programa que debe empezar por una letra y admite hasta 31 letras, dígitos, y guiones underscore `_`.

3.7. PROGRAMA EJEMPLO

EJ3-16

Este programa calcula la superficie lateral y total y el volumen de un cilindro y de un cono introduciendo como datos el radio de la base `r` y la altura `h,` y la superficie y el volumen de una esfera de radio `r`.

```
PROGRAM cuerpos
IMPLICIT NONE
REAL r, h, g, slat, stot, vol
REAL, PARAMETER:: pi=3.141592

! CILINDRO y CONO:
!  Variables de entrada:
!      r: radio de la base
!      h: altura
!  Variables de salida:
!      slat: superficie lateral
!      stot: superficie total
!      vol : volumen

WRITE (*,'(A)',ADVANCE='NO') ' radio de la base = '
READ*, r
WRITE (*,'(A)',ADVANCE='NO') ' altura = '
READ*, h

slat = 2.0*pi*r*h
stot = slat + 2.0*pi*r**2
vol = pi*r**2 * h
WRITE (*,*) ' CILINDRO'
WRITE (*,*) ' Superficie lateral = ', slat
WRITE (*,*) ' Superficie total   = ', stot
WRITE (*,*) ' Volumen      = ', vol

g = (h**2+r**2)**0.5
slat = pi*r*g
stot = slat + pi*r**2
vol = vol/3
WRITE (*,*)
WRITE (*,*) ' CONO'
WRITE (*,*) ' Superficie lateral = ', slat
WRITE (*,*) ' Superficie total   = ', stot
WRITE (*,*) ' Volumen      = ', vol

! ESFERA:
!  Variable de entrada:
!      r: radio
!  Variables de salida
!      stot: superficie total
!      vol : volumen

WRITE (*,'(/A)',ADVANCE='NO') ' radio de la esfera = '
READ*, r
stot = 4.0*pi*r**2
vol = 4.0/3.0*pi*r**3
PRINT*
WRITE (*,*) ' Superficie esferica = ', stot
WRITE (*,*) ' Volumen      = ', vol

ENDPROGRAM cuerpos
```

3.8. EJERCICIOS

1. Indicar cuáles de las siguientes constantes son ó no son válidas en Fortran:

	Constante		Constante		Constante
(a)	`4.678`	**(j)**	`37.432458948`	**(r)**	`pi`
(b)	`7,67`	**(k)**	`12345678901`	**(s)**	`7*e3`
(c)	`-3.`	**(l)**	`3.98e+22`	**(t)**	`-1e-1`
(d)	`45e0`	**(m)**	`5.8e444`	**(u)**	`.si.`
(e)	`3+i`	**(n)**	`-6.234e-14`	**(v)**	`.t.`
(f)	`(3,i)`	**(ñ)**	`5.3e-333`	**(w)**	`.f`
(g)	`(3,1)`	**(o)**	`(3,3.)`	**(x)**	`'isn''t'`
(h)	`(3.1)`	**(p)**	`(4.5,0)`	**(y)**	`'isn”t'`
(i)	`(12.3,56.78)`	**(q)**	`(4,5.0)`	**(z)**	`'isn't'`

2. Indicar la validez de las siguientes expresiones y obtener su valor si son válidas:

	Expresión	Valor		Expresión	Valor
(a)	`5+3/4`		**(h)**	`(3,2)*(5,3)`	
(b)	`5*1/5`		**(i)**	`3*(4.2,-1.5)`	
(c)	`16**(1/2)`		**(j)**	`-14/5*3**2+1`	
(d)	`16**(0.5)`		**(k)**	`4+5*3-2**6/2`	
(e)	`16**(1./2)`		**(l)**	`ser.OR.NOT.ser`	
(f)	`3*-3`		**(m)**	`a.OR.(b.AND..NOT.a)`	
(g)	`3*(-3)`		**(n)**	`'H'//'Sol'(2:2)//'las'(:2)`	

3. Escribir las siguientes expresiones en Fortran:

	Expresión	Fortran
(a)	$3a^2*(x_1+x_n)/3$	
(b)	$\frac{1}{2}\left(n^2+n(n^3-1)/3\right)$	
(c)	$\sqrt[n]{x_1(3-x_2)^2}$	
(d)	$\left(1+\frac{1}{n}\right)^n$	
(e)	$n^n e^{-n}\sqrt{2\pi n}$	
(f)	$7x^2-8xy/(x+1/y)$	

4. Escribir unas líneas en Fortran para intercambiar el contenido de las variables `a`, `b` utilizando una variable adicional `aux`. Intentar hacer dicho intercambio sin utilizar variables adicionales.

5. Escribir un programa Fortran para convertir grados Celsius a Fahrenheit.

CAPÍTULO 4. *SENTENCIAS DE CONTROL*

Sirven para alterar la ejecución secuencial de las sentencias de un programa.

4.1. SENTENCIA `CONTINUE`

La sentencia `CONTINUE` es ejecutable, pero no realiza acción alguna. Es útil para rupturas de secuencia y manejo de errores en lectura de datos. Su número de etiqueta puede ser referenciado en sentencias `DO` (ver apartado **4.7.3**) ó `GOTO`.

Ejemplos (*Ver ejemplos **EJ4-1**, **EJ4-15**, **EJ4-16**, **EJ4-17**, **EJ4-18***)

4.2. SENTENCIA `STOP`

Sintaxis: `STOP`
Variante 1: `STOP ['mensaje']`
Variante 2: `STOP [n]`
Acción: Detiene la ejecución del programa. Si está presente el literal `'mensaje'` ó el número `n` (que ha de tener de 1 a 5 dígitos y si lleva ceros iniciales no son significativos), se visualizan en pantalla.
Normas: Puede llevar etiqueta y puede formar parte de una sentencia `IF`.
Notas:
- Sirve principalmente para detener la ejecución a causa de un error y con el literal `'mensaje'` ó el número `n` puede indicarse la causa de la detención.

Ejemplos (*Ver ejemplos **EJ4.2**, **EJ4.7**, **EJ4.10**, **EJ4.21**, **EJ4.22**, **EJ4.24**, entre otros*)

4.3. SENTENCIA `GOTO` INCONDICIONAL

Sintaxis: `GOTO e`
Acción: Transfiere el control a la sentencia ejecutable con etiqueta `e` que se encuentra en la misma unidad de programa que la sentencia `GOTO`.

Ejemplo 4-1. `GOTO` *incondicional* **EJ4-1**

```
x = 3.8
GOTO 100! es lo mismo GOTO que GO TO
PRINT*, ' Esta frase no se imprime'
x = -444! Esta sentencia no se ejecuta
100 CONTINUE
y = x + 1.0
PRINT*, x, y
END
```

Normas: No se puede entrar en bloques `DO`, `IF`, `CASE` desde fuera de ellos con sentencias `GOTO`.

Notas:

- Es una sentencia cuyo uso genera mucha polémica. Un programa con gran cantidad de `GOTO` es difícil de comprender, sobre todo si hay muchas transferencias a sentencias anteriores.
  Con una adecuada programación pueden sustituirse, con facilidad, la mayoría de las sentencias `GOTO` por otras estructuras de control. Sin embargo, hay ocasiones cuya sustitución complica enormemente la lógica del programa. Es especialmente útil para tratar condiciones de error o de terminación de un bloque.
  Conviene NO abusar de esta sentencia.
- Si la siguiente sentencia a la sentencia `GOTO` no lleva etiqueta no se ejecutará nunca (código muerto). Es un síntoma de error de programación.

4.4. SENTENCIA `IF`. BLOQUES `IF`

Sirven para realizar una ejecución condicional de sentencias.

4.4.1. `IF` lógico

Sintaxis: `IF (expres) sentenc`

Acción: La expresión `expres` debe ser escalar lógica. Si es verdadera se ejecuta la sentencia `sentenc`; si es falsa no se ejecuta `sentenc` y se continúa en la sentencia siguiente.

Normas: La sentencia `sentenc` debe ser ejecutable. No puede ser la sentencia `END` ni la sentencia inicial o final de bloques `DO`, `IF`, `SELECT`, `CASE`.

Notas:

- Se usa normalmente para realizar, dependiendo de una condición: una única asignación, una sencilla escritura de datos, una parada del programa, una ramificación del flujo del programa.

Ejemplo 4-2. `IF` *lógico* **EJ4-2**

```
REAL x, y, h, v(30,20)
INTEGER i
LOGICAL xmenory

10 PRINT*, ' Introducir i entre 1 y 30: '
READ*, i
IF (i<=0 .OR. i>30) GOTO 10

x = 3.5; y = 5.6; v(i,3) = 1.4*i
xmenory = x < y
IF (x+y < v(i,3)) h = h + 1.5
IF (i == 4) STOP   ! Detiene la ejecucion del programa
IF (xmenory) PRINT*, x, y, v(i,3)
END
```

4.4.2. Bloque `IF-THEN-ENDIF`

Sintaxis: `[nomb:] IF (expres) THEN`
`bloq......`
`ENDIF [nomb]`

Acción: La expresión `expres` debe ser escalar lógica. Si es verdadera se ejecutan las sentencias del bloque `bloq` entre `THEN` y `ENDIF`. Si es falsa se continúa en la siguiente sentencia a `ENDIF`.

Normas: Si lleva nombre (opcional), debe ser un nombre válido en Fortran distinto de otros nombres en la unidad de alcance en la que está el bloque `IF`.
No se puede entrar en un bloque `THEN-ENDIF` desde fuera de él con sentencias `GOTO`. Se puede salir en cualquier lugar con sentencias `GOTO`.

Ejemplo 4-3. *Bloque* `THEN-ENDIF` **EJ4-3**

```
REAL x, y, temp
PRINT*, ' Valor de x = '
READ*, x
PRINT*, ' Valor de y = '
READ*, y
PRINT*, x, y

IF (x > y) THEN    ! intercambio: IF (...
  temp = x
  x = y
  y = temp
ENDIF ! ENDIF intercambio

PRINT*, 'Si x>y se han intercambiado x, y'
PRINT*, ' x = ', x, ' y = ', y

END
```

4.4.3. Bloque `IF-THEN-ELSE-ENDIF`

Sintaxis: `[nomb:] IF (expres) THEN`
`bloq1`
`ELSE [nomb]`
`bloq2`
`ENDIF [nomb]`

Acción: La expresión `expres` debe ser escalar lógica. Si es verdadera se ejecutan las sentencias del bloque `bloq1` entre `THEN` y `ENDIF`. Si es falsa se ejecutan las sentencias del bloque `bloq2` entre `ELSE` y `ENDIF`.

Normas: La sentencia `ELSE` puede llevar nombre sólo si las sentencias `IF` y `ENDIF` correspondientes lo llevan y, en este caso, debe ser el mismo.
No se puede entrar en un bloque `THEN-ENDIF`, `THEN-ELSE`, `ELSE-ENDIF` desde fuera de dicho bloque con sentencias `GOTO`. Se puede salir en cualquier lugar con sentencias `GOTO`.

Ejemplo 4-4. *Bloques* `THEN-ELSE-ENDIF` EJ4-4

```
PRINT*, ' Valores de x, y' ; READ*, x, y

IF (x > y) THEN
  x = x - 1.0
ELSE
  x = 2.*x
  IF (y >= 50) THEN
    z = x + y
    GOTO 500
  ENDIF
ENDIF

PRINT*, ' Inicialmente era x>y o {x<=y, y<50}'
500 PRINT*, x, y, z   ! z puede no tener valor asignado
END
```

Ejemplo 4-5. *Bloques* `THEN-ELSE-ENDIF` EJ4-5

```
PRINT*, ' Introducir i, j, k =' ; READ*, i, j, k

IF (i < 100) THEN
  PRINT*, ' i<100'
  IF (j < 10) THEN
    PRINT*, ' i<100 y j<10'
  ENDIF
ELSE
  IF (k >= 50) THEN
    PRINT*, ' i>=100 y k>=50'
  ENDIF
  PRINT*, 'i>=100'
ENDIF

END
```

4.4.4. Bloques `ELSE IF`

Sintaxis: `[nomb:] IF (expres) THEN`
`bloq1`
`[ELSEIF (expres_i) THEN [nomb]`
`bloq_i]...`
`[ELSE [nomb]`
`bloq2]`
`ENDIF [nomb]`

Acción: Cada expresión `expres`, `expres_i` debe ser escalar lógica. Si `expres` es verdadera se ejecutan las sentencias del bloque `bloq1` entre `THEN` y el primer `ELSEIF` y se pasa a la siguiente sentencia a `ENDIF`.
Si `expres` es falsa se inspeccionan en orden las expresiones `expres_i` hasta que una sea verdadera, en cuyo caso se ejecutan las sentencias del bloque `bloq_i` correspondiente y el control pasa a la siguiente sentencia a `ENDIF`.
Si la expresión `expres` y todas las espresiones `expres_i` son falsas se ejecutan las sentencias del bloque `bloq2` entre `ELSE` y `ENDIF`.

Normas: Las sentencias `ELSEIF` y `ELSE` pueden llevar nombre sólo si sus sentencias `IF` y `ENDIF` lo llevan y, en este caso, debe ser el mismo.
No se puede entrar en un bloque `IF`, `THEN`, `ELSEIF` ni `ELSE` desde fuera de él con sentencias `GOTO`. Se puede salir en cualquier lugar con sentencias `GOTO`.
Los bloques `IF` pueden anidarse.

Ejemplo 4-6. *Evaluación de una función. Sea* $f(x)=\begin{cases} 3x^2-1 & \text{si } 0\le x\le 1 \\ 6x+4 & \text{si } 5\le x\le 10 \\ -7x+1 & \text{si } 20\le x\le 40 \\ 0 & \text{en otro caso} \end{cases}$ **EJ4-6**

```
REAL x, f

PRINT*, ' valor de x = ' ; READ*, x

IF (0<=x .AND. x<=1) THEN
  f = 3*x**2 - 1
ELSEIF (5<=x .AND. x<=10) THEN
  f = 6*x + 4
ELSEIF (20<=x .AND. x<=40) THEN
  f = -7*x + 1
ELSE
  f = 0
ENDIF

PRINT*, ' x = ', x, ' f = ', f
END
```

Ejemplo 4-7. *Elementos de una matriz. Sea* $a_{ij}=\begin{cases} 1 & \text{si } i=j=1 \\ 2 & \text{si } i=j>1 \\ -1 & \text{si } |i-j|=1 \\ 0 & \text{en otro caso} \end{cases}$ **EJ4-7**

```
INTEGER, PARAMETER:: n=5
INTEGER, DIMENSION(n,n):: a, b

i = 0
10 i = i + 1

j = 0
20 j = j + 1

IF (j == i) THEN
  IF (j == 1) THEN
    a(i,j) = 1
  ELSE
    a(i,j) = 2
  ENDIF
ELSEIF (ABS(i-j) == 1) THEN   ! ABS es el valor absoluto
  a(i,j) = -1
ELSE
  a(i,j) = 0
ENDIF
```

```
! Otra forma equivalente es

b(i,j) = 0
IF (j==i .AND. j==1) b(i,j)=1
IF (j==i .AND. j>1) b(i,j)=2
IF (ABS(i-j) == 1) b(i,j)=-1
IF (a(i,j) /= b(i,j)) STOP ' Error de programa'

PRINT*, ' a(', i, ',', j, ')=', a(i,j)

IF (j < n) GOTO 20
IF (i < n) GOTO 10
END
```

Notas:

- Es conveniente no abusar de estructuras complicadas. Cuando la cantidad de `IF` anidados es grande y no se encuentran alternativas, es recomendable poner nombres a los bloques.

4.5. SELECTOR `SELECT CASE` (Nuevo en Fortran 90)

Sintaxis:
```
[nomb:] SELECT CASE (expres)
     [CASE (selector) [nomb]
 bloq]...
     [CASE DEFAULT [nomb]
 bloq0]
     ENDSELECT [nomb]
```

Acción: La expresión `expres` debe ser escalar de tipo entera, lógica (*poco interesante*) ó carácter y los valores indicados en cada selector deben ser del mismo tipo. (En el caso carácter las longitudes pueden ser diferentes pero no la clase, en los casos entero ó lógico las clases pueden ser diferentes).

Si el valor de `expres` pertenece a un selector se ejecuta su bloque de sentencias y se continúa en la siguiente sentencia a `END SELECT`. Si no pertenece a ningún selector se ejecutan las sentencias del bloque `CASE DEFAULT`, si está presente y si no lo está se continúa en la siguiente sentencia a `END SELECT`.

Normas: Si lleva nombre (opcional), debe ser un nombre válido en Fortran y distinto de otros nombres en la unidad de alcance en que se encuentra el bloque `SELECT`.

Las sentencias `CASE` y `CASE DEFAULT` pueden llevar nombre sólo si las sentencias `SELECT CASE` y `ENDSELECT` correspondientes lo llevan y, en este caso, debe ser el mismo.

Las valores de los selectores han de ser disjuntos. Se separan por comas y puede especificarse un rango de valores, también disjuntos en cada selector.

No se puede entrar en un bloque `SELECT` ó `CASE` desde fuera de él con sentencias `GOTO`. Se puede salir en cualquier lugar con sentencias `GOTO`.

Los bloques `SELECT CASE` pueden anidarse.

Notas:

- La principal diferencia entre los bloques `IF` y `CASE` es que en `CASE` sólo se evalúa una expresión cuyo valor debe estar en un conjunto predefinido de valores, mientras que en `IF` se pueden evaluar varias expresiones de naturaleza distinta.

***Ejemplo 4-8*. `SELECT CASE` *con variables* `CHARACTER` e `INTEGER`** EJ4-8

```
CHARACTER car! equivale a CHARACTER(LEN=1) car
INTEGER indice

PRINT*, ' Introducir un caracter'
READ*, car

SELECT CASE (car)
CASE ('a', 'e', 'i', 'o', 'u')
  PRINT*, ' Vocal minuscula : ', car
CASE ('A', 'E', 'I', 'O', 'U')
  PRINT*, ' Vocal MAYUSCULA : ', car
CASE ('b':'d', 'f':'h', 'j':'n', 'p':'t', 'v':'z', 'ñ')
  PRINT*, ' Consonante minuscula : ', car
CASE ('B':'D', 'F':'H', 'J':'N', 'P':'T', 'V':'Z', 'Ñ')
  PRINT*, ' Consonante MAYUSCULA : ', car
CASE ('0':'9')
  PRINT*, ' Cifra del 0 al 9 : ', car
CASE DEFAULT
  PRINT*, ' El caracter no es ni letra ni numero : ', car
ENDSELECT

PRINT*, ' Introducir un numero'
READ*, indice

SELECT CASE (indice)
CASE (2, 3, 5, 7, 11, 13, 17, 19)
  PRINT*, ' Numero primo menor que 20 : ', indice
CASE (20:29, 40:49, 60:69, 80:89)
  PRINT*, ' Numero menor que 100 con decena par : ', indice
CASE (100:999)
  PRINT*, ' Numero de 3 cifras : ', indice
CASE DEFAULT
  PRINT*, ' Resto de casos : ', indice
ENDSELECT

END
```

***Ejemplo 4-9*. *Comparación de código con bloques* `IF` y `CASE`** EJ4-9

```
PRINT*, ' Introducir i'
READ*, i

IF (i==1 .OR. i==5) THEN
  PRINT*, ' bloque 1 IF'
ELSEIF (i==3 .OR. (10<i .AND. i<=20)) THEN
  PRINT*, ' bloque 2 ELSEIF'
ELSEIF (i==6 .OR. (30<=i .AND. i<=50)) THEN
  PRINT*, ' bloque 3 ELSEIF'
ELSE
  PRINT*, ' bloque 4 ELSE'
ENDIF
```

```
SELECT CASE (i)
  CASE (1, 5)
  PRINT*, ' bloque 1 CASE'
CASE (3, 11:20)
  PRINT*, ' bloque 2 CASE'
CASE (6, 30:50)
  PRINT*, ' bloque 3 CASE'
CASE DEFAULT
  PRINT*, ' bloque 4 CASE DEFAULT'
ENDSELECT

END
```

4.6. ITERACIONES DO

Sintaxis: `[nomb:] DO [,] var = expres1, expres2[, expres3]`
`bloq`
`ENDDO [nomb]`

Acción: La variable `var` y las expresiones `expres1`, `expres2`, `expres3` (opcional) deben ser escalares enteras.
La variable `var` toma el valor inicial de `expres1`, se ejecutan las sentencias `bloq` del bloque `DO`; `var` se incrementa en `expres3`, se ejecutan las sentencias del bloque `DO`; y así sucesivamente hasta que `var>expres2` (si `expres3>0`) ó `var<expres2` (si `expres3<0`), en cuyo caso se continúa en la siguiente sentencia a `ENDDO`.

Normas: Si lleva nombre (opcional), debe ser un nombre válido en Fortran y distinto de otros nombres en la unidad de alcance en que se encuentra el bloque `DO`.
`expres3` es opcional (no puede ser cero); por defecto es `expres3=1`.
No se puede entrar en un bloque `DO` desde fuera de él con sentencias `GOTO`. Se puede salir en cualquier lugar con sentencias `GOTO`.
Los bloques `DO` pueden anidarse.

Notas:

- El número de iteraciones que se realizan en el bloque `DO` (si no se sale de él antes de terminar) es: `MAX{(expres2-expres1+expres3)/expres3, 0}`.
- Cuando `{expres1>expres2 y expres3>0}` ó `{expres1<expres2 y expres3<0}` no se ejecuta el bloque `DO`.
- Si `expres1` y/ó `expres2` y/ó `expres3` son expresiones que incluyen variables, el valor de éstas puede cambiarse dentro del bloque `DO` y esto no altera el número de iteraciones (calculado con sus valores iniciales).

Ejemplo 4-10. *Suma de los elementos de un vector* **EJ4-10**

```
INTEGER, PARAMETER:: n=100000
REAL x(n), suma1, suma2, suma3

DO i = 1, n
  x(i) = 1.0/i
ENDDO
```

```
suma1 = 0.0
DO i = 1, n
  suma1 = suma1 + x(i)
ENDDO
suma2 = 0.0
DO i = n, 1, -1
  suma2 = suma2 + x(i)
ENDDO
PRINT*, ' suma1 = ', suma1, ' suma2 = ', suma2

suma3 = 0
DO i = 1, 100000, 3    ! i toma los valores 1, 4, 7,...
  IF (suma3 <= 5) THEN
    suma3 = suma3 + x(i)
    PRINT*, ' i =', i, ' suma3 = ', suma3
  ELSE
    PRINT*, ' suma3 > 5'
    STOP
  ENDIF
ENDDO

END
```

Ejemplo 4-11. *Expresiones en los parámetros de un* `DO` EJ4-11

```
INTEGER n, m, suma

PRINT*, ' Introducir n, m (0,0 para terminar)'
READ*, n, m
IF (n==0 .AND. m==0) STOP

suma = 0
DO i = n*m, n+m+n*m, MIN(n,m)
  suma = suma + i
  PRINT*, ' n = ', n, ' m = ', m, ' Suma = ', suma
  n = 2*n -3
ENDDO
END
```

4.6.1. `DO` no limitado (Nuevo en Fortran 90)

Sintaxis: `[nomb:] DO`
`bloq`
`ENDDO [nomb]`

Acción: Se repiten las sentencias `bloq` entre `DO` y `ENDDO` indefinidamente. Se puede salir del bucle `DO` con las sentencias `EXIT`, `GOTO`.

4.6.2. `EXIT` (Nuevo en Fortran 90)

Sintaxis: `EXIT [nomb]`

Acción: Dentro de un bloque `DO` transfiere el control a la primera sentencia ejecutable después del `ENDDO` a que se refiere y termina el bloque `DO`. Si no se indica el nombre `nomb`, termina el bloque más interior en el que está contenida.

4.6.3. CYCLE (Nuevo en Fortran 90)

Sintaxis: `CYCLE [nomb]`

Acción: Dentro de un bloque `DO` transfiere el control a la sentencia `ENDDO` a que se refiere. Si no se indica el nombre `nomb`, transfiere el control al `ENDDO` del bloque más interior en el que está contenida.

Notas:

- El uso de `CYCLE` y `EXIT` permite suprimir muchas sentencias `GOTO`.

Ejemplo 4-12. `DO` *no limitado. Uso de* `CYCLE` y `EXIT` **EJ4-12**

Se introducen por teclado números enteros y se quiere contar la cantidad de números impares introducidos. La lectura termina cuando se lee un número negativo.

```
nimpar = 0
DO
  PRINT*, ' Introducir numero = '; READ*, n
  IF (n < 0) THEN
    EXIT
  ELSEIF (n-(n/2)*2 == 0) THEN
    CYCLE
  ELSE
    nimpar = nimpar + 1
  ENDIF
ENDDO

PRINT*, ' Cantidad de impares = ', nimpar
END
```

Valor de la variable `var` del bloque `DO` fuera del bloque:

El valor de la variable `var` del bloque `DO` está disponible fuera del bloque después de su ejecución. Pueden ocurrir tres situaciones:

- Si {`expres1>expres2` y `expres3>0`} ó {`expres1<expres2` y `expres3<0`} no se ejecuta el bloque `DO`. El valor de `var` es `expres1`.
- Si se sale del bloque `DO` con `GOTO` ó `EXIT`, el valor de `var` es el que tenga en ese momento.
- Si se realizan todas las iteraciones del bloque `DO`, al ejecutar `ENDDO` por última vez `var` se incrementa en `expres3` y éste será su valor fuera del bloque.

Ejemplo 4-13. *Valor de la variable del bloque* `DO` *una vez terminado* **EJ4-13**

```
PRINT*, ' Introducir n, nfin'
READ*, n, nfin

DO i = 10, n, 5
  PRINT*, ' i = ', i
  IF (i == nfin) EXIT
ENDDO

PRINT*, ' Valor de i al salir del DO = ', i
END
```

Observaciones comunes a bloques `IF`, `CASE`, `DO:`

- Cuando se anidan bloques `IF`, `CASE`, `DO` el rango de un bloque debe estar totalmente contenido en el rango de otro, esto es, no pueden solaparse.
- Es muy recomendable indentar 2 ó 3 espacios las sentencias de estos bloques.

4.7. SENTENCIAS DE CONTROL REDUNDANTES

4.7.1. `DO WHILE` (Nuevo en Fortran 90)

Sintaxis:
```
[nomb:] DO WHILE (expres)
 bloq
     ENDDO [nomb]
```
Acción: se ejecutan las sentencias del bloque `bloq` mientras la expresión `expres` (escalar lógica) sea verdadera.

Ejemplo 4-14. *Relacionado con el ejemplo* ***EJ4-12***

EJ4-14

```
n = 0; suma = 0

DO WHILE (suma <= 5)
  n = n + 1
  suma = suma + 1.0/n
  PRINT*, ' n =', n, ' suma = ', suma
ENDDO

STOP ' suma > 5'
END
```

Alternativa:

- A pesar de haber sido introducida esta sentencia en Fortran 90 está considerada redundante. Puede sustituirse por un `DO` no limitado como sigue:

```
DO
  IF (.NOT.expres) EXIT
    bloq
ENDDO
```

  Esta construcción tiene la ventaja de que la condición de salida puede colocarse en cualquier lugar, no sólo al principio del bloque. En la construcción `DO WHILE` si `expres` se vuelve falsa en medio del bloque, el resto del bloque se ejecutaría.
- El uso de `DO WHILE` entorpece notablemente la optimización del código.

4.7.2. Bloques `DO` con etiqueta

Sintaxis:
```
[nomb:] DO [etiq][,]...
 bloq
   [etiq]  ENDDO [nomb]
```

Los bloques `DO`, `DO` no limitado, `DO WHILE` pueden llevar etiqueta. El uso de la sentencia `CYCLE` y del nombre `nomb` hacen redundante el etiquetado.

4.7.3. Final de bloques `DO` en `CONTINUE`

Los bloques `DO` pueden terminarse en la sentencia `CONTINUE` con etiqueta, realizando la misma función que la sentencia `ENDDO`. Es preferible el uso de `ENDDO`, ya que además de poder llevar nombre, su uso es inconfundible como final de un bloque `DO` mientras que `CONTINUE` puede usarse para otros fines.

Ejemplo 4-15. *Bloques* `DO` *con etiquetas y* `CONTINUE` **EJ4-15**

```
n = 25
DO 300 i = 1, n
  PRINT*, ' i =', i

    DO 200 j = 1, n
      DO 100 k = 1, n
        IF (k == i+j) GOTO 300
        IF (k+i+j == 13) GOTO 150
        IF (k+i+j == 10) GOTO 250
  100 CONTINUE
      PRINT*, ' j =', j
      GOTO 200
  150 CONTINUE
      PRINT*, ' i+j+k = 13'
200 CONTINUE

250 PRINT*, ' i+j+k = 10'
300 CONTINUE
END
```

Ejemplo 4-16. *Mejor codificación del ejemplo* ***EJ4-15*** **EJ4-16**

```
n = 25
odi: DO i = 1, n
       PRINT*, ' i =', i

       odj: DO j = 1, n
         odk: DO k = 1, n
                IF (k == i+j) CYCLE odi     ! GOTO 300
                IF (k+i+j == 13) GOTO 150
                IF (k+i+j == 10) EXIT odj   ! GOTO 250
              ENDDO odk

              PRINT*, ' j =', j
              CYCLE odj              ! GOTO 200
              150 CONTINUE           ! se puede quitar CONTINUE
              PRINT*, ' i+j+k = 13'  ! y poner     150 PRINT*,...
            ENDDO odj

  PRINT*, ' i+j+k = 10'
ENDDO odi
END
```

4.8. SENTENCIAS DE CONTROL OBSOLETAS

4.8.1. Sentencias obsoletas sólo en Fortran 95

4.8.1.1. `GOTO` calculado

Sintaxis: `GOTO (et1,...,etn)[,] expres`

Acción: La expresión `expres` debe ser escalar entera y las etiquetas `et1`,...,`etn` deben estar en la unidad de alcance que contiene a la sentencia `GOTO`.
Si el valor de `expres` es `j` se continúa en la sentencia con etiqueta `etj`. Si `expres<1` ó `expres>n` se continúa en la siguiente sentencia.
Las etiquetas `etj` pueden no estar ordenadas, puede haber varias `etj` iguales y pueden no ser posteriores a la sentencia `GOTO`.

Ejemplo 4-17. `GOTO` *calculado* EJ4-17

```
10 PRINT*, ' Introducir i'    ! aqui se va si i=1
READ*, i

GOTO (10,20,30,20) i ! la coma es opcional

PRINT*, ' i<1 ó i>4, i=', i! aqui se continua si i<1 ó i>4
GOTO 40
20 PRINT*, ' i=2 o i=4, i=', i! aqui se va si i=2 o i=4
GOTO 40
30 PRINT*, ' i=3, i=', i! aqui se va si i=3
40 CONTINUE

END
```

Alternativa: Puede sustituirse por un bloque `SELECT CASE`.

4.8.2. Sentencias obsoletas en Fortran 90 y en Fortran 95

4.8.2.1. `IF` aritmético

Sintaxis: `IF (expres) et1, et2, et3`

Acción: La expresión `expres` debe ser escalar entera ó real y las etiquetas `et1`, `et2`, `et3` deben estar en la unidad de alcance que contiene a la sentencia `IF`.
Si el valor de `expres` es negativo, cero ó positivo se continúa en la sentencia con etiqueta `et1`, `et2` ó `et3`, respectivamente.
Las etiquetas `etj` pueden no estar ordenadas, puede haber dos `etj` iguales y pueden no ser posteriores a la sentencia `IF`.

Nota:

- Si la siguiente sentencia a un `IF` aritmético no lleva etiqueta, no podría ejecutarse.

Ejemplo 4-18*. `IF` *aritmético **EJ4-18**

```
PRINT*, ' Introducir m, n'
READ*, m, n

IF (m+2**n) 20,30,10
10 PRINT*, ' m+2**n > 0 : ', m+2**n     ! aqui se va si m+2**n > 0
GOTO 40
20 PRINT*, ' m+2**n < 0 : ', m+2**n     ! aqui se va si m+2**n < 0
GOTO 40
30 PRINT*, ' m+2**n = 0 : ', m+2**n     ! aquí se va si m+2**n = 0
40 CONTINUE

END
```

Alternativa: Puede sustituirse por sentencias `IF` y bloques `IF`.

4.8.2.2. Final compartido de bloques `DO`

Un bloque `DO` con etiqueta puede terminar en una sentencia distinta a `ENDDO` y `CONTINUE` con esa etiqueta. Esta sentencia debe ser ejecutable y no puede ser `GOTO`, `GOTO` asignado, `EXIT`, `CYCLE`, `IF` aritmético, `STOP`, `END` ni `RETURN`.

Los bloques `DO` anidados pueden compartir la misma sentencia etiquetada final. En este caso, sólo se puede realizar un salto a esta etiqueta desde el `DO` más interior. La sentencia `ENDDO` sólo sirve para terminar un bloque `DO`.

Ejemplo 4-19*. *Producto matricial **EJ4-19**

```
REAL a(5,7), b(7,4), c(5,4)

! Inicializacion de las matrices a, b

DO 10 i = 1, 5
DO 10 k = 1, 7
10 a(i,k) = 2.3*i + 4.1*k

DO 20 k = 1, 7
DO 20 j = 1, 4
20 b(k,j) = 1.7*k - 3.2*j

! Calculo de la matriz producto

DO 30 i = 1, 5
DO 30 j = 1, 4
c(i,j) = 0
DO 30 k = 1, 7
30 c(i,j) = c(i,j) + a(i,k)*b(k,j)

PRINT*, c
END
```

Alternativa: Compartir la sentencia final de bloques `DO` no ofrece ventajas, sino que es una fuente de errores potenciales.

4.8.3. Sentencias obsoletas en Fortran 90 y eliminadas en Fortran 95

4.8.3.1. Indices no enteros en bloques `DO`

La variable `var` del `DO` y las expresiones `expres1`, `expres2`, `expres3` pueden ser de tipo real o doble precisión. El número de iteraciones a realizar será `MAX{INT(expres2-expres1+expres3)/expres3, 0}`.

Debido a posibles errores de redondeo, si el cálculo intermedio tiene parte fraccionaria muy próxima a 1 podrían no realizarse todas las iteraciones teóricas. Asímismo, al ir incrementando la variable `var` en la cantidad `expres3` se puede perder precisión y variar el número de iteraciones a realizar.

Ejemplo 4-20. *Indices no enteros en bloques* `DO` EJ4-20

```
DO x = 0.4, 3, 0.2
  PRINT*, ' x = ', x
  IF (x == 1.8) STOP ' x = 1.8'
ENDDO
PRINT*, ' No ha ocurrido x = 1.8'
END
```

Ejemplo 4-21. *Bucle infinito debido al redondeo* EJ4-21

```
x = 0.0
DO
  x = x + 0.1
  PRINT*, ' x = ', x
  IF (x == 1.2) STOP ' x = 1.2'

! Sin la siguiente sentencia este bucle no terminaria nunca

  IF (x >= 4.0) STOP ' No ha ocurrido x = 1.2'
ENDDO
END
```

Alternativa: Utilizar variable y expresiones enteras equivalentes.

4.8.3.2. `GOTO` asignado

Es una forma de realizar saltos que consta de dos partes: sentencias `ASSIGN` y una sentencia `GOTO` asignado. El esquema es:

Sintaxis:
```
ASSIGN et1 TO ivar
   :
ASSIGN etn TO ivar
   :
GOTO ivar [[,](et1,...,etn)]
```

Acción: `et1`,...,`etn` son etiquetas de sentencias en la misma unidad de alcance e `ivar` es una variable escalar entera.
Cuando se ejecuta una sentencia `ASSIGN` su etiqueta asigna a la variable `ivar`, la cual se vuelve indefinida como variable entera y sólo puede usarse como etiqueta (a no ser que se redefina con un valor entero).
Cuando se ejecuta la sentencia `GOTO` asignado se realiza el salto a la sentencia cuya etiqueta se asignó más recientemente a `ivar`.
Si está presente la lista opcional de etiquetas, debe contener la etiqueta asignada a `ivar` y sirve para chequear esta condición. Pueden repetirse etiquetas en dicha lista.

Ejemplo 4-22. `ASSIGN` *y* `GOTO` *asignado*

EJ4-22

Este ejemplo implementa el siguiente algoritmo:
***Paso* 0**.- Leer un entero positivo *n*.
***Paso* 1**.- Escribir $n,\ n^2,\ n^3$.
Si *n* tiene 4 ó más cifras parar
Si *n* tiene 1 cifra poner $n = n+1$ y repetir el Paso 1
Si *n* tiene 2 cifras poner $n = 2\times n$ y repetir el Paso 1
Si *n* tiene 3 cifras poner $n = 3\times n$ y repetir el Paso 1

```
PRINT*, ' Introducir n>0'
READ*, n
IF (n .LE. 0) STOP ' volver a ejecutar el programa'

50 IF (1<=n .AND. n<=9) THEN
  ASSIGN 1 TO ncifras
ELSEIF (10<=n .AND. n<=99) THEN
  ASSIGN 2 TO ncifras
ELSEIF (100<=n .AND. n<=999) THEN
  ASSIGN 3 TO ncifras
ELSE
  ASSIGN 4 TO ncifras
ENDIF

GOTO 100

1 n = n +1
GOTO 50
2 n = 2*n
GOTO 50
3 n = 3*n
GOTO 50

100 PRINT*, ' n = ', n, '    n*n = ', n*n, '    n*n*n = ', n*n*n
GOTO ncifras (1,2,3,4)

4 PRINT*, ' n ya tiene mas de 3 cifras, n = ', n
END
```

Uso: Se usa cuando en una unidad de programa, desde distintos puntos se converge a una misma sentencia a partir de la cual se ejecuta una parte de código común y luego se continúa en otra sentencia que depende del punto de llegada.

Alternativa: Puede sustituirse por otras estructuras de control, o bien utilizar subprogramas internos.

Ejemplo 4-23. *Mejor codificación del ejemplo* ***EJ4-22*** EJ4-23

```
PRINT*, ' Introducir n>0'
READ*, n
IF (n <= 0) STOP ' volver a ejecutar el programa'

DO
  PRINT*, ' n = ', n, '    n*n = ', n*n, '    n*n*n = ', n*n*n

  IF (n >= 1000) THEN
    PRINT*, ' n tiene mas de 3 cifras, n = ', n
    STOP
  ENDIF

  IF (1<=n .AND. n<=9) THEN
    n = n + 1
  ELSEIF (10<=n .AND. n<=99) THEN
    n = 2*n
  ELSEIF (100<=n .AND. n<=999) THEN
    n = 3*n
  ENDIF

ENDDO

END
```

4.8.3.3. Salto a la sentencia `ENDIF`

Se puede realizar un salto a la sentencia `ENDIF` de un bloque `IF` desde fuera del bloque. El efecto es equivalente a realizar un salto a la siguiente sentencia a `ENDIF`.

Ejemplo 4-24. *Salto a* `ENDIF` EJ4-24

```
PRINT*, ' Introducir n'
READ*, n

IF (n > 10) GOTO 100
IF (n < 5) THEN
  STOP ' n<5'
ELSE
  STOP ' 5<=n<=10'
  100 ENDIF    ! Deberia ponerse la etiqueta 100 en
PRINT*, ' n>10'! la sentencia PRINT

END
```

Alternativa: Esta posibilidad no ofrece ventaja alguna. Puede sustituirse realizando el salto a la siguiente sentencia ejecutable a `ENDIF`.

4.9. PROGRAMA EJEMPLO

EJ4-25

Este programa lee desde el teclado una cantidad de números enteros (no más de 100), los ordena de menor a mayor, escribe por pantalla los números en lugar par (después de ordenados) y cuenta y escribe la cantidad de positivos, negativos y ceros.

```
PROGRAM ordenar
IMPLICIT NONE
INTEGER num(100), cpos, cneg, ceros, temp, n, i, j

PRINT*, 'cantidad de numeros (<=100)'
READ*, n
IF (n > 100) STOP ' n debe ser <=100'
DO i = 1, n
  PRINT*, 'numero ', i
  READ*, num(i)
ENDDO

!  ordenacion de menor a mayor

DO i = 1, n-1
  DO j = i+1, n
    IF (num(i) > num(j)) THEN
      temp = num(i)
      num(i) = num(j)
      num(j) = temp
    ENDIF
  ENDDO
ENDDO

!  escritura en la pantalla de los numeros de lugar par

DO i = 2, n, 2
  PRINT*, num(i)
ENDDO

!  conteo de positivos, negativos y ceros

cpos = 0
cneg = 0
ceros = 0
DO i = 1, n                    ! Otra forma equivalente es:
  IF (num(i) > 0) THEN         ! IF (num(i) > 0)  cpos=cpos+1
   cpos=cpos+1                 ! IF (num(i) == 0) ceros=ceros+1
  ELSEIF (num(i) == 0) THEN    ! IF (num(i) < 0)  cneg=cneg+1
    ceros = ceros + 1
  ELSE
    cneg = cneg + 1
  ENDIF
ENDDO

PRINT*, ' cpos=', cpos
PRINT*, ' ceros=', ceros
PRINT*, ' cneg=', cneg

ENDPROGRAM ordenar
```

4.10. EJERCICIOS

1. Sea $a_i = \dfrac{1}{i^2\left(i+2^{i/100}\right)}$ y $s_n = \sum_{i=1}^{n} a_i$. Escribir un programa que calcule y escriba en pantalla los valores $s_{10},\ s_{20},\ s_{30},\cdots,s_{200}$.

2. Escribir un programa que lea desde el teclado n números reales y que escriba en pantalla el mayor y el menor de ellos.

3. Escribir un programa que calcule aproximadamente el número π utilizando las fórmulas siguientes:

(a) $\dfrac{\pi}{4} = 1 - \dfrac{1}{3} + \dfrac{1}{5} - \dfrac{1}{7} + \cdots$

(b) $\dfrac{\pi}{8} = \dfrac{1}{1*3} + \dfrac{1}{5*7} + \dfrac{1}{9*11} + \cdots$

(c) $\dfrac{\pi}{4} = 6\,\text{arctg}\dfrac{1}{8} + 2\,\text{arctg}\dfrac{1}{57} + \text{arctg}\dfrac{1}{239}$ $\left(\text{arctg}\,x = x - \dfrac{x^3}{3} + \dfrac{x^5}{5} - \dfrac{x^7}{7} + \cdots\right)$

4. Ejecutar el siguiente programa. Habrá que pararlo con Ctrl+Break.

```
x = 0.1
DO
  x = x + 0.001
  PRINT *, x
  IF (x .EQ. 0.3) STOP ' fin del bucle DO'
ENDDO
END
```

5. *Interpolación lineal*. Dados los puntos (x_i, y_i) $i = 1,\cdots,n$, con $x_1 \le x_2 \le \cdots \le x_n$, la aproximación de x mediante interpolación lineal es y, calculada como sigue: si $x_i \le x \le x_{i+1}$ entonces $y = y_i + \dfrac{y_{i+1} - y_i}{x_{i+1} - x_i}\left(x - x_i\right)$.

Escribir un programa que lea los puntos (x_i, y_i) $i = 1,\cdots,n$, los ordene de forma que $x_1 \le x_2 \le \cdots \le x_n$, calcule $a = \min\left\{x_i\right\}$, $b = \max\left\{x_i\right\}$, divida el intervalo $[a,b]$ en 100 partes iguales y para cada punto de la división escriba en pantalla dicho punto y su aproximación por interpolación lineal.

6. *Polinomio de interpolación de Lagrange*. Dados los puntos (x_i, y_i) $i = 1,\cdots,n$ con todos los x_i distintos, el polinomio $y = p(x)$ de grado no mayor que $n-1$ que pasa por ellos puede calcularse como: $p(x) = \sum_{i=1}^{n} L_i(x) y_i$, donde $L_i(x) = \prod_{\substack{j=1 \\ j \ne i}}^{n} \dfrac{\left(x - x_j\right)}{\left(x_i - x_j\right)}$.

Escribir un programa análogo al del ejercicio **5**.

CAPÍTULO 5. *UNIDADES DE PROGRAMA. PROCEDIMIENTOS*

5.1. PROGRAMA PRINCIPAL

Un programa completo debe tener exactamente un programa principal. La forma es:

Sintaxis:
```
[PROGRAM nombprog]
  [sentencias de especificación]
  [sentencias ejecutables]
[CONTAINS
   subprogramas internos]
END [PROGRAM [nombprog]]
```

5.2. SUBPROGRAMAS EXTERNOS

Son llamados desde el programa principal o desde otros subprogramas. Pueden ser funciones ó subrutinas. Ambas pueden ser recursivas, esto es, llamarse a sí mismas. No obstante, en este Curso Básico no incluimos la recursividad por dos razones: su complejidad de uso y su, generalmente, peor eficacia computacional. A continuación describimos el uso elemental no recursivo de subprogramas externos.

5.2.1. Uso No Recursivo de Subprogramas `FUNCTION`

Sintaxis:
```
[tipo] FUNCTION nombfun ([argumentos ficticios])
         [sentencias de especificación]
         [sentencias ejecutables]
       END [FUNCTION [nombfun]]
```

Acción: Define el subprograma `FUNCTION nombfun`.
La función se invoca con: `nombfun ([argumentos actuales])`.
Se sustituyen los argumentos actuales en los ficticios y se evalúa la función.
El valor asignado a `nombfun` es el valor devuelto por la función.

Normas:

- `[tipo]` es opcional; si se omite, se toma el tipo por defecto o el que haya sido establecido por sentencias `IMPLICIT`.
- La llamada puede formar parte de una expresión ó sentencia más larga.
- Un subprograma `FUNCTION` puede contener cualquier sentencia excepto `PROGRAM`, `FUNCTION`, `SUBROUTINE` y `BLOCK DATA`. La última sentencia tiene que ser `END`.
- Las variables y etiquetas en un subprograma `FUNCTION` son locales, esto es, independientes de las del programa principal y de las de otros subprogramas.

- Los argumentos actuales deben coincidir en cantidad, orden, tipo y longitud con los argumentos ficticios. Puede no haber argumentos.
- Los argumentos actuales pueden modificarse; sin embargo, esta opción es especialmente desaconsejable.
- Una función no recursiva no puede llamarse a sí misma ni directa ni indirectamente, pero sí puede llamar a otros subprogramas.

Ejemplo 5-1. *Función no recursiva* **EJ5-1**

```
PRINT*, ' Introducir x  a  b'
READ*, x, a, b
PRINT*, ' x=', x, ' a=', a, ' b=', b, ' fun(x,a,b)=', fun(x,a,b)
END

FUNCTION fun(x,a,b)
IF ( x >= a+b ) THEN
  fun = x**a / (x+b)
ELSE
  fun = (x+a) / (x*b - a)
ENDIF
END
```

5.2.1.1. Sentencia `RETURN` en Subprogramas `FUNCTION`

- `RETURN` termina la ejecución de la función y devuelve el control a la unidad de programa que llamó a la función. Si la función no tiene sentencias `RETURN` su ejecución termina al llegar a la sentencia `END`.
- Puede ser una sentencia con etiqueta. Puede formar parte de una sentencia `IF`.

Ejemplo 5-2. *Uso de* `RETURN` *en una función* **EJ5-2**

```
REAL otrafun, x, a, b
INTEGER indice
PRINT*, ' Introducir x,  a,  b,  indice'
READ*, x, a, b, indice
PRINT*, ' x=', x, ' a=', a, ' b=', b, ' indice=', indice
otx = otrafun(x,a,b,indice)
PRINT*, ' otrafun(x,a,b,indice)=', otx
END

FUNCTION otrafun(x,a,b,indice)
REAL otrafun, x, a, b
INTEGER indice
IF (indice < 0) THEN    ! equivalente a:
  otrafun = 1.e30
  RETURN   !    otrafun = 1.e30
ENDIF!    IF (indice.LT.0) RETURN
IF (x >= a+b) THEN
  otrafun = x**a / (x+b)
ELSE
  otrafun = (x+a) / (x*b - a)
ENDIF
RETURN     ! es innecesario
END
```

5.2.2. Uso No Recursivo de Subprogramas `SUBROUTINE`

Sintaxis:
```
SUBROUTINE nombsubr [(argumentos ficticios)]
  [sentencias de especificación]
  [sentencias ejecutables]
END [SUBROUTINE [nombsubr]]
```

Acción: Define el subprograma `SUBROUTINE nombsubr`.
La llamada a la subrutina se realiza mediante la sentencia
`CALL nombsubr [(argumentos actuales)]`
Se sustituyen los argumentos actuales en los ficticios.

Normas:

- Un subprograma `SUBROUTINE` puede contener cualquier sentencia excepto `PROGRAM`, `FUNCTION`, `SUBROUTINE` y `BLOCK DATA`. La última sentencia tiene que ser `END`.
- Las variables y etiquetas en un subprograma `SUBROUTINE` son locales, independientes de las del programa principal y de las de otros subprogramas.
- Los argumentos actuales deben coincidir en cantidad, orden, tipo y longitud con los argumentos formales. Puede no haber argumentos.
- Una subrutina no recursiva no puede llamarse a sí misma ni directa ni indirectamente, pero sí puede llamar a otros subprogramas.

Ejemplo 5-3. *Subrutina para resolver ecuaciones de segundo grado* **EJ5-3**

```
REAL a, b, c, x1, x2
INTEGER indice

PRINT*, ' Introducir la ecuacion de segundo grado: a, b, c'
READ*, a, b, c

CALL segundo (a, b, c, x1, x2, indice)

SELECT CASE (indice)
  CASE (0)
  PRINT*, ' a=b=0. No existe ecuacion'
  CASE (1)
  PRINT*, ' a=0. Ecuación de primer grado. Solucion x= ', x1
  CASE (2)
  PRINT*, ' Raiz real doble x = ', x1
  CASE (3)
  PRINT*, ' Raices complejas conjugadas:', x1, ' +/- ', x2, ' i'
  CASE (4)
  PRINT*, ' Raices reales simples: x1=', x1, ' x2=', x2
ENDSELECT

END

! Subprograma SUBROUTINE

SUBROUTINE segundo (a, b, c, x1, x2, indice)
REAL a, b, c, x1, x2
INTEGER indice
REAL disc
```

```
IF (a == 0) THEN
  IF (b == 0) THEN
    indice = 0                ! no existe ecuacion
  ELSE
    indice = 1                ! ecuacion de primer grado
    x1 = -c/b
    x2 = x1
  ENDIF
ELSE
  disc = b**2 - 4*a*c
  IF (disc == 0) THEN
    indice = 2                ! raiz real doble
    x1 = -b / (2.*a)
    x2 = x1
  ELSE IF (disc < 0) THEN
    indice = 3                !  raices complejas conjugadas
    x1 = -b / (2.*a)          !  x1 contiene la parte real
    x2 = SQRT(-disc)/(2.*a)   !  x2 contiene la parte imaginaria
  ELSE
    indice = 4                !  raices reales simples
    x1 = (-b+SQRT(disc) )/(2.*a)
    x2 = (-b-SQRT(disc) )/(2.*a)
  ENDIF
ENDIF
END
```

5.2.2.1. Sentencia `RETURN` en Subprogramas `SUBROUTINE`

- `RETURN` termina la ejecución de la subrutina y devuelve el control a la unidad de programa que llamó a la función. Si la subrutina no tiene sentencias `RETURN` su ejecución termina al llegar a la sentencia `END`.
- Puede ser una sentencia con etiqueta. Puede formar parte de una sentencia `IF`.

Ejemplo 5-4. *Uso de* `RETURN` *en una subrutina* **EJ5-4**

```
REAL a, b, c, x1, x2
INTEGER indice

PRINT*, ' Introducir la ecuacion de segundo grado: a, b, c'
READ*, a, b, c

CALL segundo (a, b, c, x1, x2, indice)

SELECT CASE (indice)
  CASE (0)
  PRINT*, ' a=b=0. No existe ecuacion'
  CASE (1)
  PRINT*, ' a=0. Ecuacion de primer grado. Solucion x= ', x1
  CASE (2)
  PRINT*, ' Raiz real doble x = ', x1
  CASE (3)
  PRINT*, ' Raices complejas conjugadas:', x1, ' +/- ', x2, ' i'
  CASE (4)
  PRINT*, ' Raices reales simples: x1=', x1, ' x2=', x2
ENDSELECT

END
```

```
! Subprograma SUBROUTINE

SUBROUTINE segundo (a, b, c, x1, x2, indice)
REAL a, b, c, x1, x2
INTEGER indice
REAL disc

indice = 0                      ! no existe ecuacion
IF (a==0 .AND. b==0) RETURN
IF (a == 0) THEN
  indice = 1                    ! ecuacion de primer grado
  x1 = -c/b
  x2 = x1
  RETURN
ENDIF

disc = b**2 - 4*a*c
IF (disc == 0) THEN
  indice = 2                    ! raiz real doble
  x1 = -b / (2.*a)
  x2 = x1
  RETURN
ENDIF

IF (disc < 0) THEN
  indice = 3                    !  raices complejas conjugadas
  x1 = -b / (2.*a)              !  x1 contiene la parte real
  x2 = SQRT(-disc)/(2.*a)       !  x2 contiene la parte imaginaria
ELSE
  indice = 4                    !  raices reales simples
  x1 = (-b+SQRT(disc) )/(2.*a)
  x2 = (-b-SQRT(disc) )/(2.*a)
ENDIF

END
```

5.2.3. Argumentos de Subprogramas Externos

Los argumentos de un subprograma `FUNCTION` ó `SUBROUTINE` pueden ser de naturaleza muy diversa: constantes ó variables escalares, arrays o elementos de arrays, nombres de otros subprogramas, etc. Es necesario suministrar al compilador la información adecuada para identificar correctamente la naturaleza del argumento.

Uno de los errores más frecuentes en programas Fortran es el uso inadecuado de argumentos en llamadas a los subprogramas, especialmente si los argumentos son arrays de varias dimensiones.

5.2.3.1. Dimensiones de Argumentos Arrays

- El orden de los elementos de un array es tal que aumenta primero su primer índice, luego el segundo y así sucesivamente.

  ***Ejemplos*:**
  El orden de los elementos del array x(10,10) es:
  x(1,1),x(2,1),...,x(10,1),x(1,2),x(2,2),...,x(10,2),...,x(1,10),x(2,10),...,x(10,10)

El orden de los elementos del array a(5,6,7) es:
a(1,1,1),a(2,1,1),…,a(5,1,1),a(1,2,1),a(2,2,1),...,a(5,2,1),...,...,
a(1,6,1),a(2,6,1),...,a(5,6,1),a(1,1,2),a(2,1,2),...,a(5,1,2),...,...,
a(1,6,2),a(2,6,2),...,a(5,6,2),...,...,a(1,6,7),a(2,6,7),...a(5,6,7)

- Si un argumento de un subprograma (`FUNCTION` ó `SUBROUTINE`) es un array con d dimensiones, las d-1 primeras dimensiones del argumento ficticio en el subprograma deben ser exactamente iguales a las del argumento actual en la unidad de llamada y la d-ésima dimensión menor ó igual (suele ponerse `1` ó `*`).
- Las dimensiones de los argumentos ficticios pueden ser constantes ó variables enteras siempre que éstas también sean argumentos formales.

Ejemplo 5-5. *Producto escalar de dos vectores* **EJ5-5**

```
INTEGER, PARAMETER:: n=100
REAL, DIMENSION(n):: x, y
DO i = 1, n
  x(i) = 1.76*i
  y(i) = x(i)**2./3 - i
ENDDO
xty = escalar(x,y,n)
PRINT*, ' xty =', xty
END

REAL FUNCTION escalar (x,y,n)
REAL x(*), y(*)       ! tamaño asumido
escalar = 0.0
DO i = 1, n
  escalar = escalar + x(i)*y(i)
ENDDO
END FUNCTION escalar
```

Ejemplo 5-6. *Dimensiones de argumentos array en subprogramas* **EJ5-6**

```
REAL x(100)
INTEGER, PARAMETER:: imat=10, iterna=5, jterna=6
INTEGER mat(imat,20), terna(iterna,jterna,7)

DO i=1,100; x(i) = 1./i; ENDDO
DO i=1,8; DO j=1,15; mat(i,j)=i+j; ENDDO; ENDDO
DO i=1,3; DO j=1,2; DO k=1,4; terna(i,j,k)=i-j+2*k; ENDDO; ENDDO; ENDDO
fval = f(x, mat, terna)
PRINT*, ' f(x,mat,terna) = ', fval
CALL subrut (mat, imat, 8, 15, terna, iterna, jterna, 3, 2, 4, salida)
PRINT*, ' salida =', salida
END

FUNCTION f(a, m2, m3)
REAL a(*)
INTEGER m2(10,*), m3(5,6,*)
  f = a(30) + m2(4,5) + m3(1,2,3)
END

SUBROUTINE subrut (mat, im, n, m, terna, it, jt, ti, tj, tk, salida)
INTEGER im, n, m, it, jt, ti, tj, tk, mat(im,1), terna(it,jt,*)
  salida = mat(n,m) + terna(ti,tj,tk)
END
```

Ejemplo 5-7. *Producto de matrices* EJ5-7

```
INTEGER, PARAMETER:: mda=10, mdb=20, mdc=50
REAL a(mda,15), b(mdb,20), c(mdc,25)

! Inicializacion de las matrices a, b

DO i = 1, 2
  DO j = 1, 3
    a(i,j) = i+j-1
  ENDDO
ENDDO
DO i = 1, 3
  DO j = 1, 2
    b(i,j) = i+2*j+1
  ENDDO
ENDDO

CALL mult (a, b, c, mda, mdb, mdc, 2, 3, 2)
PRINT*, ' c = ', c(1,1), c(1,2), c(2,1), c(2,2)
END

! Subrutina que calcula el producto de dos matrices

SUBROUTINE mult (a, b, c, mda, mdb, mdc, n1, n2, n3)
REAL a(mda,*), b(mdb,*), c(mdc,*)

DO i = 1, n1
  DO j = 1, n3
    c(i,j) = 0.0
    DO k = 1, n2
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    ENDDO
  ENDDO
ENDDO
END
```

5.2.3.2. Propósito de los Argumentos (Nuevo en Fortran 90)

Los argumentos ficticios pueden tener una declaración de propósito de entrada, salida o entrada/salida. El propósito se declara con el atributo `INTENT`.

`INTENT(IN)` : declara un argumento de entrada. No debe cambiarse su valor dentro del subprograma.

`INTENT(OUT)` : declara un argumento de salida. El argumento actual debe ser una variable y se vuelve indefinida en entrada.

`INTENT(INOUT)` : declara un argumento de entrada ó salida. El argumento actual debe ser una variable.

Si un argumento ficticio no tiene declaración de propósito, el argumento actual puede ser una variable o una expresión, a no ser que el argumento ficticio se redefina en el subprograma, en cuyo caso el argumento actual debe ser una variable.

Es recomendable declarar el propósito de los argumentos ficticios, lo cual ayuda a la documentación del programa y a las verificaciones durante la compilación.

Ejemplo 5-8. *Atributo* `INTENT` **EJ5-8**

En los ejemplos ***EJ5-3*** y ***EJ5-4***, la declaración de los argumentos podría ser:

```
REAL, INTENT(IN):: a, b, c        ! argumento ficticio de entrada
REAL, INTENT(OUT):: x1, x2        ! argumento ficticio de salida
INTEGER, INTENT(OUT):: indice     ! argumento ficticio de salida
REAL disc                         ! variable local
```

Ejemplo 5-9. *Atributo* `INTENT` **EJ5-9**

En el ejemplo ***EJ5-7***, la declaración de los argumentos podría ser:

```
INTEGER, INTENT(IN):: mda, mdb, mdc, n1, n2, n3
REAL, INTENT(IN):: a(mda,*), b(mdb,*)
REAL, INTENT(OUT):: c(mdc,*)
INTEGER i, j, k             ! variables locales
```

5.2.3.3. Sentencia EXTERNAL

Sintaxis: `EXTERNAL lista`

Acción: Identifica los nombres de `lista` como subprogramas (funciones ó subrutinas) externos definidos por el usuario.

Normas:

- Es una sentencia de especificación y debe preceder a las ejecutables y a las declaraciones de funciones.
- Cuando un argumento de un subprograma es el nombre de otro subprograma, se debe declarar `EXTERNAL` en la unidad de llamada.
- Si una función intrínseca se declara `EXTERNAL` pierde su definición intrínseca en la unidad de programa asociada y se usa el subprograma del usuario.

Ejemplo 5-10. *Uso de la sentencia* `EXTERNAL` **EJ5-10**

```
EXTERNAL fun1, fun2, sin
x=1.5; n=3
CALL ameba (x, n, y1, fun1)       ! y1 = x**(5-n) = 2.25
CALL ameba (x, n, y2, fun2)       ! y2 = 3*x*(5-n) = 9
s = sin (y1, y2, 6.0, x)          ! s = y2/y1 + 6/x = 8
PRINT*, y1, y2, s
END

SUBROUTINE ameba (x, n, y, f)
  y = f(x, 5, n)
END

FUNCTION fun1 (x, i, j)
  fun1 = x**(i-j)
END

FUNCTION fun2 (x, i, j)
  fun2 = 3*x*(i-j)
END
```

```
FUNCTION sin (a, b, c, d)
  sin = b/a + c/d
END
```

5.2.3.4. Sentencia `INTRINSIC`

Sintaxis: `INTRINSIC lista`
Acción: Declara los nombres de `lista` como funciones intrínsecas.
Normas:

- Los nombres de `lista` deben ser funciones intrínsecas.
- Si un argumento de un subprograma es una función intrínseca se debe declarar `INTRINSIC` en la unidad de llamada.
- Si un nombre está en una sentencia `INTRINSIC` no puede estar en una sentencia `EXTERNAL`.

Ejemplo 5-11. Uso de la sentencia `INTRINSIC` EJ5-11

```
INTRINSIC sin, cos, exp
a=3.141592; b=-a
r = fun (sin, cos, exp, a, b, 4)
PRINT*, r
END

FUNCTION fun (f1, f2, f3, a, b, n)      ! fun = ((sin(a)+cos(b)+
fun = (f1(a) + f2(b) + f3(a+b)) ** n    ! exp(a+b))**n
END
```

5.2.3.5. Argumentos Opcionales. Palabras Clave. Bloque `INTERFACE`

Si la lista de argumentos ficticios es larga y algunos de ellos no se necesitan o no se conocen en ciertos casos, pueden declararse opcionales con el atributo `OPTIONAL` y ser llamados mediante palabras clave. (Nuevo en Fortran 90)

Normas:

- Las palabras clave son los nombres de los argumentos ficticios.
- En la llamada al subprograma puede haber una lista inicial (que puede ser vacía) de argumentos posicionales ordinarios sin palabra clave seguida de una lista de argumentos con palabras clave.
- No puede haber argumentos posicionales después del primer argumento con palabra clave.
- La función intrínseca `PRESENT(arg)` sirve para detectar dentro de un subprograma si un argumento está o no está presente en la sentencia `CALL`. Devuelve el valor `.true.` si el argumento `arg` está presente en la sentencia `CALL` de llamada, y `.false.` si no lo está.
- Para que el compilador pueda establecer las asociaciones entre los argumentos si se usa algún argumento opcional o con palabra clave, la interfaz debe ser explícita en la unidad de programa de llamada. Esto puede hacerse con un bloque `INTERFACE`. La sintaxis es:

Sintaxis: `INTERFACE`
`bloq_interface`
`END INTERFACE`
donde el bloque `bloq_interface` normalmente es una copia exacta de la cabecera del subprograma (especificaciones de los argumentos).

Ejemplo 5-12. *Atributo* `OPTIONAL` EJ5-12

Una subrutina para minimizar funciones de n variables con posibles restricciones de cotas en las variables podría tener los siguientes argumentos ficticios:

```
SUBROUTINE optim (n, fun, x, param, cotainf, cotasup, xinic)
INTEGER, INTENT(IN):: n
REAL, INTENT(OUT), DIMENSION(n):: x
REAL, INTENT(IN), OPTIONAL, DIMENSION(10):: param
REAL, INTENT(IN), OPTIONAL, DIMENSION(n):: cotainf, cotasup
REAL, INTENT(IN), DIMENSION(n):: xinic
```

La llamada a la subrutina podría ser:

```
CALL optim (n, fun, x, cotainf=a, xinic=x0)
```

5.3. SUBPROGRAMAS INTERNOS (Nuevo en Fortran 90)

Son subprogramas contenidos en el programa principal, en un subprograma externo ó en un módulo. Su uso es adecuado, a efectos de organización, para subprogramas cortos (del orden de unas 20 líneas) que sólo se necesitan en un único programa, subprograma ó módulo.

Sintaxis: `CONTAINS`
`subprogramas internos`
Acción: Define subprogramas internos.
Normas:

- Los subprogramas internos deben aparecer entre la sentencia `CONTAINS` y la sentencia `END` de la unidad de programa (***host***) a la que pertenezcan.
- Un subprograma interno no puede contener otro subprograma interno.
- Un subprograma interno sólo puede llamarse desde su host.
- Un host conoce todo acerca del interface con sus subprogramas internos, por tanto no hace falta declarar el tipo en el host para una función interna.
- Un subprograma interno tiene acceso a las variables del host.
- El host no tiene acceso a las variables locales de los subprogramas internos.
- La sentencia `IMPLICIT NONE` en un host afecta al host y también a sus subprogramas internos.
- Las etiquetas son locales: Si una sentencia tiene etiqueta, ésta debe estar en la misma unidad de alcance que la sentencia que la referencia. Esto implica que no puede haber un `GOTO` de un subprograma interno a un host.

Ejemplo 5-13. *Uso de subprogramas internos* EJ5-13

```
PROGRAM interno

  CALL coefbinomial ! invoca una subrutina interna

CONTAINS

  SUBROUTINE coefbinomial
    INTEGER n, k
    CALL leer(n)    ! invoca una subrutina interna
    DO k = 0, n
      PRINT*, k, nsobrek(n,k)     ! invoca una funcion interna
    ENDDO
  ENDSUBROUTINE coefbinomial

  SUBROUTINE leer(n)
    INTEGER n
    PRINT*, ' Introducir el valor de n'
    READ*, n
  ENDSUBROUTINE leer

  FUNCTION nsobrek(n,k)
    INTEGER nsobrek, n, k
    nsobrek = fact(n) / (fact(k)*fact(n-k)) ! invoca una funcion
  ENDFUNCTION nsobrek                       ! interna 3 veces

  FUNCTION fact(m)
    REAL fact
    INTEGER m, i
    fact = 1
    DO i = 2, m
      fact = i*fact
    ENDDO
  ENDFUNCTION fact

ENDPROGRAM interno
```

5.4. MÓDULOS (Nuevo en Fortran 90)

Un módulo permite empaquetar definiciones de datos y compartir datos entre diferentes unidades de programas que pueden incluso compilarse por separado. Sirve, especialmente, para crear grandes librerías de software. En su uso sencillo:

- ofrece posibilidades similares a `INCLUDE` (apartado **5.7.1**)
- permite compartir datos en ejecución (≈ `COMMON`, apartado **5.7.3**)
- sirve para inicializar variables (≈ `BLOCK DATA`, apartado **5.7.4**)

Sintaxis:
```
MODULE nombmod
    [sentencias de especificación]
  [CONTAINS
    subprogramas internos del módulo]
  END [MODULE [nombmod]]
```
Acción: Define el módulo `nombmod`.

El acceso al módulo se realiza con la sentencia `USE nombmod`.

Normas:

- Se puede acceder a un módulo desde el programa principal, un subprograma externo u otro módulo. Se accede a las especificaciones y variables del módulo con los valores asignados si los tienen. Las variables y datos de un módulo tienen, por defecto, un alcance **global** en todas las unidades desde las que se accede con `USE`.
- Desde un módulo se tiene acceso a las otras entidades del módulo, incluyendo subprogramas.
- Puede contener sentencias `USE` para acceder a otros módulos.
- No debe acceder a sí mismo directamente o indirectamente a través de `USE`.
- El módulo debe compilarse antes que el programa que lo usa. En la compilación se crea un fichero `*.mod` que es el que lee la sentencia `USE`. Se recomienda que un módulo sólo acceda a módulos anteriores a él.

Ejemplo 5-14. *Módulo para establecer datos globales* **EJ5-14**

```
MODULE globales
  INTEGER nvar, nfile, nyear, ncity
  PARAMETER (nvar=20, ncity=50)
  REAL x(nvar,5)
  CHARACTER(LEN=20) ciudad(ncity)
ENDMODULE globales

PROGRAM ejemplo
USE globales
  PRINT*, ' nvar=', nvar
  CALL subrut
END

SUBROUTINE subrut
USE globales
  PRINT*, ' ncity=', ncity
END
```

Ejemplo 5-15. *Uso de un módulo para compartir datos* **EJ5-15**

```
MODULE datos
  INTEGER numval
  INTEGER valores(100)
ENDMODULE datos

PROGRAM compartir
  USE datos
  IMPLICIT NONE
  INTEGER i

  PRINT*, ' numero de valores = '
  READ*, numval
  numval = MIN(numval, 100)
  DO i = 1, numval
    valores(i)=i**i
  ENDDO
  CALL escribe
ENDPROGRAM compartir
```

```
SUBROUTINE escribe
  USE datos
  IMPLICIT NONE
  INTEGER i
  DO i = 1, numval
    PRINT*, valores(i)    ! caracter global
  ENDDO
ENDSUBROUTINE escribe
```

5.5. SENTENCIA DATA

Sintaxis: `DATA nlista/vlista/[[,] nlista/vlista/]...`

Acción: asigna a los items de `nlista` los valores iniciales de su `vlista` asociada.

Normas:

- Cada `nlista` es una lista de variables, arrays, elementos de arrays, substrings, lista con `DO` implícito.
- Si un item de `nlista` es el nombre de un array, equivale a todos sus elementos dispuestos según el orden de los elementos de un array.
- Cada `vlista` es una lista de constantes; si la constante `c` se repite `n` veces consecutivas puede indicarse como `n*c`.
- Debe haber el mismo número de elementos en `nlista` que en su `vlista` asociada.
- Para items no caracteres se realiza conversión de tipo, si es necesario.
- En `nlista` no pueden estar argumentos ficticios, variables de bloques `COMMON` en blanco ni nombres de funciones.
- Las variables de un bloque `COMMON` con nombre pueden estar en sentencias `DATA` dentro del subprograma `BLOCK DATA`.
- Aunque la sentencia `DATA` puede ponerse entre las ejecutables, esto no aporta ventaja alguna e induce a confusión ya que es una sentencia de especificación. Esta posibilidad ha sido declarada obsoleta en Fortran 95.

Ejemplo 5-16. *Uso de la sentencia* `DATA` **EJ5-16**

```
INTEGER n, i, k(5), numeros(5,5)
REAL x(10)
CHARACTER(LEN=10) comunid(17)

DATA n,i/1,2/, k(2)/12345/, x/1.2,0.0,8*-3.5/
DATA ((numeros(i,j),i=1,j),j=1,5) /15*88/! DO implicito
DATA comunid/'ANDALUCIA','ARAGON',15*' '/

PRINT*, ' n=', n, ' i=', i, ' k(2)=', k
PRINT* ; PRINT*, ' x=', x
PRINT* ; PRINT*, ' numeros=', numeros
PRINT* ; PRINT*, ' dos comunidades :', comunid(1), comunid(2)
END
```

5.6. ORDEN DE LAS SENTENCIAS

Las diferentes sentencias que puede contener un programa Fortran deben escribirse con el orden que se muestra en la tabla siguiente:

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE ó MODULE</td></tr>
<tr><td colspan="3">USE</td></tr>
<tr><td rowspan="4">FORMAT</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER</td><td>IMPLICIT</td></tr>
<tr><td>PARAMETER, DATA</td><td>Tipos derivados
Bloques INTERFACE
Declaración de tipos
Sentencias de especificación</td></tr>
<tr><td colspan="2">Sentencias ejecutables</td></tr>
<tr><td colspan="3">CONTAINS</td></tr>
<tr><td colspan="3">Subprogramas internos ó Subprogramas Módulo</td></tr>
<tr><td colspan="3">END</td></tr>
</table>

5.7. SENTENCIAS REDUNDANTES EN UNIDADES DE PROGRAMA

5.7.1. Líneas `INCLUDE` (Extensión FORTRAN 77)

Sintaxis: `INCLUDE 'fichero'`

Acción: Inserta el contenido del fichero especificado en el lugar de la sentencia `INCLUDE` en el momento de la compilación. Normalmente se permiten hasta 10 niveles de anidamiento, sin recursividad.

Ejemplo 5-17. *Uso de* `INCLUDE` EJ5-17

Contenido del fichero `'bloques.517'`:

```
INTEGER mv, mlcod, mldat, mncd, mnrc
PARAMETER (mv=500, mlcod=8, mldat=20, mncd=7500, mnrc=50)
INTEGER nvar, ncasos, lr, nrc, pvr(mnrc), ancvar(mv)
CHARACTER tipvar(mv)
CHARACTER(LEN=10) nomvar(mv)
COMMON /cabec/ nvar, ncasos, lr, nrc, pvr, ancvar
COMMON /varc/ tipvar, nomvar
```

Programa FORTRAN:

```
PROGRAM frecuencias
INCLUDE 'bloques.517'
PRINT*, ' mv =', mv, ' mlcod =', mlcod, ' mldat =',mldat
CALL variab
CALL analis
END
```

```
SUBROUTINE variab
INCLUDE 'bloques.517'
PRINT*, ' mncd =', mncd, ' mnrc =', mnrc
END

SUBROUTINE analis
INCLUDE 'bloques.517'
PRINT*, ' mv =', mv, ' mlcod =', mlcod, ' mldat =',mldat
END
```

Nota: El uso de `INCLUDE` se considera redundante. Puede sustituirse por módulos.

5.7.2. Declaración de Funciones en Sentencias

Se pueden definir funciones en una sentencia similar a una asignación.

Sintaxis: `nomfun ([ argumentos ficticios ]) = expres`
Acción: define la función `nomfun` en una única sentencia.
Normas:

- Estas sentencias no son ejecutables y deben ir después de las sentencias de especificación y antes de las sentencias ejecutables.
- El nombre `nomfun` y los argumentos ficticios deben ser de naturaleza escalar. El tipo de la función se establece en una sentencia de especificación previa ó toma el tipo intrínseco por defecto. Puede no haber argumentos.
- La función `nomfun` se ejecuta cuando es llamada. Se sustituyen los valores de los argumentos actuales en los ficticios y se evalúa `expres.` Si es necesario (y posible) se realizan conversiones de tipo. En la expresión `expres` puede haber, además de argumentos ficticios, variables accesibles en la unidad de alcance que contiene a esta función.
- En la expresión `expres` se pueden referenciar constantes, variables, elementos de arrays y funciones previamente declaradas. Si hay referencias a funciones, éstas no deben redefinir los argumentos ficticios.
- Estas funciones no pueden ser argumentos de otros procedimientos.
- Una función no puede llamarse a sí misma, ni directa ni indirectamente.

Ejemplo 5-18. *Declaración y uso de funciones* **EJ5-18**

```
! Aproximacion de la funcion exp(x)

expon(x) = 1 + x + 0.5*x**2 + 1.0/6*x**3 + 1.0/24*x**4 + &
           1.0/120*x**5 + 1.0/720*x**6

DO i = 1, 20
  t = i*0.1
  et = expon(t)
  PRINT*, ' t =', t, '  e^t=', et, '  exp(t)=', EXP(t)
ENDDO
END
```

Nota: El empleo de la declaración de funciones en sentencias se considera obsoleto en Fortran 95. Puede sustituirse por subprogramas internos.

5.7.3. Sentencia `COMMON`

Esta sentencia permite que varias unidades de programa compartan variables sin tener que pasarlas como argumentos. Hay dos tipos de bloques `COMMON`: sin nombre (también llamado `COMMON` en blanco) y con nombre.

Sintaxis: `COMMON [/[nombcom]/] lista`
`nombcom`: nombre del bloque `COMMON`
`lista`: lista de variables, nombres de arrays y declaraciones de arrays (puede incluir especificación de dimensiones).

Acción: Define un área de memoria que será accesible en cada unidad de programa que contenga la sentencia `COMMON`.

Normas:

- El nombre `nombcom` es global. Debe ser distinto de los otros nombres globales (programa, subprogramas externos y otros bloques `COMMON`).
- En el bloque `COMMON` no puede haber constantes `PARAMETER`.
- Los argumentos ficticios y las funciones no pueden estar en `lista`.
- Los bloques `COMMON` son "acumulativos". Ninguna variable puede aparecer en más de un bloque `COMMON`.
- Se recomienda que en `lista` no se mezclen variables de tipo `CHARACTER` con variables de tipo no `CHARACTER`.
- Un bloque `COMMON` con nombre debe tener la misma longitud en todas las unidades de programa en las que aparezca. El bloque `COMMON` sin nombre puede tener longitudes diferentes en diferentes unidades de programa.
- Un item de `lista` no se puede inicializar al definir su tipo.
- Los nombres de `lista` pueden ser diferentes en cada aparición de la sentencia `COMMON` en distintas unidades de programa.

Ejemplo 5-19. *Los bloques* `COMMON` *son acumulativos* **EJ5-19**

```
COMMON a,b  ! estas cinco sentencias son
COMMON //r,i,s,m  ! equivalentes a las dos sentencias:
COMMON /uno/x(5),w
COMMON j(4) ! COMMON a,b,r,i,s,m,j(4)
COMMON /uno/y(3,5),z(5),t     ! COMMON /uno/x(5),w,y(3,5),z(5),t
a=1; b=2; r=3; i=4
x=88; w=10; y=99
CALL compleccion
PRINT*, ' COMMON en blanco : '
PRINT*, ' a, b, r, i, s, m = ', a, b, r, i, s, m
PRINT*, ' j = ', j
PRINT*, ' COMMON /uno/ : '
PRINT*, ' x = ', x
PRINT*, ' w = ', w
PRINT*, ' y = ', y
PRINT*, ' z = ', z
PRINT*, ' t = ', t
END
```

```
SUBROUTINE compleccion
COMMON a,b,r,i,s,m,j(4)
COMMON /uno/xxx(5),www,yyy(3,5),zzz(5),ttt

s=5; m=6; j=77
zzz=-1; ttt=3.14
END
```

Ejemplo 5-20. *Uso de sentencias* `COMMON` EJ5-20

```
PROGRAM principal
COMMON a, b, c(100), d(10,5)
COMMON /ctes/ nvar, ncasos, nfilas
a=1; b=2
nvar=5; ncasos=100
CALL calculo
CALL eval(9.0, 8.0, 7.0)
PRINT*, ' nvar, ncasos, nfilas = ', nvar, ncasos, nfilas
END

SUBROUTINE calculo
COMMON x, y, z(100), u(10,5)
COMMON /ctes/ n, m, l
l = 2000
END

SUBROUTINE eval (x,y,z)
COMMON r, s
PRINT*, ' a, b = ', r, s
PRINT*, x, y, z
END
```

Nota: El uso de `COMMON` se considera obsoleto. Puede sustituirse por módulos.

5.7.4. Subprogramas `BLOCK DATA`

Sintaxis: `BLOCK DATA [nomblock]`
`[sentencias de especificación y de inicialización]`
`END [BLOCK DATA [nomblock]]`

Acción: Inicializa variables y elementos de arrays de bloques `COMMON` con nombre.

Normas:

- `nomblock` es opcional. Sólo puede haber un subprograma `BLOCK DATA` sin nombre.
- Las sentencias que puede contener `BLOCK DATA` son: `COMMON`, `DATA`, `PARAMETER`, sentencias de especificación, `IMPLICIT`, `DIMENSION`, `END`. No puede contener sentencias ejecutables.

Ejemplo 5-21. *Subprogramas* `BLOCK DATA` EJ5-21

```
INTEGER longitud(6)
CHARACTER(LEN=12) nombres(6)
COMMON /uno/ x, y, z
COMMON /nrios/ nombres
COMMON /lrios/ longitud
```

```
  PRINT*, x, y, z
  PRINT*, nombres
  PRINT*, longitud
END

BLOCK DATA
COMMON /uno/ a, b, c
  DATA a,b,c /1.0,2.0,3.0/
ENDBLOCK DATA

BLOCK DATA datrios
INTEGER long(6)
CHARACTER(LEN=12) nomb(6)
COMMON /nrios/ nomb
COMMON /lrios/ long
  DATA nomb /'MIÑO','DUERO','TAJO','GUADIANA','GUADALQUIVIR','EBRO'/
  DATA long(3) /1008/
ENDBLOCK DATA datrios
```

Nota: El uso de subprogramas `BLOCK DATA` se considera obsoleto. Pueden sustituirse por módulos.

5.7.5. `RETURN` Alternativo en Subprogramas `SUBROUTINE`

- Un argumento actual puede ser `*e` (`RETURN` alternativo a la sentencia con etiqueta `e` del programa de llamada). El correspondiente argumento formal debe ser `*` y debe haber al menos un `RETURN n` por cada `RETURN` alternativo.

Ejemplo 5-22. *Uso de* `RETURN` *alternativo* EJ5-22

```
10 PRINT*, ' Introducir pt'  ! aqui se va si RETURN 1
   READ*, pt
   CALL vuelta (pt, *10, result, *40, *20)
   pt = pt/2! aqui se va si RETURN o RETURN n (n>3)
   GOTO 40
20 IF ( result .LE. 5) THEN   ! aqui se va si RETURN 3
     pt = pt*2
     GOTO 40
   ENDIF
40 CONTINUE ! aqui se va si RETURN 2
   PRINT*, ' result = ', result
END

SUBROUTINE vuelta (pt, *, result, *, *)
IF (pt < 0) RETURN 1
IF (0<=pt .AND. pt<=5) THEN
  result = 2.3
  RETURN 2
ENDIF
IF (7<=pt .AND. pt<=100) THEN
  result = pt**(1.0/3) -3*pt + 24.5
  RETURN 3
ENDIF
result = pt
RETURN
END
```

Nota: El uso de `RETURN` alternativo se considera obsoleto y debe evitarse. Puede utilizarse en su lugar `GOTO` calculado o `SELECT CASE`.

5.7.6. Longitud Asumida de Argumentos de Tipo Carácter

La longitud de un argumento ficticio de tipo CHARACTER puede declararse con un asterisco (longitud asumida) en cuyo caso toma como longitud la del argumento actual.

Ejemplo 5-23. *Longitud asumida de argumentos de tipo* `CHARACTER` **EJ5-23**

```
CHARACTER(LEN=9), DIMENSION(3,4):: var1
CHARACTER(LEN=10), DIMENSION(3,4):: var2
CHARACTER(LEN=*), PARAMETER:: letras = 'abcdefghijkl'
CALL asumid (3, 4, 10, var1, var2)
PRINT*, letras
PRINT*, var1
PRINT*, var2
END

SUBROUTINE asumid(n1, n2, m, car1, car2)
INTEGER n1, n2, m
CHARACTER(LEN=*), DIMENSION(n1,n2):: car1  ! longitud asumida
CHARACTER(LEN=m), DIMENSION(n1,n2):: car2  ! longitud fija
car1 = 'fiesta'
car2 = 'de navidad'
END
```

5.8. EJERCICIOS

1. Modificar el programa ejemplo ***EJ3-16*** del capítulo 3, creando un menú en el que se establezca el cuerpo (cilindro, cono, esfera) y tres subprogramas FUNCTION que calculen la superficie y volumen asociados.

2. Escribir un subprograma FUNCTION para la función definida en el ejemplo ***EJ4-6*** del capítulo 4. Idem para el ejemplo ***EJ4-7.***

3. Escribir una subrutina que realice cálculos de conversión de unidades: grados Celsius←→grados Fahrenheit, centímetros←→pulgadas, km←→millas, etc. Definir un menú en el que se seleccione la conversión a realizar.

4. Escribir un programa con un subprograma interno FUNCTION cuyo argumento de entrada sea un número entero y devuelva dicho número con las cifras en orden inverso.

5. Modificar el programa del ejemplo ***EJ1-9*** del capítulo 1 para resolver ecuaciones mediante el método de la bisección añadiendo una variable `iter` tal que si toma valor 1 (por teclado) se imprima en cada etapa el punto medio del intervalo [*a*, *b*] y el valor de la función *f* en dicho punto.

6. Elaborar una subrutina para implementar un algoritmo para minimizar funciones de una variable basado en la idea siguiente:

> En la iteración *k*-ésima se tiene el intervalo de búsqueda $[a_k, b_k]$, se divide en tres partes iguales y comparando el valor de *f* en los puntos interiores de esa división se pasa a la iteración siguiente descartando el primer o el último subintervalo

Definir la función $f(x)$ en un subprograma FUNCTION.

Utilizar el programa elaborado para minimizar las funciones siguientes:

6.1. $f(x) = \sqrt{|x-1|}$ en $[0,10]$

6.2. $f(x) = \sum_{i=1}^{20} \left(\frac{2i-5}{x-i^2} \right)^2$ para $x \notin \{1^2, 2^2, ..., 20^2\}$

7. Escribir un programa con una función lógica interna cuyo argumento de entrada sea un número entero y devuelva un valor lógico `.TRUE.` si dicho número entero es primo y `.FALSE.` si es compuesto. Utilizar la siguiente idea para comprobar si el número *n* es primo:

> Se divide *n* entre 2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31,..., (se saltan los múltiplos de 3) hasta llegar a $\sqrt{n}$. Si alguna de esas divisiones es exacta parar; *n* es compuesto, en otro caso *n* es primo.

CAPÍTULO 6. *PROCEDIMIENTOS INTRÍNSECOS*

Los procedimientos intrínsecos son funciones y subrutinas que forman parte del lenguaje Fortran estándar, suministradas con el compilador. En Fortran 95 hay 109 funciones y 6 subrutinas que pueden clasificarse en cuatro categorías de procedimientos intrínsecos:

(1) ***Procedimientos elementales***: sus argumentos son escalares o arrays. Si una función elemental se aplica a un array la función se aplica a cada elemento del array.
(2) ***Funciones interrogación***: devuelven propiedades de sus argumentos que no dependen de sus valores.
(3) ***Funciones transformacionales***: suelen tener argumentos arrays y resultados arrays cuyos elementos dependen de muchos elementos del argumento.
(4) ***Subrutinas no elementales.***

6.1. FUNCIONES NUMÉRICAS

Cada función devuelve un valor (variable o array) entero, real, complejo, lógico ó carácter y por tanto las funciones también tienen tipo. Algunas devuelven un valor del mismo tipo que el argumento. En lo sucesivo utilizaremos las abreviaturas de tipo siguientes:

I	entero
R	real
C	complejo
N	numérico (I ó R ó C)
L	lógico
CH	carácter

6.1.1. Funciones Elementales que Pueden Convertir Tipos

Nombre	*Definición*	*Tipo argumentos*	*Tipo función*
`ABS(x)`	Valor absoluto	I	I
`ABS(x)`	Valor absoluto	R	R
`ABS(z)`	Módulo complejo	C	R
`AIMAG(z)`	Parte imaginaria	C	R
`AINT(x)`	Quita decimales	R	R
`ANINT(x)`	Redondeo	R	R
`CEILING(x)`	Función techo	R	I
`CMPLX(x[,y])`	Pasa a complejo	N	C
`FLOOR(x)`	Función suelo	R	I
`INT(x)`	Pasa a entero	N	I
`NINT(x)`	Redondeo entero	R	I
`REAL(x)`	Pasa a real	N	R

Ejemplo 6-1. *Funciones que pueden realizar conversión de tipo* EJ6-1

Expresión	*Resultado*	*Expresión*	*Resultado*
`ABS(x+3*y-4)`	`|x+3y-4|`	`ABS((3.0,-4.0))`	`5`
`AIMAG((2.4,-5.8))`	`-5.8`	`AIMAG(CMPLX(1+x,3-y))`	`3-y`
`AINT(23.67)`	`23`	`AINT(-23.67)`	`-23`
`ANINT(7.2)`	`7`	`ANINT(-7.2)`	`-7`
`ANINT(7.5)`	`8`	`ANINT(-7.5)`	`-8`
`ANINT(7.9)`	`8`	`ANINT(-7.9)`	`-8`
`CEILING(7.2)`	`8`	`CEILING(-7.2)`	`-7`
`CMPLX(2)`	`2+0i`	`CMPLX(2.5)`	`2.5+0i`
`FLOOR(7.2)`	`7`	`FLOOR(-7.2)`	`-8`
`INT(7.2)`	`7`	`INT(-7.2)`	`-7`
`INT((2.3,7.9))`	`2`	`INT((-2.3,7.9))`	`-2`
`NINT(7.2)`	`7`	`NINT(7.5)`	`8`
`NINT(7.9)`	`8`	`NINT(2.3,4.5)`	`ERROR`
`REAL(2)`	`2.0`	`REAL((3.6,4.8))`	`3.6`

6.1.2. Funciones Elementales que no Convierten Tipos

El resultado de las siguientes funciones elementales es del tipo de su primer argumento. Si hay varios argumentos todos han de ser del mismo tipo.

Nombre	*Definición*	*Tipo argumentos*	*Tipo función*
`CONJG(z)`	Conjugado complejo	C	C
`DIM(x,y)`	Diferencia positiva	(I,I) ó (R,R)	I ó R
`MAX(x1,x2[,x3,...])`	Máximo	(I,I,…) ó (R,R,…)	I ó R
`MIN(x1,x2[,x3,...])`	Mínimo	(I,I,…) ó (R,R,…)	I ó R
`MOD(x,y)`	Resto de x módulo y	(I,I) ó (R,R)	I ó R
`MODULO(x,y)`	x módulo y	(I,I) ó (R,R)	I ó R
`SIGN(x,y)`	Transferencia de signo	(I,I) ó (R,R)	I ó R

Se definen:

`DIM(x,y) = máx(x-y,0)`; `SIGN(x,y) = signo(y) |x|` (si `y=0` se toma `signo(y)=1`)

`MOD(x,y) = x-INT(x/y)*y`; `MODULO(x,y) = x-FLOOR(x/y)*y`. Debe ser `y≠0`.

Ejemplo 6-2. *Funciones que no realizan conversión de tipo* EJ6-2

Expresión	*Resultado*	*Expresión*	*Resultado*
`CONJG((2.3,4.5))`	`2.3-4.5i`	`CONJG((1,3))`	`1-3i`
`DIM(2,8)`	`0`	`DIM(8,2)`	`6`
`MAX(2.4,1.8,-7.3)`	`2.4`	`MIN(2,6,-9,20)`	`-9`
`MOD(19,7)`	`5`	`MOD(-19,7)`	`-5`
`MOD(19,-7)`	`5`	`MOD(-19,-7)`	`-5`
`MOD(23.8,3.7)`	`1.599998`	`MOD(-23.8,3.7)`	`-1.599998`
`MOD(23.8,-3.7)`	`1.599998`	`MOD(-23.8,-3.7)`	`-1.599998`
`MODULO(19,7)`	`5`	`MODULO(-19,7)`	`2`
`MODULO(19,-7)`	`-2`	`MODULO(-19,-7)`	`-5`
`MODULO(23.8,3.7)`	`1.599999`	`MODULO(-23.8,3.7)`	`2.100001`
`MODULO(23.8,-3.7)`	`-2.100001`	`MODULO(-23.8,-3.7)`	`-1.599999`
`SIGN(7.3,2.8)`	`7.3`	`SIGN(7.3,-2.8)`	`-7.3`
`SIGN(-7.3,2.8)`	`7.3`	`SIGN(-7.3,-2.8)`	`-7.3`

6.1.3. Funciones Matemáticas Elementales

El resultado de las siguientes funciones elementales es del tipo de su primer argumento (frecuentemente el único).

Nombre	***Definición***	***Tipo argumentos***	***Tipo función***
ACOS(x)	Arco Coseno	R / |x|≤1	R en [0,π]
ASIN(x)	Arco Seno	R / |x|≤1	R en [-π/2,π/2]
ATAN(x)	Arco Tangente	R	R en [-π/2,π/2]
ATAN2(y,x)	Argumento número complejo	(R,R)	R en (-π,π]
COS(x)	Coseno	R ó C	R ó C
COSH(x)	Coseno hiperbólico	R	R
EXP(x)	Exponencial	R ó C	R ó C
LOG(x)	Logaritmo neperiano	R ó C	R ó C
LOG10(x)	Logaritmo decimal	R, x>0	R
SIN(x)	Seno	R ó C	R ó C
SINH(x)	Seno hiperbólico	R	R
SQRT(x)	Raíz cuadrada	R ó C	R ó C
TAN(x)	Tangente	R	R
TANH(x)	Tangente hiperbólica	R	R

Ejemplo 6-3. *Funciones matemáticas elementales* **EJ6-3**

Expresión	***Resultado***	***Expresión***	***Resultado***
ACOS(0.5)	1.047198	**ACOS**(-0.5)	2.094395
ASIN(0.5)	0.523599	**ASIN**(-0.5)	-0.523599
ATAN(1.0)	0.785398	**ATAN**(-1.0)	-0.785398
ATAN2(1.8,0.3)	1.405648	**ATAN2**(-1.8,0.3)	-1.405648
ATAN2(1.8,-0.3)	1.735945	**ATAN2**(-1.8,-0.3)	-1.735945
COS(0.7853891)	0.707113	**COS**(-1.0)	0.540302
COSH(3.5)	16.572824	**COSH**(-3.5)	16.572824
EXP(3.5)	33.115452	**EXP**(-3.5)	3.0197384e-02
LOG(33.11545)	3.500000	**LOG**(0.03019738)	-3.500000
LOG10(100.0)	2.000000	**LOG10**(2.7182818)	0.434294
SIN(0.7853891)	0.707100	**SIN**(-1)	-0.841471
SINH(3.5)	16.542627	**SINH**(-3.5)	-16.542627
SQRT(2.0)	1.414214	**SQRT**((-2.0,1))	(0.343561,1.455347)
TAN(0.7853891)	0.999982	**TAN**(-1)	-1.557408
TANH(3.5)	0.998178	**TANH**(-3.5)	-0.998178

6.1.4. Multiplicación Vectorial y Matricial

Nombre	***Definición***	***Tipo argumentos***	***Tipo función***
DOT_PRODUCT(x,y)	Producto escalar real	(I ó R, I ó R)	I ó R
DOT_PRODUCT(z,y)	Producto escalar complejo	(C,I ó R)	C
MATMUL(a,b)	Producto matricial	(N,N)	N
TRANSPOSE(a)	Matriz traspuesta	N	N

`DOT_PRODUCT(x,y)` requiere que `x` e `y` tengan una dimensión y el mismo tamaño. Si `x` es entero o real devuelve $\sum x_i y_i$; si `x` es complejo $\sum \text{conjg}(x_i) y_i$.

La función `MATMUL(a,b)` devuelve un array de dos dimensiones, tal que:

Operación	***Forma de*** a	***Forma de*** b	***Forma de*** `MATMUL(a,b)`
Matriz x Matriz	(n,m)	(m,k)	(n,k)
Vector x Matriz	(m)	(m,k)	(k)
Matriz x Vector	(n,m)	(m)	(n)

Ejemplo 6-4. *Multiplicación vectorial y matricial* **EJ6-4**

```
REAL x(5,7), y(7,20), s(5,20), a(20), b(20), c(7), d(5), st(20,5)
COMPLEX z(20), zta

DO i=1,20; a(i)=1.23*i; b(i)=2*i-1; z(i)=CMPLX(a(i),b(i)); ENDDO
DO i=1,5; DO j=1,7; x(i,j)=i+j-2; ENDDO; ENDDO
DO i=1,7; DO j=1,20; y(i,j)=i*j+1; ENDDO; ENDDO
DO i=1,7; c(i)=i**3; ENDDO
DO i=1,5; DO j=1,20; s(i,j)=i-j**2; ENDDO; ENDDO

atb = DOT_PRODUCT(a,b)
zta = DOT_PRODUCT(z,a)
st = TRANSPOSE(MATMUL(x,y))
b = MATMUL(c,y)
d = MATMUL(s,b)

PRINT*, ' atb=', atb, ' zta=', zta
END
```

6.1.5. Operaciones Globales en Arrays Numéricos

Nombre	***Definición***	***Tipo argumentos***	***Tipo función***
`MAXVAL(x)`	Máximo elemento	I ó R	I ó R
`MINVAL(x)`	Mínimo elemento	I ó R	I ó R
`PRODUCT(x)`	Producto de los elementos	I ó R	I ó R
`SUM(a)`	Suma de los elementos	I ó R	I ó R

Ejemplo 6-5. *Operaciones globales en arrays numéricos* **EJ6-5**

```
REAL x(20,30,40), y(15), maxymin, prodysum

DO i=1,20; DO j=1,30; DO k=1,40; x(i,j,k)=i+j-k**2; ENDDO; ENDDO; ENDDO
DO i=1,15; y(i)= 1.3*i-1.9; ENDDO

maxymin = MAXVAL(x) + MINVAL(y)
PRINT*, ' maxymin=', maxymin
prodysum = PRODUCT(y) + SUM(x)
PRINT*, ' prodysum=', prodysum

END
```

6.2. FUNCIONES DE CARACTERES

6.2.1. Conversiones `CHARACTER<->INTEGER`

Nombre	*Definición*	*Tipo argumentos*	*Tipo función*
`ACHAR(i)`	Carácter ASCII de lugar i	I; 0 ≤ i ≤ 127	CH (LEN=1)
`CHAR(i)`	Carácter en la posición i	I; 0 ≤ i ≤ N-1	CH (LEN=1)
`IACHAR(c)`	Posición del carácter ASCII c	CH	I
`ICHAR(c)`	Posición del carácter c	CH	I

En la función `ACHAR(i)` si `i>127` el resultado depende del procesador.
En la función `CHAR(i)` N es el número de caracteres utilizados por el procesador.

Ejemplo 6-6. *Conversiones* `CHARACTER<->INTEGER` **EJ6-6**

Expresión	*Resultado*	*Expresión*	*Resultado*
`ACHAR(65)`	A	`CHAR(65)`	A
`IACHAR('A')`	65	`ICHAR('A')`	65

6.2.2. Comparaciones Léxicas

Nombre	*Definición*	*Tipo argumentos*	*Tipo función*
`LGE(cha,chb)`	`cha` después de o igual que `chb`	(CH,CH)	L
`LGT(cha,chb)`	`cha` después de `chb`	(CH,CH)	L
`LLE(cha,chb)`	`cha` antes de o igual que `chb`	(CH,CH)	L
`LLT(cha,chb)`	`cha` antes de `chb`	(CH,CH)	L

Ejemplo 6-7. *Comparaciones léxicas* **EJ6-7**

Expresión	*Resultado*	*Expresión*	*Resultado*
`LGE('cas','sol')`	`.FALSE.`	`LGT('3n1', '3/1')`	`.TRUE.`
`LLE('lunes','2h')`	`.FALSE.`	`LLT('AeIoU','123')`	`.TRUE.`

6.2.3. Manejo de Caracteres

Nombre	*Definición*	*Tipo arg.*	*Tipo fun.*
`ADJUSTL(st)`	Ajusta por la izquierda	CH	CH
`ADJUSTR(st)`	Ajusta por la derecha	CH	CH
`INDEX(s1,s2,[BACK])`	Posición de `s2` en `s1`	(CH,CH)	I
`LEN(st)`	Longitud	CH	I
`LEN_TRIM(st)`	Longitud sin blancos finales	CH	I
`REPEAT(st,ncopias)`	Concatena `ncopias` de `st`	(CH,I)	CH
`SCAN(s1,s2,[BACK])`	Indice del primer carácter de `s1` en `s2`	(CH,CH)	I
`TRIM(st)`	Quita los blancos finales	CH	CH
`VERIFY(s1,s2,[BACK])`	Indice del primer carácter de `s1` no en `s2`	(CH,CH)	I

La variable lógica `[BACK]` es opcional. Si está presente y tiene valor `.TRUE.` las comparaciones en las funciones `INDEX`, `SCAN` y `VERIFY` empiezan por el final del dato.

EJ6-8

Ejemplo 6-8. *Manejo de caracteres*

Expresión	***Resultado***	***Expresión***	***Resultado***
`ADJUSTL(' ABC ')`	`'ABC '`	`ADJUSTR(' ABC ')`	`' ABC'`
`INDEX('banana','an')`	2	`INDEX('banana','an',.TRUE.)`	4
`LEN('luna ')`	6	`LEN_TRIM('luna ')`	4
`REPEAT('Ab',3)`	`'AbAbAb'`	`TRIM('luna ')`	`'luna'`
`SCAN('banana','an')`	2	`SCAN('banana','an',.TRUE.)`	6
`VERIFY('banana','nbc')`	2	`VERIFY('banana','nbc',.TRUE.)`	6

6.3. SUBRUTINAS DE FECHA Y HORA. NÚMEROS ALEATORIOS

6.3.1. Reloj de Tiempo Real (Nuevo en Fortran 90)

Sintaxis: `CALL DATE_AND_TIME ([DATE][,TIME][,ZONE][,VALUES])`
Tipo de argumentos: todos de salida y opcionales

`DATE: CHARACTER`
`TIME: CHARACTER`
`ZONE: CHARACTER`
`VALUES:` array `INTEGER`

Acción: Devuelve los siguientes valores

`DATE`: fecha con el formato `ccaammdd` (`ccaa`: año, `mm`:mes, `dd`:día)
`TIME`: tiempo con el formato `hhmmss.sss` (`hh`: hora, `mm`: minuto, `ss`: segundo, `sss`: milésimas de segundo)
`ZONE`: diferencia entre el tiempo local y el tiempo universal coordenado (UTC, también llamado tiempo del meridiano de Greenwich) con el formato `shhmm` (`s`: signo, `hh`: hora, `mm`: minuto)
`VALUES(1)`: año
`VALUES(2)`: mes del año
`VALUES(3)`: día del mes
`VALUES(4)`: diferencia en minutos respecto al UTC
`VALUES(5)`: hora del día
`VALUES(6)`: minutos de la hora
`VALUES(7)`: segundos del minuto
`VALUES(8)`: milisegundos del segundo

Precaución:

La subrutina `DATE_AND_TIME` no es muy precisa para calcular el tiempo de CPU. Su uso con algunos compiladores no proporciona los milisegundos y en otros compiladores deja el reloj del sistema funcionando a menor velocidad.

Sintaxis: `CALL CPU_TIME (time)` (Nuevo en Fortran 95)
Tipo de argumento:
`time: REAL`
Acción: Devuelve el tiempo del procesador en segundos.
Sirve para calcular el tiempo de ejecución de una sección de un programa:

```
REAL t1, t2
....
CALL CPU_TIME (t1)
....
....     ! seccion cuyo tiempo se quiere conocer
....
CALL CPU_TIME (t2)
PRINT*, 'tiempo de ejecucion = ', t2-t1, ' sg. '
```

6.3.2. Números Aleatorios (Nuevo en Fortran 90)

Sintaxis: `CALL RANDOM_NUMBER ([HARVEST=]aleat)`
Tipo de argumento:
`aleat: REAL.` Variable escalar o array de salida
Acción: Devuelve números seudoaleatorios uniformes en [0,1) en `aleat`.
Sintaxis: `CALL RANDOM_SEED ([SIZE][,PUT][,GET])`
Tipo de argumentos: (todos opcionales)

- `SIZE:` variable escalar `INTEGER.` Variable de salida que contiene el tamaño `N` del array semilla.
- `PUT :` array `INTEGER` de dimensión `(N).` Variable de entrada utilizada para establecer la semilla.
- `GET :` array `INTEGER` de dimensión `(N).` Variable de salida que contiene el valor actual de la semilla.

Normas: No puede especificarse más de un argumento. Si no se especifican argumentos se establece una semilla que depende del procesador.

Ejemplo 6-9. *Números aleatorios y tiempo de cálculo* **EJ6-9**

```
PROGRAM aleatorio
INTEGER t1(8), t2(8)
INTEGER i, numrep, semilla(1)
REAL x, sx, tdif
CHARACTER(LEN=8) date1, date2
CHARACTER(LEN=10) time1, time2
CHARACTER(LEN=5) zona

PRINT*, ' numero de repeticiones'
READ*, numrep
PRINT*, ' semilla inicial'
READ*, semilla
CALL RANDOM_SEED (PUT=semilla)

CALL DATE_AND_TIME (VALUES=t1, DATE=date1, ZONE=zona, TIME=time1)
DO i = 1, numrep
  CALL RANDOM_NUMBER (x)
  sx = SIN(x)
ENDDO
```

```
CALL DATE_AND_TIME (VALUES=t2, TIME=time2, DATE=date2)

PRINT*, ' zona=', zona
PRINT*, ' date1=', date1, '  date2=', date2
PRINT*, ' time1=', time1, '  time2=', time2

tdif = 0.001*(t2(8)-t1(8)) + (t2(7)-t1(7)) + 60.*(t2(6)-t1(6)) +   &
      3600.*(t2(5)-t1(5))
PRINT*, ' tdif =', tdif

ENDPROGRAM aleatorio
```

Nota.- Para asegurar la transportabilidad entre diversos compiladores es preferible utilizar subrutinas propias de generación de números aleatorios (Press y otros, 1992).

6.4. EJERCICIOS

1. Escribir una tabla de valores de diversas funciones matemáticas: SIN, COS, TAN, LOG, EXP, SQRT con precisión sencilla y doble.

2. Calcular el valor de las siguientes expresiones para $n \geq 1$ entero:

(a) $x_n = 2^{n-1}\text{sen}\frac{\pi}{n}\cdot\text{sen}\frac{2\pi}{n}\cdots\text{sen}\frac{(n-1)\pi}{n}$

(b) $y_n = (2n-1)\cdot\cot^2\frac{\pi}{2n+1}\cdot\cot^2\frac{2\pi}{2n+1}\cdots\cot^2\frac{n\pi}{2n+1}$

3. Escribir un subprograma FUNCTION para evaluar y utilizar la siguiente función:

$$f(x_1,x_2,x_3)=\sum_{i=1}^{10}\left(e^{-\frac{i}{10}x_1}-e^{-\frac{i}{10}x_2}-x_3\left(e^{-\frac{i}{10}}-e^{-i}\right)\right)^2$$

4. Escribir un subprograma FUNCTION para calcular el máximo común divisor de dos números enteros basándose en el algoritmo de Euclides, cuya idea es la siguiente:

> Sean $\bar{a},\bar{b}$ tales que $\bar{a}\geq\bar{b}>0$. Poner $a=\bar{a}$, $b=\bar{b}$. Iterar como sigue: Calcular $c=\lfloor a/b\rfloor$, $r=a-bc$. Si $r=0$ parar: $\text{mcd}(\bar{a},\bar{b})=b$; si $r>0$ poner $a=b$, $b=r$ y repetir.

5. Elaborar un programa similar al del ejemplo ***EJ6-9*** para estimar el tiempo que tardan las diferentes operaciones matemáticas elementales y las funciones matemáticas intrínsecas (SIN, COS, TAN, SQRT, EXP, LOG, etc.).

6. Escribir subprogramas SUBROUTINE para realizar las operaciones de productos escalares y matriciales elemento a elemento y obtener el máximo, mínimo, producto y suma de los elementos de un array. Comparar los tiempos empleados por los procedimientos intrínsecos con los empleados por los subprogramas propios.

CAPÍTULO 7. *ENTRADA Y SALIDA DE DATOS. FICHEROS. FORMATOS*

7.1. ELEMENTOS Y CLASES DE FICHEROS

Los conceptos fundamentales a considerar son: campo, registro y fichero.

***Campo*:** unidad de información que consta de varios caracteres que se tratan en conjunto.
***Registro*:** conjunto de campos, no necesariamente del mismo tipo.
***Fichero*:** conjunto de registros, no necesariamente con igual estructura.

Como ejemplo, un fichero de personas podría contener un registro por cada persona y los campos podrían ser: nombre, DNI, dirección, edad y teléfono.

En algunos casos, por ejemplo en bases de datos, los registros de un fichero tienen la misma estructura, esto es, el mismo número y forma de los campos.

Los tipos de ***acceso*** a un fichero en Fortran son secuencial o directo.

***Secuencial*:** Para acceder a un registro hay que recorrer todo el fichero desde el principio hasta llegar a él.
***Directo*:** Conociendo el número de orden de un registro en el fichero se puede acceder a él sin tener que recorrer los registros anteriores.

Las datos pueden almacenarse en ***forma*** formateada o no formateada.

***Formateada*:** La información se guarda como caracteres ASCII, legibles con la mayoría de los procesadores de texto.
***No formateada*:** Un fichero es una serie de registros formados por "bloques físicos".

En este Curso Básico consideraremos sólo ficheros secuenciales formateados.

7.2. LECTURA Y ESCRITURA DE DATOS

Conversión de datos.- Internamente el ordenador representa los números y caracteres con cierta codificación. Para poder interpretar unos datos de entrada o mostrar unos datos de salida de forma legible se hacen conversiones entre la representación interna y la externa mediante especificaciones de formato.

Lista de entrada/salida (lista I/O).- Las entidades a leer o escribir se llaman listas de entrada/salida (lista I/O). En entrada se deben leer variables, en salida pueden escribirse expresiones. Si un array está en una lista I/O, se consideran todos los elementos del array en el orden de almacenamiento del array. Una lista puede contener un DO implícito de variables.

Formato.- Hay tres formas de indicar el formato de los datos a leer o escribir:
(***a***) Sentencia `FORMAT` con etiqueta
Sintaxis: `e FORMAT (codform)`
Acción: `codform` especifica los códigos de formato de lectura o escritura.
Normas: `e` es un número de etiqueta. Es una sentencia no ejecutable.
(***b***) una expresión carácter que contiene el formato entre paréntesis.
(***c***) un asterisco `*` que indica formato libre (lista directa de entrada-salida).

Números de unidad.- Cada fichero ***externo*** (terminal, impresora, fichero en disco ó en cinta,…) del que se lee o en el que se escribe lleva asociado un número de unidad no negativo, generalmente en el rango 1 a 99. Un número de unidad `u` asociado a un fichero externo puede ser:
(***a***) una expresión entera con valor admisible (generalmente $1 \le u \le 99$)
(***b***) un asterisco: entrada/salida estándar por defecto (generalmente teclado y pantalla).

Entrada/salida sin número de unidad.- Toda sentencia de lectura o escritura en un fichero externo debe referirse explícitamente a su número de unidad asociado. Hay dos excepciones:
(***a***) Sentencia `READ` sin número de unidad
Sintaxis: `READ fmt [,listavar]`
Acción: Lee datos del teclado (en modo interactivo); `fmt` indica el formato.
(***b***) Sentencia `PRINT`
Sintaxis: `PRINT fmt [,listavar]`
Acción: Escribe datos en la pantalla (en modo interactivo) con el formato `fmt`.

Ejemplo 7-1. *Lista I/O de variables* **EJ7-1**

```
INTEGER n, m, v(10)
REAL x(4,5)
CHARACTER(LEN=80) linea
CHARACTER(LEN=25) nombre(5)

PRINT*, ' Introducir n (<=5), m, v(1),...,v(10)'
READ*, n, m, v
PRINT*, 'v=', (v(i),i=1,10,4)      ! DO implicito de escritura

PRINT*, ' Introducir nombre(1),...,nombre(n)'
READ*, (nombre(i),i=1,n)           ! DO implicito de lectura

DO i=1,10; DO j=1,n; x(i,j)=i+j; ENDDO; ENDDO
PRINT*, x                          ! se escriben x(1,1),...,x(4,5)
PRINT*, ((x(i,j),j=1,4),i=1,3)     ! DO's implicitos anidados

linea = REPEAT('12345',16)
PRINT*, n, m, linea(3:20+2*n), v
PRINT*, ' n2=', n*n, ' rx', SQRT(x)
PRINT*, n, (i,i=1,20)
PRINT*, (v(3),i=1,15)

END
```

EJ7-2

Ejemplo 7-2. *Entrada/salida sin número de unidad. Formatos*

```
CHARACTER(LEN=20) form1

PRINT '(A)', ' Introducir ivar1, ivar2 con formato (I6,I3)'
READ '(I6,I3)', ivar1, ivar2

form1='(I5,I8)'
PRINT '(A,A)', ' Introducir n1, n2 con formato ', form1
READ form1, n1, n2

PRINT '(A)', ' Introducir m1, m2 con formato (I7,I4)'
READ 8000, m1, m2
 8000 FORMAT (I7,I4)

PRINT '(I9,I8)', ivar1, ivar2
PRINT 9000, n1, n2
PRINT form1, m1, m2
9000 FORMAT (I11,I6)

END
```

7.3. ACCESO A FICHEROS EXTERNOS

7.3.1. Sentencia `OPEN`

Sintaxis elemental: `OPEN ([UNIT=]u, FILE=nbf)`

Acción: Conecta la unidad `u` al fichero `nbf` ("abre" la unidad `u`).

Normas: La palabra clave `UNIT=` es opcional; `u` es una expresión entera. Por defecto, el fichero se considera secuencial formateado.
`nbf` es una expresión carácter que proporciona el nombre del fichero.

EJ7-3

Ejemplo 7-3. *Conexión de ficheros*

```
CHARACTER(LEN=30) nomb
nfile = 3; nf = 3; nomb = 'cilindro.sal'
OPEN (UNIT=3, FILE='cilindro.sal')  ! estas cuatro sentencias
OPEN (3, FILE='cilindro.sal')       ! son equivalentes
OPEN (nfile, FILE='cilindro.sal')   ! si nfile=3
OPEN (nf, FILE=nomb)                ! si nf=3, nomb='cilindro.sal'
WRITE (3,'(A)') ' El fichero 3 ha sido abierto'
END
```

EJ7-4

Ejemplo 7-4. *Conexión de ficheros*

```
CHARACTER(LEN=10) ciudad(20)
j = 7
i = 5
ciudad(j) = 'madrid'
OPEN (j, FILE=ciudad(j))
OPEN (10*i+j, FILE=ciudad(j)//'.sal')
WRITE (j,'(A,I2)') ' Se ha abierto el fichero ', j
WRITE (10*i+j,'(A,I2)') ' Se ha abierto el fichero ', 10*i+j
END
```

En raras ocasiones se leen y escriben datos en un mismo fichero. Normalmente, los datos de entrada se leen de ficheros que no se desean modificar y los resultados se escriben en ficheros nuevos o se añaden a ficheros ya existentes o sustituyen la información previa en ficheros existentes.

Además de `UNIT=u` y `FILE=nbf` la sentencia `OPEN` admite numerosos especificadores.

7.3.2. Sentencia `CLOSE`

Sintaxis elemental: `CLOSE ([UNIT=]u)`
Acción: Desconecta la unidad `u` ("cierra" la unidad `u`).
Normas: La palabra clave `UNIT=` es opcional; `u` puede ser una expresión entera. Por defecto, si el programa termina normalmente se cierran todos los ficheros. Esta sentencia es útil si hay que tener abiertos A LA VEZ bastantes ficheros, ya que el número máximo de ficheros abiertos simultáneamente puede ser muy limitado.

Ejemplo 7-5. *Números de unidad. Formatos. Sentencias* `OPEN, CLOSE` **EJ7-5**

```
CHARACTER(LEN=20) form1, form2
INTEGER x(100)

! Contenido del fichero EJ7-5a.dat:  3  2 10 15 20

OPEN (8, FILE='EJ7-5a.dat')
READ (8,'(5I3)') ivar1, ivar2, n1, n2, nsal

PRINT*, 'Introducir x(1), x(2) con formato (I3,I5)'
form1='(I3,I5)'
READ (*,form1) x(1), x(2)

! Contenido del fichero EJ7-5b.dat: 9 60 61 62 63 64 65

OPEN (ivar1, FILE='EJ7-5b.dat')   ! ivar1=3
READ (ivar1,*) nvariab, (x(i),i=n1,n2)
CLOSE (ivar1)

! Contenido del fichero EJ7-5c.dat: 2 4

OPEN (ivar1, FILE='EJ7-5c.dat')   ! ivar1=3
READ (3,8000) m1, m2
8000 FORMAT (2I2)
WRITE (*,*) m1, m2
CLOSE (2+ivar1)

OPEN (m1*m2+1, FILE='EJ7-5d.sal') ! m1*m2+1=9
WRITE (m1*m2+1,'(2I4)') n1, n2
WRITE (*,8000) n1, n2
WRITE (9,*) (x(n1+i),i=0,n2-n1)
form2='(2I5)'
WRITE (nvariab,form2) n1, n2      ! nvariab=9

END
```

7.3.3. Nombres y Números Reservados de Ficheros

`Extensión FORTRAN 77`

Algunos compiladores de FORTRAN 77 tienen reservados los siguientes nombres de fichero: `AUX` (dispositivo auxiliar), `CON` (consola o terminal), `NUL` (fichero nulo, esto es, el fichero no se crea), `PRN` (impresora).

Para evitar posibles conflictos no deben utilizarse estos nombres de fichero en Fortran 90/95, ni siquiera aunque lleven extensión.

Algunos números de unidad están asignados inicialmente: `0` (pantalla), `5` (teclado), `6` (pantalla). El asterisco `*` representa siempre el teclado (entrada) y la pantalla (salida) y no puede asignarse a otros ficheros. Las unidades `0`, `5`, `6` sí pueden asignarse a ficheros externos.

***Precaución*:**

El compilador SALFORD FTN95 tiene asignados los números de unidad `1, 2, 5, 6` al teclado/pantalla. Conviene evitar estos números si, además de los ficheros correspondientes, se usa el teclado/pantalla para lectura/escritura de datos.

Ejemplo 7-6. *Números reservados de ficheros* **EJ7-6**

```
WRITE (6,'(A)',ADVANCE='NO') ' Introducir x '
READ (5,*) x
WRITE (6,'(A)',ADVANCE='NO') ' Introducir y '
READ (*,*) y

WRITE (*,*) x, y
WRITE (1,*) x**2, y**2
WRITE (2,*) x**3, y**3
WRITE (5,*) x**4, y**4
WRITE (6,*) x**5, y**5
PRINT*, x**6, y**6

END
```

7.4. ENTRADA Y SALIDA FORMATEADA

La entrada y salida sin número de unidad se ha descrito en el apartado **7.2.** (`READ` sin número de unidad, `PRINT`).

7.4.1. `READ` con número de unidad

Sintaxis: `READ ([UNIT=]u, [FMT=]fmt [,IOSTAT=ios] [,ERR=e1] [,END=e2]) listavar`

Acción: lee datos del fichero asociado a la unidad `u` según el formato `fmt` y los asigna a los items de `listavar`.

Normas:

- `u` puede ser una expresión entera, un asterisco o el nombre de un fichero interno (apartado **7.7**).
- `fmt` puede ser un número de etiqueta de una sentencia `FORMAT`, una lista de códigos de formato entre apóstrofos (si un carácter de esta lista es un apóstrofo hay que duplicarlo) ó un asterisco.
- `IOSTAT=ios` (opcional); `ios` es una variable escalar entera que tomará un valor negativo si ocurre un fin de registro, otro valor negativo distinto si ocurre un fin de fichero y un valor positivo si ocurre alguna condición de error. Valdrá 0 si no hay error en la lectura.
- `ERR=e1` (opcional); si se produce un error de lectura se continúa en la sentencia con etiqueta `e1`, si no está `ERR=e1` se para la ejecución.
- `END=e2` (opcional); si se intenta leer después del fin del fichero se continúa en la sentencia con etiqueta `e2`, si no está `END=e2` se para la ejecución.
- `listavar` es una lista de variables separadas por comas.

7.4.2. `WRITE` con número de unidad

Sintaxis: `WRITE ([UNIT=]u, [FMT=]fmt [,IOSTAT=ios] [,ERR=e1]) listavar`

Acción: escribe los datos de `listavar` en el fichero asociado a la unidad `u` según el formato `fmt`.

Normas: El significado de los parámetros es el mismo que en la sentencia `READ`. Obviamente no admite el parámetro `[,END=e2]`.

7.4.3. No avance en entrada/salida. `ADVANCE=adv` (Nuevo en Fortran 90)

Una sentencia `READ` ó `WRITE` puede incluir `ADVANCE=adv,` donde `adv` es una expresión carácter con valor `'YES'` (por defecto) o `'NO'`.
Si `adv='YES'` el fichero se reposiciona después del último registro accedido. Si `adv='NO'` entonces no se avanza al registro siguiente antes de leer o escribir y el fichero puede quedar posicionado en el interior de un registro.

Ejemplo 7-7. *Introducción de datos sin avance de línea* **EJ7-7**

```
10 WRITE (*,'(A)',ADVANCE='NO') ' Numero de casos n = '
READ (*, '(I4)', ERR=10, END=10) n
20 WRITE (*,'(A)',ADVANCE='NO') ' Numero de variables m = '
READ (*, '(I4)', ERR=20, END=20) m
30 WRITE (*,9000)
9000 FORMAT (/T5,'Analisis a realizar:'      &
             /T10,'1: Regresion multilineal' &
             /T10,'2: Correspondencias'      &
             /T10,'3: Analisis de la varianza')
WRITE (*,'(T15,A)',ADVANCE='NO') 'opcion = '
READ (*, *, ERR=30, END=30) iopc
IF (iopc<1 .OR. iopc>3) GOTO 30
END
```

Ejemplo 7-8. *Sentencias* `READ, WRITE` **EJ7-8**

El fichero de nombre `'EJ7-8.dat'` contiene los campos: nombre, dni, edad, peso y altura, con anchuras fijas de 20, 10, 2, 5, 5 posiciones, respectivamente. Se desea leer todos los registros y chequear errores (datos inválidos, peso>130, altura>220). Se quiere generar un fichero de salida que contenga los registros válidos completos y sólo un mensaje indicativo con el número de orden del registro en los que contengan algún error.

```
PROGRAM basedatos
IMPLICIT NONE

CHARACTER(LEN=20) nombre
INTEGER dni, edad
INTEGER numreg, regmal, ios
REAL peso, altura

OPEN (11, FILE='EJ7-8.dat')
OPEN (12, FILE='EJ7-8.sal')

numreg = 0
10 CONTINUE
numreg = numreg + 1
regmal = 0 ! si regmal=1 el registro tendra errores

READ (11,8000,IOSTAT=ios,ERR=20,END=30) nombre, dni, edad, peso, altura
8000 FORMAT (A20,I10,I2,2F5.0)

IF (edad<=0 .OR. peso<=0 .OR. peso>130 .OR.   &
    altura<=0 .OR. altura>220) regmal = 1

20 IF (ios/=0 .OR. regmal==1) THEN    ! registro con error
     WRITE (12,9010) numreg
     9010 FORMAT (' *** El registro',I6,' tiene errores ***')
   ELSE
     WRITE (12,8000) nombre, dni, edad, peso, altura
   ENDIF
GOTO 10

30 STOP
ENDPROGRAM basedatos
```

7.5. CÓDIGOS DE FORMATO

Los códigos de formato, también llamados descriptores de edición, indican como se realiza la conversión entre las representaciones interna y externa de los datos mediante las sentencias `READ`, `WRITE` y `PRINT`. Se clasifican en tres grupos:

- Para datos:

Enteros	Reales	Lógicos	Caracteres	General
`I, B, O, Z`	`F, E, EN, ES, D`	`L`	`A`	`G`

- Para literales: `' '` ó `" "`

- Para control:

Posición	Espacios	Signos	Escala	Fin de Formato
`X, T, TR, TL, /`	`BN, BZ`	`S, SP, SS`	`P`	`:`

Código	***Descripción***
`Iw[.m]`	Entero base 10 (decimal)
`Bw[.m]`	Entero base 2 (binario)
`Ow[.m]`	Entero base 8 (octal)
`Zw[.m]`	Entero base 16 (hexadecimal)
`Fw.d`	Real sin exponente
`Ew.d[Ee]`	Real con exponente `e`
`ENw.d[Ee]`	Real en notación ingeniera
`ESw.d[Ee]`	Real en notación científica
`Dw.d`	Real con exponente `d`
`Lw`	Lógico
`A[w]`	Carácter
`Gw.d[Ee]`	Dato general
`' '`	Literal
`" "`	Literal
`nX`	Avanzar n espacios
`Tn`	Tabulación absoluta
`TRn`	Tabulación a la derecha
`TLn`	Tabulación a la izquierda
`[r]/`	Salto de registro
`BN`	Ignora espacios en campos numéricos
`BZ`	Convierte espacios a ceros en campos numéricos
`S`	El signo + opcional se imprime o no según el procesador
`SP`	Se imprime el signo + opcional
`SS`	No se imprime el signo + opcional
`kP`	Factor de escala
`:`	Final de formato

Significado de los valores `w, m, d, e, k, n, r`:

- `w` establece la anchura del campo
- `m` indica al menos m cifras en el campo
- `d` indica el número de cifras decimales en el campo
- `e` indica el número de cifras del exponente
- `k` es un factor de escala
- `n` indica la posición en el registro desde su principio (para el descriptor `T`)
- `n` número de espacios a mover (para los descriptores `X, TR, TL`)
- `r` factor opcional de repetición, por defecto vale 1

Restricciones: $w>0,\ e>0,\ 0\le m\le w,\ 0\le d\le w,\ 0<e\le w,\ n\ge 1,\ r\ge 1,\ k\ge 0$

Normas de edición de códigos de formato:

- En Fortran 95, `w` puede ser 0 para salida en los códigos `I`, `B`, `O`, `Z`, `F` y entonces la salida tendrá la anchura mínima necesaria para contener el dato asociado.
- Los descriptores de edición se separan por comas, las cuales pueden omitirse en los siguientes casos:
  entre el factor de escala `P` y los códigos `F`, `E`, `EN`, `ES`, `D` ó `G`
  antes de `/` (si no lleva factor de repetición)
  después de `/`
  antes o después de `:`
- Pueden ponerse espacios en cualquier lugar del formato.
- Los descriptores de edición pueden anidarse entre paréntesis.
- El factor opcional de repetición `r` es una constante entera positiva opcional que puede preceder a los siguientes códigos de formato: los de datos, los espacios `X`, la barra `/` y grupos de códigos entre paréntesis.
  Se denotará por `[r]` en la sintaxis de cada código que admita repetición.
  Ejemplo.- Son equivalentes los formatos:

```
(2(3F5.1,2(I2,3X)))
(2(F5.1,F5.1,F5.1,I2,3X,I2,3X))
```

Normas de transferencia de datos con formato:

Cuando se alcanza el último paréntesis derecho de una especificación completa de formato se procede de la siguiente forma:

- Si no hay más items en la lista I/O, termina la transferencia de datos.
- Si la lista I/O tiene más items que cantidad de códigos de formato contando las repeticiones:
  - Si hay paréntesis interiores con códigos de formato, el control de formato vuelve al principio del paréntesis izquierdo correspondiente al último paréntesis derecho precedente con su factor de repetición si lo tiene y se salta al siguiente registro.
  - Si no hay paréntesis interiores con códigos de formato el control vuelve al principio de formato y se salta al siguiente registro.

    Ejemplo.- Sea el formato `(I8,6(2F10.3,5X),F8.3,F6.0)`

    Si se utiliza en escritura el primer item de la lista de salida deberá ser entero y todos los demás reales. El primer registro contendrá un dato entero con formato `I8` y 14 datos reales con el "subformato" `6(2F10.3,5X),F8.3,F6.0`; cada uno de los siguientes registros, salvo el último, contendrá sólo 14 datos reales con el citado "subformato" y el último registro contendrá $u \leq 14$ datos reales con los u primeros códigos del citado "subformato".

Ejemplo 7-9. *Repetición de códigos de formato* **EJ7-9**

```
REAL x(100), y(500)
OPEN (11, FILE='EJ7-9.sal')

DO i = 1, 100
  x(i) = SIN(0.01*i)
ENDDO
WRITE (11, 9000) (x(i),i=1,100)
9000 FORMAT (3X,'x = ',(T11,5F10.5))

DO n = 1, 500
  y(n) = SQRT(REAL(n))
ENDDO
WRITE (11, 9100) (n,y(n),n=1,500)
9100 FORMAT (5X,(' n=',I3,T20,F8.5))  ! este formato aplica el codigo
                                      ! 5X solo al primer registro
!9100 FORMAT (5X,' n=',I3,T20,F8.5)   ! este formato es preferible
END
```

Notación.- En lo que sigue un espacio en blanco en un dato se denotará por *b*

7.5.1. Código para datos enteros: `I`

Sintaxis: `[r]Iw[.m]`
`w`: anchura del campo

Normas: Si en escritura se usa la forma `rIw.m` se añaden ceros iniciales, si son necesarios, hasta completar `m` caracteres. Ha de ser `m≤n`.
Si en escritura faltan posiciones se imprimen asteriscos; si sobran, se rellenan con espacios por la izquierda.

Nota: Los códigos `B`, `O`, `Z` se usan de forma análoga al código `I`.

Ejemplo 7-10. *Formato* `I` *para datos enteros* **EJ7-10**

```
OPEN (11, FILE='EJ7-10.dat')
! OPEN (11, FILE='EJ7-10.dat', BLANK='ZERO')
OPEN (12, FILE='EJ7-10.sal')

READ (11, '(I6,3I2,I3,I4)') i, j, k, l, m, n
WRITE (12, '(2I5,I4.2,I2,I2,I6.5)') i, j, k, l, m, n
PRINT*, i, j, k, l, m, n
END
```

registro leído: `b-2345b1b23b9871bb5` (por defecto los espacios se ignoran)
asignaciones: `i=-2345, j=1, k=2, l=3, m=987, n=15`
registro escrito: `-2345bbbb1bb02b3**b00015`
salida en pantalla: `b-2345b1b2b3b987b15`

Si la unidad 11 se hubiera abierto con `BLANK='ZERO'` entonces sería:
asignaciones: `i=-2345, j=1, k=2, l=30, m=987, n=1005`
registro escrito: `-2345bbbb1bb0230**b01005`
salida en pantalla: `b-2345b1b2b30b987b1005`

7.5.2. Códigos para datos reales: `F`, `E`, `D`, `G`

7.5.2.1. Código `F` (datos reales sin exponente)

Sintaxis: `[r]Fw.d`

`w`: anchura del campo

`d`: número de decimales detrás del punto decimal

Normas:

- La entrada debe ser una constante entera o real; si lleva el punto decimal, éste prevalece sobre la especificación `d`. La variable de salida debe ser REAL ó COMPLEX.
- En salida, se debe reservar espacio para el sigmo, el punto, la parte entera y `d` decimales. Si faltan posiciones se imprimen asteriscos; si sobran, se rellenan con espacios por la izquierda.
- Un dato COMPLEX requiere dos códigos de formato para reales.

7.5.2.2. Código `E` (datos reales con exponente)

Sintaxis: `[r]Ew.d`

`w`: anchura del campo

`d`: número de decimales detrás del punto decimal

Normas:

- En entrada se usa igual que en el código `F`.
- En salida se normaliza con: signo menos (si procede), punto decimal, `d` dígitos, letra `e`, exponente. Si faltan posiciones se imprimen asteriscos; si sobran, se rellenan con blancos iniciales. Se recomienda que $n \geq d+7$.

7.5.2.3. Código `D` (datos reales con exponente d)

Sintaxis: `[r]Dw.d`

`w`: anchura del campo

`d`: número de decimales detrás del punto decimal

Normas: Se usa igual que el código `E.` En lectura sirve para leer datos en doble precisión. En escritura sirve para escribir datos con exponente `d` que posteriormente sean leídos en doble precisión.

7.5.2.4. Código `G` (datos reales con rango amplio)

Sintaxis: `[r]Gw.d`

`w`: anchura del campo

`d`: número de decimales detrás del punto decimal

Normas:

- En entrada se usa igual que en el código `E`.
- La salida es de tipo `F` ó tipo `E` dependiendo de la magnitud del dato.

Ejemplo 7-11. *Formato* `F` *para datos reales* **EJ7-11**

```
OPEN (11, FILE='EJ7-11.dat')
OPEN (12, FILE='EJ7-11.sal')

READ (11, '(5F6.3)') a, b, c, d, e
WRITE (12, '(5F10.3)') a, b, c, d, e! reg1
WRITE (12, '(E10.3,E8.2,E10.2,E6.1,E9.3)') a, b, c, d, e    ! reg2
WRITE (12, '(G10.3,G8.3,G10.4,G10.3,G12.5)') a, b, c, d, e  ! reg3
PRINT*, a, b, c, d, e
END
```

registro leído: `bbbbb6-1234598765.b1.2+7b-28-2`
asignaciones: `a=0.006, b=-12.345, c=98765.0, d=1.2e7, e=-0.028e-2`
registro reg1: `bbbbb0.006bbb-12.345b98765.000**********bbbbb0.000`
registro reg2: `b0.600E-02-.12E+02bb0.99E+05.1E+08-.280E-03`
registro reg3: `b0.600E-02****bbbb0.9877E+05b0.120E+08-0.28000E-03`
salida en pantalla: `bbb6.0000001E-03bb-12.345000bb9.8765000E+04bb1.2000000E+07 bb-2.8000001E-04`

Consejo.- El formato `G20.10` permite escribir datos reales de cualquier magnitud.

Ejemplo 7-12. *Escritura de elementos de arrays* **EJ7-12**

```
REAL x(1000), m(50,300)
OPEN (11, FILE='EJ7-12.sal')

CALL RANDOM_NUMBER (x)
CALL RANDOM_NUMBER (m)

! se escriben los 20 elementos: x(50),x(100),...,x(1000)
! con formato 5F15.5, esto es, 4 registros con 5 elementos por registro

WRITE (11, '(5F15.5)') (x(i),i=50,1000,50)

! se escriben los 36 elementos: m(1,1), m(1,2), m(2,2),
! m(1,3),m(2,3),.m(3,3),...,m(1,8),...m(8,8) con formato 5F7.4, esto es,
! 8 registros con 5,5,5,5,5,5,5,1 elementos.

WRITE (11, '(5F7.4)') ((m(i,j),i=1,j),j=1,8)

END
```

7.5.3. Código para datos lógicos: `L`

Sintaxis: `[r]Lw`
`w`: anchura del campo

Normas:

En entrada si el primer carácter no espacio es `T` ó `.T` se asigna `.TRUE.` al dato leído, si es `F` ó `.F` se asigna `.FALSE.`.
En salida se escribe `T` ó `F` precedida de `w-1` espacios.

Ejemplo 7-13. *Formato* L

EJ7-13

```
LOGICAL log1, log2, log3
OPEN (11, FILE='EJ7-13.dat')
OPEN (12, FILE='EJ7-13.sal')

READ (11, '(2L3,L5)') log1, log2, log3
WRITE (12, '(3L2)') log1, log2, log3
PRINT*, log1, log2, log3
END
```

registro leído: `␣.T␣T␣FALSO`
asignaciones: `log1=.TRUE., log2=.TRUE., log3=.FALSE.`
salida en pantalla: `␣T␣T␣F`

7.5.4. Código para datos carácter: A

Sintaxis: `[r]A[w]`

`w`: anchura del campo

Normas:

Si se usa la forma `A` sin especificar `w`, se leen o escriben un número de caracteres igual a la longitud del item correspondiente de la lista I/O.

En entrada sea `lon` la longitud del dato carácter a leer con formato `Aw`.

- Si `w≥lon` se leen los `lon` caracteres más a la derecha del campo.
- Si `w<lon` se leen `w` caracteres del campo, se asignan justificados por la izquierda al dato carácter y se completa con `lon-w` espacios finales.

En salida sea `lon` la longitud del dato carácter a escribir con formato `Aw`.

- Si `w>lon` se escriben `w-lon` espacios seguidos de `lon` caracteres del dato carácter.
- Si `w≤lon` se escriben los `w` caracteres iniciales del dato carácter.

Ejemplo 7-14. *Formato* A

EJ7-14

```
CHARACTER(LEN=4) car1, car2, car3, car4
OPEN (11, FILE='EJ7-14.dat')
OPEN (12, FILE='EJ7-14.sal')

READ (11, '(A3,A,A4,A5)') car1, car2, car3, car4
WRITE (12, '(A,A2,A6,A4)') car1, car2, car3, car4
PRINT*, car1, car2, car3, car4
END
```

registro leído: `ABCabcdIJKLxyz`
asignaciones: `car1='ABC␣', car2='abcd', car3='IJKL', car4='yz␣␣'`
registro escrito: `ABC␣ab␣␣IJKLyz␣␣'`
salida en pantalla: `␣ABC␣abcdIJKLyz`

Consejo.- La descripción precisa del uso de los códigos de formato para datos es demasiado compleja para un estudio detallado en un curso básico. Es recomendable que los códigos de formato para datos sean sencillos y adecuados a su longitud.

7.5.5. Literales

Sintaxis: `'texto'` ó `"texto"`

En salida escribe el literal `texto`. Si uno de los caracteres es un apóstrofo y se usa el delimitador apóstrofo hay que duplicarlo. Análogamente, si uno de los caracteres es una comilla y se usa el delimitador comilla hay que duplicarla.

No puede utilizarse con sentencias `READ`.

Ejemplo 7-15. *Literales. Duplicación de apóstrofos* **EJ7-15**

```
CHARACTER(LEN=*), PARAMETER:: mensaje = &
       ' Aplicando la regla de L''Hôpital el limite es:'
flim = 3.5687
WRITE (*, '(A,F10.3)') mensaje, flim
END
```

7.5.6. Códigos de control

Espacio: `nX`

En entrada salta `n` caracteres sin cambiar de registro.

En salida deja `n` espacios en blanco antes de escribir el próximo item.

El límite de tabulación izquierda es la primera posición del registro, excepto si la operación previa sobre el fichero fue una transferencia de datos con formato y con `ADVANCE='NO'`, en cuyo caso el límite de tabulación izquierda es la posición actual en el registro.

Tabulación absoluta: `Tn`

Sitúa la posición de lectura/escritura justo antes de la columna `n` del registro actual, respecto al límite de tabulación izquierdo.

Tabulación a la derecha: `TRn`

Sitúa la posición de lectura/escritura `n` columnas a la derecha a partir de la actual en el registro actual.

Tabulación a la izquierda: `TLn`

Sitúa la posición de lectura/escritura `n` columnas a la izquierda a partir de la actual en el registro actual, con el límite de la tabulación izquierda.

Salto de registro: `/`

Indica el fin de transferencia de datos al (del) registro actual, esto es, se salta al principio del siguiente registro para leer o escribir.

Si hay n barras `/` consecutivas al principio o al final de la sentencia `FORMAT` se saltan `n` registros; si están en el interior sólo `n-1`.

Fin de formato: Dos puntos `:`

Si al llegar a los `:` no quedan más items en la lista I/O acaba el formato, si quedan más se ignoran los `:`. Es útil en salida.

Ejemplo 7-16. *Códigos de control* **EJ7-16**

```
OPEN (11, FILE='EJ7-16.sal')

x=-8.24 ; y=123.91
WRITE (11,9000) x, y
9000 FORMAT (2X,'x=',F10.5,5X,'y=',F5.1,:,'c=',F7.2//T8,'d=',F9.2)

! registro escrito:     bbx=bb-8.24000bbbbby=123.9

i=12 ; j=34 ; k=56
WRITE (11,9010) i, j, k
9010 FORMAT (1X,I2/2X,I2//3X,I2)

! 4 registros escritos:   b12
! bb34
!
! bbb56

WRITE (11,9020) i, j, k
9020 FORMAT (T5,I2,3X,I4,T15,I2)

! registro escrito:     bbbb12bbbbb34b56

a=73.2 ; b=5.7 ; n=327 ; m=542
WRITE (11,9030) a, b, n, m
9030 FORMAT (T20,F4.1,TL9,F3.1,TL15,I3,TR2,I3)

! registro escrito:     bb327bb542bbbb5.7bb73.2

END
```

Control de carro: `('b', '0', '1', '+')`

Las sentencias de salida formateada fueron diseñadas en su origen para impresoras de líneas, con el concepto de línea y página. Cuando la sentencia WRITE envía datos a una impresora, el primer carácter de cada registro se interpreta como control de carro y no se imprime. El efecto del primer carácter es:

- `b` : empieza en una nueva línea
- `0` : salta una línea
- `1` : avanza hasta el principio de la página siguiente
- `+` : no avanza, permanece en la misma línea

Nota.- El salto de página en impresoras conectadas a un PC al imprimir un fichero se obtiene con (Ctrl+L) ó Alt 12 (`CHAR(12)` en Fortran).

Consejo.- Es una buena práctica de programación insertar un blanco como primer carácter de cada registro cuando la salida se envía a la pantalla o a una impresora. Puede hacerse comenzando los formatos por `(1X,...` ó por `(T2,...`

7.6. POSICIONAMIENTO DE FICHEROS

7.6.1. Sentencia `BACKSPACE`

Sintaxis elemental: `BACKSPACE u`
Acción: Si el fichero conectado a la unidad `u` está posicionado dentro de un registro se vuelve al principio del registro actual; si está posicionado entre registros se vuelve al principio del registro precedente.
Sintaxis completa: `BACKSPACE ([UNIT=]u [,IOSTAT=ios] [,ERR=er])`
`IOSTAT=ios, ERR=er` tienen el mismo significado que en `READ`.

Ejemplo 7-17. *Uso de la sentencia* `BACKSPACE` **EJ7-17**

```
OPEN (10, FILE='EJ7-17.sal')
BACKSPACE 10    ! sintaxis elemental
nb=3 ; i=28
BACKSPACE (nb+i/4)   ! el numero de unidad puede ser una expresion
BACKSPACE (nb+i/4,IOSTAT=ios,ERR=999)
999 PRINT*, ' ios =', ios
END
```

Uso principal.- Sirve para releer registros y para reemplazar registros escritos.

7.6.2. Sentencia `REWIND`

Sintaxis elemental: `REWIND u`
Acción: Posiciona el fichero conectado a la unidad `u` al principio de su primer registro.
Sintaxis completa: `REWIND ([UNIT=]u [,IOSTAT=ios] [,ERR=er])`
`IOSTAT=ios, ERR=er` tienen el mismo significado que en `READ`.

Ejemplo 7-18. *Uso de la sentencia* `REWIND` **EJ7-18**

```
OPEN (10, FILE='EJ7-18.sal')
REWIND 10 ! sintaxis elemental
nb=3 ; i=28
REWIND (nb+i/4)! el numero de unidad puede ser una expresion
REWIND (nb+i/4,IOSTAT=ios,ERR=999)
999 PRINT*, ' ios =', ios
END
```

7.6.3. Sentencia `ENDFILE`

Sintaxis elemental: `ENDFILE u`
Acción: Escribe un registro de fin de fichero en el fichero conectado a la unidad `u`. Se posiciona después del registro de fin de fichero.
Sintaxis completa: `ENDFILE ([UNIT=]u [,IOSTAT=ios] [,ERR=er])`
`IOSTAT=ios, ERR=er` tienen el mismo significado que en `READ`.

Normas: Se escribe automáticamente un registro de fin de fichero si:

- - se ejecuta `BACKSPACE` ó `REWIND` después de `WRITE` en la unidad `u`
- - se cierra el fichero con `CLOSE`
- - se ejecuta `OPEN` con la unidad `u`
- - el programa termina normalmente

Ejemplo 7-19. *Uso de la sentencia* `ENDFILE` EJ7-19

```
OPEN (10, FILE='EJ7-19.sal')
! ENDFILE 10! sintaxis elemental
nb=3 ; i=28
! ENDFILE (nb+i/4)     ! el numero de unidad puede ser una expresion
ENDFILE (nb+i/4,IOSTAT=ios,ERR=999)
  999 PRINT*, ' ios =', ios
END
```

7.7. FICHEROS INTERNOS

Un fichero interno es una variable carácter, un array de caracteres, un elemento de un array de caracteres o un substring de caracteres. Hay dos tipos básicos:

1) Variable carácter, elemento de un array de caracteres o substring de caracteres. El fichero tiene un único registro de la longitud de la variable, elemento del array ó substring correspondiente.

2) Array de caracteres. El fichero es una secuencia de tantos registros como elementos tiene el array de caracteres. El orden de los registros es el de los elementos del array y todos tienen la misma longitud, que es la de los elementos del array.

Un fichero interno tiene tipo de acceso secuencial y forma de almacenamiento de datos formateados. La transferencia de datos se realiza con las sentencias `READ` y `WRITE`. Sirven para convertir valores entre representaciones externas de caracteres y representaciones internas en memoria. Son útiles para construir formatos variables.

La lectura de un fichero interno convierte los valores ASCII en valores numéricos, lógicos o caracteres. La escritura en un fichero interno convierte valores numéricos, lógicos o caracteres en sus representaciones ASCII. Si se escribe menos de un registro completo en un fichero interno el resto del registro se rellena con blancos.

Ejemplo 7-20. *Formato variable mediante ficheros internos* EJ7-20

```
CHARACTER(LEN=14) fmt
INTEGER k(8)
REAL x(20)

OPEN (11, FILE='EJ7-20.sal')
DO i=1,8; k(i)=i**3; ENDDO
CALL RANDOM_NUMBER (x)
```

```
DO n = 1, 8
  DO m = 2, 4
    WRITE (fmt,9000) n, m                      ! se construye el formato:
    9000 FORMAT ('(',I1,'I5,3X,',I1,'F7.4)')  ! (nI5,3X,mF7.4)
    PRINT*, fmt
    WRITE (11, fmt) (k(i),i=1,n), (x(j),j=1,m)
  ENDDO
ENDDO
END
```

Ejemplo 7-21. *Lectura de un fichero interno con diferentes formatos* **EJ7-21**

Se tiene un fichero en el que cada registro contiene datos personales de un cliente, datos de una empresa o direcciones de correo electrónico, según que el primer carácter del registro sea '1', '2' ó '3', respectivamente. Para leer la información correcta y tratarla adecuadamente el esquema podría ser el siguiente:

```
CHARACTER(LEN=80) linea
CHARACTER(LEN=20) nombre, direcc
CHARACTER(LEN=10) telef, nif, tipo, fax, cif
CHARACTER(LEN=30) email

   OPEN (12, FILE='EJ7-21.dat')
10 READ (12,'(A80)') linea
   SELECT CASE (linea(1:1))
     CASE ('1')
       READ(linea,'(1X,2A20,3A10)') nombre, direcc, telef, nif, tipo
!      CALL cliente
     CASE ('2')
       READ(linea,'(1X,A15,A20,3A10)') nombre, direcc, telef, fax, cif
!      CALL empresa
     CASE ('3')
       READ(linea,'(1X,A30)') email
!      CALL correo
     CASE ('0')
       STOP ' Fin del programa'
   ENDSELECT
   GOTO 10
END
```

7.8. PROGRAMA EJEMPLO

Se desean generar ficheros de datos simulados para ciertas variables de naturaleza meteorológica. En una situación real, estos ficheros ya estarían grabados con cierto formato y se desearían leer y analizar.

a) Simulación de la temperatura `temp`

En el mes `im` se define `tmax=35-(im-6.5)**2`

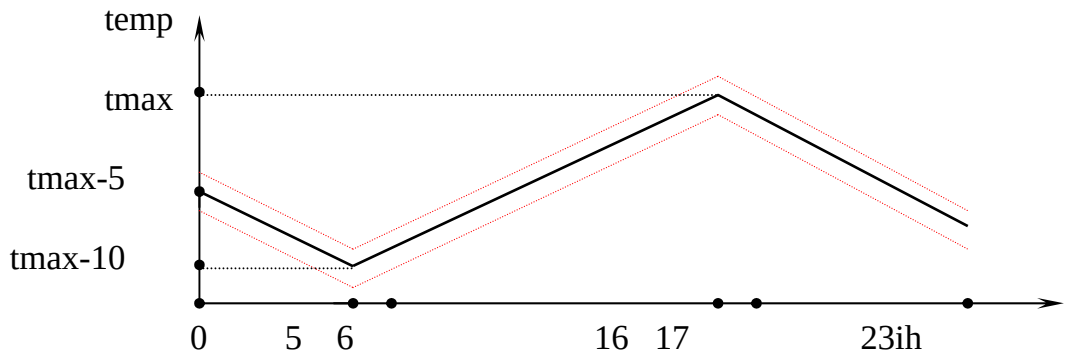
Para la hora `ih` se simula la tempetarura `temp` como sigue:

Si `0 <= ih <= 5, temp = tmax-5 - ih + u1`

Si `6 <= ih <= 16, temp = tmax-10 + (10/11)*(ih-5) + u1`

Si `17 <= ih <= 23, temp = tmax - (4/7)*(ih-16) + u1`

siendo `u1≡U(-4,4)`.

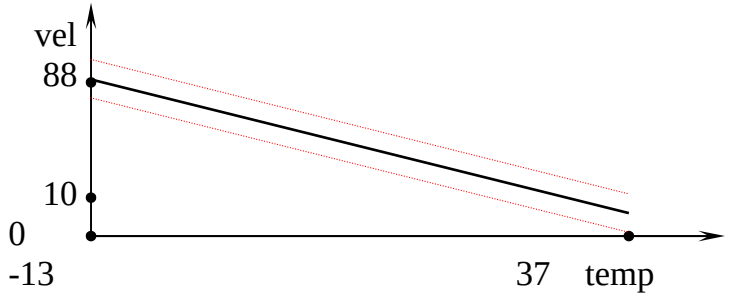


b) Simulación de la velocidad del viento `vel`

Si la temperatura es `temp` se simula la velocidad del viento como sigue:

```
vel = 88 – 1.6*(temp+13) + u2
```

siendo `u2≡U(-8,8)`.



c) Simulación de la humedad relativa del aire `hum`

Si la temperatura es `temp` y la velocidad del viento es `vel` se simula la humedad relativa como sigue:

```
hum = 60 – 0.8*(temp+13)*u3 + 0.4*vel*u4
```

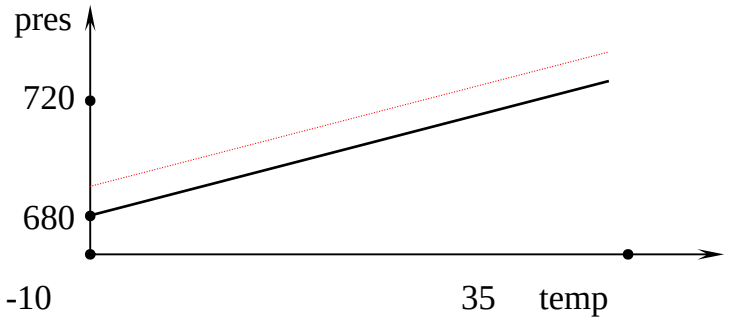
siendo `u3, u4≡U(0,1)`.

d) Simulación de la presión atmosférica `pres`

Si la temperatura es `temp` se simula la presión atmosférica como sigue:

```
pres = 680 + 0.9*(temp+5) + u5
```

siendo `u5≡U(0,4)`.



El programa considera diferentes lugares, años y meses y genera un fichero para cada terna lugar-año-mes, con datos de las cuatro variables para cada día y hora.

En este ejemplo no se pretende que los datos simulados puedan corresponder a situaciones reales.

***Ejemplo 7-22**. Simulación de datos meteorológicos*

EJ7-22

```
PROGRAM meteor
IMPLICIT NONE

! Este programa genera ficheros con datos simulados de las variables
!    temp  : temperatura
!    vel   : velocidad del viento
!    humed : humedad relativa
!    pres  : presion atmosferica

! Descripcion de variables:
!    maxlug : maximo numero de lugares admitidos
!    nlug   : numero de lugares considerados
!    lugar(nlug) : nombres de los lugares
!    nyear1 : año inicial
!    nyear2 : año final
!    ndias  : numero de dias de un mes
!    il : indice de lugar
!    ia : indice de año
!    im : indice de mes
!    id : indice de dia
!    ih : indice de hora
!    isem : semilla inicial para el generador de numeros aleatorios

! Nombres de los ficheros generados: DATmmaa.lug
! Cada fichero tiene un registro con datos por cada dia y hora

! Simulacion de datos:
! Sea tmax = 35 -(im-6.5)**2
!   si 0 <= ih <= 5   temp = tmax-5 - ih + u1               u1≡U(-4,4)
!   si 6 <= ih <= 16  temp = tmax-10 + (10/11)*(ih-5) + u1  u1≡U(-4,4)
!   si 17 <= ih <= 23 temp = tmax - (4/7)*(ih-16) + u1      u1≡U(-4,4)
! vel   = 88 – 1.6*(temp+13) + u2                          u2≡U(-8,8)
! humed = 60 – 0.8*(temp+13)*u3 + 0.4*vel*u4               u3,u4≡U(0,1)
! pres  = 680 + 0.9*(temp+5) + u5                          u5≡U(0,4)

INTEGER, PARAMETER:: maxlug=3
INTEGER nlug, nyear1, nyear2, ndias, i, il, ia, im, id, ih
INTEGER isem(1)
REAL u(5), tmax, temp, vel, humed, pres
CHARACTER(LEN=20) lugar(maxlug), fent, nomfile
CHARACTER(LEN=2) mm, aa, lug*3

WRITE (*,8000,ADVANCE='NO')
8000 FORMAT (/1X,'Fichero de nombres y dimensiones = ')
READ (*,'(A)') fent
OPEN (3, FILE=fent)
READ (3,'(3I5)') nlug, nyear1, nyear2
READ (3,'(A)') (lugar(i),i=1,nlug)

WRITE (*,8010,ADVANCE='NO')
8010 FORMAT (/1X,'Semilla inicial = ')
READ*, isem
CALL RANDOM_SEED (PUT=isem)
```

```
DO il = 1, nlug
  lug = lugar(il)(:3)
  DO ia = nyear1, nyear2
    WRITE (aa,'(I2.2)') MOD(ia,100)

    DO im = 1, 12
      WRITE (mm,'(I2.2)') im
      nomfile = 'DAT' // mm // aa // '.' // lug
      PRINT*, nomfile
      OPEN (4, FILE=nomfile)

      SELECT CASE (im)
        CASE (1,3,5,7,8,10,12)
          ndias = 31
        CASE (4,6,9,11)
          ndias = 30
        CASE (2)
          ndias = 28
        IF (MOD(ia,4) == 0) THEN
          ndias = 29
          IF (MOD(ia,100)==0 .AND. MOD(ia,400)/=0) ndias=28
        ENDIF
      ENDSELECT

      tmax = 35.0 - (im-6.5)**2

      DO id = 1, ndias
        DO ih = 0, 23
          CALL RANDOM_NUMBER (u)
          SELECT CASE (ih)
            CASE (0:5)
              temp = tmax-5 - ih + (-4.0+8.0*u(1))
            CASE (6:16)
              temp = tmax-10 + 10.0/11.0*(ih-5) + (-4.0+8.0*u(1))
            CASE (17:23)
              temp = tmax - 4.0/7.0*(ih-16) + (-4.0+8.0*u(1))
          ENDSELECT
          vel = 88.0 –1.6*(temp+13) + (-8.0+16*u(2))
          humed = 60.0 –0.8*(temp+13)*u(3) + 0.4*vel*u(4)
          pres = 680.0 + 0.9*(temp+5) + 4.0*u(5)
          WRITE (4,'(2I3,4F8.1)') id, ih, temp, vel, humed, pres
        ENDDO    ! ih
      ENDDO   ! id
      CLOSE (4)

    ENDDO   ! im
  ENDDO   ! ia
ENDDO   ! il

END
```

7.9. EJERCICIOS

1. Escribir una secuencia de código que dada una variable CHARACTER cuente el número de caracteres alfabéticos, numéricos y especiales que contiene y que saque por pantalla esta información.

2. Escribir un programa que lea los enteros n, m y cree un fichero ASCII que contenga los bordes de un rectángulo de dimensiones (n,m) y en su interior asteriscos.

3. Sea `x` una matriz entera de dimensiones (37,48). Elaborar un formato para escribir `x` por filas, en líneas con 5 datos por línea, indicando el número de la fila al comienzo de su primera línea, separando las filas con una línea en blanco. Los elementos de `x` deben escribirse con 8 caracteres y separarse con 4 espacios.

4. Sea `x` una matriz entera de dimensiones arbitrarias (n,m). Elaborar un formato para escribir una tabla de esas dimensiones, con sus celdas separadas con líneas sencillas y los bordes de la tabla dobles, que contenga los datos de la matriz `x`, siendo k la anchura de cada celda.

5. Los registros de un fichero pueden contener letras, números y símbolos especiales. Si el primer carácter del registro es `'1'` el registro contiene los campos siguientes: nombre del cliente (3-32), NIF (35-44), dirección (47-76), teléfono (79-87).
Si el primer carácter del registro es `'2'` el registro contiene los campos siguientes: número de factura (3-10), fecha (13-22), importe (25-34), IVA (37-39, 'SI' ó 'NO'), descuento aplicado (42-45, con formato F5.1), NIF del cliente (48-57), indicador de pago (60-61, 'SI' ó 'NO').
Se desea sacar un listado de las facturas pendientes de cada cliente con el importe total de las mismas.

6. Dado un fichero de entrada con los siguientes campos por registro: día (I2), mes (I2), año (I4), temperaturas en cada hora (24F4.1) crear un fichero para cada par (mes=mm, año=aa) con el nombre Fmmaa.tem, con tantos registros como días tiene ese mes y con los siguientes campos por registro: día (I2), media y desviación típica de las temperaturas (2F5.2).

7. Elaborar un programa que lea los datos generados por el ejemplo ***EJ7-22***. Se desea crear un fichero por cada lugar y año que contenga un registro por día con los datos medios de las horas de ese día.

8. Escribir una subrutina que lea un registro de entrada de hasta 132 caracteres de un fichero externo y lo pase a un fichero interno. Después debe clasificarlo como una línea de comentario Fortran, una línea inicial sin etiqueta, una línea inicial con etiqueta, una línea de continuación o una línea multisentencia.

9. Modificar el programa del ejemplo ***EJ1-9*** del capítulo 1 para resolver ecuaciones mediante el método de la bisección en el sentido del ejercicio **5** del capítulo 5 (variable `iter`) escribiendo la información de salida en ficheros.

CAPÍTULO 8. *ELABORACIÓN DE PROGRAMAS*

Es importante cuidar la elaboración del programa fuente y procurar satisfacer varios objetivos, entre otros: que sea claro y legible tanto para el autor del programa como para otros potenciales usuarios, que sea fácil detectar errores, que sea eficiente en tiempo, precisión o memoria, que permita introducir cambios con facilidad, etc.

8.1. ESTILO DE PROGRAMACIÓN

Algunos detalles que favorecen el estilo de programación son:

- Amplio uso de comentarios: incluir una breve descripción de algoritmos o procedimientos al principio de cada unidad de programa, en secciones de código diferenciadas, en límites de arrays, en sentencias que deberían cambiarse para ejecución con otros datos, etc.
- Descripción del significado de cada variable.
- Declaración organizada de variables (alfabética, por tipos, agrupada por similaridades, etc.).
- Líneas en blanco de separación entre secciones de código (bucles, bloques `IF`,...) y entre subprogramas.
- Desplazamiento ("Indentación") de las sentencias de estructuras (bucles, bloques `IF`, `CASE`,...) unos espacios (2 ó 3) a la derecha.
- Inserción de espacios en blanco a ambos lados de algunos operadores y del signo igual, después de algunas comas. En expresiones largas suele ser preferible no abusar de los blancos.
- Las sentencias `FORMAT` pueden agruparse todas juntas al final de la unidad de programa. También puede ponerse el formato en la sentencia de entrada/salida.
- La sentencia `GOTO` complica la legibilidad de un programa y empeora la eficiencia. La mayoría de los `GOTO` pueden evitarse con estructuras adecuadas. Sin embargo, a veces para suprimir un `GOTO` hay que complicar la redacción con estructuras y variables lógicas y es preferible mantener la sentencia `GOTO`.
- Se debe evitar el uso de sentencias obsoletas ya que podrían desaparecer en futuras versiones de Fortran.

8.2. DEPURACIÓN DE ERRORES

Los errores que pueden cometerse en la elaboración de un programa Fortran son de clase muy diversa: sintaxis, diseño del programa, programación, algorítmicos, instalación del software, errores de tamaño de memoria, etc. Una vez realizada una correcta instalación del software, los otros errores son imputables al usuario ó a limitaciones de software o hardware. La ley de MURPHY no falla cuando se aplica en programación: "La primera codificación de un programa contiene errores". Algunos detalles que favorecen la detección y corrección de errores son:

- Una redacción clara con suficientes comentarios.
- Evitar estructuras de control, formatos y expresiones complicados.
- Si un programa es muy largo conviene partirlo en subprogramas. Es difícil corregir un subprograma de más de unas 300 líneas ejecutables. Por otra parte, algunos compiladores no pueden optimizar subprogramas largos.
- En primeras versiones de un programa conviene incluir sentencias de escritura (a pantalla ó fichero) después de secciones diferenciadas de código con objeto de aislar posibles errores o comprobar el buen funcionamiento de partes de código.
- Aunque prácticamente todos los compiladores tienen un sistema de depuración de errores (DEBUG) es conveniente que la redacción del programa sea tal que si se cometen errores se puedan localizar y corregir con la mayor facilidad posible.
- Siempre que sea posible conviene probar el funcionamiento del programa con ejemplos de los que se sepa cuáles deben los resultados del programa.
- Los errores de sintaxis se muestran en el momento de compilación.
- Un error muy frecuente es el acceso a elementos de un array con subíndices fuera de sus cotas. Los compiladores tienen opciones de compilación para detectar tal circunstancia en ejecución.

8.3. OPTIMIZACIÓN DE PROGRAMAS

Algunos detalles que afectan a la eficiencia de un programa son:

- Uso de la opción de compilación para optimizar la velocidad en ejecución.

- Potenciación `a**b`:
  si `b` es entero, `a**b` se obtiene con multiplicaciones
  si `b` es real, `a**b` se calcula como: `EXP(b*LOG(a))` y debe ser `a>0`
  Ejemplos:
  `2.4**5` se calcula como `2.4*2.4*2.4*2.4*2.4`
  `2.4**5.0` se calcula como `EXP(5.0*LOG(2.4))`

- La raíz cuadrada es una operación rápida:
  `SQRT(x)**3` es mejor que `x**1.5`

- Siempre que se pueda conviene ahorrar operaciones y simplificar fórmulas, aunque las expresiones pueden ser numéricamente distintas. Ejemplos:
  `x+y` es mejor que `(x**2-y**2)/(x-y)`
  `COS(2.0*x)` es mejor que `2.0*COS(x)**2 - 1.0`

- Si no hay peligro de confusión conviene reutilizar variables, vectores y matrices. Si los elementos de un vector o matriz son conocidos, se puede prescindir de ellos: ¡es inadmisible dimensionar la matriz identidad de orden 5000!. El acceso a elementos de arrays consume tiempo.

8.4. EJERCICIOS

La ÚNICA forma de aprender cualquier lenguaje de programación es haciendo programas. Como sugerencia, además de los ejercicios de cada capítulo, los siguientes pueden servir como primeras prácticas. En algunos de ellos es necesario conocer el correspondiente método numérico.

(a) Encontrar todos los números de 3 cifras que son iguales a la suma de los cubos de sus 3 cifras.

(b) Calcular todos los números primos menores que `N`.

(c) Sea `X` cualquier número de 4 cifras, no todas iguales. Sea `X1` el número obtenido ordenando las cifras de `X` de mayor a menor y `X2` de menor a mayor. Sea ahora `X=X1-X2`. Demostrar que repitiendo este proceso a lo sumo siete veces se obtiene el número 6174.

(d) Encontrar una multiplicación correcta, de forma que cada cifra del 0 al 9 esté exactamente dos veces y tenga el siguiente formato:

```
     * * *
     * * *
---------
     * * *
   * * *
* * *
---------
* * * * *
```

(e) Escribir las permutaciones de `N` elementos.

(f) Operaciones aritméticas elementales con números de gran cantidad de cifras. Como aplicación:
(f1) calcular 10000 cifras del número e.
(f2) calcular y escribir 1000!

(g) Resolución de triángulos: conociendo tres datos de entre los lados y los ángulos calcular los otros tres y el área. Elaborar un menú de opciones.

(h) Cálculos estadísticos para una variable: media, desviación típica, momentos, coeficientes de asimetría y curtosis, etc., de una muestra de datos.

(i) Cálculos estadísticos para dos variables: momentos, regresión, correlación, análisis de la varianza.

(j) Cálculo del número π por simulación estimando superficies relacionadas con el número π mediante números aleatorios.

(k) Cálculo numérico de la integral definida de una función por el método del trapezoide, fórmula de Simpson, reglas adaptativas, etc.

(l) Resolución numérica de sistemas lineales de ecuaciones.

(m) Estadísticas de caracteres en un texto. Contar vocales, sílabas, palabras, secuencias de letras, etc.

(n) Crear mediante simulación una base de datos con los siguientes campos: DNI, nombre, apellidos, dirección, teléfono. Elaborar subrutinas para ordenarla por un campo dado.

APÉNDICE. *CARACTERÍSTICAS AVANZADAS EN Fortran*

A.1. TIPOS Y CLASES DE DATOS

Parámetro KIND.
Funciones: KIND, SELECTED_INT_KIND, SELECTED_REAL_KIND.
Constantes con clase, argumento de clase en funciones intrínsecas.
Sentencias de especificación. Sentencia IMPLICIT.
Atributos PUBLIC, PRIVATE, SAVE.
Sentencias de especificación redundantes.
Punteros. Atributos POINTER, TARGET.
Tipo derivado de datos.
Modelos numéricos para datos enteros y reales.
Datos binarios, octales y hexadecimales.

A.2. TRATAMIENTO DE ARRAYS

Arrays de tipo intrínseco. Constructores de arrays. Secciones. Subíndices vectoriales. Operaciones con arrays.
Arrays de tamaño cero.
Argumentos array. Forma explícita, tamaño asumido, forma asumida.
Arrays con subíndice vectorial.
Objetos automáticos.
Memoria dinámica. Sentencias ALLOCATE, DEALLOCATE. Función ALLOCATED. Atributo ALLOCATABLE.
Funciones con resultado array.
Máscaras. Sentencia y bloques WHERE.
Funciones intrínsecas para arrays: reducción, multiplicación, interrogación, construcción, manipulación, subíndices.

A.3. PROCEDIMIENTOS

Unidades de alcance. Asociación. Definición.
Subprogramas recursivos.
Procedimientos intrínsecos. Funciones numéricas de interrogación.
Manipulación de bits.

A.4. FICHEROS

Ficheros de acceso directo.
Ficheros no formateados.
Condiciones de error.
Posicionamiento de ficheros.
Opciones de las sentencias OPEN, INQUIRY.
Sentencia NAMELIST.

A.5. INTERFACES Y OPERADORES

Bloques INTERFACE.
Definición de operadores. Sobrecarga.

A.6. COMPILADORES

Utilidad DEBUG.
Utilidades gráficas.
Entorno Windows.

A.7. LIBRERÍAS MATEMÁTICAS

Librerías de los compiladores.
Subrutinas NAG, IMSL. NUMERICAL RECIPES.
Otras librerías gráficas, numéricas, estadísticas.

A.8. Fortran EN INTERNET

Información. Páginas web.
Diversos compiladores.
Software de dominio público en código fuente.

BIBLIOGRAFÍA

BIBLIOGRAFÍA BÁSICA

ADAMS, J.C., BRAINERD, W.S., MARTIN, J.T. SMITH, B.T. y WAGENER, J.L. (1997) "Fortran 95 Handbook. Complete ISO/ANSI Reference" MIT Press

BRAINERD, W.S., GOLDBERG, C.H. y ADAMS, J.C. (1996) “Programmer’s Guide to Fortran 90” McGraw-Hill

HAHN, B.D. (1997) "Fortran 90 for Scientists and Engineers" Arnold

METCALF, M. y REID, J. (1996) “FORTRAN 90/95 Explained” Oxford University Press

METCALF, M. y REID, J. (1999) “FORTRAN 90/95 Explained” Second Edition, Oxford University Press

METCALF, M., REID, J. y COHEN, M. (2004) “FORTRAN 95/2003 Explained” Oxford University Press

REDWINE, C. (1995) “Upgrading to Fortran 90” Springer Verlag

Página web principal: http://www.fortran.com

BIBLIOGRAFÍA COMPLEMENTARIA

CHAPMAN, S.J. (2007) "Fortran 95/2003 for Scientists and Engineers” McGraw-Hill

PRESS, W.H., TEUKOLSKY, S.A., VETTERLING, W.T. y FLANNERY, B.P. (1992) "Numerical Recipes in Fortran. The Art of Scientific Computing" Second Edition, Cambridge University Press

PRESS, W.H., TEUKOLSKY, S.A., VETTERLING, W.T. y FLANNERY, B.P. (1996) "Numerical Recipes in Fortran 90. The Art of Parallel Scientific Computing" Second Edition, Cambridge University Press

VETTERLING, W.T., TEUKOLSKY, S.A., PRESS, W.H. y FLANNERY, B.P. (1992) "Numerical Recipes. Example Book (Fortran)" Second Edition, Cambridge University Press