NOTAS DE FORTRAN 77

C. García-Recio, L. L. Salcedo
Departamento de Física Atómica, Molecular y Nuclear,
Universidad de Granada, E-18071 Granada, Spain
E-mail: g_recio@ugr.es, salcedo@ugr.es

24 de mayo de 2018

Resumen

Manual de consulta rápida de FORTRAN. Se incluyen las construcciones más usuales aunque no todas son estándar. En caso de duda, para ver cómo funciona exactamente una construcción lo más práctico es preguntar al compilador: hacer un pequeño programa de prueba y ver cómo responde (aunque a veces el resultado puede depender del compilador de FORTRAN).

Índice

1. Estructura general, variables ***2***

1.1. Estructura de un programa . . . 2
1.2. Estructura de una línea FORTRAN . . . 2
1.3. Caracteres . . . 3
1.4. Tipos de variables o constantes . . . 3
1.5. Declaraciones . . . 3
1.5.1. Declaraciones de tipo . . . 3
1.5.2. Sentencia `IMPLICIT` . . . 3
1.5.3. Sentencia `DIMENSION` . . . 4
1.5.4. Sentencia `COMMON` . . . 4
1.5.5. Sentencia `DATA` . . . 4
1.5.6. Sentencia `PARAMETER` . . . 4
1.6. Expresiones, operaciones, asignaciones. . . . 4

2. Sentencias de control y bucles ***5***

2.1. Sentencias `STOP, RETURN` . . . 5
2.2. Sentencia `IF` . . . 5
2.3. Sentencia `DO WHILE` . . . 6
2.4. Sentencia `DO` . . . 6
2.5. Sentencia `DO` *con etiqueta* . . . 6
2.6. Sentencia `EXIT` . . . 6
2.7. Sentencia `GO TO` . . . 6

3. Subprogramas subrutina ***7***

3.1. Declaración y uso 7

3.2. Argumentos de una subrutina 7

3.3. `COMMON` 7

3.4. Matrices como argumento 8

3.5. Subprograma como argumento 8

4. Subprogramas función ***9***

4.1. Funciones externas 9

4.2. Funciones sentencias 9

4.3. Funciones intrínsecas 9

5. Lectura-Escritura ***10***

5.1. Método simple 10

5.2. Ficheros 10

5.3. Formatos de escritura y lectura 11

6. Linux ***12***

6.1. UNIX 12

6.2. Comandos útiles de Linux 12

Para mayor claridad en estas notas usamos mayúsculas para las palabras reservadas pero en realidad el compilador de FORTRAN no distingue entre mayúsculas y minúsculas. Tampoco ve los espacios (los caracteres espacio en blanco) excepto en variables de tipo carácter y similares.

1. Estructura general, variables

1.1. Estructura de un programa

```
PROGRAM ejemplo       ! sentencias no ejecutables
INTEGER i             ! sentencias no ejecutables
i = 3                 ! sentencias ejecutables
WRITE(*,*) ' i =',i   ! sentencias ejecutables
END                   ! sentencias ejecutables
```

`PROGRAM` > `IMPLICIT NONE` > `IMPLICIT` > `REAL, INTEGER, LOGICAL, DIMENSION, PARAMETER` > `COMMON, SAVE, DATA` > funciones sentencia > sentencias ejecutables > `END`.

1.2. Estructura de una línea FORTRAN

Comentario:	`C` *ó cualquier carácter no numérico en columna 1.*
Etiquetas:	*Número de etiqueta en columnas 1-5 (usualmente 2-5). Se utilizan para* `CONTINUE` *y* `FORMA`
Continuación de línea:	*Carácter en la columna 6. Típicamente* `F` *ó* `1,2,3,...`*, ó* `$`*.*
Sentencia:	*Columnas 7 a 72.*

```
   C  Esto es un comentario
C23456789012345678901
 100  a = SIN(x)     ! Esto es la sentencia a=SIN(x)*COS(x) innecesariamente escrita en dos líneas,
     $ *COS(x)       ! Se admiten hasta 19 continuaciones de línea.
                     ! El signo de admiración es otra forma de poner un comentario.
```

Se recomienda el uso del tabulador para ajustar el inicio de sentencias, y el uso del sangrado para resaltar la estructura de los programas.

1.3. Caracteres

Alfanuméricos: `a−z A−Z 0−9`
Especiales: `= + − * | ( ) , . ' '' ! : _`
FORTRAN no distingue entre mayúsculas y minúsculas. No admite ñ ni acentos (excepto en comentarios).

1.4. Tipos de variables o constantes

Tipo	*En el ordenador*	*Ejemplos*
Entero 2 bytes	`INTEGER` *ó* `INTEGER*2`	`15, -11305`
Entero 4 bytes	`INTEGER*4`	`15, -11305, -1119832`
Real 4 bytes	`REAL` *ó* `REAL*4`	`-10.35, 12., 3.1E-15, 0.00317`
Real doble precisión	`DOUBLE PRECISION` *ó* `REAL*8`	`-10.35D0, 3.1D-15`
Complejo	`COMPLEX` *ó* `COMPLEX*8`	`(3.,0.2E-7)`
Complejo doble precisión	`DOUBLE COMPLEX` *ó* `COMPLEX*16`	`(3.D0,0.2D-7)`
Carácter	`CHARACTER, CHARACTER*23`	`'hola'`, ``Hola''
Lógico	`LOGICAL`	`.TRUE., .false., .T., .F.`
Vectores (enteros, reales,...)	`INTEGER id(50), DIMENSION v(-20:43),...`	`id(5), v(-3), a(55)`
Matrices (enteros, reales,...)	`INTEGER im(5,10,50), REAL am(-1:20,100)`	`im(4,1,40), am(-1,55)`

1.5. Declaraciones

1.5.1. Declaraciones de tipo

Los nombres de variables tienen a lo sumo seis caracteres, estos son de tipo alfanumérico, y el primero deben ser alfabético. Por omisión (si no se declaran explícitamente de otra forma) las variables que empiezan por `i`, `j`, ..., `n`, *son enteras, el resto reales.*

```
REAL a3, b(10,2)
INTEGER xint
LOGICAL y, pepe
CHARACTER*30 file, file1
COMPLEX*16 z, zdim(-10,10)
```

1.5.2. Sentencia `IMPLICIT`

```
IMPLICIT NONE       ! Implica que hay que declarar todas las variables

IMPLICIT COMPLEX*16 (c)   ! Las variables que empiezan por c son complejas doble precisión
                          ! por omisión
IMPLICIT DOUBLE PRECISION (a,b,d-h,o-z) ! y todas éstas son DOUBLE PRECISION por omisión
INTEGER dint       ! Pero esta variable es entera
```

1.5.3. Sentencia `DIMENSION`

```
DIMENSION b(10,2), x(3), c(10,100,8)
DIMENSION a(-3:3,0:1,3)
```

En una matriz el primer índice corre primero, luego el segundo y así sucesivamente (al revés que el sistema decimal). El elemento `b(3,2)` *de la matriz* `b` *antes dimensionada a* `(10,2)`*, está en la posición 10+3=13 de la memoria. La matriz* `b` *de dimensión* `(10,2)` *está almacenada como sigue:*

```
b(1,1), b(2,1), b(3,1),..., b(10,1), b(1,2), b(2,2), b(3,2),..., b(10,2)
```

Independientemente de su estructura, FORTRAN ve toda matriz como un vector.

1.5.4. Sentencia `COMMON`

Comparte variables entre dos o más unidades (o módulos) de programación.

```
COMMON datos, b3z         ! Éste es el COMMON ‘‘blanco’’, sin nombre.
COMMON rz(30), lg
COMMON /zona1/ rz, rx     ! Éste es un COMMON con nombre: zona1.
```

El nombre de las variables identificadas no tiene porqué coincidir (ni el tipo ni nada) en los distintos módulos. El nombre de un `COMMON` *(que sigue las mismas reglas que para variables) es visible en todo el programa, y eso es malo para la portabilidad (el nombre puede estar ya usado).*

1.5.5. Sentencia `DATA`

Se usa para inicializar los valores de variables al principio de un programa o subprograma. El valor de esas variables se puede redefinir en otras partes del programa.

```
DIMENSION a(3), b(5)
DATA a/1.,2.,3./
DATA b/2*0.,1.,2*0./   ! Equivalente a definir b(1)=0.,b(2)=0.,b(3)=1.,b(4)=0., b(5)=0.
DATA i,j/2,3/,k,l/3,5/
```

1.5.6. Sentencia `PARAMETER`

Se usa para definir constantes (al principio de un programa o subprograma). El valor de estas constantes no se puede redefinir; queda fijado en todo la ejecución. Las constantes sólo se pueden usar después de haberse definido.

```
COMPLEX i,i2
PARAMETER (i=(0.,1.), i2=2.*i, pi = 3.141593)
PARAMETER (nmax=100)       ! nmax es el nombre de una constante igual a 100
REAL*8 a5(nmax)
```

1.6. Expresiones, operaciones, asignaciones.

```
ant = b1 * 3.5 + ant
```

`+ - * / **` *(enteros, reales, complejos)*

3 3 2 2 1 *(prioridad de la operación)*

La división de enteros trunca a entero:

```
write(*,*) -5/3       !Escribe -1
```

Tipos mixtos:

```
DOUBLE PRECISION x
a = 3       ! Convierte a REAL. Guarda a como 3.0000000 (salvo redondeo)
```

```
i = -3.2       ! Convierte a INTEGER. Guarda i como -3
a = FLOAT(2)/3       ! FLOAT convierte a REAL. Guarda a como -0.6666666 (salvo redondeo)
x = 2./3       ! Evalúa a REAL. Guarda x como -0.6666666000000000D0 (salvo redondeo)
x = DBLE(2.)/3       ! Guarda x como -0.6666666666666666D0 (salvo redondeo)
x = DBLE(2)/3        ! Guarda x como -0.6666666666666666D0 (salvo redondeo)
x = DBLE(2/3)        ! Guarda x como 0.0D0
```

Operaciones lógicas, comparaciones:

.EQ.	.GT.	.LT.	.GE.	.LE.	.NE.	.AND.	.OR.	.NOT.
$=$	$>$	$<$	$\geq$	$\leq$	$\neq$	*y*	*o*	*no*

2. Sentencias de control y bucles

2.1. Sentencias `STOP`, `RETURN`

`STOP`*: detiene la ejecución de un programa. En su defecto la ejecución termina en el* `END` *del programa principal.*
`RETURN`*: en un subprograma (subrutina o función) devuelve el control de ejecución al módulo que invocó dicho subprograma.*

2.2. Sentencia `IF`

```
IF (a.GT.b) b = a
```

es equivalente a:

```
LOGICAL l
l = a.GT.b
IF (l) b = a
```

```
IF (l) THEN
  b = a
  a = a+1
END IF
```

```
IF (a.NE.0) THEN
  a = 0
ELSE
  a = 1
END IF
```

```
IF (a.LE.0) THEN
  a = 0
ELSE IF (a.GE.10) THEN
  a = 1
ELSE
  a = 2
END IF
```

```
IF (a.EQ.0) THEN
  STOP
ELSE IF (a.NE.1.OR.a.NE.3) THEN
  WRITE(*,*) a
END IF
```

2.3. Sentencia DO WHILE

```
i = 5
DO WHILE (i.NE.0)
  WRITE(*,*) i     ! escribe 5,4,3,2,1
  i = i-1
END DO
WRITE(*,*) i       ! escribe 0
```

2.4. Sentencia DO

```
DO i = 1, 15
  WRITE(*,*) i     ! escribe 1,2,3,...,14,15
END DO
```

```
DO i = 13,-6,-5
  WRITE(*,*) i     ! escribe 13, 8, 3, -2
END DO
```

```
DO x = 0.2, -0.4, -0.05  ! Algunos compiladores admiten contadores reales.
  WRITE(*,*) x           ! Debe escribir 0.2, 0.15, 0.10, ..., -0.35, -0.40
END DO                 ! No se recomienda su uso por no ser estándar. Puede dar resultados
                       ! imprevistos en distintos ordenadores por errores de redondeo
```

```
DO i = 1, 5 ! bucles anidados
  DO j = 1, 10 ! bucles anidados
    WRITE(*,*) '(',i,',',j,')'   ! escribe (1,1),(1,2), ...,(1,10),(2,1),...,(5,10)
  END DO
END DO
```

El contador `i` *no se puede modificar (por ejemplo* `i = i+1`*) dentro un bucle "*`DO i = imin, imax, idel`*", ..., "*`END DO`*''. Sin embargo, las variables* `imin, imax` *sí se pueden modificar.*

2.5. Sentencia DO con etiqueta

```
      DO 30 i = 1, 15, 3
30    WRITE(*,*) i       ! escribe 1,4,7,10,13
```

Se recomienda usar los `DO` *sin etiqueta.*

2.6. Sentencia EXIT

Sale de un bucle `DO`*; la ejecución continúa con la primera sentencia posterior al fin del* `DO`*.*

2.7. Sentencia GO TO

```
      GO TO 100
      ...
100   CONTINUE
```

Se puede salir de un bucle con un `GO TO`*, pero no entrar en él. Debe evitarse el uso de la sentencia* `GO TO`*. Además, se recomienda que siempre envíe la ejecución a una sentencia* `CONTINUE`*.*

3. Subprogramas subrutina

3.1. Declaración y uso

Se declaran como el programa principal, pero en vez de "`PROGRAM nombre`*", se usa como primera línea*[1]*:*

```
SUBROUTINE nombre (arg1,arg2,...)
```

y acaban con la línea:

```
END
```

Al final de la ejecución vuelven al punto donde se llamó la subrutina.
Se invocan con

```
CALL nombre (p1,p2,...)
```

El nombre de la subrutina es global (visible en todo el programa), los nombres de los argumentos son locales (visibles sólo dentro de la subrutina), ejemplo:

```
CALL sub (x,y)
...
SUBROUTINE sub (a,b)
```

3.2. Argumentos de una subrutina

Al llamar a la subrutina, se asocian temporalmente los argumentos con las variables o constantes de la unidad invocante. Así, en el ejemplo anterior, `a` *es* `x` *y* `b` *es* `y`*. La asociación es por valor y/o por variable.*

```
...
b = 5.
y = 2.
CALL sub (1., y)
WRITE(*,*) y, b
```

```
SUBROUTINE sub (a,b)
...
WRITE(*,*) a, b
b = 3.
```

Primero escribe `1., 2.` *(o sea* `a, b`*) en* `sub` *y luego escribe* `3., 5.` *(o sea* `y, b`*) en el módulo o programa que invoca a la subrutina. (Por supuesto, "*`a = 3.`*" ó "*`a = ...`*" en* `sub` *produciría un error.)*

La asociación desaparece cuando acaba la ejecución de la subrutina (al llegar al `END` *o* `RETURN` *dentro de la subrutina). En principio el valor de la variable del subprograma se pierde (no se puede usar al volver a invocar la subrutina) excepto si:*
- está en un `DATA` *y no se ha redefinido*
- está en un `COMMON`
- está en una sentencia `SAVE`

```
SAVE a, b !guarda a, b (variables locales)
SAVE      !guarda todas las variables locales
```

`SAVE` *es una sentencia no ejecutable, se sitúa antes de las ejecutables. Los argumentos del subprograma no se guardan con* `SAVE`*.*

3.3. `COMMON`

Otro método de comunicarse con el subprograma es a través de un `COMMON`*. Este método tiene sus ventajas (no hay que enumerar los argumentos) y sus inconvenientes (la variable es visible fuera del subprograma y se puede alterar su valor por error. El programa es menos portable).*
No se puede poner una variable a la vez como argumento y en un `COMMON`*.*

[1] La subrutina también puede no tener argumentos: "`SUBROUTINE nombre''`.

3.4. Matrices como argumento

Al pasar una matriz (o vector, etc) como argumento lo que se hace es identificar el elemento inicial de la matriz en la unidad invocante con el elemento inicial de la matriz en el subprograma.

- *Ejemplo 1:*

```
DIMENSION a(10)                    SUBROUTINE sub(x)
CALL sub(a)                        DIMENSION x(*)
```

En este ejemplo la variable "`x`*" de "*`sub`*" se identifica con la variable "*`a`*" de la unidad invocante:* `x(1) = a(1), ..., x(10) = a(10)`*. También hubiera hecho el mismo efecto* `DIMENSION x(1)` *o* `DIMENSION x(27)`*: la dimensión de "*`x`*" es la de "*`a`*".*

- *Ejemplo 2:*

```
DIMENSION a(10,3,5)                SUBROUTINE sub(x)
CALL sub(a)                        DIMENSION x(10,3,*)
```

Identifica `x(i,j,k)` *con* `a(i,j,k)`*. Para que la identificación sea correcta, las dimensiones en la subrutina deben coincidir con las dimensiones externas, excepto la última.*

- *Ejemplo 3:*

```
DIMENSION a(10,3,5)                SUBROUTINE sub(ndim1,ndim2,x)
CALL sub(10,3,a)                   DIMENSION x(ndim1,ndim2,*)
```

También identifica `x(i,j,k)` *con* `a(i,j,k)`*. En este caso se han pasado las dimensiones como argumento (excepto la última, que no hace falta).*

- *Ejemplo 4:*

```
DIMENSION a(10)                    SUBROUTINE sub(x)
CALL sub(a(5))                     DIMENSION x(*)
```

En este caso `x(1) = a(5),...,x(6) = a(10)`*.*

- *Ejemplo 5:*

```
DIMENSION a(10,3)                   SUBROUTINE sub(x)
CALL sub(a(1,2))                    DIMENSION x(*)
```

En este caso `x(1) = a(1,2),...,x(10) = a(10,2),x(11) = a(1,3),....`

- *Ejemplo 6:*

```
COMPLEX a                           SUBROUTINE sub(x,y)
CALL sub(a)                         REAL x,y
```

Identifica `x` *con la parte real de* `a` *e* `y` *con la parte imaginaria de* `a`*.*

3.5. Subprograma como argumento

Requiere el uso de `EXTERNAL` *en las especificaciones:*

```
EXTERNAL fun              ! (fun es el nombre auténtico del subprograma en la
CALL sub(x,y,fun)         ! unidad de programación que llama a la subrutina)

SUBROUTINE sub(a,b,subs)
EXTERNAL subs             ! (subs es el nombre simbólico)
```

Si el argumento en la unidad invocante es una función intrínseca (no definida por el usuario) hay que usar INTRINSIC en lugar de EXTERNAL (además debe ser una función intrínseca específica, no genérica):

```
INTRINSIC COS                       SUBROUTINE sub7(a,b,fun)
CALL sub7(x,y,COS)                  EXTERNAL fun
```

4. Subprogramas función

4.1. Funciones externas

Son subprogramas totalmente análogos a las subrutinas, excepto que el nombre de la función es a su vez una variable

```
                                                    FUNCTION f(x,y)
REAL y, f                                           REAL x,y,f
y = 2.                                              f= x**2+1.
WRITE(*,*) 2*f(3.,y) ! Escribe 10.                  RETURN
                                                    END
```

Otra diferencia es que las subrutinas pueden no tener argurmentos:

```
CALL sub                SUBROUTINE sub
```

en cambio las funciones deben tener al menos los paréntesis:

```
                                FUNCTION ran()
WRITE(*,*) ran()                REAL*4 ran
                                ...
```

4.2. Funciones sentencias

Son sentencias que no están separadas sino en el programa principal o un subprograma. Son de la forma:

```
INTEGER n
REAL f,x
f(x)= 2.+x**n       ! Define la función f(x) (que también depende de n)
```

- *La sentencia* `f(x)= 2.+x**n`*, que define la función* `f` *es no ejecutable y debe estar antes de todas las sentencias ejecutables y después de las especificaciones.*
- *En* `f(x)= 2.+x**n`*, la variable* `x` *es local (visible) a nivel de sentencia. Es un argumento simbólico y no existe fuera de la sentencia (excepto por ser definida de tipo* `REAL` *en las especificaciones). El símbolo* `x` *puede usarse como variable en otra parte de la unidad de programación sin que se identifique con el* `x` *argumento de* `f`*.*
- *En cambio la variable* `n` *es visible en toda la unidad de programación. Es una variable (no un argumento de* `f`*) y toma el valor que tenga en la unidad de programación en el momento de llamar a* `f`*.*
- *Una función puede tener cualquier número de argumentos y de cualquier tipo. Si no se declara el tipo se supone el tipo implícito.*
- *Una función sentencia es totalmente equivalente a reemplazar la expresión de* `f` *(y de sus argumentos) en el momento de la compilación.*
- *Una función sentencia sólo debe ocupar una sentencia (con continuaciones de línea si es preciso).*

4.3. Funciones intrínsecas

Están ya definidas y son reconocidas por el compilador.

- *Si se define una nueva función con el mismo nombre que una intrínseca, prevalece la definida por el usuario.*
- *Las principales funciones intrínsecas son:*

  `INT, REAL, DBLE, CMPLX, ABS, MOD, SIGN, MAX, MIN, AIMAG, CONJG, SQRT, EXP, LOG, LOG10, SIN, COS, TAN, ASIN, ACOS, ATAN, ATAN2, SINH, COSH, TANH.`

- *Frecuentemente hay una versión genérica (*`LOG`*) que sirve para cualquier tipo de argumento numérico y una específica para cada tipo (*`ALOG, DLOG, CLOG`*).*

5. Lectura-Escritura

5.1. Método simple

```
READ(*,*) a, b        ! Lectura por pantalla y sin formato.
WRITE(*,*) a, b       ! Escritura por pantalla y sin formato.
```

Es válido

```
WRITE(*,*) 3, a, 'hola'
```

pero no

```
READ(*,*) 3, a, 'hola'       ! (3 y 'hola' son constantes)
```

Si `a` *es una vector, la sentencia*

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

lee todos los elementos de `a` *(ordenados:* `a(1), a(2), ...`*). Ídem en escritura.*

La sentencia (`DO` *implícito)*

```
WRITE(*,*) (i,a(i), i = 1, 5)
```

escribe `1, a(1), 2, a(2), ...5, a(5)`*. La lista puede tener cualquier expresión evaluable en términos de* `i`*:*

```
WRITE(*,*) (2*i,(b(i,j),j=1,30,2), i=1,10)
```

En lectura el ordenador (es decir, durante la ejecución) lee la lista de izquierda a derecha. Así:

```
READ(*,*) i, a(i)
```

Si lee "3 5.0", equivale a "`i = 3`*" y "*`a(3) = 5.0`*".*

5.2. Ficheros

Con la sentencia ejecutable

```
OPEN (3, FILE='datos')
```

se asocia la unidad 3 con el fichero datos. Así:

```
WRITE(3,*) a       ! escribe a en la unidad 3 (fichero datos) sin formato
READ(3,*) a        ! lee a de la unidad 3 (fichero datos) sin formato
```

- *Por omisión la unidad 5 es lectura por pantalla y la unidad 6 es escritura por pantalla:*

  ```
  READ (5,*) a
  WRITE (6,*) b
  ```

  equivalen a

  ```
  READ (*,*) a
  WRITE (*,*) b
  ```

- *El número de la unidad se refiere al mismo fichero en todo el programa. Por ejemplo, el fichero puede abrirse con* `OPEN` *en el programa principal o un subprograma y la sentencia de lectura* `READ` *en otro subprograma distinto.*

- *La asociación entre fichero y unidad rompe (se cierra el fichero) con la sentencia* `CLOSE`*. Por ejemplo:*

  ```
  CLOSE (3)       ! (3 es el número de la unidad de lectura-escritura)
  ```

  Los ficheros también se cierran automáticamente al terminar el programa.

- *El fichero puede existir previamente o no. Si no existe se crea al ejecutar una sentencia de escritura.*

- *Cuando se lee o escribe inmediatamente después de abrir un fichero, siempre se empieza por la primera línea del fichero. Posteriormente, cada sentencias* `READ` *o* `WRITE` *pasa una línea (o más, de acuerdo con el formato). En todo caso siempre se pasan líneas completas y la siguiente lectura-escritura a partir del primer carácter de la siguiente línea. Así:*

```
      WRITE (3, *) a
      WRITE (3, *) b
```
escribe `a` *y* `b` *en dos líneas. Ídem en lectura.*

- *En el sistema operativo UNIX (LINUX), se puede usar* `READ(*,*)` *y* `WRITE(*,*)` *y leer y escribir en ficheros con la sintaxis*
  `$ programa < datos > resultados`
  El ejecutable `programa` *lee del fichero* `datos` *y escribe en el fichero* `resultados`. *Con* `>` *se pierde lo que hubiera en* `resultados`. *Para que escriba al final de* `resultados` *conservando lo que hubiera se debe usar*
  `$ programa < datos >> resultados`
  Tanto `<` *como* `>` *son opcionales. Así son válidos también*
  `$ programa < datos`
  `$ programa >> resultados`

5.3. Formatos de escritura y lectura

```
      a = -21.0053
      n1= 2
      n2= -5
      WRITE(*,157) a, n1, n2
 157  FORMAT(F13.6,2X,2I7)
```
produce (salvo redondeo)
```
  -21.005300      2     -5
```

(F13.6) usa un total de 13 espacios para escribir `a` *(tipo REAL) con 6 decimales (añadiendo blancos a la izquierda si hace falta) incluyendo el signo. (*`2X`*) deja dos espacios en blanco a continuación y (*`2I7`*) usa 7 espacios para escribir* `n1` *(incluyendo signo) dejando blancos a la izquierda y otros 7 espacios para escribir* `n2`.

- *La sentencia* `FORMAT` *debe tener etiqueta (ej.* `157`*).* `FORMAT` *puede estar en cualquier lugar en la unidad de programación (pero después de* `PROGRAM, SUBROUTINE o FUNCTION` *y antes de* `END`*). No es ejecutable.*

- *En* `FORMAT`*, El carácter* `X` *indica dejar un espacio (uso:* `1X, 2X`*, etc, pero no* `X` *sin número delante). El carácter* `/` *pasa una línea (*`//` *pasa dos líneas, etc).*

- *Tipos de datos (los números son simplemente ejemplos):*

`I6`	*Datos tipo* `INTEGER`
`F13.6`	*Datos tipo* `REAL` *y* `REAL*8`
`E13.6` *ó* `D13.6`	*Datos tipo* `REAL` *y* `REAL*8`. *Escribe con exponente:* `-0.320E-04`
`G13.6`	*El compilador elige escribir como* `F13.6` *o como* `E13.6` *(ó* `D13.6`*).*
`L5`	*Datos tipo* `LOGICAL`. *En escritura produce* `T` *o* `F`, *En lectura acepta* `T`, `F`, `.TRUE.` *y* `.FALSE.`
`A`	*Datos tipo* `CHARACTER`
`A5`	*En lectura, lee los 5 últimos caracteres (es decir, los que están a la derecha) En escritura, escribe los 5 primeros caracteres.*

- *Los paréntesis dentro de formatos indican repetición de esa parte del formato:*
```
      WRITE(*,15) (I,a(I),b(I),I=1,10)
 15   FORMAT(I10,/,2(1X,E20.16))
```

- *Truco para escribir sin que cambie de línea:*
```
      DO k = 1, 100000
      WRITE(*,'(1a1,i25,1x,f23.5,$)') char(13), k, k-0.5
      END DO
```

6. Linux

6.1. UNIX

El sistema UNIX distingue entre mayúsculas y minúsculas.

Usando flechas (arriba y abajo) repite comandos dados previamente.

<TAB> completa comandos y nombres de ficheros cuando no hay ambigüedad.

Los nombres de ficheros pueden tener hasta 31 caracteres. Admite caracteres alfanuméricos y algunos especiales. Los ficheros pueden tener una extensión que indica de qué tipo son. Ej. `chufa.f` *hace que muchas aplicaciones presupongan que el fichero contiene la fuente de un programa FORTRAN.*

Los datos están organizados en directorios (carpetas) y dentro de ellos en ficheros. Los directorios pueden contener a su vez otros subdirectorios. Al entrar como usuario se entra en el directorio del usuario, "`/home/nombre_usuario`*". En cada momento el directorio actual es ".", el de arriba "..", el de más arriba "../..", etc. Todos los directorios cuelgan del directorio raíz "/".*

Por lo dicho antes "`../chufa`*" se refiere a un fichero llamado* `chufa` *en el directorio superior, mientras que "*`kk/kk1/chufa`*" o "*`./kk/kk1/chufa`*" se refiere a un fichero llamado* `chufa` *en el subdirectorio* `kk1` *que cuelga del sudirectorio* `kk` *que a su vez está en el directorio actual. Si el directorio actual era "*`/home/macfly`*", la dirección absoluta sería "*`/home/macfly/kk/kk1/chufa`*".*

6.2. Comandos útiles de Linux

IMPORTANTE: Linux no suele pedir confirmación. Presupone que lo que se le pide es lo que realmente se pretende hacer. Conviene guardar copias de seguridad.

`ls` *da la lista de ficheros en el directorio actual.*

`ls *chuf*` *da la lista de todos los ficheros cuyo nombre contenga la cadena* `chuf`*.*

`file chufa` *dice (intenta adivinar) de qué tipo es el fichero llamado* `chufa`*.*

`ps` *da la lista de procesos corriendo colgados de la sesión actual.*

`man` *nombre_comando da información sobre un comando (por ejemplo* `man ls`*).*

`cp chufa1 chufa2` *crea o modifica* `chufa2` *de modo que pasa a ser idéntico a* `chufa1`*. El fichero* `chufa1` *no cambia.*

`mv chufa1 chufa2` *como* `cp` *excepto que* `chufa1` *deja de existir.*

`cd nombre_subdirectorio` *cambia al subdirectorio indicado*

`cd ..` *cambia al directorio de encima.*

`cd` *cambia al directorio de inicio del usuario.*

`emacs chufa` *abre una ventana para editar el fichero llamado* `chufa`*.*

`gfortran nombre_fuente.f` *compila un programa FORTRAN contenido en el fichero* `nombre_fuente.f` *y pone el ejecutable en el fichero* `a.out`*. Éste se puede ejecutar con "*`./a.out`*" (*`./` *indica a la shell que busque el ejecutable en el directorio actual ".")*

`gfortran nombre_fuente.f -o nombre_ejecutable` *como antes pero pone el ejecutable en el fichero* `nombre_ejecutable`*. Éste se puede ejecutar con "*`./nombre_ejecutable`*". Por ejemplo*

`gfortran chufa.f -o chufa.x ; ./chufa.x`

compila e inmediatamente ejecuta el programa (si no ha habido errores graves de compilación). En general ";" encadena comandos.

`./chufa.x < fichero_entrada > fichero_salida` *ejecuta* `chufa.x` *leyendo de* `fichero_entrada` *y escribiendo en* `fichero_salida`*.*

Análogamente para un progama en C

```
gcc chufa.c -o chufa.x; ./chufa.x
```