Capítulo 2

Sentencias de control: bifurcaciones y bucles

Son parte fundamental de cualquier lenguaje. Sin ellas, las instrucciones de un programa sólo podrían ejecutarse en el orden en que están escritas. Las sentencias de control permiten modificar este orden. Hay dos categorías de sentencias de control:

Bifurcaciones o condicionales: permiten que se ejecuten conjuntos distintos de instrucciones, en función de que se verifique o no una determinada condición.

Bucles o repeticiones: permiten que se ejecute repetidamente un conjunto de instrucciones, bien un número pre-determinado de veces, o bien hasta que se verfique una determinada condición.

En términos de un lenguaje de programación, que se verifique una condición se traduce en que una expresión lógica tome el valor `.TRUE.`.

2.1 Expresiones lógicas

Imprescindicles para verificar **condiciones** son las **expresiones lógicas**, es decir, expresiones cuya evaluación produce un **valor lógico**.

Las más simples son aquéllas en las que se **comparan** dos datos.

2.1.1 Operadores relacionales o de comparación

Estos operadores permiten **comparar dos datos**, que pueden ser numéricos escalares o de caracteres. Siempre actúan entre dos operandos:

```
expresion-1  operador  expresion-2
```

Cada operador de comparación se puede escribir, en FORTRAN90, de dos formas. En la tabla siguiente se muestran ambas (los puntos que aparecen delante y detrás **forman parte** del operador):

Descripción	Símbolo 1	Símbolo 2
menor que	<	.LT.
mayor que	>	.GT.
igual que	==	.EQ.
menor o igual que	<=	.LE.
mayor o igual que	>=	.GE.
distinto de	/=	.NE.

Las comparaciones entre cadenas de caracteres se realizan caracter a caracter, comenzando por la izquierda. Así, `'azz'<'baa'` da como resultado `.TRUE.` En la ordenación de "menor" a "mayor" van en primer lugar los caracteres especiales, luego los dígitos decimales y por último los caracteres alfabéticos, en el orden habitual.

2.1.2 Operadores lógicos

Son los que actúan entre operandos de tipo lógico. Permiten construir expresiones que representen condiciones más complicadas, como que se verifiquen varias condiciones a la vez, que se verifique una entre varias condiciones, etc.

En la tabla siguiente se muestran los operadores lógicos de FORTRAN:

Descripción	Símbolo
Negación (NO)	.NOT.
Conjunción (Y)	.AND.
Disyunción (O)	.OR.
Equivalencia lógica	.EQV.
No equivalencia lógica	.NEQV.

El primero de ellos, `.NOT.`, actúa sobre un sólo operando lógico, dando como resultado el contrario de su valor: si la variable `var` toma el valor `.TRUE.`, entonces `.NOT.var` da como resultado `.FALSE.` y viceversa.

Los restantes operadores de la tabla actúan siempre entre dos operandos lógicos. La tabla siguiente muestra las correspondientes "tablas de verdad":

var1	var2	var1.AND.var2	var1.OR.var2	var1.EQV.var2	var1.NEQV.var2
.TRUE.	.TRUE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.

En una expresión pueden aparecer varios operadores lógicos. En ese caso, el orden en que se evalúan, es el siguiente:

1. `.NOT.`
2. `.AND.` y `.OR.`
3. `.EQV.` y `.NEQV.`
4. En caso de igual precedencia, se evalúan de izquierda a derecha.

2.1.3 Orden general de evaluación de las expresiones

En una expresión general pueden aparecer operadores de los tipos aritmético, de comparación y lógicos, así como funciones. El orden de evaluación general es el siguiente:

- Si en un expresión hay paréntesis, lo primero que se evalúa es su contenido. Si hay paréntesis anidados, se comienza por los más internos. Si hay varios grupos de paréntesis disjuntos, se comienza por el que esté más a la izquierda.
- En una expresión sin paréntesis de agrupamiento, el orden de evaluación es el siguiente:
  1. Las funciones. Si el argumento de la función es una expresión, se le aplican estas reglas. Si hay varias funciones, se comienza por la de la izquierda.
  2. Los operadores aritméticos, en el orden ya indicado.
  3. Los operadores de concatenación de caracteres, en orden de izquierda a derecha.
  4. Los operadores de comparación.
  5. Los operadores lógicos, en el orden antes mencionado.

2.2 Construcciones condicionales

Estas instrucciones o "bloques de instrucciones" (de ahí la denominación construcción) sirven para ejecutar distintos conjuntos de instrucciones, en función de que se verifiquen o no determinadas condiciones.

Los esquemas básicos, escritos en lenguaje corriente, son los siguientes:

Esquema condicional tipo 1 (IF):

- Si se verifica la condición `C`, ejecutar las instrucciones `I`.

Esquema condicional tipo 2 (IF - ELSE):

- Si se verifica la condición `C`, ejecutar las instrucciones $\texttt{I}_1$.
- Si no se verifica la condición `C`, ejecutar las instrucciones $\texttt{I}_2$.

Esquema condicional tipo 3 (bloque IF):

- Si se verifica la condición $\texttt{C}_1$, ejecutar las instrucciones $\texttt{I}_1$.
- Si no se verifica $\texttt{C}_1$, pero sí se verifica la condición $\texttt{C}_2$, ejecutar las instrucciones $\texttt{I}_2$.
- Si no se verifican $\texttt{C}_1$ ni $\texttt{C}_2$, pero sí se verifica la condición $\texttt{C}_3$, ejecutar las instrucciones $\texttt{I}_3$.
- ...
- Si no se verifica ninguna de las anteriores, ejecutar las instrucciones $\texttt{I}_k$

Esquema condicional tipo 4 (SELECT CASES):

- En el caso de que A valga a_1, ejecutar las instrucciones I_1.
- En el caso de que A valga a_2, ejecutar las instrucciones I_2.
- En el caso de que A valga a_3, ejecutar las instrucciones I_3.
- ...
- Si no se da ninguno de los casos anteriores, ejecutar las instrucciones I_k.

2.2.1 La sentencia IF más sencilla

Permite escribir un esquema condicional de tipo 1 en el que I se reduce a **una instrucción**. Su forma general es

```
IF ( expresion-logica ) instruccion
```

y su funcionamiento es el siguiente: si la expresion-logica toma el valor .TRUE. se ejecuta la instruccion. Si toma el valor .FALSE., no se ejecuta. En cualquiera de los casos la ejecución del programa continúa por la instrucción siguiente al IF.

```
print*, ' Escribe un valor real cualquiera: '
read*, x
func=x*x-4.*x
if (func < 0.) func=0.
print*, func
```

El trozo de programa anterior calcula el valor de la función f^+ (parte positiva de f) con $f(x) = x^2 - 4x$, en un punto x que se lee desde el teclado.

2.2.2 La construcción IF - THEN - END IF

Sirve para escribir un esquema condicional de tipo 1 en el que I se compone de más de una instrucción. Su forma general es

```
IF ( expresion-logica ) THEN
  ... instrucciones ...
END IF
```

y su funcionamiento es similar al de la sentencia IF, salvo porque las instrucciones a ejecutar pueden ser más de una. Tras finalizar, la ejecución del programa continuará por la instrucción siguiente al END IF.

```
print*, ' Escribe un valor real cualquiera: '
read*, x
func=x*x-4.*x
if (func < 0.) then
   func=0.
   print*, ' Valor de la función negativo'
end if
print*, func
```

Este ejemplo hace lo mismo que el anterior, pero, además, cuando cambia un valor negativo por cero imprime un mensaje.

2.2.3 La construcción IF - THEN - ELSE -END IF

Sirve para escribir un esquema condicional de tipo 2, es decir, en el que hay una alternativa: se hace una cosa o se hace otra. Su forma general es

```
IF ( expresion-logica ) THEN
  ... instrucciones-1 ...
ELSE
  ... instrucciones-2 ...
END IF
```

y su funcionamiento es el siguiente: si la `expresion-logica` toma el valor `.TRUE.` entonces se ejecuta el conjunto de `instrucciones-1` y si la `expresion-logica` toma el valor `.FALSE.`, se ejecuta el conjunto de `instrucciones-2`. En ambos casos la ejecución del programa continúa por la instrucción siguiente al `END IF`.

```
print*, ' Escribe un valor entero cualquiera: '
read*, num
if (mod(num,7)==0) then
   print*, ' El numero ',num,' es multiplo de 7'
else
   print*, ' El numero ',num,' no es multiplo de 7'
end if
```

Este ejemplo imprime un mensaje u otro, en función de que el número leído sea o no múltiplo de 7.

2.2.4 El bloque IF más general

Es la versión más elaborada de la " instrucción" `IF`, y permite escribir un esquema condicional de tipo 3. Su forma general es

```
IF ( expresion-logica ) THEN
  ... instrucciones-1 ...
ELSE IF ( expresion-logica-2 ) THEN
  ... instrucciones-2 ...
ELSE IF ( expresion-logica-3 ) THEN
  ... instrucciones-3 ...
...
ELSE
  ... instrucciones-k ...
END IF
```

y su funcionamiento es el siguiente:

1. Si `expresion-logica-1` toma el valor `.TRUE.`, entonces se ejecuta el conjunto de `instrucciones-1`.

2. Si `expresion-logica-1` toma el valor `.FALSE.`, y `expresion-logica-2` toma el valor `.TRUE.`, entonces se ejecuta el conjunto de `instrucciones-2`.

3. Si `expresion-logica-1` y `expresion-logica-2` toman ambas el valor `.FALSE.`, y `expresion-logica-3` toma el valor `.TRUE.`, entonces se ejecuta el conjunto de `instrucciones-3`.

4. Si ninguna de las `expresiones-logicas` anteriores toma el valor .TRUE., entonces se ejecuta el conjunto de `instrucciones-k`.

5. En cualquiera de los casos, la ejecución del programa continúa por la instrucción siguiente al `END IF`.

En un bloque `IF` puede haber un numero cualquiera de `ELSE IF (...) THEN` con sus correspondientes conjuntos de instrucciones. La opcion `ELSE` junto con su conjunto de instrucciones puede no existir.

2.2.5 Algunas observaciones sobre las construcciones condicionales

Las siguientes observaciones se aplican a cualquiera de las construcciones `IF` descritas anteriormente:

1. Obsérvese que en un bloque `IF`, se ejecuta, como máximo, uno de los conjuntos de instrucciones. Si está presente la opción `ELSE`, entonces siempre se ejecuta uno de los bloques. Si no está presente la opción `ELSE`, puede ocurrir que no se ejecute ninguno de los conjuntos de instrucciones.

2. La afirmación "la ejecución del programa continúa por la instrucción siguiente al `END IF`" es sólo una verdad a medias: puede ocurrir que, dentro del conjunto de instrucciones de uno de los casos del `IF` haya una instrucción de "ruptura incondicional de secuencia del programa", como por ejemplo una instrucción `STOP`, que provocará la inmediata finalización de la ejecución, o bien determinadas intrucciones (que se describirán más adelante), que envíen el flujo de ejecución a otra parte del programa. La afirmación debe entenderse, pués, en el sentido de que, si no existe este tipo de instrucción dentro de un bloque, una vez ejecutado el mismo se "saltan" todos los demás casos y se continúa por lo que sigue al bloque `IF`.

3. Las distintas construcciones `IF` pueden ser **anidadas**, es decir, puede incluirse un bloque `IF` dentro del conjunto de instrucciones de uno de los casos de otro bloque `IF`. Como es lógico, no puede haber solapamiento, es decir, una construcción `IF` debe estar completamente contenida dentro del conjunto de instrucciones de uno de los casos de otra.

4. El sangrado de las lineas de código que están dentro de uno de los casos del bloque no es necesaria, aunque puede facilitar la lectura del programa.

5. Se puede añadir una etiqueta identificativa a un bloque `IF`. Esto puede resultar útil en programas que contengan bloques `IF` muy largos o anidados. La forma de hacerlo es

```
nombre: IF ( expresion-logica ) THEN
           ... instrucciones-1 ...
        ELSE nombre
           ... instrucciones-2 ...
        END IF nombre
```

2.2.6 El ejemplo típico: resolución de una ecuación de segundo grado

El objetivo aquí es escribir un programa FORTRAN que lea los valores de los tres coeficientes reales de una ecuación de segundo grado y calcule la o las soluciones, especificando el caso de que se trata (dos raíces reales, una raíz real o dos raíces complejas conjugadas). Vamos a tratar de recorrer todo el camino, desde el planteamiento teórico del problema hasta una versión del programa final lo más optimizada posible.

Comenzamos por escribir, en lenguaje corriente, el algoritmo para resolver esta tarea. Esta etapa no debe ser obviada, ya que es fundamental para plantear correctamente el esquema lógico del programa. En este caso, el algoritmo podría ser:

Algoritmo

Leer los valores de los tres coeficientes a, b y c

Calcular el valor del discriminante $d = b^2 - 4ac$

Si $d > 0$ **entonces**

> **Calcular** las dos raíces, $x_1 = (-b + \sqrt{d})/2a$ y $x_2 = (-b - \sqrt{d})/2a$
> **Imprimir** "La ecuación tiene dos raices reales:"
> **Imprimir** los valores de x_1 y x_2

Si no, si $d < 0$ **entonces**

> **Calcular** las partes real e imaginaria de las raices complejas conjugadas, $x_r = -b/2a$ y $x_i = \sqrt{-d}/2a$.
> **Imprimir** "La ecuación tiene dos raices complejas conjugadas:"
> **Imprimir** las partes real e imaginaria x_r y x_i

Si no

> **Calcular** la raiz doble, $x = -b/2a$.
> **Imprimir** "La ecuación tiene una raiz doble:"
> **Imprimir** el valor de x

Fin del Si

Este sería, en esencia, el algoritmo que utilizaríamos para resolver el problema "a mano". Para resolverlo en el ordenador, hay que cambiarlo ligeramente. La razón es la forma en que se almacenan los números en el ordenador. Debido, en primer lugar, a la codificación en base

2 de los números y, en segundo lugar, a los errores de redondeo que se producen por tener que truncar la parte decimal para almacenarlos, es en la práctica imposible (estadísticamente hablando) que un número real calculado en el ordenador valga **exactamente** cero (o cualquier otro valor).

Por ello es importante la siguiente **observación de carácter general**:

en un programa no tiene sentido preguntar si un número real vale exactamente cero (o cualquier otro valor).

Lo que se debe hacer, en su lugar, es preguntar si el número en cuestión es "sumamente pequeño", es decir, menor en valor absoluto que una cantidad que se considere suficientemente pequeña.

Modificando conveniente el algoritmo anterior de acuerdo con esta observación, la escritura del programa es inmediata, utilizando un bloque `IF`.

Antes de presentarlo, merece la pena hacer los siguientes comentarios:

1. El primer paso en el algoritmo descrito es la lectura de los datos, en este caso de los coeficientes de la ecuación. Cuando se escribe un programa debe tenerse en cuenta que puede ser usado por una persona distinta del programador, o incluso por el mismo programador pasado un cierto tiempo, y en consecuencia se debe procurar, cuando se piden datos desde el programa, que quede bien claro qué datos se están pidiendo. Por ejemplo, unas personas escriben la ecuación de segundo grado como $ax^2+bx+c=0$ y otras pueden escribirla $cx^2+bx+a=0$. No será, pués, suficiente pedir "los coeficientes a, b y c".

2. Es también muy conveniente introducir en el código del programa algunas líneas de comentarios que ayuden a comprenderlo por otra persona o por el mismo programador pasado un tiempo. Un programa bien comentado debe contener unas cuantas líneas a continuación de la primera, en donde se explique, de manera clara pero concisa lo que hace el programa, así como el nombre del programador y eventualmente, la fecha. Intercaladas en el programa se deben incluir las líneas necesarias para identificar las etapas más importantes o para realizar aclaraciones. Esto debe hacerse también de forma concisa. No se trata de escribir una novela, sino de incluir unas escuetas claves para comprender el programa o refrescar la memoria. No tiene mucho sentido, en general, que un programa tenga más líneas de comentarios que de código.

Finalmente, el programa correspondiente al algoritmo anterior podría ser:

```
      program raices
!-----------------------------------------------------------------
! Programa para discutir y resolver una ecuación de segundo grado
! de coeficientes reales.
!-----------------------------------------------------------------
      real*4, parameter :: eps=1.e-7
      real*4  :: a,b,c,disc,x1,x2
      real*4  :: a2
!  Lectura de datos
      print*,"Escribe los valores de los tres coeficientes de la"
      print*,"ecuacion a*x^2 + b*x  + c = 0"
      print*," a, b, c ? ::"
      read*, a,b,c
      print*," Los valores leidos son:"
      print*," a= ",a
      print*," b= ",b
      print*," c= ",c
!  Calculos previos
      disc=b*b-4.*a*c
      a2=2.*a
!  Discusion
      if (disc > eps) then
!  Caso en que hay dos raices reales distintas
        disc=sqrt(disc)
        x1=(-b+disc)/a2
        x2=(-b-disc)/a2
        print*," La ecuacion tiene dos raices reales distintas"
        print*," x1 = ",x1
        print*," x2 = ",x2
      else if (disc < -eps) then
!  Caso en que hay dos raices complejas
        disc=sqrt(-disc)
        x1=-b/a2
        x2=disc/a2
        print*," La ecuación tiene dos raices complejas conjugadas"
        print*," Parte real =       ",x1
        print*," Parte imaginaria = ",x2
      else
!  Caso en que hay una sola raiz
        x1=-b/a2
        print*," La ecuacion tiene una raiz doble"
        print*," x = ",x1
      end if
!
      stop
      end
```

2.2.7 La construcción SELECT CASE

Esta construcción permite escribir un esquema condicional de tipo 4, en el que el flujo del programa se bifurca en varios caminos a partir de un mismo punto, en función del valor que tome una determinada expresión. Su forma general es:

```
SELECT CASE ( expresion-de-control )
CASE ( valores-1 )
  ... instrucciones-1 ...
CASE (valores-2 )
  ... instrucciones-2 ...
CASE ( valores-3 )
  ... instrucciones-3 ...
...
CASE DEFAULT
  ... instrucciones-k ...
END SELECT
```

en donde `expresion-de-control` es una expresion escalar numérica o de caracteres y `valores-i` puede ser un sólo valor, dado por una constante o expresion constante y también puede ser un rango de valores, en cuyo caso se escribe `valor-i1:valor-i2`. También se puede escribir `:valor-i`, indicándose con ello el caso en que la expresión que hace de índice toma cualquier valor **hasta** (i.e. menor o igual que) `valor-i`. Análogamente, se puede escribir `valor-i:` para indicar " mayor que". No puede haber casos que se solapen.

El funcionamiento de esta construcción es el siguiente: al "entrar" en ella se evalúa la `expresion-de-control`. Si el valor obtenido encaja en alguno de los **casos**, se ejecuta el correspondiente conjunto de instrucciones. Si no encaja en ninguno se ejecuta el conjunto de instrucciones del sub-bloque `CASE DEFAULT`, si existe. Si no existe se termina la ejecución del `SELECT CASE`. En todos los casos, la ejecución continúa por la instrucción siguiente al `END SELECT` (con las salvedades explicadas en **2.2.5**).

```
select case (num)
  case (:5)
    print*," num es menor que 5"
  case (6)
    print*, "num es igual a 6"
  case (7:10)
    print*, "num esta entre 7 y 10"
  case default
    print*, "num es mayor que 11"
 end select
```

2.4 Construcciones de repetición

Este tipo de construcciones permiten que se ejecute un conjunto determinado de instrucciones de forma repetida. En FORTRAN90 se pueden escribir los siguientes tipos de construcciones repetitivas:

Esquema repetitivo tipo 1 (DO):

- Repetir indefinidamente el conjunto de instrucciones `I`.

Esquema repetitivo tipo 2 (DO indexado):

- Para cada valor del índice `IND` en un rango fijado, ejecutar el conjunto de instrucciones `I`.

Esquema repetitivo tipo 3 (DO WHILE):

- Mientras que se verifique la condición `C`, ejecutar el conjunto de instrucciones `I`.

2.4.1 La construcción `DO` y la instrucción `EXIT`

Permite escribir un esquema de tipo 1, en el que un determinado conjunto de instrucciones, simplemente, se ejecuta indefinidamente. Su forma general es:

```
DO
... instrucciones ...
END DO
```

Lógicamente, en un programa que tenga sentido, el conjunto de instrucciones en cuestión deberá contener algún mecanismo que interrumpa en algún momento la repetición, bien deteniendo el programa (`STOP`), bien enviando el control de la ejecución a otro punto del mismo. Una forma de hacerlo es utilizar la instrucción

```
EXIT
```

que finaliza la ejecución del bucle y pasa el control a la instrucción siguiente al `END DO`.

```
program cota
integer :: i=1,ifact=1,limit
integer :: ifacto=1
print*," Escribe el valor de la cota"
read*,limit
do
  i=i+1
  ifacto=ifacto*i
  if (ifacto > limit) exit
  ifact=ifacto
end do
print*, " El número buscado es : ",i-1
print*, " El valor de su factorial es : ",ifact
end program cota
```

Este programa lee un número entero y calcula el número entero más grande cuyo factorial es menor que el primero.

2.4.2 El DO indexado

Permite escribir un esquema de tipo 2, en el que una variable, denominada índice del DO, va tomando una serie de valores en un rango prefijado, y para cada nuevo valor se ejecuta un determinado conjunto de instrucciones. Su forma general es:

```
DO indice=valor-inicial,valor-final,incremento
  ... instrucciones ...
END DO
```

Su funcionamiento es el siguiente:

1. Inicialmente, se hace `indice=valor-inicial`.
2. En el caso en que `incremento` es positivo, si `indice` es menor o igual que `valor-final`, se ejecuta el conjunto de `instrucciones`. Si no, se termina la ejecución del `DO`. En el caso en que `incremento` es negativo, si `indice` es mayor o igual que `valor-final`, se ejecuta el conjunto de `instrucciones`. Si no, se termina la ejecución del `DO`.
3. Se incrementa el valor del índice, es decir, se hace `indice=indice+incremento` y se repite la etapa anterior hasta que, en alguna iteración, se termina la ejecución del `DO`.
4. Cuando se termina la ejecución del `DO`, el programa continúa por la instrucción siguiente al `END DO`.

Algunas observaciones sobre esta construcción:

a) El índice del `DO`, `indice`, debe ser una variable entera. Su valor puede ser utilizado dentro del conjunto de instrucciones que forman el cuerpo del `DO`, pero no debe ser modificado.

b) `valor-inicial`, `valor-final` y `incremento` pueden ser constantes o variables enteras o expresiones que produzcan valores enteros. Sus valores son determinados al comienzo del `DO`, es decir, antes de que ninguna instrucción sea ejecutada. Cualquier modificación dentro del cuerpo del `DO` no tendrá ninguna consecuencia sobre el ámbito del `DO`.

c) Si, inicialmente, `valor-final` es mayor que `valor-inicial` (recíprocamente, menor que, si `incremento` es negativo), entonces el conjunto de `instrucciones` **no se ejecuta ninguna vez**, debido a que en la primera iteración ya se verifica la condición de parada.

d) Si el valor de `incremento` es `1`, entonces no es necesario indicarlo, es decir, se puede escribir, solamente:

```
DO indice=valor-inicial,valor-final
```

e) Cuando se termina la ejecución del `DO`, la variable-índice conserva el último valor que tomó (tras el último incremento), que puede no ser el último valor para el que se ejecutó el bucle, como puede observarse en el ejemplo siguiente.

```
      program suma
!-----------------------------------------------------
!  Calculo de la suma de los primeros numeros impares
!-----------------------------------------------------
      integer :: i,n,isuma
      print*, " Calcular suma 1+3+5+...+n "
      print*, " Escribe el valor de n :"
      read*, n
      isuma=0
      do i=1,n,2
        isuma=isuma+i
      enddo
      print*, " La suma 1+3+5+...+",i-2," vale: ",isuma
      end program suma
```

El DO indexado puede también escribirse de la forma siguiente

```
      DO iet indice=valor-inicial,valor-final,incremento
        ... instrucciones ...
iet   CONTINUE
```

donde iet es un **número de etiqueta**, es decir: un número del 1 al 99999 que se pone delante de determinadas instrucciones, en las columnas 1-5, para identificarlas. El programa anterior se escribiría, con este formato:

```
      program suma
      integer :: i,n,isuma
      print*, " Calcular suma 1+3+5+...+n "
      print*, " Escribe el valor de n :"
      read*, n
      isuma=0
      do 10 i=1,n,2
        isuma=isuma+i
10    continue
      print*, " La suma 1+3+5+...+",i-2," vale: ",isuma
      end program suma
```

2.4.3 El DO WHILE

Permite escribir un esquema de tipo 3, en el que un conjunto de instrucciones se ejecutará en tanto se verifique una determinada condición. Su forma general es:

```
DO WHILE ( expresion-logica )
  ... instrucciones ...
END DO
```

Su funcionamiento es el siguiente: al comenzar cada iteración, se evalúa la `expresion-logica` y, si toma el valor `.TRUE.`, se ejecuta el conjunto de instrucciones y se vuelve a iterar, pero si toma el valor `.FALSE.` se detiene la ejecución del `DO WHILE` y el programa se sigue ejecutando por la instrucción siguiente al `END DO`.

```
program tonto
logical :: log=.TRUE.
character*1 :: letra
do while (log)
  print*," Escribe una letra minuscula"
  read*, letra
  if(letra=="ñ") log=.FALSE.
enddo
print*
print*," ACERTASTE !!! "
end program tonto
```

2.4.4 Las instrucciones `CYCLE` y `EXIT`

Dentro de cualquier construcción `DO`, la instrucción `CYCLE` hace que se interrumpa la ejecución de la iteración actual y se comience la siguiente, si procede. Es decir, se detiene la ejecución actual del conjunto de instrucciones, **pero no se detiene** la ejecución del `DO`.

Por el contrario la instrucción `EXIT` hace que se detenga la ejecución del bucle de repetición al que está asociada.

2.4.5 Construcciones anidadas

Estas construcciones, lo mismo que los otros tipos de `DO`, pueden **anidarse**, es decir, se puede incluir un bloque `DO` (de cualquier tipo) dentro del conjunto de instrucciones de otro bloque.

Cuando se hace esto, puede ser útil la opción de poner a la construcción un `nombre`, como ya se vió, para el `IF`, en **2.2.5**. La forma de hacerlo aquí es:

```
nombre:  DO ...
  ...instrucciones...
END DO nombre
```

Las instrucciones `EXIT` y `CYCLE`, usadas en bucles anidados, se aplican con carácter general al bucle más interno. Pero pueden referirse a otro si se etiquetan:

```
primero: DO ...
  segundo: DO ...
    ... CYCLE primero
    tercero:     DO ...
      ...EXIT segundo
    END DO tercero
  END DO segundo
END DO primero
```

2.4.6 Una instrucción "casi" obsoleta: GO TO

Utilizando números de etiqueta, se puede, en FORTRAN, implementar otros tipos de ruptura de secuencia en las instrucciones, mediante la instrucción

```
GO TO num-etiq
```

Esta instrucción hace que la ejecución del programa continúe por la instrucción, dentro del mismo programa o subprograma, que tenga la etiqueta `num-etiq`.

```
      program repite
      integer :: num
11    print*,">>> Escribe un numero mayor que 3 : "
      read*,num
      if (num<4) go to 11
      print*,">>> El numero escrito es:",num
      end program repite
```

Esta instrucción está desaconsejada si se desea realizar una programación estructurada. Su uso hace, en general, que los programas sean más complicados y difíciles de corregir.