Capítulo 1

Estructuras de Control - IF()

1.1. Formas de realizar una pregunta: IF()

La sentencia **IF()** permite hacer una pregunta y tomar decisiones a partir de la respuesta obtenida a dicha interrogación. Esta posibilidad es extremadamente útil cuando se plantean cálculos a resolver dónde los resultados de ciertas operaciones indican y permiten la elección de las fórmulas y algoritmos matemáticos para la solución del problema. Por lo cual puedo hacer una pregunta sobre ciertas variables y a partir de esa respuesta elegir el camino que sigue el programa. Por ejemplo, en resolución de una ecuación de segundo grado si el discriminante ($B^2 - 4AC$) es menor que cero, la solución está en el campo de los números complejos y no en la de los reales, por lo cual se deben tomar previsiones para este caso. Otro ejemplo, mucho más complicado podría darse si se están calculando modelos de la estructura de una estrella, para cierta capa a una profundidad dada podría haber sólo transferencia de energía por radiación o por convección según la física local (presión, temperatura, etc). Con lo cual el programa podría evaluar esas variables y decidir la física a utilizar.

Hay 3 formas distintas de usar esta orden. La primera de ellas no la usaremos porque ya es obsoleta (conocida como forma aritmética), nos concentraremos en las dos formas modernas de usarla: Esto es: el **IF()** sentencia y en el **IF()** en bloque (block if en inglés).

1.1.1. IF() Sentencia

El **IF()** sentencia funciona haciendo una pregunta a una expresión o variable, cuyo resultado es una respuesta Booleana, es decir estamos en el caso que la respuesta a la interrogación es un **Verdadero** o un **Falso.** Se hace la pregunta y si la respuesta es verdadera se ejecuta la sentencia que está a la derecha en el mismo renglón que la pregunta. Si es falsa esa orden escrita a la derecha no se ejecuta y se continua con el programa. La sentencia tiene esta forma:

IF(algo) sentencia

donde “algo” en su forma más general es una expresión booleana cuya resolución nos da un resultado verdadero o falso. También podría ser una variable Lógica sola cuyo valor asignado en el programa sea un verdadero o un falso.

Si la pregunta tiene que ver con comparar números, se deberán usar los **Operadores de Relación**.

Operadores de Relación

Estos son: $>$, $\geq$, =, $\neq$, $<$, $\leq$ pero como esos símbolos matemáticos no existían en los teclados antiguos, Fortran y otros lenguajes tienen su propia manera de escribirlos, ver tabla 1.1.

Operador	Escritura en Fortran
$>$	.GT.
$\geq$	.GE.
=	.EQ.
$\neq$	.NE.
$<$	.LT.
$\leq$	.LE.

Tabla 1.1. Forma de escribir en Fortran los operadores relacionales. Note que se escriben con puntos al comienzo y al final del nombre. Esta notición con puntos permite al compilador no confundirlos con una variable que podría tener esos nombres.

Ejemplo del uso de IF() com estos operadores:

IF(A.GT.10) A = B2 + C**2**

en este caso estoy indicando que si A $>$ 10, el valor de A cambiará por el cálculo de A = B^2 + C^2 de lo contrario seguirá con el valor que ya tenía asignado.

Es importante volver a notar que la repuesta a la pregunta del **IF()** es Booleana, por lo cual nada evita que la haga de esta manera:

Logical L ! Ahora L es una variable lógica. Esta sentencia tiene que estar al principio del programa, en la zona donde defino variables.
L= A.GT.10 ! Comparo A con 10 y si es cierto guardo en L un verdadero y si no los es un Falso
IF(L) A = B2 + C**2** ! Pregunto que sobre L, ya que ahí se guardó el resultado del cálculo de la expresión anterior que tiene como resultado lógico un verdadero o un falso.

Operadores Lógicos

Para tener una mayor cantidad de opciones existen los Operadores Lógicos. Estos sirven para realizar varias preguntas simultáneas. Ya que permiten comparar distintos resultados lógicos con reglas que impongo a través de estos operadores. A estos operadores los conocen de la asignatura Álgebra I. Recordemos los de uso más común, ver tabla 1.2

Prioridades de las operaciones

El resultado Booleano que uno espera en una expresión lógica, como hemos visto, puede ser la combinación de una cantidad de operaciones matemáticas, de relación y por último lógicas. Por lo cual se puede hacer tabla con el orden de prioridades de cómo estas operaciones son resueltas. Veamos la tabla 1.3:

Operador	Fortran	Operación
Negación	.NOT.	Cambia el valor de la expresión lógica a su opuesto
Conjunción	.AND.	Cierto únicamente si ambas expresiones lógicas son ciertas
Disyunción Inclusiva	.OR.	Cierto si una de las expresiones es cierta
Disyunción exclusiva	.XOR.	Cierto únicamente si una de las expresiones es cierta
Equivalente	.EQV.	Cierto si ambas expresiones tienen el mismo valor
No equivalente	.NEQV.	Cierto si ambas expresiones no tienen el mismo valor

Tabla 1.2. Forma de escribir en Fortran los operadores relacionales. Note que se escriben con puntos al comienzo y al final del nombre, al igual que vimos en el caso de los operadores de relación.

Tipo de Operación	Operador	Asociatividad	Prioridad
Aritmética	** (potencia)	Derecha a izquierda	1
	*, /	Izquierda a derecha	2
	+,-	Izquierda a derecha	3
Relacionales	.GT., .GE., .EQ., .NE., .LT., .LE.	No tienen	4
Lógicos	.NOT.	Derecha a izquierda	5
	.AND.	Izquierda a derecha	6
	.OR.	Izquierda a derecha	7
	.EQV., .NEQV.	Izquierda a derecha	8

Tabla 1.3. Prioridades de los operadores en la forma más general de una expresión aritmética/boolena posible. Estas prioridades pueden sobre escribirse poniendo los paréntesis adecuados

1.1.2. Sentencia GOTO

El IF sentencia tiene una limitación muy importante y es que si la condición se cumple sólo puede ejecutar una sola orden y no más. Históricamente forma de programar se la ha combinado con una sentencia llamada **GOTO** y suele considerarse como una muy manera poco feliz de realizar programas, ya que los códigos escritos de esta forma son muy confusos y difíciles de modificar. La orden GOTO funciona de la siguiente manera:

```
      GOTO 99
      ⋮ se saltean estas líneas
99    sigue el programa...
```

Cuando la ejecución de un programa encuentra la orden **GOTO** en vez de seguir con la sentencia que está debajo salta a la que tiene la etiqueta (99). Esta etiqueta puede ser una sentencia posterior o incluso anterior a la que está el **GOTO**.

Esta orden tiene en si una utilidad, por ejemplo, para romper un ciclo **DO** de la siguiente manera: si en un cálculo tengo programados un bucle DO muy largo y descubro que no necesito terminarlo, porque el resultado ya esta calculado (ejemplo: una serie convergió muy rápido) puedo preguntar si esto pasó y escapar del **DO** antes que termine y ahorrar un montón de tiempo de computación. Así:

```
      DO i=1,100000
```

```
      ⋮
      IF (algo) GOTO 20
      ⋮
      ENDDO
20    CONTINUE
  sigue el programa...
```

Ejemplo: cálculo la serie de Taylor de la función cos(x), con un error menor a error $< 10^{-6}$

$$Cos(x) = \sum_{i=1}^{\infty} \frac{(-1)^i x^{2i}}{(2i)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ... + \left|error(\xi)\right|$$

Veamos el programa:

El programa es simple, vamos calculando los términos de la serie y sumándolos, pero al mismo tiempo calculamos el término del error. Evaluando ese término sabemos si llegamos al valor del error que requerimos. Si no llegamos a satisfacer la condición de tener un error más pequeño del que establecimos como satisfactorio, calculamos un término más a la serie y continuamos el proceso. Pero si obtenemos un error que nos determina que ya calculamos un valor útil para nuestras necesidades, el IF() nos permite terminar el programa y no hacer más cálculos innecesarios consumiendo recursos de la computadora.

```
      Program cos_de_angulo [1,2]

      write(*,*) 'ingrese el valor del ángulo n'
      read(*,*) omega
      pi=3.14159265358979
      omega=omega/180*pi
      xmax=pi/2

      coseno=0
      do i=0,1000000

C     Calculo el factorial para el término y para del error: n! y (n+2)!
      facto=1.
      do ii=1,2*i
       facto=facto*ii
      enddo
      facto2=facto*(i+1)*(i+2)
```

[1] Atención: Este programa es muy **ineficiente**, ya que se repiten cálculos innecesariamente. Puede ser mejorado muy fácilmente, pero se volverá muy confuso y dejaría de tener utilidad con el fin de que sea un ejemplo. Se deja como ejercicio al lector el intentar mejorar su eficiencia, modificándolo para que realice la menor cantidad de cálculos posibles e igual arribe al resultado correcto.

[2] Note que hemos asumido que $0^0 = 1$ cuando esta operación da como resultado un valor indeterminado. Muchos lenguajes de computación (FORTRAN, Python, etc) consideran que el resultado es $0^0 = 1$ por razones de simetría. Aunque este criterio puede ser discutible y no está generalizado a todos los lenguajes.

```
C       Calculo el termino i
        term=omega**(2*i)*(-1)**(i)/facto

C       Calculo el término y el error
        coseno=coseno+term
        eterm= xmax**(2*i+2)/facto2

C       Pregunto si llegué al error deseado
        if(eterm.LT.1E-8) goto 99
       enddo

99       write(*,*) i, coseno, cos(omega)

       end
```

1.1.3. Bloque IF() (Block IF)

El **IF** utilizado como bloque permite subsanar la deficiencia que tiene el IF sentencia de que sólo permite la ejecución de una sola orden, con este comando es posible realizar una serie muy larga de órdenes . Funciona de esta manera:

```
IF (expresión lógica) THEN
sentencia 1
sentencia 2
...
ELSEIF(expresión lógica 2) THEN
sentencia
sentencia
...
ELSEIF(expresión lógica 3) THEN
sentencia
sentencia
....
ELSE
sentencia
sentencia
...
ENDIF
```

¿Cómo funciona esta estructura de órdenes?
La primera sentencia sólo se diferencia del caso anterior porque a su derecha está la palabra **THEN** que le indicaría al compilador que estamos en el caso del **IF()** en bloque y no en el caso anterior del IF() sentencia. Esta orden se podría traducir al lenguaje coloquial como: *Si pasa esto entonces hacemos todo esto que sigue y pero si no pasó esto pero pasó esto otro hacemos todas estas otras cosas.* Y en cada caso lo que se hace no es ejecutar una sola orden sino todo un conjunto de órdenes sin límite de cantidad. Pueden escribirse todas las preguntas diferentes que se necesiten

para resolver el problema. Veámoslo con más detalle:

Viendo el esquema, si la primera expresión lógica es verdadera se cumplen la sentencia 1, luego la 2 y así hasta que estas se acaban. Si la pregunta dio como resultado un verdadero, todas las órdenes que siguen se saltean y luego el programa continua su ejecución después del **ENDIF**.

Si la expresión lógica es falsa. Se continua con las siguientes preguntas en orden. Si la expresión lógica de primer **ELSEIF** es cierta se cumplen con las sentencias asociadas a este **ELSEIF()** y luego el programa continua con las sentencias que siguen al **ENDIF**. Si la expresión lógica no es cierta, se pregunta si es cierta la del segundo **ELSEIF()** y así sucesivamente con todos los **ELSEIF()** que hayan. Si ninguna de las expresiones del **IF()** o de los **ELSEIF()** son verdaderas, y sólo si en ese caso, se activan las sentencias del **ELSE** final. Este actúa en la forma de que solo se cumplen las sentencias debajo de él en el caso de que ninguna pregunta que se haya hecho haya sido verdadera.

Salvo la orden inicial **IF() THEN** y la última **ENDIF** todas las demás sentencias internas pueden o no estar. Nada impide tener la cantidad de sentencias **ELSEIF() THEN** que se requieran para la tarea, o ninguna. Y en el caso del **ELSE** si lo lógica de lo que está programando no lo necesita esta orden no se la utiliza y por consiguiente no se la escribe. Básicamente el **IF** en bloque es una estructura de módulos de los cuales se eligen los que se requieran para la resolver la tarea.

Ejemplos:

- Función a trazos
  Supongamos que en alguna parte de un programa, deberíamos calcular la siguiente función a trazos:

$$f(x) = \begin{cases} 0 & \text{si } x \leq 30 \\ x - 30 & \text{si } 30 < x < 60 \\ 30 & \text{si } 60 \leq x \end{cases} \tag{1.1}$$

  La parte del programa que hace este trabajo tendría la siguiente forma:

```
⋮
IF (X.LE.30) THEN
      F = 0

 ELSEIF(X.GT.30.AND.X.LT.60) THEN
      F = X - 30

 ELSE
      F = 30

 ENDIF
⋮
```

Es decir, primero se verifica que X $\leq$ 30. Si la pregunta es cierta se ejecuta la orden F = 0 y se termina la sentencia **IF**. Si no lo es, se pregunta si 30 $<$ x $<$ 60 y si es veraz, corre F = X - 30 y se termina el **IF**. Si tampoco es verdadera se ejecuta si o si lo que sigue a la sentencia **ELSE** que es F = 30 y se continua con el programa.

- Solución de una ecuación de segundo grado

Resolver una ecuación cuadrática tiene interés porque según sus coeficientes, los resultados son diversos. Las soluciones pueden ser dos números reales, el mismo número repetido dos veces, o dos números complejos. Por lo cual al hacer un programa hay que tomar cuenta estos 3 casos por separado y por lo tanto utilizar la sentencia **IF()** para determinar en cual de estas distintas situaciones se encuentra nuestra solución.
Por lo cual, si mi ecuación es $AX^2 + BX + C = 0$ tengo que ver si $B^2 - 4 * A * C$ es mayor que cero (dos raíces reales), igual a cero, o sea dos raíces iguales, o menor que cero por lo cual las raíces estarán el campo de los números complejos.

El programa seria:

```
program cuad
complex z1,z2
read(*,*) a,b,c
disc=b*b-4*a*c
if(disc.lt.0) then
   z1=(-b+sqrt(complex(disc,0)))/(2*a)
   z2=(-b-sqrt(complex(disc,0)))/(2*a)
   write(*,*)'raices complejas:',z1,z2
elseif(disc.eq.0) then
   x=-b/(2*a)
   write(*,*)' el disc. es cero, x=',x
else
   x1=(-b+sqrt(disc))/(2*a)
   x2=(-b-sqrt(disc))/(2*a)
   write(*,*) x1,x2
endif
end
```

Veremos parte por parte que hace este programa y sobre todo la función If que en este caso decidirá la forma de resolver la ecuación:

```
program cuad
complex z1,z2
```

Comienza el programa y por las dudas defino dos variables complejas que usaré llegado el caso de tener raíces complejas, en el caso de tenerlas no las utilizaré.

read(*,*) a,b,c

disc=b*b-4*a*c

Leo los coeficientes de las ecuación y calculo el discriminante, el cual determinará el tipo de solución

if(disc.lt.0) then

z1=(-b+sqrt(complex(disc,0)))/(2*a)

z2=(-b-sqrt(complex(disc,0)))/(2*a)

write(*,*)'Raices complejas:',z1,z2

este punto del programa, se pregunta: ¿Es el discriminante menor que cero? Si es así cálculo las dos raíces complejas. Como la solución tiene una parte imaginaria que proviene de tomar la raíz cuadrada de un número negativo, debo primero construir ese número como complejo, así la operación se realizará en el campo de los números imaginarios. Al poner la orden **complex(disc,0)**[3] estoy convirtiendo la variable disc en un número complejo con parte real (disc) y parte imaginaria 0, Al hacer esto cualquier operación matemática de aquí en adelante con este número será en el campo complejo. Estas operaciones se harán conservando las reglas del álgebra para números complejos.
De los cálculos z1 y z2 serán las soluciones complejas de la ecuación, la cuales escribo en la pantalla.
Pero, si el discriminante no es negativo, esta parte del IF no se cumple y se continua con la siguiente pregunta:

elseif(disc.eq.0) then

x=-b/(2*a)

write(*,*)' el disc. es cero, x=',x

este caso, si el discriminante es cero, se calcula la solución (en este caso es trivial). Luego se imprime un aviso sobre que caso fue y la solución. Pero si esta pregunta tampoco es cierta si o si estamos en el última caso y se calcula como la solución para raíces reales:

x1=(-b+sqrt(disc))/(2*a)

x2=(-b-sqrt(disc))/(2*a)

Y luego se imprime el resultado:

write(*,*) x1,x2

[3] Recordar que en Fortran los números complejos se escriben como un par ordenado, con la siguiente estructura: (parte real, parte imaginaria)