

1. Prefacio

La meta de este tutorial de Fortran es dar una rápida introducción a las características más comunes del lenguaje de programación Fortran 90. No se pretende que sea una referencia completa, muchos detalles han sido omitidos, la presentación del material se enfoca al cómputo científico. El tutorial fue inspirado en el libro *Fortran 90/95 for Scientist and Engineers*. Se han tomado algunos ejemplos del tutorial hecho por la Facultad de Ciencias.

2. ¿Qué es Fortran?

Fortran es lenguaje de propósito general, principalmente orientado a la computación matemática, por ejemplo en ingeniería. Fortran es un acrónimo de FORMula TRANslator, y originalmente fue escrito con mayúsculas como FORTRAN. Sin embargo la tendencia es poner sólo la primera letra con mayúscula, por lo que se escribe actualmente como Fortran. Fortran fue el primer lenguaje de programación de alto nivel. El desarrollo de Fortran inicio en la decada de 1950 en IBM y han habido muchas versiones desde entonces. Por convención, una versión de Fortran es acompañada con los últimos dos dígitos del año en que se propuso la estandarización. Por lo que se tiene:

- Fortran 66
- Fortran 77
- Fortran 90 (95)

La versión más común de Fortran actualmente es todavía Fortran 77, sin embargo Fortran 90 esta creciendo en popularidad. Fortran 95 es una versión revisada de Fortran 90 la cual fue aprobada por ANSI en 1996. Hay también varias versiones de Fortran para computadoras paralelas. La más importante es HPF (High Performance Fortran), la cual es de hecho el estándar.

Los usuarios deben ser cuidadosos con los compiladores de Fortran 77, ya que pueden manejar un superconjunto de Fortran 77, por ejemplo contienen extensiones no estandarizadas. En este tutorial se hará énfasis al estándar ANSI Fortran 77.

Razones para aprender Fortran

Fortran es un lenguaje de programación dominante usado en aplicaciones de ingeniería y matemáticas, por lo que es importante que se tenga bases para poder leer y modificar un código de Fortran. Algunas opiniones de expertos han dicho que Fortran será un lenguaje que pronto decaerá en popularidad y se extinguirá, lo que no ha sucedido todavía. Una de las razones para esta sobrevivencia es *la inercia del software*, ya que una vez que una compañía ha gastado muchos millones de dólares y de años en el desarrollo de software, no le es conveniente traducir el software a un lenguaje diferente, por el costo que implica y por ser una tarea difícil y laboriosa.

Portabilidad

Una ventaja principal de Fortran es que ha sido estandarizado por ANSI y ISO, por lo que un programa escrito en ANSI Fortran 90/95 se podrá ejecutar en cualquier computadora que tenga un compilador de Fortran 90/95

3. Introducción a Fortran 90

Un programa de Fortran es una secuencia de líneas de texto. El texto debe de seguir una determinada *sintaxis* para ser un programa válido de Fortran. Se analiza el siguiente ejemplo:

```
program circulo
  implicit none
  real  :: r, area

; Este programa lee un número real r y muestra
; el área del círculo con radio r.

  write (*,*) 'Escribe el radio r:'
  read  (*,*) r
  area = 3.14159*r*r
  write (*,*) 'Area = ', area

end program circulo
```

Las líneas que comienzan con el caracter "!" son *comentarios* y no tienen otro propósito más que hacer los programas más legibles. Originalmente todos los programas de Fortran tenían que ser escritos solamente con letras mayúsculas, actualmente se pueden escribir con minúsculas con lo cual se mejora la legibilidad, por lo que se hará de esa forma.

Organización del Programa

Un programa de Fortran por lo general consiste de un programa principal o *main* (o manejador) y posiblemente varios subprogramas (o procedimientos o subrutinas). Por el momento se considerara que todas las sentencias están en el programa principal; los subprogramas se revisarán más adelante. La estructura del programa principal es:

```
program name

implicit none

Real      ::
Integer   ::
Logical   ::
Carácter  :: } declarations

Read (*,*)
Write(*,*) } Instrucciones ejecutables

... Comandos ejecutables...
STOP
END
```

La sentencia `stop` es opcional y podría ser vista como redundante ya que el programa terminará cuando alcance el fin, pero se recomienda que el programa termine con la sentencia `stop` para resaltar que la ejecución del programa ahí termina.

Reglas para la posición en columnas

Para Fortran 90 se permite el libre formato.

Comentarios

Una línea que inicia con "!", "c" o un asterisco en la primera columna es un comentario. El comentario puede aparecer en cualquier lugar del programa. El colocarlos en el lugar preciso incrementan la legibilidad. Los códigos comerciales de Fortran contienen un 50% de comentarios. El signo de exclamación puede aparecer en cualquier parte.

Continuación

Ocasionalmente una sentencia no cabe en una sola línea. Se puede dividir la sentencia en dos o más líneas, y usar la marca de continuación "&" al final de la línea.

Ejemplo:

```
program circulo
! ejemplo de continuacion de linea
.
.
.
area = 3.14159      &
      *r*r
.
.
end program circulo
```

Nombre de Variables

Los nombres de variables en Fortran 90 consisten de caracteres elegidos de la a a la z y de los dígitos del 0 al 9. El primer carácter debe ser una letra. (Nota: en Fortran 90 se permiten nombres de longitud arbitraria).

Tipos y declaraciones

Cada variable *debería* ser definida con una *declaración*. Esto indica el *tipo* de la variable. F90 maneja principalmente los siguientes tipos de variables :

```
integer  lista de variables
real     lista de variables
complex  lista de variables
logical  lista de variables
character lista de variables
```

La lista de variables consiste de nombres de variables separadas por comas. Cada variable deberá ser declarada exactamente una vez. Si una variable no está declarada, Fortran 90 arrojará un error en la compilación.

Variables- Integer

Una variable Integer consiste de un número entero positivo, entero negativo y cero. La cantidad de memoria usada para almacenar una variable integer es 4-byte integer. Por lo tanto, el valor más pequeño y más grande serán: $-2,147,384,648 (-2^{n-1})$ y $2,147,483,647(2^{n-1}-1)$ donde n es el número de bits. Un byte contiene 8 bits.

Variables- Real (punto flotante)

Una variable real almacena números en notación científica. Por ejemplo, la velocidad del sonido en el vacío es 299,800,000 m/s. El número es más fácil escribirlo como 2.998×10^8 m/s. Las dos partes de un número en notación científica son: la **mantisa** y el **exponente**. La mantisa del número anterior es 2.998 y el exponente es 8 (base decimal).

Los números reales en una computadora son manejados similarmente excepto que se usa la **base binaria (0,1)** en lugar de la base decimal. La mantisa contiene un número entre -1.0 y 1.0 y el exponente contiene la potencia requerida para escalar el número a su valor real.

Números reales generalmente ocupan 32 bits (4bytes) de la memoria de la computadora dividido en dos componentes: 24 bits para la mantisa y 8 bits para el exponente.

La forma más simple de una expresión es una *constante*. Hay seis tipos de constantes, que corresponden con los tipos de datos que maneja Fortran. Se muestran a continuación

Ejemplos de constantes **integer**:

1
0
-100
32767
+15

Ejemplos de constantes de tipo **real**:

1.0
-0.25
2.0E6
3.333E-1

La notación *E* se usa para números que se encuentran escritos en *notación científica*. Por lo que 2.0E6 es dos millones, mientras que 3.333E-1 es aproximadamente la tercera parte de uno

Para constantes que rebasen la capacidad de almacenamiento de un real se puede usar doble precisión. En este caso, la notación usada es el símbolo "D" en vez de "E". Por ejemplo:

2.0D-1
1D99

Por lo que *2.0D-1* es la quinta parte de uno, almacenada en un tipo doble precisión y *1D99* es un uno seguido por 99 ceros.

El siguiente tipo son constantes **complejas**. Los cuales son indicados por un par de constantes (enteras o reales), separadas por una coma y encerrados entre paréntesis. Por ejemplo:

(2, -3)
(1., 9.9E-1)

El primer número denota la parte real y el segundo la parte imaginaria.

El quinto tipo, son las constantes **lógicas**. Estas sólo pueden tener uno de dos posibles valores, que son:

.TRUE.
.FALSE.

Observar que se requiere que se encierren con punto las palabras.

El último tipo son las constantes de **caracteres**. Estas son por lo general usadas como un *arreglo* de caracteres, llamado *cadena*. Estas consisten de una secuencia arbitraria de caracteres encerradas con apóstrofes (comillas sencillas), por ejemplo:

```
'ABC'  
'¡Cualquier cosa!'  
'Es un magnífico día'
```

Las constantes de cadenas son **sensibles** a mayúsculas y minúsculas. Se presenta un problema cuando se quiere poner un apóstrofe dentro de la cadena. En este caso la cadena debe ser encerrada entre comillas dobles, por ejemplo:

```
"Hay muchos CD's piratas"
```

Expresiones

Las expresiones más sencillas son de la forma

$\boxed{\text{operando}}$ $\boxed{\text{operador}}$ $\boxed{\text{operando}}$

y un ejemplo es

$x + y$

El resultado de una expresión es por si misma otro operando, por lo que se puede hacer anidamiento como lo siguiente:

$x + 2 * y$

Con lo anterior se presenta la pregunta de la precedencia, ¿la última expresión significa $x + (2*y)$ o $(x+2)*y$? La precedencia de los operadores aritméticos en Fortran 9' es (de la más alta a la más baja):

```
**    {exponenciación}  
*    /    {multiplicación, división}  
+    -    {suma, resta}
```

Todos los operadores son evaluados de izquierda a derecha, excepto el operador de exponenciación, el cual tiene precedencia de derecha a izquierda. Si se desea cambiar el orden de evaluación, se pueden usar paréntesis.

Todos los operadores anteriores son binarios. Se tienen también operadores unarios, uno de ellos es el de negación - y que tiene una precedencia mayor que los anteriores. Por lo que la expresión $-x+y$ significa lo que se esperaría.

Se debe tener cuidado al usar el operador de división, el cual devolverá distintos valores dependiendo del tipo de datos que se estén usando. Si ambos operandos son enteros, se hace una división entera, de otra forma se hace una división que devuelve un tipo real.

Por ejemplo, $3/2$ es igual a 1 y $3./2.$ es igual a 1.5 .

Asignación

Una asignación tiene la forma

```
nombre_de_variable = expresión
```

La interpretación es como sigue: se evalúa el lado derecho y se asigna el valor del resultado a la variable de la izquierda. La expresión de la derecha puede contener otras variables, pero estas nunca cambiarán de valor. Por ejemplo:

```
area = pi * r**2
```

no cambia el valor de `pi`, ni de `r`, solamente de `area`.

Conversión de Tipos

Cuando diferentes tipos de datos ocurren en una expresión se lleva a cabo una *conversión de tipos* ya sea explícita o implícitamente. Fortran realiza algo de conversión implícita.

Por ejemplo

```
real x
x = x + 1
```

el uno será convertido al tipo real, y se incrementa la variable `x` en uno. Sin embargo, en expresiones más complicadas, es una buena práctica de programación forzar explícitamente la conversión de tipos. Para números se tienen las siguientes funciones disponibles:

```
int
real
dble
ichar
char
```

Las primeras tres no requieren explicación. Con `ichar` se toma un carácter y se convierte a un entero y con `char` se hace lo contrario.

Ejemplo: ¿Cómo multiplicar dos variables tipo real `x` e `y` usando doble precisión y guardando el resultado en una variable de doble precisión `w`?

```
w = dbble(x)*dbble(y)
```

Observar que es diferente de:

w = dble(x*y)

Ejercicios

Exercicio A

Calcular el valor de las siguientes expresiones de Fortran a mano:

$$2+1-10/3/4$$

$$2**3/3*2-5$$

$$-(3*4-2)**(3-2**1+1)/-2$$

7. Expresiones Lógicas

Una expresión lógica puede tener solamente el valor de `.TRUE.` o de `.FALSE.` Una valor lógico puede ser obtenido al comparar expresiones aritméticas usando los siguientes *operadores relacionales*:

<code>.LT.</code>	<code>meaning < significando</code>	Menor que
<code>.LE.</code>	<code><=</code>	Menor o igual que
<code>.GT.</code>	<code>></code>	Mayor que
<code>.GE.</code>	<code>>=</code>	Mayor o igual que
<code>.EQ.</code>	<code>==</code>	Igual
<code>.NE.</code>	<code>/=</code>	Diferente

Las expresiones lógicas pueden ser combinadas con los *operadores lógicos* `.AND.` `.OR.` `.NOT.` que corresponden a los operadores lógicos conocidos Y, O y negación respectivamente.

Asignación de Variables Lógicas

Los valores booleanos pueden ser guardados en *variables lógicas*. La asignación es de forma análoga a la asignación aritmética. Ejemplo:

```
Logical    :: a, b
a = .TRUE.
```

Las expresiones lógicas son usadas frecuentemente en sentencias condicionales como `if`.

Ejercicio

Exercicio A

Calcular el valor de las siguientes expresiones lógicas:

`(2.LT.2)` `(5 .EQ. 11/2)`

`(2 < 2)` `(5 == 11/2)`

8. La sentencia `if`

Una parte importante de cualquier lenguaje de programación son las *sentencias condicionales*. La sentencia más común en Fortran es `if`, la cual tiene varias formas de uso. La forma más simple de la sentencia `if` es:

```
if (expresión lógica) sentencia
```

Lo anterior tiene que ser escrito en una sola línea. El siguiente ejemplo obtiene el valor absoluto de `x`:

```
if (x < 0) x = -x
```

Si más de una sentencia necesita ser ejecutada dentro de la sentencia `if`, entonces la siguiente sintaxis deberá ser usada:

```
if (expresión lógica) then  
    sentencias  
endif
```

La forma más general más general de la sentencia `if` tiene la siguiente forma:

```
if (expresión lógica) then  
    sentencias  
elseif (expresión lógica) then  
    sentencias  
:  
:  
else  
    sentencias  
endif
```

El flujo de ejecución es de arriba hacia abajo. Las expresiones condicionales son evaluadas en secuencia hasta que se encuentra una que es verdadera. Entonces el código asociado es ejecutado y el control salta a la siguiente sentencia después de la sentencia `endif`.

Sentencias `if` anidadas

La sentencia `if` puede ser anidada varios niveles. Para asegurar la legibilidad es importante sangrar las sentencias. Se muestra un ejemplo:

```
if (x > 0) then
  if (x > y) then
    write(*,*) 'x es positivo y x >= y'
  else
    write(*,*) 'x es positivo pero, x < y'
  endif
elseif (x < 0) then
  write(*,*) 'x es negativo'
else
  write(*,*) 'x es cero'
endif
```

Se debe evitar anidar muchos niveles de sentencias `if` ya que es difícil de seguir.

Ejercicios

Ejercicio A

Escribir un segmento de programa en Fortran 90 que asigne a una variable tipo real t los siguientes valores (suponiendo que x e y han sido declarados previamente):

$x+y$	si x e y son ambos positivos
$x-y$	si x es positivo e y es negativo
y	si x es negativo
0	si x o y es cero

9. Ciclos

Para la repetir la ejecución de sentencias se usan los *ciclos*. Si se esta familiarizado con otros lenguajes de programación se habrá escuchado de los *ciclos-for* y de los *ciclos-until*, Fortran 90 tiene solamente una construcción de ciclo, conocida como el ciclo `do`. El ciclo-`do` corresponde al ciclo-*for* que existe en otros lenguajes de programación. Otros ciclos pueden ser simulados usando las sentencias `if` y `goto`.

Ciclos-do

El ciclo-`do` es usado para repetir un conjunto de sentencias una determinada cantidad de veces. Se muestra el siguiente ejemplo donde se calcula la suma de los enteros desde el *1* hasta *n* (suponiendo que a *n* se le ha asignado un valor previamente):

```
integer  :: i, n, suma
:
:
:

suma = 0

Do i = 1, n
  suma = suma + i
  write(*,*) 'i =', i
  write(*,*) 'suma =', suma
End do
```

La variable en la sentencia `do` es incrementada en 1 por default. Sin embargo, se puede usar cualquier otro entero para el *paso o incremento*. El siguiente segmento de programa muestra los números pares en forma decreciente entre el 1 y 10:

```
integer i

do i = 10, 1, -2
  write(*,*) 'i =', i
continue
```

La forma general del ciclo `do` es la siguiente:

```
do var = expr1, expr2, expr3
  sentencias
continue
```

donde:

var es la variable del ciclo (conocida con frecuencia como el *índice del ciclo*) el cual deberá ser del tipo `integer`.

expr1 indica el valor inicial de *var*,

expr2 es el valor hasta el que llegará el índice, y *expr3* es el incremento (step).

Nota: La variable del ciclo `do` nunca deberá ser modificada por otras sentencias dentro del ciclo, ya que puede generar errores de lógica.

Ciclos while

La forma más intuitiva para escribir un ciclo `while` es

```
do while (expr lógica)
  sentencias
enddo
```

Las sentencias en el cuerpo serán repetidas mientras la condición en el ciclo `while` sea verdadera. A pesar de que esta sintaxis es aceptada por muchos compiladores (incluyendo el de Linux), no forma parte del ANSI Fortran 77. La forma correcta es usando las sentencias `if` y `goto`:

```
etiq if (expr lógica) then
  sentencias
  goto etiq
endif
```

A continuación se tiene un ejemplo que calcula y muestra el doble de todos los número anterior comenzando con el 2 y que son menores a 100:

```
integer n
n = 1
10 if (n .lt. 100) then
  n = 2*n
  write (*,*) n
  goto 10
endif
```

Ciclos-until

Es un ciclo el cual el criterio de terminación esta al final en vez del inicio. En pseudocódigo tendríamos el siguiente formato:

```
haz
  sentencias
hasta (expr lógica)
```

lo cual nuevamente, puede ser implementado en Fortran 77 usando las sentencias `if` y `goto`:

```
etiq continue
  sentencias
  if (expr lógica) goto etiq
```

Observar que la expresión lógica en la última versión deberá ser la negación de la expresión dada en pseudocódigo.

Ciclos en Fortran 90

Fortran 90 ha adoptado la construcción `do-endo` como su ciclo (el `f77` de linux la reconoce como válida). Por lo que el ejemplo de decrementar de dos en dos queda como:

```
do i = 10, 1, -2
  write(*,*) 'i =', i
heñido
```

para simular los ciclos *while* y *until* se puede usar la construcción `do-endo`, pero se tiene que agregar una sentencia condicional de salida `exit (salida)`. El caso general es:

```
do
  sentencias
  if (expr lógica) exit
  sentencias
end do
```

Si se tienen la condición de salida al principio es un ciclo *while*, y si esta al final se tiene un ciclo *until*.

Ejercicios

Ejercicio a

Reescribe los siguientes pseudocódigos en código de Fortran 90. Evitar usar la sentencia `goto` si es posible.

```
i = 1
mientras (i<100) haz
  suma = suma + i
  i = i+2
fin_mientras
```

```
i = 0
x = 1.0
repite
  x = f(x)
  i = i+1
hasta que (x<0)
muestra i, x
```

10. Arreglos

Muchos cálculos científicos usan vectores y matrices. El tipo de dato usado en Fortran para representar tales objetos es el *array*. Un arreglo unidimensional corresponde a un vector, mientras que un arreglo bidimensional corresponde a una matriz. Para entender como son usados en Fortran 90, no solamente se requiere conocer la sintaxis para su uso, sino también como son guardados estos objetos en la memoria.

Arreglos Unidimensionales

El arreglo más sencillo es el de una dimensión, el cual es sólo un conjunto de elementos almacenados secuencialmente en memoria. Por ejemplo, la declaración

```
Real, dimension(20) :: d
```

declara a *d* como un arreglo del tipo real con 20 elementos. Esto es, *d* consiste de 20 números del tipo real almacenados en forma contigua en memoria. Por convención, los arreglos en Fortran 90 están indexados a partir del valor 1. Por lo tanto el primer elemento en el arreglo es $d(1)$ y el último es $d(20)$. Sin embargo, se puede definir un rango de índice arbitrario para los arreglos como se observa en los siguientes ejemplos:

```
real, dimension (0:19)      :: b
real, dimension (-162:237) :: c
```

En el caso de *b* es similar con el arreglo *d* del ejemplo previo, excepto que el índice corre desde el 0 hasta el 19. El arreglo *c* es un arreglo de longitud $237 - (-162) + 1 = 400$.

El tipo de los elementos de un arreglo puede ser cualquiera de los [tipos básicos de datos](#) ya vistos. Ejemplos:

```
Integer, dimension (10)  :: i
Logical, dimension (0:1) :: a
```

Cada elemento de un arreglo puede ser visto como una variable separada. Se referencia al *i*-ésimo elemento de un arreglo *a* por $a(i)$. A continuación se muestra un segmento de código que guarda los primeros 10 cuadrados en un arreglo *cuad*

```
integer i,
real, dimension (10) :: cuad

do i=1, 10, 1
  cuad(i) = i**2;
  write(*,*) cuad(i)
enddo
```

Un error común en Fortran es hacer que el programa intente acceder elementos del arreglo que están fuera de los límites. Lo anterior es responsabilidad del programador, ya que tales errores no son detectados por el compilador.

Arreglos Bidimensionales

Las matrices son muy importantes. Las matrices son usualmente representadas por arreglos bidimensionales. Por ejemplo, la declaración

```
Real, dimension(3,5) :: A
```

define un arreglo bidimensional de $3 \times 5 = 15$ números del tipo real. Es útil pensar que el primer índice es el índice del renglón, y el segundo índice corresponde a la columna. Por lo tanto se vería como:

	1	2	3	4	5
1					
2					
3					

Un arreglo bidimensional podría también tener índices de rango arbitrario. La sintaxis general para declarar el arreglo es:

```
(índice1_inf : índice1_sup, índice2_inf : índice2_sup)
```

El tamaño total del arreglo es de

```
tamaño = (índice1_sup - índice1_inf + 1) x (índice2_sup - índice2_inf + 1)
```

Forma de Almacenamiento para un arreglo bidimensional

Fortran almacena los arreglos de más de una dimensión como una secuencia contigua lineal de elementos. Es importante saber que los arreglos bidimensionales son guardados *por columnas*. Por lo tanto en el ejemplo anterior, el elemento del arreglo (1,2) está después del elemento (3,1), luego sigue el resto de la segunda columna, la tercera columna y así sucesivamente.

Considerando otra vez el ejemplo donde solamente se usa la submatriz de 3 x 3 del arreglo de 3 x 5. Los primeros 9 elementos que interesan se encuentran en las primeras nueve localidades de memoria, mientras que las últimas seis celdas no son usadas. Lo anterior funciona en forma transparente porque la *dimensión principal* es la misma para ambos, el arreglo y la matriz que se guarda en el arreglo. Sin embargo, frecuentemente la dimensión principal del arreglo será más grande que la primera dimensión de la matriz. Entonces la matriz *no* será guardada en forma contigua en memoria, aún si la arreglo es contiguo. Por ejemplo, supongamos que la declaración hubiera sido $A(5,3)$ entonces hubiera habido dos celdas "sin usar" entre el fin de la primera columna y el principio de la siguiente columna (suponiendo que asumimos que la submatriz es de 3 x 3).

Esto podría parecer complicado, pero actualmente es muy simple cuando se empieza a usar. Si se tiene en duda, puede ser útil hallar la *dirección* de un elemento del arreglo. Cada arreglo tendrá una dirección en la memoria asignada a partir del arreglo, que es el elemento (1,1). La dirección del elemento (i,j) esta dada por la siguiente expresión:

$$\text{dirección}[A(i,j)] = \text{dirección}[A(1,1)] + (j-1)*\text{princ} + (i-1)$$

donde `princ` es la dimensión principal (la columna) de `A`. Observar que `princ` es en general diferente de la dimensión actual de la matriz. Muchos errores de lógica en Fortran son causados por lo anterior, por lo tanto es importante entender la diferencia.

Arreglos Multi-dimensionales

Fortran 90 permite arreglos de hasta 7 dimensiones. La sintaxis y forma de almacenamiento es análoga al caso de dos dimensiones.

La forma para declarar un arreglo en Fortran 90 es la siguiente:

```
Integer , parameter :: n=10  
Real, dimensión (n) :: A, x
```

Ejercicios

Ejercicio A

Escribir una subrutina que haga el producto matricial escalar $A = B*C$. A,B,C son Matrices cuadradas y compatibles

11. Subprogramas

Cuando un programa tiene más de cien líneas, es difícil de seguir. Los códigos de Fortran que resuelven problemas reales de ingeniería por lo general tienen decenas de miles de líneas. La única forma para manejar códigos tan grandes, es usar una aproximación *modular* y dividir el programa en muchas unidades independientes pequeñas llamadas *subprogramas*.

Un subprograma es una pequeña pieza de código que resuelve un subproblema bien definido. En un programa grande, se tiene con frecuencia que resolver el mismo subproblema con diferentes tipos de datos. En vez de replicar el código, estas tareas pueden resolverse con subprogramas. El mismo subprograma puede ser llamado varias veces con distintas entradas de datos.

En Fortran se tienen dos tipos diferentes de subprogramas, conocidas como *funciones* y *subrutinas*.

Funciones

Las funciones en Fortran son bastante similares a las funciones matemáticas: ambas toman un conjunto de variables de entrada (parámetros) y regresan un valor de algún tipo. Al inicio de la sección se comentó de los subprogramas *definidas por el usuario*, pero Fortran 77 tiene también funciones *incorporadas*.

Un ejemplo simple muestra como usar una función:

```
x = cos(pi/3.0)
```

En este caso la función coseno `cos` de 60°, asignará a la variable `x` el valor de 0.5 (si `pi` ha sido definido correctamente; Fortran 90 no tiene constantes incorporadas). Hay varias funciones incorporadas en Fortran 90. Algunas de las más comunes son:

<code>abs</code>	<i>valor absoluto</i>
<code>min</code>	<i>valor mínimo</i>
<code>max</code>	<i>valor máximo</i>
<code>sqrt</code>	<i>raíz cuadrada</i>
<code>sin</code>	<i>seno</i>
<code>cos</code>	<i>coseno</i>
<code>tan</code>	<i>tangente</i>
<code>atan</code>	<i>arco tangente</i>
<code>exp</code>	<i>exponencial (natural)</i>
<code>log</code>	<i>logaritmo (natural)</i>

En general, una función siempre tiene un *tipo*. Varias de las funciones incorporadas mencionadas anteriormente son sin embargo *genéricas*. Por lo tanto en el ejemplo anterior `pi` y `x` podrían ser del tipo `real` o del tipo `double precision`. El compilador revisará los tipos y usará la versión correcto de la función `cos` (`real` o `double precision`).

Desafortunadamente, Fortran no es un lenguaje *polimórfico*, por lo que en general, el programador debe hacer coincidir los tipos de las variables y las funciones.

Se revisa a continuación como implementar las funciones escritas por el usuario. Supongamos el siguiente problema: un meteorólogo ha estudiado los niveles de precipitación en el área de una bahía y ha obtenido un modelo (función) $ll(m,t)$ donde ll es la cantidad de lluvia, m es el mes, y t es un parámetro escalar que depende de la localidad. Dada la fórmula para ll y el valor de t , calcular la precipitación anual

La forma obvia de resolver el problema es escribir un ciclo que corra sobre todos los meses y sume los valores de ll . Como el cálculo del valor de ll es un subproblema independiente, es conveniente implementarlo como una función. El siguiente programa principal puede ser usado:

```
program lluvia
real r, t, suma
integer m

read (*,*) t
suma = 0.0
do m = 1, 12
  suma = suma + ll(m, t)
end do
write (*,*) 'La precipitación Anual es ', suma, 'pulgadas'

stop
end
```

Además, la función ll tiene que ser definida como una función de Fortran. La fórmula del meteorólogo es:

$$ll(m,t) = t/10 * (m**2 + 14*m + 46) \text{ si la expresión es positiva}$$
$$ll(m,t) = 0 \text{ otro caso}$$

La correspondiente función en Fortran es

```
real function ll(m,t)
integer m
real t

ll = 0.1*t * (m**2 + 14*m + 46)
if (ll .LT. 0) ll = 0.0

return
end
```

Se puede observar que la estructura de una función es parecida a la del programa principal. Las diferencias son:

- Las funciones tienen un tipo. El tipo debe coincidir con el tipo de la variable que recibirá el valor.
- El valor que devolverá la función, deberá ser asignado en una variable que tenga el mismo nombre que la función.
- Las funciones son terminadas con la sentencia *return* en vez de la sentencia *stop*.

Para resumir, la sintaxis general de una función en Fortran 77 es:

```

tipo function nombre (lista_de parámetros)
  declaraciones
  sentencias
  return
end

```

La función es llamada simplemente usando el nombre de la función y haciendo una lista de argumentos entre paréntesis.

Subrutinas

Una función de Fortran puede devolver únicamente un valor. En ocasiones se desean regresar dos o más valores y en ocasiones ninguno. Para este propósito se usa la construcción subrutina. La sintaxis es la siguiente:

```

subroutine nombre (lista_de parámetros)
  declaraciones
  sentencias
  return
end

```

Observar que las subrutinas no tienen tipo y por consecuencia no pueden hacerse asignación al momento de llamar al procedimiento.

Se da un ejemplo de una subrutina muy sencilla. El propósito de la subrutina es intercambiar dos valores enteros.

```

subroutine iswap (a, b)
  integer a, b
c Variables Locales
  integer tmp

  tmp = a
  a = b
  b = tmp

  return
end

```

Se debe observar que hay dos bloques de declaración de variables en el código. Primero, se declaran los parámetros de entrada/salida, es decir, las variables que son comunes al que llama y al que recibe la llamada. Después, se declaran las *variables locales*, esto es, las variables que serán sólo conocidas dentro del subprograma. Se pueden usar los mismos nombres de variables en diferentes subprogramas.

Llamada por referencia

Fortran 77 usa el paradigma de *llamada por referencia*. Esto significa que en vez de pasar los valores de los argumentos a la función o la subrutina (*llamada por valor*), se pasa la dirección (apuntadores) de los argumentos.

```

program llamaint
  integer m, n
c
  m = 1

```

```

n = 2

call iswap(m, n)
write(*,*) m, n

stop
end

```

La salida de este programa es "2 1", justo como se habría esperado. Sin embargo, si Fortran 77 hubiera hecho una llamada por valor entonces la salida hubiera sido "1 2", es decir, las variables m y n hubieran permanecido sin cambio. La razón de esto último, es que solamente los valores de m y n habrían sido copiados a la subrutina *iswap*, a pesar de que a y b hubieran sido cambiados dentro de la subrutina, por lo que los nuevos valores no sería regresados al programa que hizo la llamada.

En el ejemplo anterior, la llamada por referencia era lo que se quería hacer. Se debe tener cuidado al escribir código en Fortran, porque es fácil introducir *efectos laterales* no deseados. Por ejemplo, en ocasiones es tentador usar un parámetro de entrada en un subprograma como una variable local y cambiar su valor. No se deberá hacer *nunca*, ya que el nuevo valor se propagará con un valor no esperado.

Se revisará más conceptos cuando se vea la sección de [Arreglos en subprogramas](#) para pasar arreglos como argumentos.

Ejercicios

Ejercicios A

Escribir una función llamada *fac* que tome un entero n como entrada y regrese $n!$ (factorial de n). Probar la función usando el siguiente programa main

```

program prbfac
c
c Ejercicio A, seccion 11.
c Programa Main para probar la función factorial.
c
integer n, fac

10 continue
write(*,*) 'Dame n: '
read (*,*) n
if (n.gt.0) then
write(*,*) ' El factorial de', n, ' es', fac(n)
goto 10
endif
c End of loop

stop
end

```

(Tip: Se tiene que usar un ciclo para implementar la función ya que Fortran 77 no soporta llamadas recursivas.)

Ejercicio B

Escribir una subrutina *cuad* que tome tres números reales a, b, c como entrada y encuentre las raíces de la ecuación $ax^{**2} + bx + c = 0$. Si las raíces son complejas, se deberá mostrar un mensaje de error como el siguiente:

```
write(*,*) 'Advertencia: Raices Complejas.'
```

También se deberá escribir un programa main que pruebe la subrutina.

12. E/S Básica

Una parte importante de cualquier programa de cómputo es manejar la entrada y la salida. En los ejemplos revisados previamente, se han usado las dos construcciones más comunes de Fortran que son: `read` and `write`. La E/S con Fortran puede ser un poco complicada, por lo que nos limitaremos a los casos más sencillos en el tutorial.

Lectura y Escritura

La sentencia `read` es usada para la entrada y la sentencia `write` para la salida. El formato es:

```
read (núm_unidad, núm_formato) lista_de_variables
write(núm_unidad, núm_formato) lista_de_variables
```

El número de unidad se puede referir a la salida estándar, entrada estándar o a un archivo. Se describirá más adelante. El número de formato se refiere a una etiqueta para la sentencia `format`, la cual será descrita brevemente.

Es posible simplificar estas sentencias usando asteriscos (*) para algunos argumentos, como lo que se ha hecho en los ejemplos anteriores. A lo anterior se le conoce como una lectura/escritura de *lista dirigida*.

```
read (*,*) lista_de_variables
write(*,*) lista_de_variables
```

La primera sentencia leerá valores de la entrada estándar y asignará los valores a las variables que aparecen en la lista, y la segunda escribe la lista en la salida estándar.

Ejemplos

Se muestra un segmento de código de un programa de Fortran:

```
read(*,*) m, n
read(*,*) x, y
```

Se ingresan los datos a través de la entrada estándar (teclado), o bien, redireccionando la entrada a partir de un archivo. Un archivo de datos consiste de *registros* de acuerdo a los formatos válidos de Fortran. En el ejemplo, cada registro contiene un número (entero o real). Los registros están separados por espacios en blanco o comas. Por lo que una entrada válida al programa anterior puede ser:

```
-1 100
-1.0 1e+2
```

O se pueden agregar comas como separadores:

```
-1, 100
```

-1.0, 1e+2

Observar que la entrada en Fortran 90 es sensible a la línea, por lo que es importante contar con el número apropiado de elementos de entrada (registros) en cada línea. Por ejemplo, si se da la siguiente entrada en una sola línea

-1, 100, -1.0, 1e+2

entonces a m y a n se asignarán los valores de -1 y 100 respectivamente, pero los dos últimos valores serán descartados, dejando a x e y sin definir.

13. Sentencia `Format`

En las secciones anteriores se ha mostrado el *formato libre* de entrada/salida. Éste caso sólo usa una reglas predefinidas acerca de como los diferentes tipos (integers, reals, characters, etc.) serán mostrados. Por lo general un programador desea indicar algún formato de entrada o salida, por ejemplo, el número de decimales que tendrá un número real. Para este propósito Fortran 90 tiene la sentencia *format*. La misma sentencia *format* puede ser usada para la entrada o salida.

Sintaxis

```
write(*, etiqueta) lista_de_variables
etiqr format códigos_de_formato
```

Un ejemplo simple muestra como trabaja. Supongamos que se tiene una variable entera que se quiere mostrar con un ancho de 4 caracteres y un número real que se quiere mostrar en notación de punto fijo con 3 decimales.

```
write(*, 900) i, x
900 format (I4,F8.3)
```

La etiqueta 900 de la sentencia *format* es escogida en forma arbitraria, pero es una práctica común numerar las sentencias *format* con números más grandes que las etiquetas de control de flujo. Después de la palabra *format* se ponen los códigos de formato encerrados entre paréntesis. El código `I4` indica que un entero tendrá un ancho de 4 y `F8.3` significa que el número deberá mostrarse en notación de punto fijo con un ancho de 8 y 3 decimales.

La sentencia *format* puede estar en cualquier lugar dentro del programa. Hay dos estilos de programación: agrupar por parejas las sentencias (como en el ejemplo), o poner el grupo de sentencias *format* al final del (sub)programa.

Códigos comunes de formato

Las letras para códigos de formato más comunes son:

- A - cadena de texto
- D - números de doble precisión, notación científica
- E - números reales, notación científica
- F - números reales, formato de punto fijo
- I - entero
- X - salto horizontal (espacio)
- / - salto vertical (nueva línea)

El código de formato F (y similarmente D y E) tiene la forma general $Fa.d$ donde a es una constante entera indicando el ancho del campo y d es un entero constante que indica el número de dígitos significativos.

Para los enteros solamente el campo de ancho es indicado, por lo que la sintaxis es $\text{I}a$. En forma parecida las cadenas de caracteres pueden ser especificadas como $\text{A}a$ pero el campo de ancho por lo general no es usado.

Si un número o cadena no llena todo el ancho del campo, espacios son agregados. Usualmente el texto será ajustado a la derecha, pero las reglas exactas varían de acuerdo a los códigos de formato.

Para un espaciado horizontal, el código nX es usado. Donde n indica el número de espacios horizontales. Si n es omitido se asume $n=1$. Para espaciado vertical (nuevas líneas) se usa el código $/$. Cada diagonal corresponde a una nueva línea. Observar que cada sentencia `read` o `write` por defecto termina con un salto de línea (a diferencia de C).

Algunos Ejemplos

El siguiente código de Fortran

```
x = 0.025
write(*,100) 'x=', x
100 format (A,F)
write(*,110) 'x=', x
110 format (A,F5.3)
write(*,120) 'x=', x
120 format (A,E)
write(*,130) 'x=', x
130 format (A,E8.1)
```

genera la siguiente salida una vez que es ejecutado:

```
x=      0.0250000
x=0.025
x=  0.2500000E-01
x=  0.3E-01
```

Observar que espacios en blanco son automáticamente puestos del lado izquierdo y que el ancho del campo por default para números tipo real es de usualmente de 14. Se puede ver también que Fortran 90 sigue la regla de redondeo donde los dígitos del 0-4 son redondeados hacia abajo y los dígitos del 5-9 son redondeados hacia arriba.

En este ejemplo cada sentencia `write` usa una sentencia `format` diferente. Pero es correcto usar la misma sentencia `format` varias veces con distintas sentencias `write`. De hecho, esta es una de las principales ventajas de usar sentencias `format`. Esta característica es buena cuando se muestra el contenido de una tabla por ejemplo, y se desea que cada renglón tenga el mismo formato. `format`.

Cadenas de formato en las sentencias read/write

En vez de indicar el código de formato en una sentencia format por separado, se puede dar el código de formato en la sentencia read/write directamente. Por ejemplo, la sentencia

```
write (*,'(A, F8.3)') 'La respuesta es x = ', x
```

que es equivalente a

```
write (*,990) 'La respuesta es x = ', x
990 format (A, F8.3)
```

Algunas veces cadenas de texto son dadas en las sentencias de formato, por ejemplo la siguiente versión es también equivalente:

```
write (*,999) x
999 format ('La respuesta es x = ', F8.3)
```

Ciclos Implícitos y Repetición de Formatos

Ahora se mostrará un ejemplo más complejo. Supongamos que se tiene un arreglo bidimensional de enteros y que se desea mostrar la submatriz izquierda 5 por 10, con 10 valores cada 5 renglones.

```
do 10 i = 1, 5
  write(*,1000) (a(i,j), j=1,10)
10 continue
1000 format (I6)
```

Se tiene un ciclo explícito do loop sobre los renglones y un ciclo *implícito* sobre el índice j para la columna.

Con frecuencia una sentencia format involucra repetición, por ejemplo:

```
950 format (2X, I3, 2X, I3, 2X, I3, 2X, I3)
```

Hay una notación abreviada para lo anterior, que es:

```
950 format (4(2X, I3))
```

Es también posible permitir la repetición sin hacerlo explícitamente indicando las veces que el formato deberá repetirse. Supongamos que tenemos un vector, del cual se desea mostrar los primeros 50 elementos, con 10 elementos en cada línea. Se muestra una forma de hacerlo:

```
write(*,1010) (x(i), i=1,50)
1010 format (10I6)
```

La sentencia format dice que 10 números deberán ser mostrados. Pero en la sentencia write, se hace con los primeros 50 números. Después de que los primeros 10 números han sido mostrados, la misma sentencia format es automáticamente usada para los siguientes 10 números y así sucesivamente.

Ejercicios

Ejercicio A

Pueden suceder cosas extrañas si no hay una correspondencia adecuada con la sentencia format. Intentar los siguientes ejemplos:

```
write(*,100) 12, 12345
write(*,110) 0.12345
write(*,110) 123.45
write(*,110) 12345.0
100 format (I4, 2X, I4)
110 format (F6.2)
```

14. E/S de Archivos

Se han estado haciendo ejemplos donde la salida/entrada se ha realizado a los dispositivos estándares de entrada/salida. También es posible leer o escribir de *archivos* los cuales son guardados en algún dispositivo externo de almacenamiento, por lo general un disco (disco duro, floppy) o una cinta. En Fortran cada archivo esta asociado con un *número de unidad*, un entero entre 1 y 99. Algunos números están reservados: 5 es la entrada estándar, 6 es la salida estándar.

Abriendo y cerrando un archivo

Antes de que pueda usarse un archivo se requiere que sea *abierto (open)*. El comando es `open (lista_de_especificadores)` donde los especificadores más comunes son:

```
[UNIT=]  u
IOSTAT=  ios
ERR=     err
FILE=    nomb_arch
STATUS=  sta
ACCESS=  acc
FORM=    frm
RECL=    rl
```

El número de unidad *u* es un número en el rango de 9-99 para algún archivo, el programador lo escoge debiendo ser un número único.

ios es el identificador del estado de la E/S y debe ser una variable entera. El valor que regresa *ios* es cero si la sentencia fue exitosa y sino, regresa un valor diferente de cero.

err es una etiqueta a la cual el programa saltará si hay un error.

nomb_arch es una cadena de caracteres que contiene el nombre del archivo.

sta es una cadena de caracteres que tiene que ser NEW, OLD o SCRATCH. Esta muestra el estatus del archivo. Un archivo scratch es aquel que es creado y borrado cuando el archivo es cerrado (o el programa termina).

acc deberá ser SEQUENTIAL o DIRECT. El valor predefinido es SEQUENTIAL.

frm deberá ser FORMATTED o UNFORMATTED. El valor predefinido es UNFORMATTED.

rl indica la longitud de cada registro en un archivo de acceso directo.

Para más detalles en los especificadores, se recomienda que se revise un buen libro de Fortran 90.

Una vez que un archivo ha sido abierto, se puede acceder con sentencias de lectura y escritura. Cuando se manipula un archivo y se termina de usar, deberá ser cerrado usando la sentencia.

```
close ([UNIT=u], IOSTAT=ios, ERR=err, STATUS=sta])
```

donde, los parámetros en bracket *[]* son opcionales

Complemento de Read and write

El único cambio necesario de los ejemplos previos de las sentencias read/write, es que el número de unidad debe ser indicado. Pero se pueden incluir especificadores adicionales. Se muestra como:

```
read ([UNIT=u], [FMT=fmt], IOSTAT=ios, ERR=err, END=s)
write([UNIT=u], [FMT=fmt], IOSTAT=ios, ERR=err, END=s)
```

donde la mayoría de los especificadores han sido descritos anteriormente. El especificador END=*s* define a que sentencia saltará el programa si se alcanza el fin del archivo (eof).

Ejemplo

Se da un archivo de datos con las coordenadas xyz de un montón de puntos. El número de puntos es dado en la primera línea. El nombre del archivo de datos es *puntos.dat*. El formato para cada coordenada es de la forma F10.4 Se muestra un programa que lee los datos y los pone en tres arreglos x, y, z.

```
program entdat
i
; Este programa lee n puntos desde un archivo de datos y los guarda en
; 3 arreglos x, y, z.
i
    integer    :: nmax, u
    integer, parameter :: nmax=1000, u=20
    real, dimension (nmax) :: x,y,z

; Abrir el archivo de datos
    open (u, FILE='puntos.dat', STATUS='OLD')

; Leer el número de puntos
    read(u,*) n
    if (n > nmax) then
        write(*,*) 'Error: n = ', n, 'es más largo que nmax =', nmax
        goto 9999
    endif
```

```

; Ciclo para recuperar los datos
  do i= 1, n
    read(u,100) x(i), y(i), z(i)
  enddo
100 format (3(F10.4))

; Cerrar el archivo
  close (u)

; Ahora se procesarán los datos
; ...

9999 stop
  end

```

Using 32 bits to represent a number, positive or negative, the range of possible values is large but there are circumstances when bigger number representations are needed. The way to do this is to use floating point numbers.

A floating point number is one in which the position of the point is determined within the number itself. Working with floating point values, the position of the point in each number must be determined at processing time. This means that these calculations are much slower than fixed point. The reason for using floating point representation is that the range of possible values is much greater.

Floating point representation is similar to scientific notation and details of how values are represented vary from one machine to another. One of the standard floating point representations is the IEEE standard and it is useful to understand how this works.

The number uses a 32-bit string. This string has 3 distinct parts:

- The sign bit 1 bit. With the value 1 for negative and 0 for positive.
- The mantissa 23 bits
- The exponent 8 bits

Using binary, the value of the number is:

$$\text{Sign (+ / -) } 1.\text{mantissa} * 2^{\text{exponent}}$$

Example 1
Sign bit 1 negative.

Sign	Exponent	Mantissa
1	10000111	1011001101000000000000

Exponent. The exponent is shown with excess 127. This means that the machine exponent is the actual exponent with 127 added. This has the effect of giving a range of 0 to 255 for the machine representation while the range of actual values is -128 to +127. If the excess was not used there would have to be a mechanism for showing a negative exponent.

In the example the actual exponent is 1000 (decimal 8).

The machine representation is:

$$0000\ 1000\ (8) + 0111\ 1111\ (127) = 1000\ 0111\ (\text{decimal } 135)$$

Mantissa. The mantissa is the number being encoded. Before the number is encoded the point is moved(or floated) left or right until the value of the number is in the range:

$$1 < n < 2$$

This is known as *normalisation*. Since the first digit is now always going to be 1 there is no need to encode this digit. The mantissa only includes the digits after the point and the leading 1 is assumed.

The number shown above is $-1.1011\ 0011\ 01 \times 2^8$

In un-normalised form this is -110110011.01

Example 2

To encode the binary number 101.1110 01:

Step 1 The sign bit is 0 for positive.

Step2: normalise Move the point 2 places to the left:
1.01111 001
the digits after the point are the mantissa. All trailing bits are 0.

Step 3 The point was moved 2 places to the left. The exponent is now used to multiply the mantissa by 2^2 This effectively returns the mantissa to its original size. However, remember that excess 127 is used so the exponent will be $2 + 127 = 129$.

Sign	Exponent	Mantissa
0	10000001	011110010000000000000000

Using this representation the 32-bit number has 23 bits giving the detail of the value and 8 bits in the exponent which means that the number can be increased or decreased by 2^{127} .

A 32-bit integer can show 32 bits of detail but has lower magnitude. This is the trade-off between floating point and fixed point numbers.