Elementos de programación Fortran.

Hay dos formas de escribir programas sin errores.
Sólo la tercera funciona.

Pablo Santamaría

v0.1 (Abril/Mayo 2008)

Índice

1. Primeros pasos en programación.

La resolución de un problema científico con una computadora, tarde o temprano, conduce a la escritura de un *programa* que implemente la solución del problema. Un programa es un conjunto de instrucciones, ejecutables sobre una computadora, que permite cumplir una función específica. Ahora bien, ¡la creación del programa no comienza directamente en la computadora! El proceso comienza en papel diseñando un *algoritmo* para resolver el problema. Un algoritmo es un conjunto de pasos (o instrucciones) *precisos*, *definidos* y *finitos* que a partir de ciertos datos conducen al resultado del problema.

Características de un algoritmo.

- *preciso*: el orden de realización de cada paso está especificado,
- *definido*: cada paso está especificado sin ambigüedad,
- *finito*: el resultado se obtiene en un número finito de pasos.
- *entrada/salida*: dispone de cero o más datos de entrada y devuelve uno o más resultados.

Para describir un algoritmo utilizamos un *pseudocódigo*. Un pseudocódigo es un lenguaje de especificación de algoritmos donde las instrucciones a seguir se especifican de forma similar a como las describiríamos con nuestras palabras.

Consideremos, como ejemplo, el diseño de un algoritmo para calcular el área de un círculo. Nuestro primer intento, bruto pero honesto, es:

```
Calcular el área de un círculo.
```

Sin embargo, este procedimiento no es un algoritmo, por cuanto no se especifica, como dato de entrada, cuál es el círculo a considerar ni tampoco cual es el resultado. Un mejor procedimiento sería:

```
Leer el radio del círculo.
Calcular el área del círculo de radio dado.
Imprimir el área.
```

Sin embargo, éste procedimiento aún no es un algoritmo por cuanto la segunda instrucción no especifica cómo se calcula el área de un círculo de radio dado. Explicitando la fórmula matemática tenemos finalmente un algoritmo apropiado:

```
Leer el radio del círculo.
Tomar π = 3.141593.
Calcular área = π × radio^2.
Imprimir el área.
```

Una manera complementaria de describir un algoritmo es realizar una representación gráfica del mismo conocida como *diagrama de flujo*. El correspondiente diagrama de flujo para el algoritmo anterior se ilustra en la figura 1.

Una vez que disponemos del algoritmo apropiado, su implementación en la computadora requiere de un *lenguaje de programación*. Un lenguaje de programación es un lenguaje utilizado para escribir programas de computadora. Como todo lenguaje, cada lenguaje de programación tiene una sintaxis y gramática particular que debemos aprender para poder utilizarlo. Por otra

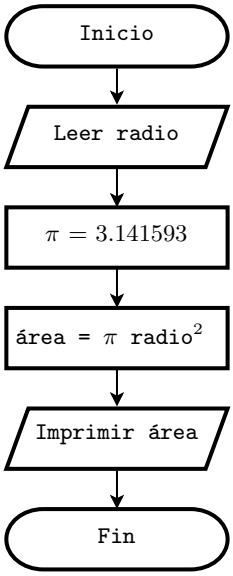

Figura 1. Diagrama de flujo para el algoritmo que calcula el área de un círculo.

parte, si bien existen muchos lenguajes de programación para escoger, un lenguaje adecuado para problemas científicos es el denominado lenguaje Fortran[1]. El proceso de implementar un algoritmo en un lenguaje de programación es llamado *codificación* y su resultado *código* o *programa fuente*. Siguiendo con nuestro ejemplo, la codificación del algoritmo en el lenguaje Fortran conduce al siguiente programa:

```
      program CalcularArea
*     ------------------------------------------------
*     Cálculo del área de un círculo de radio dado
*     ------------------------------------------------
*     Declaración de variables
*     ------------------------------------------------
      implicit none
      real pi, radio, area
      parameter (pi = 3.141593)
*     ------------------------------------------------
*     Entrada de datos
*     ------------------------------------------------
      write(*,*) 'Ingrese radio del círculo'
      read(*,*) radio
*     ------------------------------------------------
*     Calcular area
*     ------------------------------------------------
      area = pi*radio**2
*     ------------------------------------------------
*     Imprimir resultado
*     ------------------------------------------------
      write(*,*) 'Area = ', area
*     ------------------------------------------------
*     Terminar
*     ------------------------------------------------
      stop
      end
```

[1] El nombre es un acrónimo en inglés de *formula translation*.

Aún cuando no conozcamos todavía la sintaxis y gramática del Fortran, podemos reconocer en el código anterior ciertas características básicas que se corresponden con el algoritmo original. En particular, notemos que los nombres de las variables involucradas son similares a la nomenclatura utilizada en nuestro algoritmo, que los datos de entrada y salida estan presentes (junto a un mecanismo para introducirlos y mostrarlos) y que el cálculo del área se expresa en una notación similar (aunque no exactamente igual) a la notación matemática usual. Además hemos puesto una cantidad de comentarios que permiten comprender lo que el código realiza.

Un algoritmo es independiente del lenguaje de programación.

Si bien estamos utilizando Fortran como lenguaje de programación debemos enfatizar que un algoritmo es independiente del lenguaje de programación que se utiliza para su codificación. De este modo un mismo algoritmo puede ser implementado en diversos lenguajes.

Disponemos ya de un código para nuestro problema. Ahora bien, ¿cómo lo llevamos a la computadora para obtener un programa que se pueda ejecutar? Este proceso involucra dos etapas: la *edición* y la *compilación*. Comencemos, pues, con la edición. Nuestro código fuente debe ser almacenado en un archivo de *texto plano* en la computadora. Para ello debemos utilizar un *editor de texto*, el cual es un programa que permite ingresar texto por el teclado para luego almacenarlo en un archivo. El archivo resultante se conoce como *archivo fuente* y para un código escrito en Fortran puede tener el nombre que querramos pero debe terminar con la *extensión* `.f`. Si bien en Linux existen varios editores de texto disponibles, nosotros utilizaremos el editor `emacs`. Así para ingresar el código en un archivo que llamaremos `area.f` ejecutamos en la línea de comandos de la terminal el comando:

```
$ emacs area.f &
```

El editor `emacs` es un editor de texto de propósito general pero que adapta sus posibilidades al contenido del archivo que queremos guardar. En particular permite que la programación en Fortran resulte muy cómoda al resaltar con distintos colores las diversas instrucciones que posee el lenguaje y facilitar, además, la generación de código sangrado apropiadamente para mayor legibilidad.

Nuestro programa fuente, almacenado ahora en el archivo fuente `area.f`, *no es todavía un programa que la computadora pueda entender directamente*. Esto se debe a que Fortran es uno más de los denominados *lenguajes de alto nivel*, los cuales están diseñados para que los programadores (es decir, nosotros) puedan escribir instrucciones con palabras similares a los lenguajes humanos (en general, como vemos en nuestro código, en idioma inglés)[2]. En contraste, una computadora no puede entender directamente tales lenguajes, pues las computadoras solo entienden *lenguajes de máquina* donde las instrucciones se expresan en términos de los dígitos binarios 0 y 1. De este modo, nuestro programa fuente, escrito en un lenguaje de alto nivel, debe ser traducido a instrucciones de bajo nivel para que la computadora pueda ejecutarlo. Esto se realiza con ayuda de un programa especial conocido como *compilador*. Así pues, el compilador toma el código fuente del programa y origina un *programa ejecutable* que la computadora puede entender directamente. Este proceso es denominado *compilación*. En Linux el compilador de Fortran es llamado `gfortran`. Así, para compilar el archivo fuente `area.f` y generar un programa ejecutable, que llamaremos `area`, escribimos en la línea de comandos:

```
$ gfortran -Wall -o area area.f
```

Debería ser claro que la opción `-o` permite dar el nombre para el programa ejecutable resultante de la compilación. Por otra parte la opción `-Wall` le dice al compilador que nos advierta de posibles errores (no fatales) durante el proceso de compilación[3].

[2] De hecho, Fortran es el abuelo de todos los lenguajes de alto nivel, pues fue el primero ellos.

[3] *Wall* significa, en inglés, *warnings all*.

Si no se producen errores, habremos creado nuestro primer programa. El programa se puede ejecutar desde la línea de comandos con sólo teclar su nombre:

```
$ ./area
```

¿Qué significa ./?

Puesto que el carácter . representa al directorio actual y / al separador de directorios, ./ es el directorio actual de trabajo.

Cuando las cosas fallan. Tres tipos de errores se pueden presentar: *errores de compilación*, *errores de ejecución* y *errores lógicos*. Los errores de compilación se producen normalmente por un uso incorrecto de las reglas del lenguaje de programación (típicamente *errores de sintaxis*). Debido a ellos el compilador no puede generar el programa ejecutable. Por otra parte, los errores de ejecución se producen por instrucciones que la computadora puede comprender pero no ejecutar. En tal caso se detiene abruptamente la ejecución del programa y se imprime un mensaje de error. Finalmente, los errores lógicos se deben a un error en la lógica del programa. Debido a estos errores, aún cuando el compilador nos da un programa ejecutable, el programa dará resultados incorrectos. Todos estos tipos de errores obligan a revisar el código, originando un ciclo de desarrollo del programa que consta de compilación, revisión y nueva compilación hasta que resulten subsanados todos los errores.

Ftnchek.

El programa `ftnchek` permite identificar la mayoría de los errores de sintaxis y ciertos tipos de errores de gramática que pueden ocurrir en un código Fortran.

Los dialectos de Fortran. El primer compilador Fortran data de 1957. Al igual que los lenguajes humanos, con el paso del tiempo el lenguaje va evolucionando adaptándose a las necesidades (y filosofía) de programación de la época. Cada cierto período de tiempo un comité internacional fija la sintaxis y gramática del lenguaje originando un *estándar*. Con un estándar del lenguaje a seguir, todos los programadores nos podemos asegurar que los programas funcionaran en cualquier computadora que tenga un compilador Fortran. Tradicionalmente, los estándares de Fortran se denominan con la fecha de su constitución. El primer estándar fue Fortran 66, al cual le siguió el Fortran 77, el cual introdujo mejoras significativas al lenguaje y aún hoy continúa siendo ampliamente utilizado en la comunidad científica. Una mayor revisión del lenguaje fue el Fortran90. El Fortran 90 contiene como caso particular al Fortran 77 pero va más allá incorporando muchas nuevas características. Una revisión menor del Fortran 90 condujo al estándar Fortran 95. El estándar más reciente es el Fortran 2003, el cual constituye una nueva revisión completa del lenguaje.

En estas notas trabajaremos con Fortran 77, utilizando ciertas extensiones estandarizadas en el Fortran 90 que facilitan la escritura (y lectura) de programas Fortran.

2. Estructura general de un programa Fortran.

Un programa en Fortran consiste de un programa principal (*main*, en inglés) y posiblemente varios subprogramas. Por el momento asumiremos que el código consta sólo de un programa principal. La estructura del programa principal es

Columna	Uso
1	Blanco, o un carácter `c` o `*` para indicar un comentario.
2–5	Etiqueta de sentencia (*label* en inglés).
6	Carácter (usualmente `+` o &) para indicar la continuación de una línea anterior.
7–72	Sentencias.

Tabla 1. Disposición de las columnas en el formato fijo del código fuente para Fortran 77.

```
program nombre
declaraciones de tipo
sentencias ejecutables
stop
end
```

Cada línea del programa forma parte de una *sentencia* que describe una instrucción a ser llevada a cabo y tales sentencias se siguen en el orden que estan escritas. En general, las sentencias se clasifican en *ejecutables* (las cuales realizan acciones concretas como ser cálculos aritméticos o entrada y salida de datos) y *no ejecutables* (las cuales proporcionan información sin requerir en sí ningún cómputo, como ser las declaraciones de tipos de datos involucrados en el programa). La sentencia (no ejecutable) `program` especifica el comienzo del programa principal. La misma está seguida de un nombre identificatorio para el programa el cual no necesita ser el mismo que el de su código fuente ni el del programa ejecutable que genera el compilador. La sentencia (no ejecutable) `end` indica el final lógico del programa principal. La sentencia (ejecutable) `stop` detiene la *ejecución* del programa.

End y stop.

Es importante comprender la diferencia entre estas dos sentencias. La sentencia `end`, siendo la última sentencia del programa principal, indica *al compilador* la finalización del mismo. Por su parte, la sentencia `stop` detiene la ejecución del programa, *cuando éste es ejecutado*, devolviendo el control al sistema operativo. Así mientras sólo puede existir una sentencia `end` para indicar el final del programa principal, puede haber más de una sentencia `stop` y puede aparecer en cualquier parte del programa que la necesite. Por otra parte, una sentencia `stop` inmediatamente antes de una sentencia `end` es, en realidad, opcional, puesto que el programa terminará cuando alcance el fin. Sin embargo su uso es recomendado para resaltar que la ejecución del programa termina allí.

Los programas en Fortran 77 tienen una idiosincrasia particular: las sentencias *no* son dispuestas en el archivo fuente en un formato libre, sino que, por el contrario existen un conjunto de reglas estrictas sobre la disposición de las mismas. Cada línea dispone de un máximo de 72 columnas que se emplean como se indica en la tabla 1

Nota 1. Las estrictas reglas en Fortran 77 sobre la disposición en columnas de una línea son una reliquia de los tiempos en que el programa se *cargaba* en la computadora con tarjetas perforadas. Fortran 90, por su parte, permite un formato libre de las líneas.

Nota 2. Originalmente la codificación de un programa Fortran solo podía utilizar letras mayúsculas. En la actualidad todos los compiladores aceptan que el programa esté escrito con letras minúsculas. Debe tenerse presente, sin embargo, que el compilador Fortran *no* distinguirá entre las letras mayúsculas y minúsculas (excepto en las constantes literales de carácter).

Nota 3. Un comentario es una indicación del programador que permite aclarar o resaltar lo que realiza cierta parte del código. Los comentarios son ignorados por el compilador. Una extensión (presente en Fortran 90) permite comentar en la misma línea que la sentencia si el comentario está precedido con el carácter `!`.

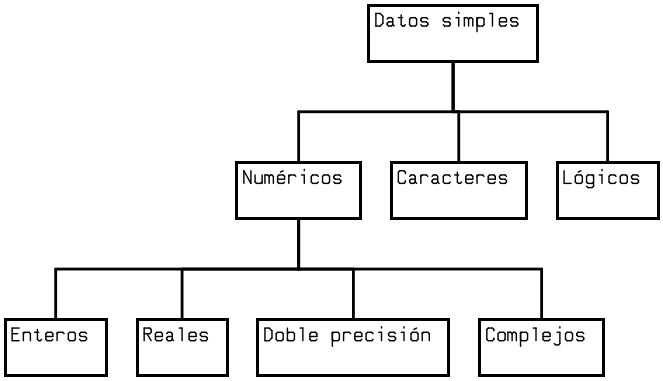

Figura 2. Tipos de datos simples.

NOTA 4. En el estilo de programación estructurada que desarrollaremos las etiquetas de sentencia no serán necesarias.

NOTA 5. Los espacios en blanco en una sentencia son ignorados en Fortran 77. Por lo tanto, si se remueven todos los espacios en blanco en un programa Fortran 77 el programa es sintácticamente correcto (pero no es muy legible que digamos). En Fortran 90, por el contrario, los espacios en blanco son significativos.

3. Tipos de datos simples.

Los diferentes objetos de información con los que un programa puede operar se conocen colectivamente como *datos*. Todos los datos tienen un *tipo* asociados a ellos. La asignación de tipos a los datos permite indicar la clase de valores que pueden tomar y las operaciones que se pueden realizar con ellos. Fortran 77 dispone de seis tipos de datos simples: *enteros*, *reales*, *doble precisión*, *complejos*, de *carácter* y *lógicos* (siendo los cuatro primeros datos de tipo numérico). Esta clasificación se ilustra en la figura 2. Por otra parte, los datos aparecen en el programa como *constantes literales*, como *variables* o como *parámetros*. Una constante literal es un valor de cualquiera de los tipos de datos que se utiliza como tal y por lo tanto permanece invariable. Una variable, en cambio, es un dato, referido con un nombre, susceptible de ser modificado por las acciones del programa. Un parámetro es un valor referenciado con un nombre cuyo valor permanece fijo, es decir, no puede ser alterado por las acciones del programa.

Existen razones tanto matemáticas como computacionales para considerar distintos tipos de datos numéricos. Un dato entero permite la representación *exacta* de un número entero en la computadora (dentro de cierto rango). Un dato real y un dato de doble precisión permiten la representación de un número real, aunque tal representación *no* es exacta, sino aproximada. La diferencia entre los tipos real y doble precisión radica en la diferencia de *precisión* en tales aproximaciones.

Precisión de los datos reales y de doble precisión.

En las computadoras personales (PC) actuales, un dato de tipo real tiene una precisión de unos 6–7 dígitos significativos mientras que un dato de doble precisión tiene entre 15–16 dígitos significativos. Por tal motivo, en general, la solución de problemas científicos requiere el uso de datos de doble precisión.

Una *constante literal entera* es un número entero, como ser `-1`, `0`, `2`, y por lo tanto no puede tener parte decimal. Una *constante literal real* es un número real con punto decimal, como ser `3.14159`,

`-18.8`, `0.0056`. Números muy grandes o muy pequeños se representan en notación científica en la forma

$$\pm \texttt{m}\ \texttt{e} \pm \texttt{n},$$

donde `m` es un número decimal y la `e` (de exponente) significa multiplicar por 10 a la potencia entera $\pm$`n`. Así, por ejemplo, 3.49×10^{-2} se codifica como `3.49e-2`. Una *constante literal de doble precisión* se codifica a través de la notación científica pero reemplazando la `e` por una `d`. Así, 0.31×10^{-7} se escribe en doble precisión como `0.31d-7`, en tanto que 1.5 se escribe como `1.5d0` o bien `0.15d1`.

Constantes literales numéricas.

Fortran establece la diferencia entre constantes literales enteras y reales por la presencia o ausencia del punto decimal, respectivamente. Y la diferencia entre constantes literales reales y de doble precisión por la ausencia o presencia del carácter `d`, respectivamente. Estas tres formas *no* son intercambiables puesto que se almacenan y procesan de forma diferente en la computadora.

Por razones de claridad y estilo es conveniente utilizar el cero explícitamente al escribir constante reales sin parte fraccionaria con parte entera nula. Así escribimos `11.0` en vez de `11.` y `0.2` en vez de `.2`. También resulta conveniente codificar el cero real como `0.0` y el cero de doble precisión como `0.0d0`.

Un dato de tipo complejo permite representar un número complejo mediante un par ordenado de datos reales: uno que representa la parte real, y otro, la parte imaginaria del número complejo. Una *constante literal compleja* consiste un par ordenado de constantes literales reales encerradas entre paréntesis y separadas por una coma. Por ejemplo `(3.5,4.0)` corresponde al número complejo con parte real 3.5 y parte imaginaria 4.0. Un dato tipo carácter permite representar caracteres alfanuméricos, esto es letras (a, b, A, B, etc.) y dígitos (tal como 6), y caracteres especiales (tales como $, &, `*`, el espacio en blanco, etc.). Una *constante literal de carácter* consiste en una secuencia de caracteres encerradas entre apóstrofes (comillas sencillas), por ejemplo: `'ABC'`. Finalmente, un dato de tipo lógico es un dato que solo puede tomar un valor entre dos valores posibles: verdadero o falso. Sus valores literales se codifican en Fortran como `.true.` y `.false.` respectivamente.

Las variables son los datos de un programa cuyo valores pueden cambiar durante la ejecución del mismo. Existen tantos tipos de variables como tipos de datos diferentes. Una variable, en realidad, es una región dada en la memoria de la computadora a la cual se le asigna un nombre simbólico. El nombre simbólico o identificador se llama *nombre de la variable* mientras que el valor almacenado en la región de memoria se llama *valor de la variable*. La extensión de la región que ocupa la variable en la memoria viene dada por el tipo de dato asociado con ella. Una imagen mental útil de la situación es considerar a las variables como cajas o buzones, cada una de las cuales tiene un nombre y contiene un valor. En Fortran los nombres de las variables sólo pueden formarse con letras o números y siempre deben comenzar con una letra. Fortran 77 sólo permite seis caracteres como máximo para formar el nombre, en tanto que Fortran 90 permite hasta 31 caracteres (además Fortran 90 considera al guión bajo `_` como un carácter válido). En la práctica, aún cuando dentro del estándar del Fortran 77 sólo puedan utilizarse seis caracteres, todos los compiladores actuales permitirán utilizar más.

Sobre los nombres de las variables

Constituye una buena práctica de programación utilizar nombres de variables que sugieran lo que ellas representan en el contexto del problema considerado. Esto hace al programa más legible y de más fácil comprensión.

Antes de ser utilizadas, *las variables deben ser declaradas* en la sección de declaración de variables del programa. Para ello se utilizan las siguientes sentencias de acuerdo al tipo de dato que almacenarán:

`integer` *lista de variables enteras*
`real` *lista de variables reales*
`double precision` *lista de variables de doble precisión*
`complex` *lista de variables complejas*
`character(`*tamaño*`)` *lista de variables carácter*
`logical` *lista de variables lógicas*

Si la lista de variables cuenta con más de un elemento, las mismas están separadas por comas. Puede también usarse una sentencia de declaración de tipo por cada variable del programa.

Declaración de variables implícita y explícita.

Fortran dispone de una (desafortunada) característica cual es la *declaración implícita de tipos*: nombres de variables que comienzan con las letras en el rango `a-h`, `o-z` se consideran variables reales, mientras que el resto, es decir las que comienza con las letras en el rango `k-n`, se consideran variables enteras. Este estilo de programación es altamente desaconsejado ya que es una fuente continua de errores. Por el contrario, nosotros propiciamos la declaración explícita de todas las variables utilizadas en el programa. Mas aún, con el fin de forzar al compilador la inhabilitación de la posibilidad de declaraciones implícitas utilizamos la sentencia `implicit none` al comienzo de la declaración de tipo. Con esta sentencia el uso de variables no declaradas generará un error de compilación.

Palabras reservadas.

En la mayoría de los lenguajes de programación las palabras que constituyen instrucciones del lenguaje no pueden ser utilizadas como nombres de variables (se dice que son *palabras reservadas*). En Fortran, sin embargo, *no* existen palabras reservadas y, por lo tanto, por ejemplo, es posible dar el nombre `stop` a una variable. Esta forma de proceder, sin embargo, no es recomendable porque puede introducir efectos no deseados.

Finalmente, los parámetros permiten asignar un nombre simbólico a una constante literal y, como tal, no podrán ser alterados. Esto resulta útil para nombrar números irracionales tales como π o constantes físicas como la velocidad de la luz c que aparecen repetidamente, y también para reconfigurar valores que pueden intervenir en un algoritmo. La declaración de un parámetro implica declarar primero su tipo y luego asignar su valor a través de la siguiente sentencia:

`parameter (`*nombre* `=`*constante*`)`

En el estilo de programación que estamos desarrollando las sentencias `parameter` se colocan junto a las de declaración de tipo, antes de cualquier sentencia ejecutable. Téngase presente también que los valores de los parámetros son asignados en tiempo de compilación y no en tiempo de ejecución del programa.

Símbolo	Significado	Ejemplo
+	adición	`a+b`
-	sustracción	`a-b`
	opuesto	`-b`
/	división	`a/b`
*	multiplicación	`a*b`
**	potenciación	`a**b`

Tabla 2. Codificación de los operadores aritméticos básicos

4. Sentencias de asignación.

La *sentencia de asignación* permite asignar (almacenar) un valor en una variable. La operación de asignación se indica tanto en el pseudocódigo de nuestros algoritmos como en un programa Fortran en la forma general

variable = *expresión*

Aquí el signo = no debe ser interpretado como el signo de igualdad matemático, sino que representa la operación en la cual el valor de la expresión situada a la derecha se almacena en la variable situada a la izquierda. Por *expresión* entendemos aquí un conjunto de variables o constantes conectadas entre sí mediante los operadores que permiten sus tipos.

Nota 1. El tipo de dato correspondiente a la expresión debe ser el mismo tipo de dato correspondiente a la variable. Para tipos numéricos si éste no es el caso ciertas conversiones de tipo implícitas se realizan (las cuales serán discutidas enseguida).

Nota 2. La operación de asignación es una operación *destructiva* para la variable del miembro de la izquierda, debido a que cualquier valor almacenado previamente en dicha variable se pierde y se sustituye por el nuevo valor. Por el contrario, los valores de cualesquiera variables que se encuentren en la expresión del miembro de la derecha de la asignación no cambian sus valores.

Nota 3. En una sentencia de asignación cualquier cosa distinta de un nombre de variable en el miembro de la izquierda conduce a una sentencia incorrecta.

Entre los tipos de expresiones que podemos considerar nos interesan especialmente las *expresiones aritméticas*, las cuales son un conjunto de datos *numéricos* (variables y constantes) unidos por *operadores aritméticos* y *funciones* sobre los mismos. Cuando una expresión aritmética se evalúa el resultado es un dato numerico y por lo tanto puede ser asignada una variable de tipo numérico a través de una sentencia de asignación aritmética. Los operadores aritméticos son los usuales, cuya codificación en Fortran se indica en la tabla 2.

Pero además Fortran dispone de un conjunto de *funciones intrínsecas* que implementan una gran variedad de funciones matemáticas, algunas de las cuales se presentan en la tabla 3. Para utilizar una función específica se emplea el nombre de la función seguido por la expresión sobre la que se va a operar (argumento) dentro de un juego de paréntesis. Por ejemplo, para calcular el valor absoluto de `x` y asignarlo a `y` escribimos `y = abs(x)`.

Funciones implícitas del compilador gfortran.

Una lista de todas las funciones implícitas que proporciona el compilador `gfortan` puede verse en la página info del mismo, ejecutando en una terminal el comando:

```
$ info gfortran
```

y dirigiéndonos a la sección titulada `Intrinsic Procedures`. (Para salir de la página info simplemente presionamos la tecla `q`).

`* = sin(rd)`	seno del ángulo en radianes
`* = cos(rd)`	coseno del ángulo en radianes
`* = tan(rd)`	tangente del ángulo en radianes
`* = asin(rd)`	arco seno (en el rango $-\pi/2$ a $+\pi/2$)
`* = acos(rd)`	arco coseno (en el rango 0 a $+\pi$)
`* = atan(rd)`	arco tangente (en el rango $-\pi/2$ a $+\pi/2$)
`* = atan2(rd,rd)`	arco tangente de arg1/arg2 (en el rango $-\pi$ a π)
`* = sqrt(rd)`	raíz cuadrada
`* = exp(rd)`	función exponencial
`* = log(rd)`	logaritmo natural
`* = log10(rd)`	logaritmo decimal
`* = abs(ird)`	valor absoluto
`* = mod(ird,ird)`	resto de la división de arg1 por arg2
`i = int(ird)`	convierte a un tipo entero truncando la parte decimal
`r = real(ird)`	convierte a tipo real
`d = dble(ird)`	convierte a tipo doble precisión

Tabla 3. Funciones intrínsecas básicas proporcionadas por Fortran. El tipo del dato del argumento y del resultado que admiten es indicado por una letra: `i` = entero, `r` = real, `d` = doble precisión. Un asterisco en el miembro de la derecha indica que el resultado es del mismo tipo que el argumento.

Orden de precedencia de las operaciones aritméticas. Cuando en una expresión aparecen dos o más operadores se requiere de un *orden de precedencia* de las operaciones que permita determinar el orden en que se realizaran. En Fortran estas reglas son las siguientes:

1) Todas las subexpresiones entre paréntesis se evalúan primero. Las expresiones con paréntesis anidados se evalúan de adentro hacia fuera: el paréntesis más interno se evalúa primero.
2) Dentro de una misma expresión o subexpresión, las funciones se evalúan primero y luego los operadores se evalúan en el siguiente orden de prioridad: potenciación; multiplicación y división; adición, substracción y negación.
3) Los operadores en una misma expresión o subexpresión con igual nivel de prioridad se evalúan de izquierda a derecha, con excepción de la potenciación, que se evalúa de derecha a izquierda.

Así, por ejemplo, `a + b*c/d` es equivalente a `a + ((b*c)/d)`, mientras que `x**y**z` es equivalente a `x**(y**z)`.

Conversión implícita de tipos. Cuando en un operador aritmético los dos operandos tienen el mismo tipo de dato numérico, el resultado tiene el mismo tipo que éstos. En particular, *la división de dos tipos enteros es un entero*, mientras que la división de dos tipos reales es un real. Si, en cambio, los operandos tienen tipos numéricos distintos, una *conversión de tipo implícita* se aplica, llevando el tipo de uno de ellos al otro, siguiendo la siguiente dirección: enteros se convierten a reales, reales se convierten a doble precisión. La excepción a esta regla es la potenciación: cuando la potencia es un entero el cálculo es equivalente a la multiplicación repetida (por ejemplo, con `x` real o de doble precisión `x**2 = x*x`) Sin embargo, si la potencia es de otro tipo numérico el cómputo se realiza implícitamente a través de las funciones logarítmica y exponencial (por ejemplo, `x**2.0` es calculado como $e^{2.0 \log x}$). De la misma manera una conversión de tipo implícita se realiza cuando en una sentencia aritmética la expresión y la variable difieren en tipo numérico, en tal caso, la expresión después de ser evaluada es convertida al tipo de la variable a la que se asigna el valor. Ciertamente, estas conversiones implícitas, producto de la "mezcla" de tipos en una misma expresión o sentencia debe ser consideradas con mucho cuidado. En expresiones complejas conviene hacerlas explícitas a través de las apropiadas *funciones intrínsecas de conversión de tipo* que se detallan en la tabla 3.

5. Entrada y salida de datos por lista.

Si se desea utilizar un conjunto de datos de entrada diferentes cada vez que se ejecuta un programa debe proporcionarse un método para leer dichos datos. De manera similar, para visualizar los resultados del programa debe proporcionarse un mecanismo para darles salida. El conjunto de instrucciones para realizar estas operaciones se conocen como *sentencias de entrada/salida*. En los algoritmos tales instrucciones las describimos en pseudocódigo como

```
Leer lista de variables de entrada.
Imprimir lista de variables de salida.
```

Existen dos modos básicos para ingresar datos en un programa: *interactivo* y por *archivos*. Aquí solo discutiremos el primero, dejando el segundo para una sección posterior. En el modo interactivo el usuario ingresa los datos por teclado mientras ejecuta el programa. La sentencia Fortran apropiada para ésto es

`read(*,*)` *lista de variables*

donde la lista de variables, si contiene más de un elemento, está separada por comas. Con el fin de guiar al usuario en la entrada de datos interactiva es conveniente imprimir un mensaje indicativo previo a la lectura de los datos. Por ejemplo,

```
write(*,*) 'Ingrese radio del círculo'
read(*,*) radio
```

Para dar salida a los datos por pantalla utilizamos la sentencia

`write(*,*)` *lista de variables*

Nuevamente podemos utilizar constantes literales de carácter para indicar de que trata el resultado obtenido. Por ejemplo,

```
write(*,*) 'Area del círculo = ', area
```

6. Estructuras de control

Las *estructuras de control* permiten especificar el orden en que se ejecutaran las instrucciones de un algoritmo. Todo algoritmo puede diseñarse combinando tres tipos básicos de estructuras de control:

1. *secuencial*: las instrucciones se ejecutan sucesivamente una después de otra,
2. *de selección*: permite elegir entre dos conjuntos de instrucciones dependiendo del cumplimiento (o no) de una condición,
3. *de iteración*: un conjunto de instrucciones se repite una y otra vez hasta que se cumple cierta condición.

Combinando estas tres estructuras básicas es posible producir un flujo de instrucciones más complejo pero que aún conserve la simplicidad inherente de las mismas. La implementación de un algoritmo en base a estos tres tipos de estructuras se conoce como *programación estructurada* y este estilo de programación conduce a programas más fáciles de escribir, leer y modificar.

Estructura secuencial. La estructura de control más simple esta representada por una sucesión de operaciones donde el orden de ejecución coincide con la aparición de las instrucciones en el algoritmo (o código del programa). La figura 3 ilustra el diagrama de flujo correspondiente a esta estructura. En Fortran una estructura secuencial es simplemente un conjunto de sentencias simples unas después de otra.

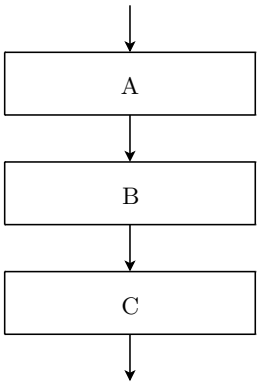

Figura 3. Estructura secuencial.

Estructura de selección. La *estructura de selección* permite que dos conjuntos de instrucciones alternativas puedan ejectuarse según se cumpla (o no) una determinada condición. El pseudocódigo de esta estructura es descrito en la forma *si ... entonces ... sino ...*, ya que si p es una condición y A y B respectivos conjuntos de instrucciones, la selección se describe como *si* p es verdadero *entonces* ejecutar las instrucciones A, *sino* ejecutar las instrucciones B. Codificamos entonces esta estructura en pseudocódigo como sigue.

```
Si condición entonces
      instrucciones para condición verdadera
sino
      instrucciones para condición falsa
fin_si
```

El diagrama de flujo correspondiente se ilustra en la figura 4. En Fortran su implementación tiene la siguiente sintaxis:

```
if (condición) then
      sentencias para condición verdadera
else
      sentencias para condición falsa
endif
```

Sangrado ("indentación")

Mientras que la separación de líneas en la codificación de una estructura de control es sintácticamente necesaria, el sangrado en los conjuntos de sentencias es opcional. Sin embargo, la sangría favorece la legibilidad del programa y, por lo tanto, constituye una buena práctica de programación.

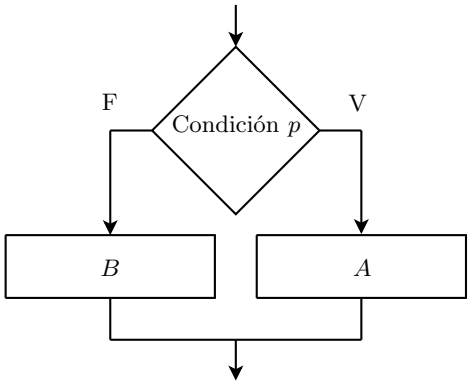

Figura 4. Estructura de selección.

Operador	Significado
`.lt.`	$<$ menor que (*less than*)
`.gt`	$>$ mayor que (*greater than*)
`.eq.`	$=$ igual a (*equal to*)
`.le.`	$\leq$ menor o igual que (*less than o equal to*)
`.ge.`	$\geq$ mayor o igual que (*grater than o equal to*)
`.ne.`	$\neq$ distinto a (*not equal to*)

Tabla 4. Operadores relacionales.

Operador	Significado
`.not.`	negación
`.and.`	conjunción
`.or.`	disyunción (inclusiva)
`.eqv.`	equivalencia

Tabla 5. Operadores lógicos.

La condición en la estructura de selección es especificada en Fortran por una *expresión lógica*, esto es, una expresión que devuelve un dato de tipo lógico: verdadero (`.true.`) o falso (`.false.`). Una expresión lógica puede formarse comparando los valores de expresiones aritméticas utilizando *operadores relacionales* y pueden combinarse usando *operadores lógicos*. El conjunto de operadores relacionales involucra a las relaciones de igualdad, desigualdad y de orden, las cuales son codificadas en Fortran como se indica en la tabla 4. Por otro lado, los operadores lógicos básicos son la *negación*, la *conjunción*, la *disyunción* (inclusiva) y *equivalencia*, cuya codificación en Fortran se indica en la tabla 5. El operador `.not.` indica la negación u opuesto de la expresión lógica. Una expresión que involucra dos operandos unidos por el operador `.and.` es verdadera si ambas expresiones son verdaderas. Una expresión con el operador `.or.` es verdadera si uno cualquiera o ambos operandos son verdaderos. Finalmente en el caso de equivalencia lógica la expresión es verdadera si ambos operandos conectados por el operador `.eqv.` son ambos verdaderos.[4].

Cuando en una expresión aparecen operaciones aritméticas, relacionales y lógicas, el *orden de precedencia* en la evaluación de las operaciones es como sigue:

1. Operadores aritméticos.
2. Operadores relacionales.
3. Operadores lógicos, con prioridad: `.not.`, `.and.` y `.or.`, `.eqv.`.

Operaciones que tienen la misma prioridad se ejecutan de izquierda a derecha. Por supuesto, las prioridades pueden ser modificadas mediante el uso de paréntesis.

En muchas circunstancias se desea ejecutar un conjunto de instrucciones sólo si la condición es verdadera y no ejecutar ninguna instrucción si la condición es falsa. En tal caso, la estructura de control se simplifica, codificándose en pseudocódigo como sigue (y con un diagrama de flujo como se indica en la figura 5)

[4]Estos enunciados no son más que las conocidas *tablas de verdad* de la lógica matemática.

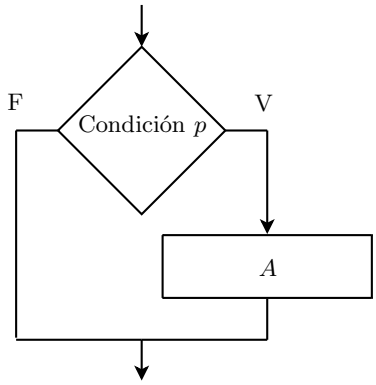

Figura 5. Estructura de selección sin sección *sino*.

```
Si condición entonces
      instrucciones para condición verdadera
fin_si
```

La sintaxis correspondiente en Fortran es

```
if (condición) then
      sentencias para condición verdadera
endif
```

Si, además, sólo debe realizarse *una* sentencia ejecutable cuando la condición es verdadera, Fortran permite codificar esta situación en un *if lógico*:

```
if (condición) sentencia ejecutable
```

Otra circunstancia que se presenta es la necesidad de elegir entre más de una alternativa de ejecución. En este caso podemos utilizar la *estructura multicondicional* cuya lógica podemos expresar como *si ... entonces sino si ... sino ...*, y cuyo pseudocódigo es descrito como sigue:

```
Si condición 1 entonces
      instrucciones para condición 1 verdadera
sino si condición 2 entonces
      instrucciones para condición 2 verdadera
...
sino si condición N entonces
      instrucciones para condición N verdadera
sino
      instrucciones para todas las condiciones falsas
fin_si
```

El diagrama de flujo correspondiente se ilustra en la figura 6. Aquí, cada condición se prueba por turno. Si la condición no se se satisface, se prueba la siguiente condición, pero si la condición es verdadera, se ejecutan las instrucciones correspondientes para tal condición y luego se va al final de la estructura. Si ninguna de las condiciones son satisfechas, se ejecutan las instrucciones especificadas en el bloque correspondientes al *sino* final. Debería quedar claro entonces que para que un estructura condicional sea eficiente sus condiciones deben ser *mutuamente excluyentes*. La codificación de esta estructura en Fortran se indica a continuación.

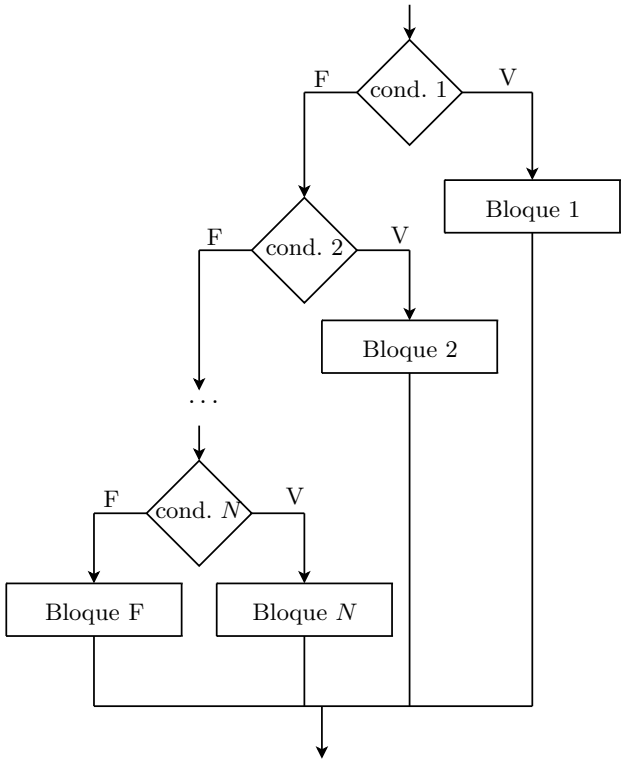

Figura 6. Estructura multicondicional.

```
if (condición 1) then
      sentencias para condición 1 verdadera
elseif (condición 2) then
      sentencias para condición 2 verdadera
...
elseif (condición N) then
      sentencias para condición N verdadera
else
      sentencias para todas las condiciones falsa
endif
```

Ejemplo. Consideremos el diseño e implementación de un algoritmo para calcular las raíces de una ecuación cuadrática

$$a\,x^2 + b\,x + c = 0,$$

esto es,

$$x_{\{1,2\}} = \frac{-b \pm \sqrt{b^2 - 4\,a\,c}}{2a}.$$

La naturaleza de estas raíces dependerá del signo del discriminante $\Delta = b^2 - 4\,a\,c$, lo cual nos lleva a implementar una estructura de decisión: si $\Delta > 0$ las raíces serán números reales, de lo contrario serán números complejos conjugados uno del otro. Por otra parte, al ingresar los coeficientes, deberíamos asegurarnos que el coeficiente a no es nulo, pues, de lo contario la fórmula conduciría a una división por cero. Nuevamente, una estructura de decisión permite verificar ésto. El pseudocódigo de nuestro algoritmo es el siguiente.

```
Leer a, b, c.
Si a = 0 entonces
      escribir 'La ecuación no es cuadrática',
      salir.
```

```
fin_si.
Calcular Δ = b^2 − 4 a c.
Si Δ > 0 entonces
      calcular x1 = (−b + √Δ)/2a,
      calcular x2 = (−b − √Δ)/2a,
de lo contrario
      calcular x1 = (−b + i√−Δ)/2a,
      calcular x2 = x̄1,
fin_si.
Imprimir x1 y x2.
```

A continuación damos una implementación de este algoritmo. Nótese que *no* utilizamos variables complejas sino dos variables reales que contendrán la parte real y la parte imaginaria de las posibles raíces complejas.

```
      program ecuadratica
*     ------------------------------------------------------------------
*     Se calculan las raíces de la ecuación cuadrática ax**2+bx+c=0
*     ------------------------------------------------------------------
*     Bloque de declaración de tipos
*     ------------------------------------------------------------------
      implicit none
      double precision a,b,c            ! coeficientes de la ecuación
      double precision discr            ! discriminante de la ecuación
      double precision x1,x2            ! variables para soluciones
*     ------------------------------------------------------------------
*     Entrada de datos
*     ------------------------------------------------------------------
      write(*,*) 'Ingrese coeficientes a,b,c'
      read(*,*) a,b,c
      if(a.eq.0.0d0) then
        write(*,*) 'La ecuación no tiene término cuadrático'
        stop
      endif
*     ------------------------------------------------------------------
*     Bloque de procesamiento y salida
*     ------------------------------------------------------------------
      discr = b**2 - 4.0d0*a*c
      den   = 2.0d0*a
      term  = sqrt(abs(discr))
      if (discr.ge.0.0d0) then
        x1 = (-b+term)/den
        x2 = (-b-term)/den
        write(*,*) 'x1 = ', x1
        write(*,*) 'x2 = ', x2
      else
        x1 = -b/den
        x2 = term/den
        write(*,*) 'x1 = (', x1, ' ,',  x2, ')'
        write(*,*) 'x1 = (', x1, ' ,', -x2, ')'
      endif
*     ------------------------------------------------------------------
*     Terminar
*     ------------------------------------------------------------------
      stop
      end
```

Estructura de iteración. La *estructura de control iterativa* permite la repetición de una serie determinada de instrucciones. Este conjunto de instrucciones a repetir se denomina *bucle* (*loop*,

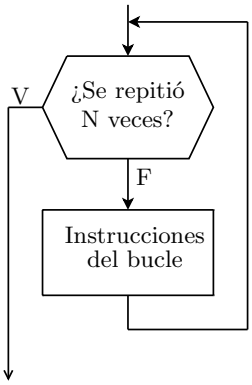

Figura 7. Estructura de iteración.

en inglés) y cada repetición del mismo se denomina *iteración*. Podemos diferenciar dos tipos de bucles:

1. Bucles donde el número de iteraciones es fijo y conocido de antemano.
2. Bucles donde el numero de iteraciones es desconocido de antemano. En este caso el bucle se repite mientras se cumple una determinada condición (*bucles condicionales*).

Para estos dos tipos de bucles disponemos de sendas formas básicas de la estructura de control iterativa.

Comencemos con un bucle cuyo número de iteraciones es conocido *a priori*. La estructura iterativa, en tal caso, puede expresarse en pseudocódigo como sigue.

```
Desde indice = valor inicial hasta valor final hacer
      instrucciones del bucle
fin_desde
```

El diagrama de flujo correspondiente se ilustra en la figura 7.

Nota 1. *indice* es una variable entera que, actuando como un *contador*, permite controlar el número de ejecuciones del ciclo.

Nota 2. *valor inicial* y *valor final* son valores enteros que indican los límites entre los que varía *indice* al comienzo y final del bucle.

Nota 3. Está implícito que en cada iteración la variable *índice* toma el siguiente valor, incrementándose en una unidad. En seguida veremos que en Fortran se puede considerar incrementos mayores que la unidad e incluso negativos.

Nota 4. El número de iteraciones del bucle es $N =$ *valor final* - *valor inicial* $+1$.

Nota 5. Dentro de las instrucciones del bucle *no es legal* modificar la variable *indice*. Asimismo, al terminar todas las iteraciones el valor de la variable no tiene porque tener un valor definido, por lo tanto, la utilidad de la variable *indice* se limita a la estructura de iteración.

En Fortran la estructura repetiva se codifica como sigue.

```
do indice = valor inicial, valor final, incremento
      sentencias del bucle
enddo
```

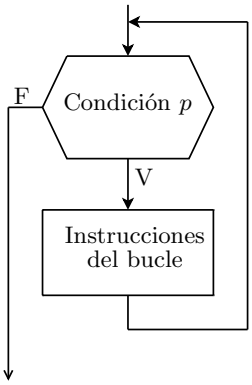

Figura 8. Estructura de iteración condicional.

Aquí *indice* es una variable entera, mientras que *valor inicial*, *valor final* e *incremento* pueden ser variable, constantes o expresiones enteras y pueden ser negativos. Si *incremento* no se especifica se asume igual a la unidad. El número de iteraciones del bucle está dado por

$$N = \text{máx}\{\, [\, (\textit{valor final} - \textit{valor inicial} + \textit{incremento})/\textit{incremento}\,]\, , 0\, \},$$

donde [ ] denota tomar la parte entera, descartando la fracción decimal. Sólo si N es mayor que 0 se ejecutará el bucle.

Consideremos, finalmente, los bucles condicionales. Aquí, el número de iteraciones no es conocido *a priori*, sino que el bucle se repite *mientras* se cumple una determinada condición. En este caso, la estructura iterativa se describe en pseudocódigo de la siguiente forma (y su diagrama de flujo se ilustra en la figura 8)

```
Mientras condición hacer
      instrucciones del bucle
fin_mientras
```

Nota 1. La condición se evalúa antes y después de cada iteración del bucle. Si la condición es verdadera las instrucciones del bucle se ejecutarán y, si es falsa, el control pasa a la instrucción siguiente al bucle.

Nota 2. Si la condición es falsa cuando se ejecuta el bucle por primera vez, las instrucciones del bucle no se ejecutarán.

Nota 3. Mientas que la condición sea verdadera el bucle continuará ejecutándose indefinidamente. Por lo tanto, para terminar el bucle, en el interior del mismo debe tomarse alguna acción que modifique la condición de manera que su valor pase a falso. Si la condición nunca cambia su valor se tendrá un *bucle infinito*, la cual no es una situación deseable.

En Fortan un bucle condicional se codifica como sigue.

```
do while (condición)
      sentencias del bloque
enddo
```

donde *condición* es una expresión logica.

Ejemplos. Consideremos la determinación de la suma de los n primeros enteros positivos, esto es, el valor de $\sum_{i=1}^{n} i$. Aquí un bucle iterativo `do` permite calcular la suma puesto el número de términos a sumar es conocido de antemano. El siguiente pseudocódigo muestra el algoritmo que realiza la suma.

```
Leer n.
Iniciar suma = 0.
Desde i = 1 hasta n hacer
      Tomar suma = suma + i
fin_desde.
Escribir suma.
Terminar.
```

Nótese la inicialización a cero de la variable `suma` utilizada como acumulador para la suma pedida. La implementación en Fortran es como sigue.

```
      program sumar
*     ----------------------------------------------------------------------
*     Se calcula la suma de los n primeros enteros positivos
*     ----------------------------------------------------------------------
*     Bloque de declaración de tipos
*     ----------------------------------------------------------------------
      implicit none
      integer n       ! número de términos a sumar
      integer suma    ! valor de la suma
      integer i
*     ----------------------------------------------------------------------
*     Entrada de datos
*     ----------------------------------------------------------------------
      write(*,*) 'Ingrese el número de enteros positivos a sumar'
      read(*,*) n
*     ----------------------------------------------------------------------
*     Bloque de procesamiento
*     ----------------------------------------------------------------------
      suma = 0
      do i=1,n
         suma = suma + i
      enddo
*     ----------------------------------------------------------------------
*     Salida de datos
*     ----------------------------------------------------------------------
      write(*,*) 'Suma = ', suma
*     ----------------------------------------------------------------------
*     Terminar
*     ----------------------------------------------------------------------
      stop
      end
```

Considérese ahora el problema de determinar el primer valor n para el cual la suma $\sum_{i=1}^{n}$ excede a 10 000. En este problema *no* puede utilizarse un bucle `do` ya que el número de términos a sumar no es conocido de antemano. Es claro, entonces, que debemos utilizar un bucle condicional, tal como se muestra en el siguiente pseudocódigo.

```
Iniciar n = 0.
Iniciar suma = 0.
Mientras suma <10000 hacer
      Tomar n = n +1
      Tomar suma = suma + n
fin_mientras.
Escribir 'El número de términos necesarios es', n.
Terminar.
```

La implementación en Fortran de este algoritmo se muestra a continuación.

```
      program sumar
```

```
*     ------------------------------------------------------------------
*     Se determina el primer valor de n para el cual la suma de los n
*     primeros enteros positivos excede a 10000.
*     ------------------------------------------------------------------
*     Bloque de declaración de tipos
*     ------------------------------------------------------------------
      implicit none
      integer n       ! número de términos a sumar
      integer suma    ! valor de la suma
*     ------------------------------------------------------------------
*     Bloque de procesamiento
*     ------------------------------------------------------------------
      suma = 0
      n    = 0
      do while(suma.lt.10000)
         n    = n +1
         suma = suma + n
      enddo
*     ------------------------------------------------------------------
*     Salida de datos
*     ------------------------------------------------------------------
      write(*,*) 'Número de términos utilizados ', n
*     ------------------------------------------------------------------
*     Terminar
*     ------------------------------------------------------------------
      stop
      end
```

7. Modularización: subrutinas y funciones

Frente a un problema complejo una de las maneras más eficientes de diseñar (e implementar) un algoritmo para el mismo consiste en descomponer dicho problema en *subproblemas* de menor dificultad y éstos, a su vez, en subproblemas más pequeños y así sucesivamente hasta cierto grado de refinamiento donde cada subproblema involucre una sola tarea específica bien definida y, preferiblemente, independiente de los otros. El problema original es resuelto, entonces, combinando apropiadamente las soluciones de los subproblemas.

En una implementación computacional, cada subproblema es implementado como un *subprograma*. De este modo el programa consta de un *programa principal* (la unidad del programa de nivel más alto) que llama a subprogramas (unidades del programa de nivel más bajo) que a su vez pueden llamar a otros subprogramas. La figura 9 representa esquemáticamente la situación en un diagrama conocido como *diagrama de estructura*. Por otra parte en un diagrama de flujo del algoritmo, un subprograma es representado como se ilustra en la figura 10 el cual permite indicar que la estructura exacta del subprograma será detallada aparte en su respectivo diagrama de flujo.

Este procedimiento de dividir el programa en subprogramas más pequeños se denomina *programación modular* y la implementación de un programa en subprogramas que van desde lo más genérico a lo más particular por sucesivos refinamientos se conoce como *diseño descendente* (*top-down*, en inglés).

Un diseño modular de los programas provee la siguientes ventajas:

- El programa principal consiste en un resumen de *alto nivel* del programa. Cada detalle es resuelto en los subprogramas.
- Los subprogramas pueden ser planeados, codificados y comprobados independientemente unos de otros.
- Un subprograma puede ser modificado internamente sin afectar al resto de los subprogramas.

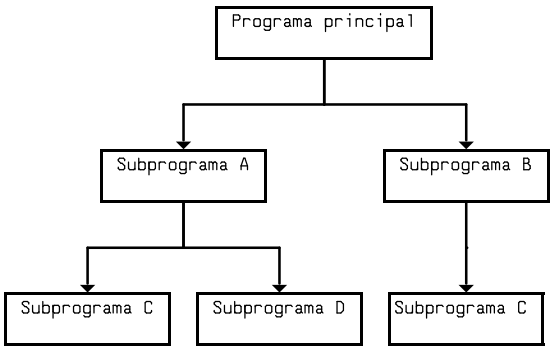

Figura 9. Diagrama de estructura de un algoritmo o programa modularizado.

Figura 10. Representación de un subprograma en un diagrama de flujo.

- Un subprograma, una vez escrito, pueden ser ejecutado todas las veces que sea necesario a través de una invocación al mismo.
- Una vez que un subprograma se ha escrito y comprobado, se puede utilizar en otro programa (*reusabilidad*).

Fortran provee dos maneras de implementar un subprograma: *funciones* y *subrutinas*. Estos subprogramas pueden ser *intrínsecos*, esto es, provistos por el compilador o *externos*. Ya hemos mencionado (y utilizado) las funciones intrínsecas que provee el propio lenguaje. Por otra parte los subprogramas externos pueden ser escritos por el propio usuario o bien formar parte de un paquete o *biblioteca* de subprogramas (*library*, en inglés) desarrollado por terceros. En lo que sigue consideraremos la creación de subprogramas propios. Para ello presentamos, como ejemplo, una versión modularizada del código para el cálculo del área de un círculo.

```
      program principal
*     ---------------------
*     Declaración de tipos
*     ---------------------
      implicit none
      real radio,area
      real calcular_area
*     ---------------------
*     Invocar subprogramas
*     ---------------------
      call leer_radio(radio)
      area = calcular_area(radio)
      call imprimir_area(area)
*     ---------------------
*     Terminar
*     ---------------------
      stop
      end
```

```
      subroutine leer_radio(radio)
*     ---------------------
*     Declaración de argumentos formales
*     ---------------------
      implicit none
      real radio
*     ---------------------
*     Ingreso de datos
*     ---------------------
      write(*,*) 'Ingrese radio'
      read(*,*) radio
*     ---------------------
*     Retornar
*     ---------------------
      return
      end
```

```
      real function calcular_area(radio)
*     ---------------------
*     Declaración de argumentos formales
*     ---------------------
      implicit none
      real radio
*     ---------------------
*     Declaración de variables
*     ---------------------
      real pi
      parameter (pi = 3.14159)
*     ---------------------
*     Calculo de la función
*     ---------------------
      calcular_area = pi*radio**2
*     ---------------------
*     Retornar
*     ---------------------
      return
      end
```

```
      subroutine imprimir_area(a)
*     ---------------------
*     Declaración de argumentos formales
*     ---------------------
      implicit none
      real a
*     ---------------------
*     Salida de datos
*     ---------------------
      write(*,*) 'Area =', a
*     ---------------------
*     Retornar
*     ---------------------
      return
      end
```

Como vemos subrutinas y funciones comparten características comunes, por cuanto son subprogramas, pero poseen también ciertas diferencias. Desde el punto de vista de la implementación de un subprograma, *la diferencia fundamental entre una función y una subrutina es que las funciones permiten devolver un único valor a la unidad del programa* (programa principal o subprograma) *que la invoca mientras que una subrutina puede devolver cero, uno o varios valores*. La forma sintáctica de una función y una subrutina es como sigue:

tipo `function` *nombre* (*argumentos*)
  declaraciones de tipo
  sentencias
  nombre = expresión
  `return`
`end`

`subroutine` *nombre* (*argumentos*)
  declaraciones de tipo
  sentencias
  `return`
`end`

A continuación detallamos las características comunes y distintivas de subrutinas y funciones.

Nota 1. Un programa siempre tiene uno y sólo un programa principal. Subprogramas, ya como subrutinas o funciones, pueden existir en cualquier número.

Nota 2. Cada subprograma es por sí mismo una unidad de programa independiente con sus propias variables, constantes literales y parámetros. Por lo tanto cada unidad tiene sus respectivas sentencias `implicit none` y de declaraciones de tipo. Ahora, mientras que el comienzo de un programa principal es declarado con la sentencia `program`, el comienzo de una subrutina está dado por la sentencia `subroutine` y, para una función, por la sentencia `function`. Cada una de estas unidades del programa se extiende hasta su respectiva sentencia `end`, la cual indica su fin lógico al compilador.

Nota 3. Los subprogramas deben tener un nombre que los identifique. El nombre escogido debe seguir las reglas de todo identificador en Fortran. Ahora bien, como una función devuelve un valor, *el nombre de una función tiene un tipo de dato asociado* el cual debe, entonces, ser declarado tanto en la sentencia `function` como en la unidad de programa que la invoca (como se observa en nuestro código para la función `calcular_area`). Por el contrario, al nombre de una subrutina no se le puede asignar un valor y por consiguiente *ningún tipo de dato está asociado con el nombre de una subrutina.*

Nota 4. **Alcance de los identificadores**: en Fortran 77 los nombres del programa principal y de los subprogramas son *globales*, esto es, conocidos por todas las unidades del programa. Por lo tanto, tales

nombres deben ser únicos a lo largo de todo el programa. Los restantes identificadores (correspondientes a nombres de variables, parámetros y funciones intrínsecas) dentro de una unidad de programa son *locales* al mismo. Esto significa que una unidad de programa desconoce la asociación de un identificador con un objeto local de *otra* unidad y por lo tanto el mismo nombre pueden ser utilizado en ambas para designar datos independientes. (Por ejemplo, en nuestro código, el parámetro `pi` sólo es conocido por la función `calcular_area`, mientras que el identificador `radio` utilizado en el programa principal y dos de los subprogramas se refieren a variables distintas).

Nota 5. El nombre del subprograma es utilizado para la invocación del mismo. Ahora bien, una función es invocada utilizando su nombre como *operando* en una expresión dentro de una sentencia, mientras que una subrutina se invoca en una sentencia específica que utiliza la instrucción `call`.

Nota 6. Al invocar un subprograma el control de instrucciones es transferido de la unidad del programa que realiza la invocación al subprograma. Entonces las respectivas instrucciones del subprograma se ejecutan hasta que se alcanza una sentencia `return`, momento en el cual el control vuelve a la unidad del programa que realizó la invocación. Ahora bien, en la invocación de una función el control de instrucciones retorna a la misma sentencia que realizó el llamado, mientras que en una subrutina el control retorna a la sentencia siguiente a la del llamado.

Nota 7. La forma de compartir información entre una unidad de programa y el subprograma invocado es a través de una *lista de argumentos*. Los argumentos pueden ya sea pasar información de la unidad del programa al subprograma (*argumentos de entrada*), del subprograma hacia la unidad de programa (*argumentos de salida*) o bien en ambas direcciones (*argumentos de entrada/salida*). Los argumentos utilizados en la declaración de una subrutina o función son conocidos como *argumentos formales* o *ficticios* y consisten en una lista de nombres simbólicos separados por comas y encerrados entre paréntesis. La lista puede contar con cualquier número de elementos, inclusive ninguno, pero, por supuesto, no puede repetirse ningún argumento formal. Si no hay ningún argumento entonces los paréntesis pueden ser omitidos en las sentencias de llamada y definición de una subrutina, pero para una función, por el contrario, siempre deben estar presentes. En Fortran 77 la distinción entre argumentos de entrada, salida y entrada/salida no es especificada de ningún modo en la lista de argumentos formales, sólo la forma en que se usan indican la diferencia. En la práctica, sin embargo, es una buena costumbre listar en una subrutina primero los argumentos formales que serán de entrada, luego los de entrada/salida y finalmente los de salida. *En una función los argumentos formales serán utilizados solamente como argumentos de entrada*, es el nombre de la función en sí mismo el que es utilizado para devolver un valor a la unidad de programa que lo invocó. De este modo, el nombre de una función puede ser utilizada como una variable dentro de la misma y debe ser asignada a un valor antes de que la función devuelva el control. (En nuestro ejemplo, el argumento formal `radio` en la definición de la subrutina `leer_radio` actúa como un argumento de salida, mientras que el argumento formal `a`, de la subrutina `imprimir_area`, actúa como un argumento de entrada. Por su parte en la función `calcular_area`, vemos que el argumento formal `radio` actúa efectivamente como un dato de entrada y que la devolución del valor de la función es dada por una sentencia en donde se asigna el valor apropiado a `calcular_area`). Por razones de estilo y claridad es costumbre también separar la declaración de tipo de los argumentos formales de las declaraciones de variables locales.

Los argumentos que aparecen en una invocación de un subprograma son conocidos como *argumentos actuales*. La asociación entre argumentos actuales y formales se realiza cada vez que se invoca el subprograma y de este modo se transfiere la información entre la unidad del programa y el subprograma. *La correspondencia entre los argumentos actuales y los formales se basa en la posición relativa que ocupan en la lista*. No existe correspondencia a través de los nombres (esto es, los nombres de variables en los argumentos formales y actuales no deben ser necesariamente los mismos, por ejemplo, en nuestro código el argumento formal `a` de la subrutina `imprimir_area` se corresponde con el argumento actual `area` del programa principal a través de la llamada correspondiente). Pero, *el tipo de dato de un argumento actual debe coincidir con el tipo de dato del argumento formal correspondiente*. Además, en Fortran 77 debe haber el mismo número de argumentos actuales y formales (argumentos *opcionales* no son permitidos).

Verificar siempre la correspondencia de tipos.

Debido a que las unidades del programa son independientes es difícil para el compilador revisar errores de tipo en la correspondencia de la lista de argumentos formales con la lista de argumentos actuales. Es nuestra responsabilidad asegurarnos que tal correspondencia de tipos es correcta.

Un argumento formal de salida (y de entrada/salida) es una variable en el subprograma cuyo valor será asignado dentro del mismo y sólo puede corresponderse con un argumento actual que sea una variable (del

mismo tipo, por supuesto) en la unidad de programa que invoca al subprograma. Un argumento formal de entrada, por otra parte, es una variable que preserva su valor a través de todo el subprograma y puede corresponderse a un argumento actual de la unidad del programa que pueden ser no sólo una variable sino también a una expresión (incluyendo una constante). Si al argumento actual es una expresión, ésta es evaluada antes de ser transferida. Nótese que una variable y una constante pueden ser transformadas en un argumento formal de tipo expresión encerrándola entre paréntesis.

Pasaje por referencia.

En programación existen varias alternativas para implementar la manera en la cual los argumentos actuales y formales son transmitidos y/o devueltos entre las unidades de programa. Fortran 77, independientemente de si los argumentos son de entrada, salida o entrada/salida, utiliza el *paradigma de pasaje por referencia*. En este método en vez de pasar los valores de los argumentos a la función o subrutina (*pasaje por valor*), se pasa la dirección de memoria de los argumentos. Esto significa que el argumento actual y formal comparten la misma posición de memoria y por lo tanto, cualquier cambio que realice el subprograma en el argumento formal es *inmediatamente* "visto" en el argumento actual. Debido a ésto debe tenerse especial cuidado en no introducir *efectos laterales* (*side-effects*) no deseados: el pasaje por referencia permite que un argumento de entrada, que en un subprograma debe ser una variable cuyo valor no cambie, pueda ser, de hecho, cambiada. Esto debe ser *evitado siempre*, ya que el nuevo valor se propagará en el programa con un valor no esperado.

Nota 8. Fortran permite que un subprograma sea utilizado como un argumento actual en la llamada de otro subprograma, pero para ello el nombre del subprograma debe ser declarado dentro de la unidad que realiza la invocación en una sentencia `external` si el subprograma que se pasa como argumento es una función o subrutina externa, o en una sentencia `intrinsic` si, por el contrario, es intrínseca. En este último caso el nombre de la función intrínseca en el argumento actual debe ser su nombre *específico* y no su nombre genérico. *Esta es la única circunstancia en la cual debe utilizarse los nombres específicos de las funciones intrínsecas.*

Nota 9. Fortran 77 no permite subprogramas *recursivos*. Esto es, un subprograma no puede, dentro de su cuerpo, llamarse a sí mismo directa o indirectamente. Por ejemplo, refiriéndonos al diagrama de estructura de la figura 9, el subprograma `C` no podría llamar a su vez al subprograma `A`. Asociado a ésto se tiene la restricción de que funciones que en su cuerpo utilizan sentencias de entrada/salida no deberían ser invocadas en expresiones que involucren sentencias de entrada/salida. Esto evita que las operaciones de entrada/salida sean utilizadas recursivamente.

Nota 10. Bajo circunstancias normales, las variables locales de un subprograma resultan en un estado *indefinido* tan pronto como el control vuelve a la unidad de programa que lo llamó. La sentencia `save` puede ser utilizada en aquellas (raras) circunstancias en que debe asegurarse que las variables locales preserven sus valores entre sucesivas llamadas al subprograma.

Compilación por separado de las unidades del programa. En la práctica parecería que las diversas unidades que conforman un programa deben ser guardadas en un único archivo fuente para proceder a su compilación. Ciertamente nada impide proceder siempre de esta forma. Sin embargo, es posible compilar el programa a partir de las distintas unidades que lo conforman cuando éstos son guardados en archivos fuente separados. Por ejemplo, consideremos nuevamente nuestro código modularizado para el cálculo del área del círculo. Asumiendo que el programa principal es guardado en el archivo fuente `area.f` mientras que los subprogramas son guardados en los archivos `leer-radio.f`, `calcular-area.f`, `imprimir-area.f` respectivamente, el programa ejecutable `area` puede ser compilado con la siguiente línea de comandos

```
$ gfortran -Wall -o area area.f leer-radio.f calcular-area.f imprimir-area.f
```

Más aún las distintas unidades pueden ser compiladas separadamente y luego unidas para generar el programa ejecutable. Para ello utilizamos el compilador con la opción `-c` sobre cada unidad del programa como sigue:

```
$ gfortran -Wall -c area.f
$ gfortran -Wall -c leer-radio.f
$ gfortran -Wall -c calcular-area.f
$ gfortran -Wall -c imprimir-area.f
```

Estos comandos generan, para cada archivo fuente, un nuevo archivo con el mismo nombre pero con extensión `.o`. Estos archivos, conocidos como *archivos objeto*, contienen las instrucciones de máquina que corresponden a los archivos fuente, pero con posiciones de memoria relativas en lugar de absolutas. El programa ejecutable resulta de la unión de los archivos objetos (procedimiento conocido como *linking*), lo cual se logra con la siguiente línea de comandos:

```
$ gfortran -o area area.o leer-radio.o calcular-area.o imprimir-area.o
```

Aunque esta forma de proceder puede parecer innecesaria para pequeños programas, resulta de gran versatilidad para la compilación de programas que son construidos a partir de un gran número de subprogramas. Esto se debe a que si se efectúan cambios sobre unas pocas unidades del programa, sólo ellas necesitan ser recompiladas. Por supuesto, para obtener el ejecutable final, el proceso de *linking* debe repetirse. Todo este proceso puede ser automatizado con herramientas apropiadas como ser `make`.

8. Entrada/salida por archivos

Hasta ahora hemos asumido que los datos que necesita un programa han sido entrados por teclado durante la ejecución del programa y que los resultados son mostrados por pantalla. Es claro que esta forma de proceder es adecuada sólo si la cantidad de datos de entrada/salida es relativamente pequeña. Para problemas que involucren grandes cantidades de datos resulta más conveniente que los mismos estén guardados en *archivos*. En lo que sigue veremos las instrucciones que proporciona Fortran para trabajar con archivos.

Un archivo es un conjunto de datos almacenado en un dispositivo (tal como un disco rígido) al que se le ha dado un nombre. Para la mayoría de las aplicaciones los únicos tipos de archivos que nos interesa considerar son los *archivos de texto*. Un archivo de texto consta de una serie de líneas o *registros* separadas por una marca de fin de línea (*newline*, en inglés). Cada línea consta de uno o más *datos* que es un conjunto de caracteres alfanuméricos que, en el procesamiento de lectura o escritura, se trata como una sola unidad. El acceso a los datos del archivo de texto procede en forma *secuencial*, esto es, se procesan línea por línea comenzando desde la primera línea hacia la última. Esto implica que no es posible acceder a una línea específica sin haber pasado por las anteriores.

Para fijar ideas consideremos el problema de calcular el baricentro de un conjunto de N puntos (x_i, y_i) del plano. Esto es, queremos computar las coordenadas del baricentro, definidas por

$$\bar{x} = \frac{\sum_{i=1}^{N} x_i}{N}, \qquad \bar{y} = \frac{\sum_{i=1}^{N} y_i}{N}.$$

Supongamos que las coordenadas de los N puntos se encuentran dispuestas en un archivo llamado `coordenadas.dat` que consta de una línea por cada punto, cada una de las cuales tiene dos columnas: la primera corresponde a la coordenada x y la segunda a la coordenada y. Así, este archivo consiste de N líneas, cada una de los cuales consta de dos datos de tipo real. Por otra parte, supondremos que el resultado $(\bar{x}, \bar{y})$ se quiere guardar en un archivo denominado `baricentro.sal`. El siguiente código Fortran efectúa lo pedido.

```
      program baricentro
*     ------------------------------------------------------
*     Declaración de variables
```

```
*     ------------------------------------------------------
      implicit none
      integer n,i
      real x,y,bar_x,bar_y
*     ------------------------------------------------------
*     Leer de la terminal el número de puntos
*     ------------------------------------------------------
      write(*,*) 'Ingrese el número de puntos'
      read (*,*) n
*     ------------------------------------------------------
*     Abrir archivo de datos
*     ------------------------------------------------------
      open(8,file='coordenadas.dat')
*     ------------------------------------------------------
*     Leer los datos (y procesarlos)
*     ------------------------------------------------------
      bar_x = 0.0
      bar_y = 0.0
      do i=1,n
         read(8,*) x,y
         bar_x = bar_x + x
         bar_y = bar_y + y
      end do
*     ------------------------------------------------------
*     Cerrar el archivo de datos
*     ------------------------------------------------------
      close(8)
*     ------------------------------------------------------
*     Calcular las coordenadas del baricentro
*     ------------------------------------------------------
      if(n.ne.0) then
        bar_x = bar_x/real(n)
        bar_y = bar_y/real(n)
      endif
*     ------------------------------------------------------
*     Abrir archivo de salida
*     ------------------------------------------------------
      open(9,file='baricentro.sal')
*     ------------------------------------------------------
*     Imprimir resultados en el archivo de salida
*     ------------------------------------------------------
      write(9,*) 'Cooordenadas del baricentro'
      write(9,*) 'x = ', bar_x
      write(9,*) 'y = ', bar_y
*     ------------------------------------------------------
*     Cerrar archivo de salida
*     ------------------------------------------------------
      close(9)
*     ------------------------------------------------------
*     Terminar
*     ------------------------------------------------------
      stop
      end
```

En base a este programa podemos detallar las características básicas de la entrada/salida por archivos.

Nota 1. En Fortran, un programa referencia indirectamente a un archivo a través de un *número de unidad lógica* (*lun*, del inglés *logic unit number*), el cual es un entero positivo pequeño (pero distinto de 5 y 6 pues

estas unidades están *pre-conectadas* a la entrada y salida estándar por teclado y pantalla, respectivamente). Así, para trabajar con un archivo, el primer paso consiste en establecer la relación entre el nombre del archivo y una unidad lógica. Esta conexión se realiza con la sentencia `open` y el proceso se conoce como *abrir* el archivo (en nuestro ejemplo, el número de unidad 8 es asignado al archivo `coordenadas.dat`, mientras que el número de unidad 9 es asignado al archivo `baricentro.sal`). Nótese que excepto por esta sentencia, los archivos son referidos dentro del programa a través de su unidad lógica y *no* por su nombre.

NOTA 2. Para poder leer o escribir datos de una línea del archivo, conectado a la unidad *número*, Fortran utiliza las sentencias `read` y `write`, respectivamente, en la forma

`read(`*número*`,*)` *variables*
`write(`*número*`,*)` *variables*

Cada dato tiene su correspondiente variable del tipo apropiado en la lista de variables.

En nuestro ejemplo la lectura procede en un bucle `do` desde el inicio hasta el final del archivo, avanzando línea por línea en cada lectura y asignando, cada vez, los dos datos del registro en sendas variables. Esta operación de lectura se comprende mejor introduciendo el concepto de *posición actual de línea*. A medida que el bucle `do` se ejecuta imaginemos que un *puntero* se mueve a través de las líneas del archivo de modo que la computadora conoce de cual línea se deben leer los datos. Comenzando con la primera línea, la primer sentencia `read` asigna a `x` e `y` los dos datos de dicha línea y luego *mueve el puntero a la siguiente línea*. Así, la segunda vez que la sentencia `read` es ejecutada los datos de la segunda línea son asignados a las variables `x` e `y`. El proceso se repite N veces hasta alcanzar el final del archivo. De manera similar, cada vez que se ejecuta una sentencia de escritura `write` se comienza en una nueva línea en el archivo.

NOTA 3. Así como un programa debe ser abierto para poder trabajar con el mismo, una vez finalizada la lectura o escritura el archivo debe ser *cerrado*, esto es, debe terminarse la conexión existente entre el archivo y la unidad lógica respectiva. Esta operación se realiza con la sentencia `close` seguida por el número de unidad entre paréntesis. Constituye una buena práctica de programación cerrar el archivo tan pronto no se necesita más. Nótese que mientras un archivo esté abierto su número de unidad no debe ser utilizado para abrir otro archivo. Sin embargo, una vez cerrado un archivo, el número de unidad correspondiente puede ser reutilizado. Por otra parte, un archivo que ha sido cerrado puede ser nuevamente abierto con una sentencia `open`. Todos los archivos que no han sido cerrados explícitamente con una sentencia `close` serán cerrados automáticamente cuando el programa termine (salvo que un error aborte el programa).

NOTA 4. Los números de unidades son un recurso *global*. Un archivo puede ser abierto en cualquier unidad del programa, y una vez abierto las operaciones de entrada/salida pueden ser realizadas por cualquier unidad del programa (en tanto se utilice el mismo número de unidad). Nótese que los números de unidades puede ser guardados en variables enteras y ser pasados a un subprograma como argumento. Por otra parte, un programa resultará más modular si en lugar de utilizar directamente los números de unidad en las sentencias de entrada/salida se utilizan constantes con nombres (esto es, parámetros) para referirnos a los mismos.

NOTA 5. Desde el punto de vista del programa los archivos se utilizan o bien para *entrada*, o bien para *salida*. Un archivo es de entrada cuando el programa toma datos del mismo y los usa en el programa (como el archivo `coordenadas.dat` en nuestro ejemplo), y es de salida cuando los resultados del programa son almacenados en él (como el archivo `baricentro.sal`). Es claro, entonces, que para que un programa funcione correctamente los archivos de entrada deben existir previamente a la ejecución del mismo. Esto puede controlarse fácilmente utilizando las cláusulas `status` y `iostat` en la sentencia `open`:

`open (`*número*`, file='`*nombre del archivo*`', 'status='old', iostat=`*variable entera*`)`

La cláusula `status='old'` indica que el archivo a abrirse debe existir previamente. Por otra parte la asignación de una variable entera en la cláusula correspondiente a `iostat`[5] permite discernir si el archivo fue abierto o no, puesto que la variable entera tomará el valor cero si el archivo se abrió exitosamente o un valor positivo si hubo un error (en este caso, el error es que el archivo no existe). Esto permite implementar una estructura de selección para manejar el error que puede producirse:

```
if (variable entera.ne.0) then
  write(*,*) 'El archivo de entrada no puede ser leído'
  stop
endif
```

[5]El nombre `iostat` hace referencia a *Input/Output status*, esto es estado de entrada/salida.

En el caso de un archivo de salida, éste puede o no existir previamente. Si no existe, la sentencia `open` lo creará (estando vacío hasta que se escriba algo en él), pero si éste existe previamente será *sobrescrito* por las sentencias `write` que operen sobre él. Un error que puede producirse es que no se puede escribir sobre el archivo (por falta de permisos adecuados, por ejemplo). Esta situación puede detectarse nuevamente con la cláusula `iostat` en la sentencia `open`:

```
open (número, file='nombre del archivo, iostat=variable entera)
if (variable entera.ne.0) then
  write(*,*) 'El archivo de salida no puede ser escrito'
  stop
endif
```

Si se quiere evitar que un archivo de salida sea reescrito (en el caso de existir previamente) se puede utilizar la cláusula `status='new'` junto con la cláusula `iostat`:

```
open (número, file='nombre del archivo, status='new', iostat=variable entera)
if (variable entera.ne.0) then
  write(*,*) 'El archivo de salida no puede ser escrito: quizas ya exista.'
  stop
endif
```

Finalmente nótese que un archivo de texto puede ser abierto para lectura o escritura, pero no para ambas operaciones a la vez.

Nota 6. Una situación que se presenta muchas veces es la necesidad de leer un archivo de entrada cuyo número de líneas es arbitrario (esto es, no está fijado de antemano por el problema). Bajo esta circunstancia un bucle `do` no resulta adecuado. Para implementar la alternativa (un bucle `do while`) se necesita disponer de una forma de detectar el final del archivo conforme éste se va recorriendo. Esto puede lograrse agregando la cláusula `iostat=`*variable entera* a la sentencia de lectura. La correspondiente variable entera asignada tomará el valor cero si la lectura se realizó sin error, un valor positivo si se produjo un error (por ejemplo, intentar leer un dato de tipo distinto al que se está considerando) y un valor negativo si el final del archivo es encontrado. Utilizando un contador para el número de datos leídos y el bucle `do while` podemos resolver el problema con el siguiente fragmento de código.

```
n  = 0
io = 0
do while(io.ge.0)
  read(numero,*,iostat=io) variables
  if(io.eq.0) then
    n = n+1
    procesar variables leídas
   endif
enddo
```

A continuación presentamos una revisión del código teniendo en cuenta las notas anteriores.

```
      program baricentro
*     ------------------------------------------------------
*     Declaración de variables
*     ------------------------------------------------------
      implicit none
      integer in,out,io
      parameter (in=8,out=9)
      character(80) arch_in, arch_out
      integer n
      real x,y,bar_x,bar_y
*     ------------------------------------------------------
*     Preguntar nombres de archivos
*     ------------------------------------------------------
      write(*,*) 'Ingrese nombre del archivo de datos'
      read(*,*) arch_in
      write(*,*) 'Ingrese nombre del archivo de salida'
```

```
      read(*,*) arch_out
*     ------------------------------------------------------
*     Abrir archivo de datos
*     ------------------------------------------------------
      open(in,file=arch_in,status='old',iostat=io)
      if(io.ne.0) then
        write(*,*) 'No puede leerse el archivo: ',arch_in
        stop
      endif
*     ------------------------------------------------------
*     Leer los datos (y procesarlos)
*     ------------------------------------------------------
      bar_x = 0.0
      bar_y = 0.0
      n     = 0
      io    = 0
      do while(io.ge.0)
         read(in,*,iostat=io) x,y
         if(io.eq.0) then
            n     = n + 1
            bar_x = bar_x + x
            bar_y = bar_y + y
         endif
      end do
*     ------------------------------------------------------
*     Cerrar el archivo de datos
*     ------------------------------------------------------
      close(in)
*     ------------------------------------------------------
*     Calcular las coordenadas del baricentro
*     ------------------------------------------------------
      if(n.ne.0) then
        bar_x = bar_x/real(n)
        bar_y = bar_y/real(n)
      endif
*     ------------------------------------------------------
*     Abrir archivo de salida
*     ------------------------------------------------------
      open(out,file=arch_out,iostat=io)
      if(io.ne.0) then
        write(*,*) 'No puede escribirse el archivo: ', arch_out
        stop
      endif
*     ------------------------------------------------------
*     Imprimir resultados en el archivo de salida
*     ------------------------------------------------------
      write(out,*) 'Número de puntos = ', n
      write(out,*) 'Cooordenadas del baricentro'
      write(out,*) 'x = ', bar_x
      write(out,*) 'y = ', bar_y
*     ------------------------------------------------------
*     Cerrar archivo de salida
*     ------------------------------------------------------
      close(out)
*     ------------------------------------------------------
*     Terminar
*     ------------------------------------------------------
      stop
```

Tabla 6. Descritores de formatos más comunes.

Descriptor de formato	Uso
`I`w ó `I`$w.m$	Dato entero
`F`$w.d$	Dato real en notación decimal
`E`$w.d$	Dato real en notación científica
`A` ó `A`w	Dato carácter
'$x \ldots x$'	Cadena de caracteres
n`X`	Espaciado horizontal

w: constante positiva entera que especifica el ancho del campo.
m: constante entera no negativa que especifica el mínimo número de dígitos a leer/mostrar.
d: constante entera no negativa que especifica el número de dígitos a la derecha del punto decimal.
x: un carácter.
n: constante entera positiva especificando un número de posiciones.

```
end
```

9. Formato de los datos en la entrada/salida.

Otro aspecto a considerar en la entrada/salida de datos (ya sea por archivos o por teclado/pantalla) es la forma de estos datos. Hasta el momento hemos dejado que el compilador escoja automáticamente el formato apropiado para cada tipo de dato. Sin embargo es posible dar una forma precisa de la representación de los datos con una *especificación de formato* la cual consiste en una cadena de caracteres de la forma `'(` *lista de especificaciones de formatos* `)'`. Cada conjunto de especificación de formato consiste de un *descriptor de edición* de una letra, dependiente del tipo de dato, un tamaño de campo y una ubicación decimal si es pertinente. La tabla 6 muestra los descriptores de formatos de más utilidad. Para hacer efectiva esta especificación de formato se coloca la misma en la correspondiente sentencia de entrada/salida reemplazando al segundo carácter '`*`'.

Por ejemplo, para imprimir el resultado de nuestro programa ejemplo con las coordenadas con tres decimales, podemos escribir

```
write(9,'(A)') 'Coordenadas del baricentro'
write(9,'(A,F7.3)') 'x = ', bar_x
write(9,'(A,F7.3)') 'y = ', bar_y
```

Nota 1. En la especificación de formato `F`$w.d$ de un dato real debe ser $w \geq d + 3$ para contemplar la presencia del signo del número, el primer dígito y el punto decimal.

Nota 2. Con la especificación de formato `E`$w.d$ de un dato real, éste es mostrado en forma *normalizada*, esto es, con un signo menos (si es necesario) seguido de una parte entera consistente de un cero, el punto decimal, y una fracción decimal de d dígitos significativos, y una letra `E` con un exponente de dos dígitos (con su signo menos si es apropiado). Por lo tanto, debe ser $w \geq d + 7$.

Nota 3. En la especificación de formatos de un dato real si el número de dígitos de la parte fraccionaria excede el tamaño asignado en la misma los dígitos en exceso serán eliminados *después* de redondear el número. Por ejemplo, la especificación `F8.2` hace que el número 7.6543 se escriba como 7.65, mientras que el número 3.9462 se escribe como 3.95.

Nota 4. En el descriptor `A` para variables carácter si el ancho del campo no se especifica se considera que el mismo es igual a la longitud de la variable especificada en su declaración de tipo.

Nota 5. La especificación n`X` hace que n posiciones no se tomen en cuenta en la lectura o bien que se llenen n espacios en blanco en la escritura.

Nota 6. Cuando varios datos tienen las mismas especificaciones de formato, puede repetirse una sola especificación colocando un número entero frente a ella. Encerrando un conjunto de especificaciones entre paréntesis y colocando un número frente al conjunto indicamos que dicho conjunto será repetido tal número de veces.

Nota 7. Si la especificación de formato es demasiado pequeña para el dato que se quiere procesar el campo será llenado con asteriscos. ¡Esta es una situación que suele ser muy desconcertante!

Nota 8. La especificación de formato puede ser almacenada en una variable o parámetro de tipo carácter y ser utilizada como tal en la correspondiente sentencia de lectura o escritura.

Nota 9. En general para *leer* datos (ya sea por teclado o archivo) la lectura por lista sin formato es el método adecuado. Sólo si los datos tienen una forma específica predeterminada debe utilizarse la lectura con formato.

10. Tipos de datos indexados: arreglos

11. Utilizando bibliotecas de funciones externas