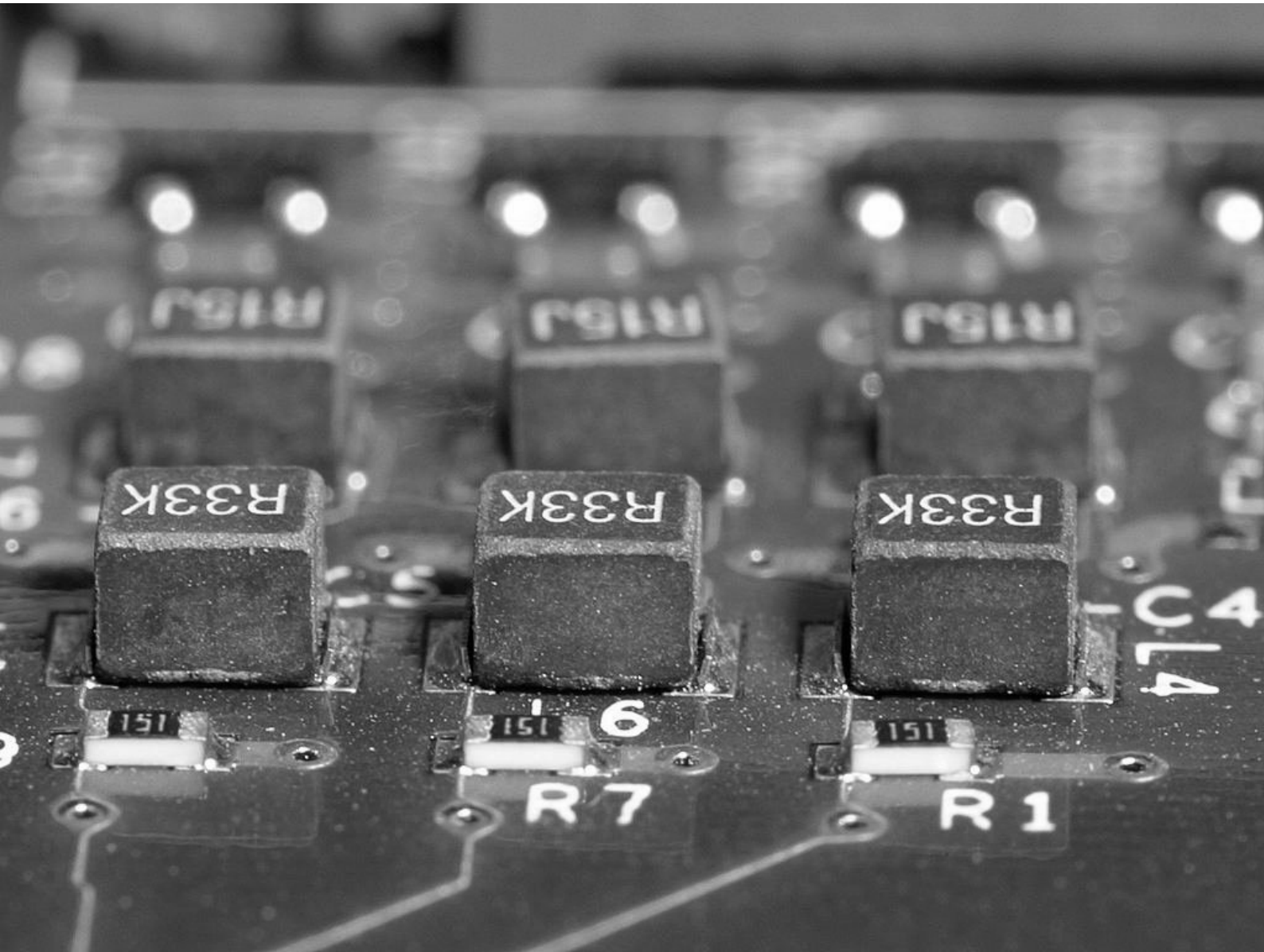
Facultad de Ciencias Astronómicas y Geofísicas - UNLP -

Elementos de programación Fortran

Pablo J. Santamaría



Elementos de programación Fortran.
Pablo J. Santamaría.
Junio 2012 - Versión 0.1.5
e-mail: pablo@fcaglp.unlp.edu.ar
web: http://gcp.fcaglp.unlp.edu.ar/integrantes:psantamaria:fortran:start

Las imágenes que ilustran los capítulos son de dominio público y fueron obtenidas en los sitios http://www.public-domain-image.com y http://www.publicdomainpictures.net.

Prefacio

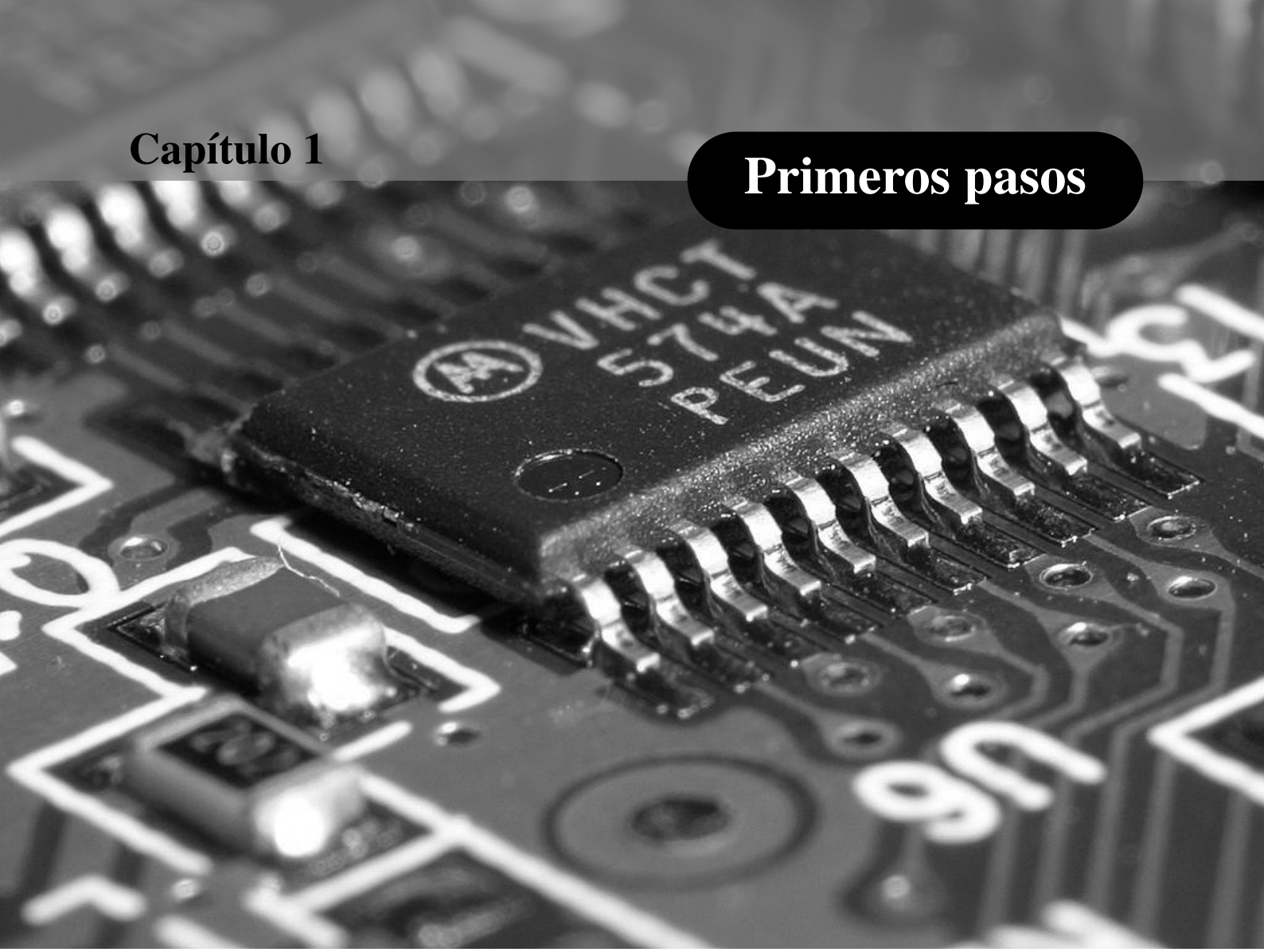
Fortran es, desde sus inicios, el lenguaje de programación ampliamente escogido en el ámbito de la computación científica. El primer compilador Fortran data de 1957. Pero al igual que los lenguajes humanos, con el paso del tiempo un lenguaje de programación evoluciona, adaptándose a las necesidades (y filosofía) de la programación de la época. En el caso de Fortran, 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 usuarios nos podemos asegurar que los programas funcionarán en cualquier computadora que tenga un compilador Fortran que lo implemente. 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 utilizado en la comunidad científica). Una mayor revisión del lenguaje fue el Fortran 90. 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. Los estándares más recientes, Fortran 2003 y 2008, introducen aún más nuevas características en el lenguaje. En esta guía trabajaremos con el estándar Fortran 95 y algunas extensiones del Fortran 2003 implementadas en el compilador `gfortran`, el compilador Fortran de la suite de compiladores **GCC**. Más aún asumiremos que el usuario accederá al mismo en una computadora personal ejecutando el sistema operativo y utilidades de usuario **GNU/Linux**. Puesto que el objetivo de esta guía es proporcionar una rápida introducción al lenguaje haciendo énfasis en la escritura de código bien estructurado y modular, muchas características obsoletas o no recomendadas proveniente de estándares previos no son siquiera nombradas. Asi mismo, el carácter introductorio de la guía, no nos permite tampoco extendernos a conceptos y características más avanzadas introducidas en los últimos estándares. Sin embargo, espero que esta guía, con los conceptos presentados y sus ejercicios, sea de utilidad a aquellos que comienzan implementar algoritmos en Fortran.

Pablo J. Santamaría.
Marzo 2012.

Índice general

Capítulo 1

Primeros pasos



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

1.1. 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.

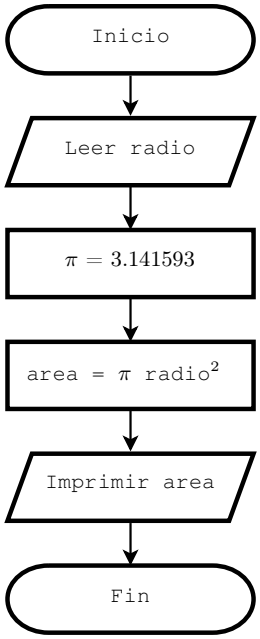


Figura 1.1. Diagrama de flujo para el algoritmo que calcula el área de un círculo.

Para describir un algoritmo utilizaremos 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 × radio².
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 1.1.

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 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:

Código 1.1. Cálculo del área de un círculo

```
PROGRAM calcular_area
! -----------------------------------------------
! Cálculo del área de un círculo de radio dado
! -----------------------------------------------
! Declaración de variables
! -----------------------------------------------
IMPLICIT NONE
REAL :: radio                      ! radio del círculo
REAL :: area                       ! área del  círculo
REAL, PARAMETER :: PI = 3.141593   ! número pi
! -----------------------------------------------
! Entrada de datos
! -----------------------------------------------
WRITE(*,*) 'Ingrese radio del círculo'
READ(*,*) radio
! -----------------------------------------------
! Calcular area
! -----------------------------------------------
area = PI*radio**2
! -----------------------------------------------
! Imprimir resultado
! -----------------------------------------------
WRITE(*,*) 'Area = ', area
! -----------------------------------------------
! Terminar
! -----------------------------------------------
STOP
END PROGRAM calcular_area
```

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*

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

y para un código escrito en Fortran puede tener el nombre que querramos pero debe terminar con la *extensión* `.f90`. 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.f90` ingresamos en la línea de comandos de una terminal:

```
$ emacs area.f90 &
```

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.f90`, *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.f90` y generar un programa ejecutable, que llamaremos `area`, escribimos en la línea de comandos:

```
$ gfortran -Wall -o area area.f90
```

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].

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
```

1.1.1. Ejercicios

Ejercicio 1.1 Utilizar el editor `emacs` para almacenar el código 1.1 en el archivo fuente `area.f90`.

Ejercicio 1.2 Compile el programa fuente del ejercicio anterior. Verifique que se genera, efectivamente, el programa ejecutable.

Ejercicio 1.3 Verificar que el programa da resultados correctos. En particular verificar que para un radio igual a la unidad el área que se obtiene es π y que el área se cuadruplica cuando el radio es igual a 2.

Ejercicio 1.4 Modificar el programa para que calcule el área $A = \pi ab$ de una elipse de semiejes a y b.

[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*.

1.2. 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.

1.2.1. Ejercicios

Ejercicio 1.5 Procediendo con la compilación y ejecución, identificar que tipo de errores se producen en las siguientes situaciones con nuestro código.

a) La sentencia `WRITE(*,*) 'Ingrese el radio del círculo'` del código es cambiada por `WROTE(*,*) 'Ingrese radio del círculo'`.
b) Durante la ejecución se ingresa una letra cuando se pide el radio del círculo.
c) El valor de la constante `PI` es asignada a 2.7182818 en el código.

1.3. 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

```
PROGRAM nombre
declaraciones de tipo
sentencias ejecutables
STOP
END PROGRAM nombre
```

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 están 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 PROGRAM` indica el final lógico del programa principal. La sentencia (ejecutable) `STOP` detiene la *ejecución* del programa.

✏ **END PROGRAM y STOP.**

Es importante comprender la diferencia entre estas dos sentencias. La sentencia `END PROGRAM`, 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 PROGRAM` 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 PROGRAM` 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.

☞ Una sentencia Fortran sobre una línea puede extenderse hasta 132 caracteres. Si la sentencia es más larga ésta puede continuarse en la siguiente linea insertando al final de la línea actual el carácter &. De esta manera una sentencia puede ser continuada sobre 40 líneas, en Fortran 95, o 256 líneas en Fortran 2003.

☞ Originalmente la codificación de un programa Fortran sólo podía utilizar letras mayúsculas. En la actualidad todos los compiladores aceptan que en el programa haya 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).

☞ Cualquier oración precedida de un signo de exclamación `!` es un *comentario*. Un comentario es una indicación del programador que permite aclarar o resaltar lo que realiza cierta parte del código. Por lo tanto los comentarios son de gran ayuda para documentar la operación de un programa o una parte de ella y deben usarse profusamente. Nótese que los comentarios son ignorados por el compilador.

☞ Es recomendable ser consistente con cierto estilo al escribir un programa Fortran. Aquí adoptaremos el estilo en que las instrucciones de Fortran, tales como `PROGRAM`, `READ`, `WRITE`, son escritas con mayúsculas, mientras que las variables definidas por el programador son escritas con minúsculas. Asimismo los nombres que sean compuestos a partir de varias palabras son unidos con el guión bajo, como ser, `mi_primer_programa`. Por otra parte, constantes con nombres, utilizadas para valores que no cambian durante la ejecución del programa, como ser `PI` para almacenenar el valor del número π, son escritos con mayúsculas.

1.3.1. Ejercicios

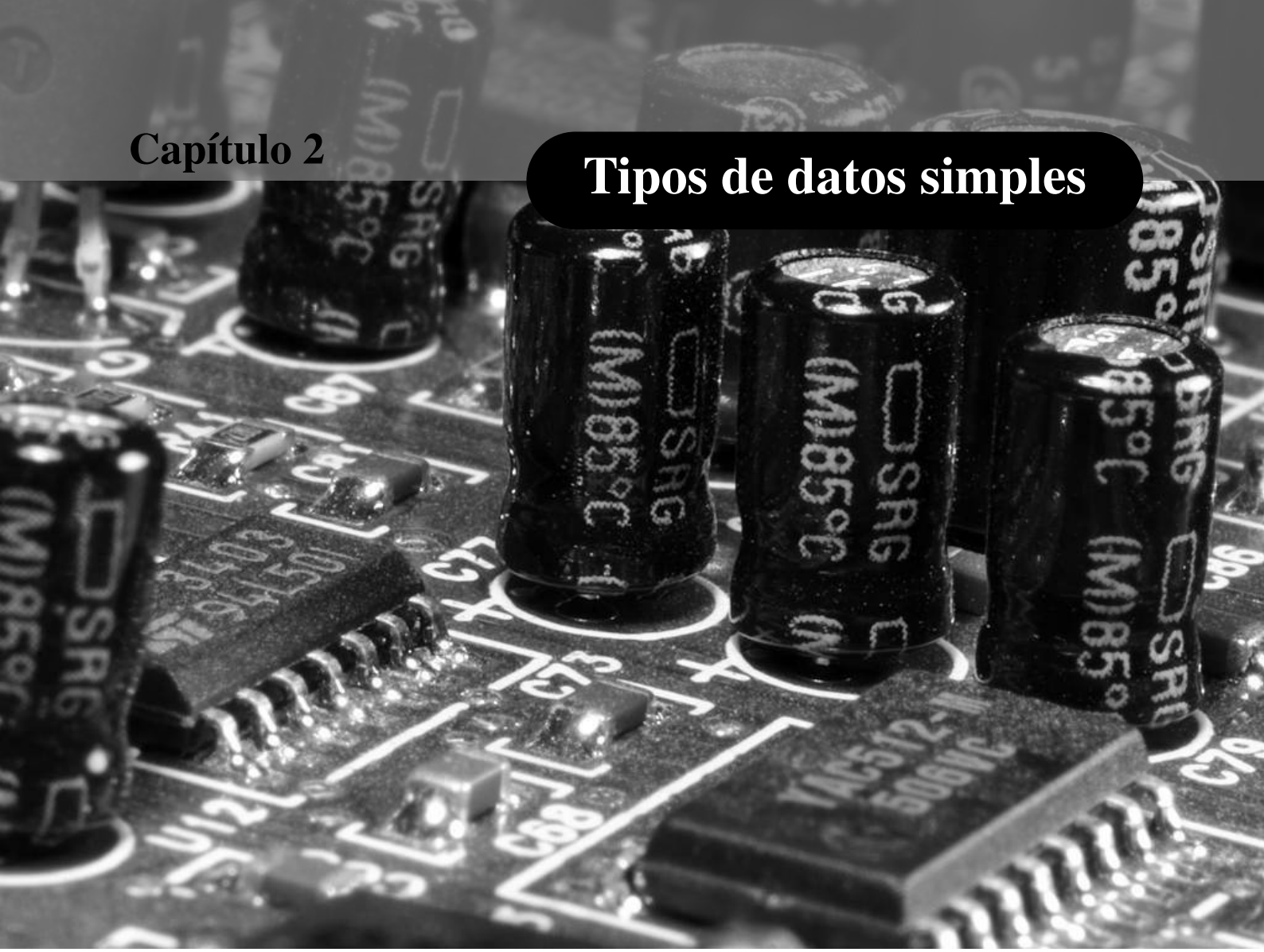
Ejercicio 1.6 Identificar y clasificar las sentencias del programa dado en el código 1.1.

Ejercicio 1.7 Indicar los errores en el siguiente programa Fortran.

```
PROGRAMent
* Un simple programa
integer :: ent
ent = 12
WRITE(*,*) 'El valor del entero es ',
ent
END PROGRAM ent
STOP
```

Capítulo 2

Tipos de datos simples



Dios es real
A menos que sea declarado entero.

2.1. 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 dispone de cinco tipos de datos simples: *enteros*, *reales*, *complejos*, de *carácter* y *lógicos* (siendo los cuatro primeros datos de tipo numérico). Esta clasificación se ilustra en la figura 2.1. Por otra parte, los datos aparecen en el programa como *constantes literales*, como *variables* o como *constantes con nombres*. 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. Una constante con nombre es un valor referenciado con un nombre y su 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 permite la representación de un número real, aunque tal representación *no* es exacta, sino aproximada.

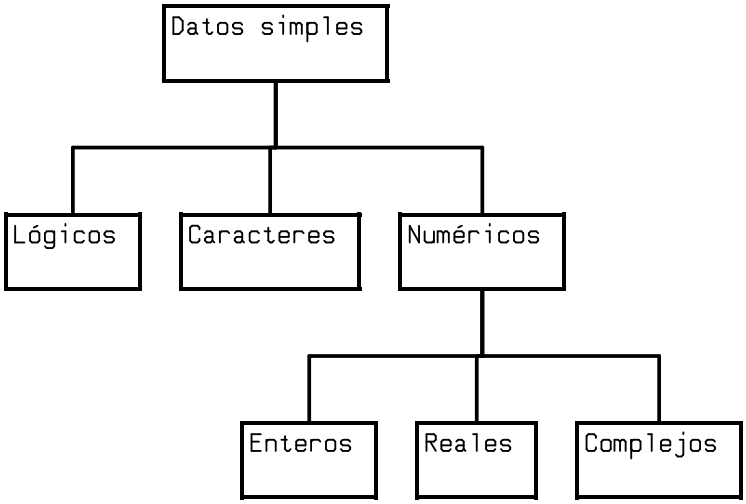


Figura 2.1. Tipos de datos simples.

✏ **Precisión de los datos reales.**

En las computadoras personales (PC) actuales, un dato de tipo real tiene una precisión de no más de 7 dígitos significativos y puede representar números reales en el rango de 10^{-38} a 10^{38}. Este tema es abordado más exahustivamente en el capítulo 7.

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

$$\pm \texttt{m.n E} \pm \texttt{p},$$

donde `m.n` es un número decimal y la `E` (de exponente) significa multiplicar por 10 a la potencia entera $\pm$`p`. Así, por ejemplo, 3.49×10^{-2} se codifica como `3.49E-2`.

✏ **Constantes literales numéricas.**

Fortran establece la diferencia entre constantes literales enteras y reales por la presencia o ausencia del punto decimal, respectivamente. Estas dos 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`.

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 $, &, $\star$, el espacio en blanco, etc.). Una *constante literal de carácter* consiste en una secuencia de caracteres encerradas entre apóstrofes (comillas sencillas), por ejemplo: `'AbC'`, o comillas dobles, `"AbC"` por ejemplo.

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 (incluyendo al guión bajo _ como un carácter válido) o números pero siempre deben comenzar con una letra. El número máximo de caracteres que permite Fortran 95 para formar el nombre de una variables es de 31 caracteres, aunque en Fortran 2003 el límite es de 63.

✏ **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 tipo de variables. Para ello se utilizan las siguientes sentencias no ejecutables de acuerdo al tipo de dato que almacenarán:

```
INTEGER :: lista de variables enteras
REAL :: lista de variables reales
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 deben estar separadas por comas. Puede también usarse una sentencia de declaración de tipo por cada variable del programa. Esto último permite introducir un comentario para el uso que se hará en el programa, construyendo así un útil *diccionario de datos* que permite comprender fácilmente el uso dado a las variables 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 `i-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 evitar completamente 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.

Finalmente, las constantes con nombres permiten asignar un nombre simbólico a una constante literal y, como tal, no podrán ser alteradas. 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 una constante con nombre se realiza en su declaración de tipo como sigue:

```
tipo, PARAMETER :: nombre = constante
```

2.1.1. Ejercicios

Ejercicio 2.1 La siguiente lista presenta una serie de constantes válidas o inválidas en Fortran. Indicar si cada una de ellas es válida o inválida. En el caso que sea válida, especificar su tipo. Si es inválida indicar porque razón.

$$10.0,\ -100,000,\ 123\text{E-}5,\ \text{'Es correcto'},\ 3.14159,\ \text{'3.14159'},\ \text{'Distancia =}, 17.8\text{E+}6,\ 13.0\text{\^{}}2.$$

Ejercicio 2.2 Escribir los siguientes números como constantes literales reales.

$$256,\ 2.56,\ -43\,000,\ 10^{12},\ 0.000000492,\ -10,\ -10^{-16}.$$

Ejercicio 2.3 Escribir los siguientes números complejos como constantes literales complejas.

$$i, \quad 3+i, \quad 1.$$

Ejercicio 2.4 Indicar cuales de los siguientes son identificadores aceptables para variables en Fortran.

```
gamma, j79-12, epsilon, a5, 5a, is real, is_real, r(2)19, stop, _ok.
```

> ✏ **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.

Ejercicio 2.5 Escriba las declaraciones de tipo apropiadas para declarar la variables enteras `i`, `contador`, las variables reales `x`, `y`, `vx`, `vy`, la variable lógica `clave` y la variable carácter `mensaje` (la cual tendrá un máximo de 80 caracteres).

Ejercicio 2.6 Escriba sentencias de declaración apropiadas para definir como parámetros la velocidad de la luz en el vacío $c = 2.9979 \times 10^8$ m s^{-1} y la constante de Plank $h = 6.6256 \times 10^{-34}$ J s.

2.2. 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.

☞ 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).

☞ 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.

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

Tabla 2.1. Codificación de los operadores aritméticos en Fortran.

☞ 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 es indicada en la tabla 2.1.

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 2.2. Para utilizar una función se emplea el nombre de la misma seguido por la expresión sobre la que se va a operar (*argumento*) dentro de un juego de paréntesis. Por ejemplo, la sentencia `y = ABS(x)` calcula el valor absoluto de `x` y lo asigna a `y`.

✏ **Funciones intrínsecas 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`).

2.2.1. Ejercicios

Ejercicio 2.7 Escribir sentencias de asignación que efectúen lo siguiente:

a) Incrementar en 1 el valor actual de una variable entera `n` y remplazar el valor de `n` por dicho incremento. Una variable entera que se incrementa en una unidad o una cantidad constante se conoce como *contador*.
b) Incrementar en `x` (siendo `x` una variable numérica) el valor de la variable numérica `suma` y reemplazar el valor de `suma` por tal incremento. Una variable que actúa de esta forma se conoce como un *acumulador*.
c) Asignar a una variable lógica el valor verdadero.
d) Intercambiar el valor de dos variables `a` y `b` del mismo tipo (*Ayuda*: el intercambio requiere de una tercera variable).

Ejercicio 2.8 Considere el siguiente programa Fortran.

```
PROGRAM problema
IMPLICIT NONE
REAL :: x,y
y = x + 1.0
WRITE(*,*) y
STOP
END PROGRAM problema
```

`r = SIN(r)`	seno del ángulo en radianes
`r = COS(r)`	coseno del ángulo en radianes
`r = TAN(r)`	tangente del ángulo en radianes
`r = ASIN(r)`	arco seno (en el rango $-\pi/2$ a $+\pi/2$)
`r = ACOS(r)`	arco coseno (en el rango 0 a $+\pi$)
`r = ATAN(r)`	arco tangente (en el rango $-\pi/2$ a $+\pi/2$)
`r = ATAN2(r,r)`	arco tangente de arg1/arg2 (en el rango $-\pi$ a π)
`r = SQRT(r)`	raíz cuadrada
`r = EXP(r)`	función exponencial
`r = LOG(r)`	logaritmo natural
`r = LOG10(r)`	logaritmo decimal
`* = ABS(ir)`	valor absoluto
`* = MOD(ir,ir)`	resto de la división de arg1 por arg2
`* = MAX(ir,ir)`	devuelve el máximo entre arg1 y arg2
`* = MIN(ir,ir)`	devuelve el mínimo entre arg1 y arg2
`i = INT(r)`	convierte a un tipo entero truncando la parte decimal
`r = REAL(i)`	convierte a tipo real

Tabla 2.2. Funciones intrínsecas importantes proporcionadas por Fortran. El tipo del dato del argumento y del resultado que admiten es indicado por una letra: `i` = entero, `r` = real. Un asterisco en el miembro de la derecha indica que el resultado es del mismo tipo que el argumento.

¿Qué sucede al ejecutarlo varias veces? ¿A qué se debe tales resultados?

✏ **Variables no inicializadas.**

Una variable en el lado derecho de una sentencia de asignación debe tener un valor antes de que la sentencia de asignación se ejecute. Hasta que una sentencia no le de un valor a una variable, esa variable no tendrá un valor definido. Una variable a la que no se le ha dado un valor se dice que no se ha *inicializado*. Entonces si, por ejemplo, `x` no tiene un valor antes de ejecutarse la sentencia `y = x + 1.0`, se produce un error lógico. Muchos lenguajes de programación inicializan automáticamente sus variables numéricas en cero. Sin embargo, este no es el caso de Fortran. Así una variable sin inicializar contendrá esencialmente un valor espurio proveniente de lo que exista en dicho momento en la posición de memoria correspondiente a la variable. Su uso ciertamente conducirá a una situación de error en los datos de salida. En Fortran 95 la inicialización de una variable puede ser realizada *en tiempo de compilación* asignando su valor en la sentencia de declaración correspondiente, como ser, por ejemplo, `REAL :: x = 0.0`.

2.3. Orden de precedencia de las operaciones aritméticas y conversión implícita de tipo.

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 las operaciones. En Fortran estas reglas son las siguientes:

- 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.
- 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.
- 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)`.

Cuando en una expresión aritmética los 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 complejos. 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, `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 2.2.

2.3.1. Ejercicios

Ejercicio 2.9 Determinar el valor de la variable real `a` o de la variable entera `i` obtenido como resultado de cada una de las siguientes sentencias de asignación aritmética. Indicar el orden en que son realizadas las operaciones aritméticas y las conversiones de tipo implícitas (si existen).

a) `a = 2*6 + 1`
b) `a = 2/3`
c) `a = 2.0*6.0/4.0`
d) `i = 2*10/4`
e) `i = 2*(10/4)`
f) `a = 2*(10/4)`
g) `a = 2.0*(10.0/4.0)`
h) `a = 2.0*(1.0e1/4.0)`
i) `a = 6.0*1.0/6.0`
j) `a = 6.0*(1.0/6.0)`
k) `a = 1.0/3.0 + 1.0/3.0 + 1.0/3.0`
l) `a = 1/3 + 1/3 + 1/3`
m) `a = 4.0**(3/2)`
n) `a = 4.0**3.0/2.0`
ñ) `a = 4.0**(3.0/2.0)`
o) `i = 19/4 + 5/4`
p) `a = 19/4 + 5/4`
q) `i = 100*(99/100)`
r) `i = 10**(2/3)`
s) `i = 10**(2.0/3.0)`

Ejercicio 2.10 Supongase que las variables reales `a,b,c,d,e,f` y la variable entera `g` han sido inicializadas con los siguientes valores:

`a=3.0, b=2.0, c=5.0, d=4.0, c=5.0, d=4.0, e=10.0, f=2.0, g=3`

Determine el resultado de las siguientes sentencias de asignación, indique el orden en que se realizan las operaciones.

a) `resultado = a*b+c*d+e/f**g`
b) `resultado = a*(b+c)*d+(e/f)**g`
c) `resultado = a*(b+c)*(d+e)/f**g`

Ejercicio 2.11 Escriba sentencias de asignación aritméticas para los siguientes expresiones matemáticas.

a) $t = 3 \times 10^3 x^4$,
b) $y = (-x)^n$,
c) $x = a^{(1/n)}$,
d) $z = \dfrac{xy}{\sqrt{x^2+y^2}}$,
e) $y = \left(\dfrac{2}{\pi x}\right)^{1/2} \cos x$,
f) $z = \cos^{-1}(|\log x|)$.

g) $y = \frac{1}{2}\log\frac{1+\sin x}{1-\sin x}$,

Ejercicio 2.12 Considere el siguiente programa

```
PROGRAM test
IMPLICIT NONE
REAL :: a,b,c
! -----------------------------------
READ(*,*) a,b
c = ( (a+b)**2 - 2.0*a*b - b**2 )/a**2
WRITE(*,*) c
! -----------------------------------
STOP
END PROGRAM test
```

¿Cuál sería el valor esperado de `c`, cualquiera sean los valores de `a` y `b` ingresados? Ejectue el programa ingresando los pares de valores `a` = 0.5, `b` = 888.0; `a` = 0.0001, `b` = 8888.0 y `a` = 0.00001, `b` = 88888.0. ¿A qué se debe los resultados obtenidos?

2.4. Entrada y salida 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 otra práctica. 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
```

2.4.1. Ejercicios

Ejercicio 2.13 Escribir la sentencia de entrada (para leer por teclado) y la sentencia de salida (por pantalla) para los siguientes datos:

a) `0.1E13`
b) `(0.0,1.0)`
c) `'Hola mundo!'`
d) `3 1.5 -0.6`
e) `.TRUE.`

2.5. Implementando lo aprendido.

Los siguientes ejercicios plantean diversos problemas. Para cada uno de ellos se debe diseñar un algoritmo apropiado el cual debe ser descrito en pseudocódigo y graficado por su diagrama de flujo. Luego implementar dicho algoritmo como un programa Fortran. Testear el programa utilizando datos de entrada que conducen a resultados conocidos de antemano.

Ejercicio 2.14 Dado los tres lados a, b y c de un triángulo, calcular su área A por la fórmula de Herón,

$$A = \sqrt{s(s-a)(s-b)(s-c)},$$

donde $s = (a+b+c)/2$.

Ejercicio 2.15 Dadas las coordenadas polares (r,θ) de un punto en el plano $\mathbb{R}^2$, se desea calcular sus coordenadas rectangulares (x,y) definidas por

$$\begin{cases} x = r\cos\theta, \\ y = r\sin\theta. \end{cases}$$

Ejercicio 2.16 Calcular el área A y el volumen V de una esfera de radio r,

$$A = 4\pi r^2, \quad V = \frac{4}{3}\pi r^3.$$

Implementar el algoritmo de manera de minimizar el número de multiplicaciones utilizadas.

> ✏ **Eficiencia de un algoritmo.**
> Una manera de medir la eficiencia de un algoritmo (y un programa) es contabilizar el número de operaciones utilizadas para resolver el problema. Cuanto menor sea este número más eficiente será el algoritmo (y el programa).

Ejercicio 2.17 La temperatura medida en grados centígrados t_C puede ser convertida a la escala Farenheit t_F según la fórmula:

$$t_F = \frac{9}{5}t_C + 32$$

Dado un valor decimal de la temperatura en la escala centígrada se quiere obtener su valor en la escala Fahrenheit.

Ejercicio 2.18 La magnitud de la fuerza de atracción gravitatoria entre dos masas puntuales m_1 y m_2, separadas por una distancia r está dada por la fórmula

$$F = G\frac{m_1 m_2}{r^2},$$

donde $G = 6.673 \times 10^{-8}$ cm^3 s^2 g^{-1} es la constante de gravitación universal. Se desea evaluar, dada la masas de dos cuerpos y la distancia entre ellos, la fuerza de gravitación. El resultado debe estar expresados en dinas; una dina es dimensionalmente igual a un g cm s^{-2}.Nótese que las unidades de una fórmula deben ser *consistentes*, por lo cual, en este problema, las masas deben expresarse en gramos y la distancia en centímetros.

Capítulo 3

Estructuras de control

Se encuentra un programador muerto en la bañadera.
En una de sus manos hay una botella de shampoo que dice:
Modo de uso: Aplicar, enjuagar, repetir.

3.1. 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:

- *secuencial*: las instrucciones se ejecutan sucesivamente una después de otra,
- *de selección*: permite elegir entre dos conjuntos de instrucciones dependiendo del cumplimiento (o no) de una condición,
- *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.

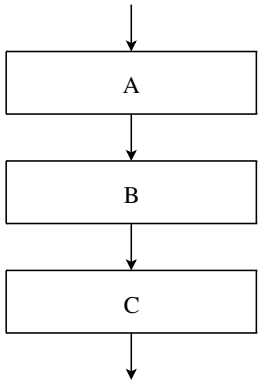


Figura 3.1. Estructura secuencial.

3.2. Estructura secuencial.

La estructura de control más simple está 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 3.1 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.

3.3. Estructura de selección.

La *estructura de selección* permite que dos conjuntos de instrucciones alternativas puedan ejecutarse 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 3.2. 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.

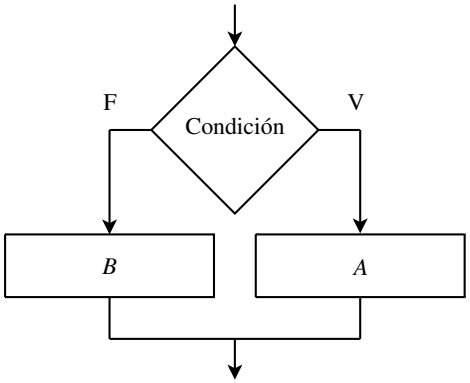


Figura 3.2. Estructura de selección.

Operador	Significado
<	menor que
>	mayor que
==	igual a
<=	menor o igual que
>=	mayor o igual que
/=	distinto a

Tabla 3.1. Operadores relacionales.

Operador	Significado
`.NOT.`	negación
`.AND.`	conjunción
`.OR.`	disyunción (inclusiva)
`.EQV.`	equivalencia

Tabla 3.2. 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 3.1. 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 3.2. 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.[1].

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 3.3)

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

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

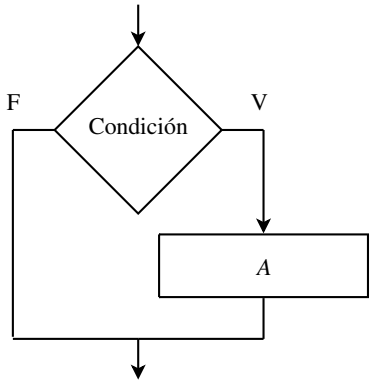


Figura 3.3. Estructura de selección sin sección *sino*.

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 suele presentar es la necesidad de elegir entre más de una alternativa de ejecución. En este caso podemos utilizar la *estructura multicondicional* cuya lógica se puede expresar en la forma *si . . . entonces sino si . . . sino . . . .* El pseudocódigo correspondiente es descrito como sigue (y su diagrama de flujo se ilustra en la figura 3.4)

```
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
```

Aquí, cada condición se prueba por turno. Si la condición no se se satisface, se prueba la siguiente, 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.

```
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
```

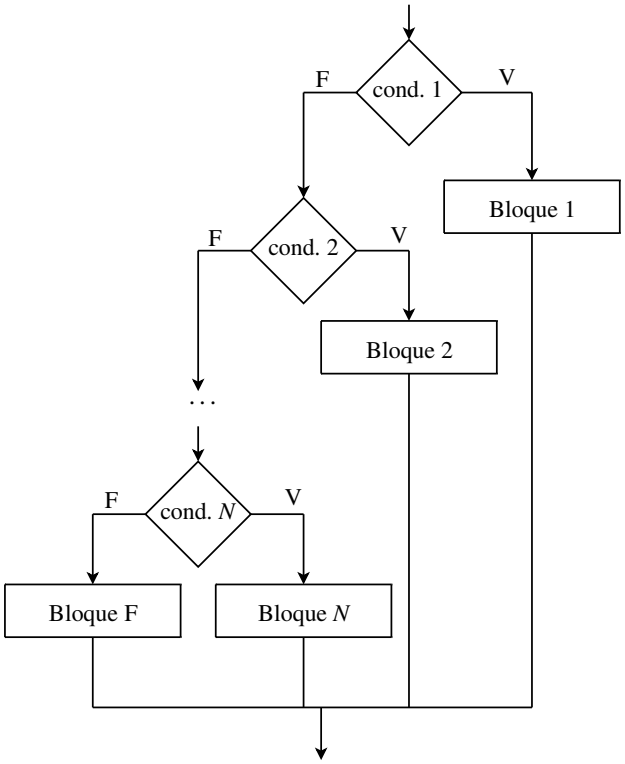


Figura 3.4. Estructura multicondicional.

Consideremos, como ejemplo de una estructura de selección, el diseño e implementación de un algoritmo para calcular las raíces de una ecuación cuadrática

$$ax^2+bx+c=0,$$

esto es,

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

La naturaleza de estas raíces dependerá del signo del discriminante $\Delta=b^2-4ac$, lo cual nos lleva a implementar una estructura de selección: 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 = b² −4ac.
Si Δ ≥ 0 entonces
  Calcular x₁ = (−b+√Δ)/2a
  Calcular x₂ = (−b−√Δ)/2a
sino
  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.

Código 3.1. Cálculo de las raíces de la ec. cuadrática

```
PROGRAM ecuadratica
  ! ------------------------------------------------------------------
  ! Se calculan las raíces de la ecuación cuadrática ax**2+bx+c=0
  ! ------------------------------------------------------------------
  ! Declaración de tipos
  ! ------------------------------------------------------------------
  IMPLICIT NONE
  REAL :: a,b,c            ! coeficientes de la ecuación
  REAL :: discr            ! discriminante de la ecuación
  REAL :: x1,x2            ! variables para soluciones
  REAL :: term, den
  ! ------------------------------------------------------------------
  ! Entrada de datos
  ! ------------------------------------------------------------------
  WRITE(*,*) 'Ingrese coeficientes a,b,c'
  READ(*,*) a,b,c
  IF( a == 0 ) THEN
     WRITE(*,*) 'La ecuación no tiene término cuadrático'
     STOP
  ENDIF
  ! ------------------------------------------------------------------
  ! Bloque de procesamiento y salida
  ! ------------------------------------------------------------------
  discr = b**2 - 4.0*a*c
  den   = 2.0*a
  term  = SQRT(ABS(discr))
  IF (discr >= 0 ) 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 PROGRAM ecuadratica
```

3.3.1. Ejercicios

Ejercicio 3.1 Dadas las variables con los valores que se indican:

```
a =  2.0     d =  2.5    i =  2    f = .FALSE.
b =  5.0     e = -4.0    j =  3    t = .TRUE.
c = 10.0                 k = -2
```

deducir el valor lógico de cada una de las expresiones lógicas siguientes. Indicar el orden en que se evalúan.

a) `t .AND. f .OR. .FALSE.`
b) `a**i+b <= b/c+d`
c) `i/j == 2+k .AND. b/c+d >= e+c/d-a**j`

d) `(b*j+3.0) == (d-e) .AND. (.NOT. f)`

Ejercicio 3.2 Implemente una estructura de decisión para verificar si un número es negativo o no negativo (positivo o cero).

Ejercicio 3.3 Implemente una estructura de decisión para determinar si un número entero es par o impar (*Ayuda:* utilice la función `MOD(x,y)`, la cual devuelve el resto de la división de `x` por `y`).

Ejercicio 3.4 Dada una esfera de radio R, considerando su centro como origen de coordenadas se quiere determinar si un punto de coordenadas (x,y,z) está dentro o fuera de la esfera. Implemente un algoritmo para éste problema.

Ejercicio 3.5 Considérese en el plano un rectángulo dado por las coordenadas (x_S, y_S) de su vértice superior izquierdo y las coordenadas (x_I, y_I) de su vértice inferior derecho. Se quiere determinar si un punto (x,y) del plano está dentro o fuera del rectángulo. Implemente la solución en un algoritmo.

Ejercicio 3.6 Dado tres números reales *distintos* se desea determinar cual es el mayor. Implemente un algoritmo apropiado (*Ayuda:* considere ya sea un conjunto de estructuras de selección *anidadas* o bien dos estructuras de selección en secuencia).

✏ **Estructuras de selección anidadas**

La sentencia que comienza en el bloque de instrucciones verdadero o falso de la estructura de selección puede ser cualquiera, incluso otra sentencia `IF-THEN-ELSE`. Cuando ésto ocurre en una o ambas bifurcaciones de la estructura, se dice que las sentencias `IF` están *anidadas*.

Ejercicio 3.7 Compile y ejecute el siguiente programa. ¿A qué puede atribuirse el resultado que se obtiene?

```
PROGRAM test_igualdad
IMPLICIT NONE
REAL :: a
a = 2.0
IF (1.0/a == 0.5) THEN
  WRITE(*,*) '1/2 es igual a 0.5'
ELSE
  WRITE(*,*) '1/2 es distinto a 0.5'
ENDIF
a = 10.0
IF (1.0/a == 0.1) THEN
   WRITE(*,*) '1/10 es igual a 0.1'
ELSE
   WRITE(*,*) '1/10 es distinto a 0.1'
ENDIF
STOP
END PROGRAM test_igualdad
```

✏ **Igualdad entre datos reales.**

Las cantidades reales que son algebraicamente iguales pueden producir un valor lógico falso cuando se comparan los respectivos datos reales con == ya que la mayoría de los números reales no se almacenan exactamente en la computadora.

Ejercicio 3.8 Implemente un algoritmo que intercambie los valores de dos números reales si están en orden creciente pero que no realice ninguna acción en caso contrario.

Ejercicio 3.9 Implementar el cálculo del valor absoluto de un número real con un if lógico.

Ejercicio 3.10 Implementar una estructura multicondicional para la evaluación de la función

$$f(x) = \begin{cases} e^{-x} & \text{si } x < -1, \\ e & \text{si } -1 \leq x \leq 1, \\ e^{x} & \text{si } x > 1. \end{cases}$$

Ejercicio 3.11 Un examen se considera desaprobado si la nota obtenida es menor que 4 y aprobado en caso contrario. Si la nota está entre 7 y 9 (inclusive) el examen se considera destacado, y entre 9 y 10 (inclusive) sobresaliente. Implementar un algoritmo para indicar el *estatus* de un examen en base a su nota.

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

- Bucles donde el número de iteraciones es fijo y conocido de antemano.
- 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 índice = valor inicial hasta valor final hacer
   instrucciones del bucle
fin_desde
```

El diagrama de flujo correspondiente se ilustra en la figura 3.5.

☞ El *índice* es una variable entera que, actuando como un *contador*, permite controlar el número de ejecuciones del ciclo.

☞ Los *valor inicial* y *valor final* son valores enteros que indican los límites entre los que varía *índice* al comienzo y final del bucle.

☞ 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.

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

☞ Dentro de las instrucciones del bucle *no es legal* modificar la variable *índice*. 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 *índice* se limita a la estructura de iteración.

En Fortran la estructura repetitiva se codifica como sigue.

```
DO índice = valor inicial, valor final, incremento
  sentencias del bucle
ENDDO
```

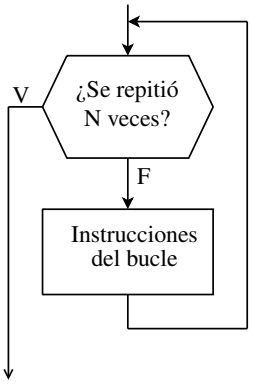


Figura 3.5. Estructura de iteración.

Aquí *índice* es una variable entera, mientras que *valor inicial*, *valor final* e *incremento* pueden ser variables, 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 cualquier fracción decimal. Sólo si N es mayor que 0 se ejecutará el bucle.

Consideremos ahora 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 3.6)

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

☞ 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.

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

☞ 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 lógica.

Fortran, además del `DO WHILE`, dispone de una estructura más general para bucles condicionales, cuya codificación es la siguiente:

```
DO
   sentencias del bloque pre-condición
   IF (condición) EXIT
   sentencias del bloque post-condición
ENDDO
```

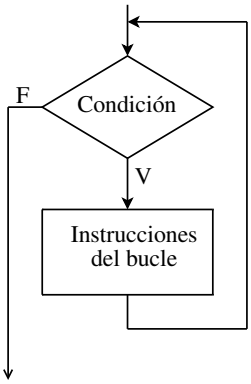


Figura 3.6. Estructura de iteración condicional.

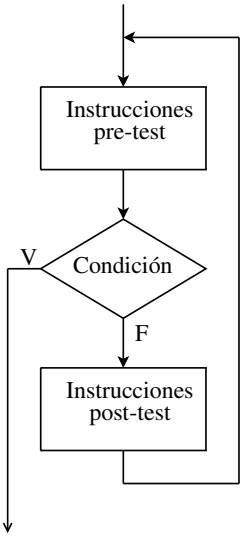


Figura 3.7. Estructura de iteración condicional general.

El pseudocódigo correspondiente puede ser descrito como sigue (y su diagrama de flujo se ilustra en la figura 3.7):

```
Repetir
  instrucciones pre-condición
  Si condición terminar repetir
  instrucciones post-condición
fin_repetir
```

☞ Las sentencias correspondientes tanto a los bloques previo y posterior al test de la condición son ejecutadas indefinidamente mientras la condición sea falsa. Cuando la condición resulta verdadera la repetición es terminada y la ejecución continúa con la sentencia que sigue a la estructura de control. Nótese que si la condición es verdadera cuando se inicia el bucle por primera vez, las sentencias del bloque pre-condición serán ejecutadas una vez y luego el control es transferido a la sentencia siguiente a la estructura, sin ejecutarse ninguna de las instrucciones correspondientes al bloque posterior a la condición.

☞ Un bucle condicional `DO WHILE`

```
DO WHILE (condición)
   sentencias del bloque
ENDDO
```

es equivalente a un bucle condicional general de la forma

```
DO
   IF (.NOT. condición) EXIT
   sentencias del bloque
ENDDO
```

Esto es, la condición lógica que controla la repetición del bucle es evaluada al comienzo del bucle y es la negación lógica a la condición que controla al bucle `DO WHILE`.

☞ Un bucle condicional general de la forma

```
DO
  sentencias del bloque
  IF (condición) EXIT
ENDDO
```

repetirá las sentencias del bloque hasta que la condición se haga verdadera. Pero debido a que la condición se verifica después de que el cuerpo del bucle se ha ejecutado, las instrucciones correspondientes se ejecutarán al menos una vez sin importar si la condición es verdadera o falsa. Este tipo de bucle condicional es conocido como *repetir hasta que* (*repeat-until*).

Consideremos, como ejemplo de estructura de iteración, 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.

Código 3.2. Suma de los n primeros enteros positivos

```
PROGRAM sumar
  ! ------------------------------------------------------------------
  ! Se calcula la suma de los n primeros enteros positivos
  ! ------------------------------------------------------------------
  ! Declaración de tipos
  ! ------------------------------------------------------------------
  IMPLICIT NONE
  integer :: n        ! número de términos a sumar
  integer :: suma     ! valor de la suma
  integer :: i        ! indice del bucle
  ! ------------------------------------------------------------------
  ! 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 PROGRAM sumar
```

Considérese ahora el problema de determinar el primer valor n para el cual la suma $\sum_{i=1}^{n}$ excede a 10000. 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 n
Terminar
```

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

Código 3.3. Determinar valor a partir del cual la suma de los n primeros enteros positivos excede un límite

```
PROGRAM sumar
  ! ------------------------------------------------------------------
  ! Se determina el primer valor de n para el cual la suma de los n
  ! primeros enteros positivos excede a 10000.
  ! ------------------------------------------------------------------
  ! Declaración de tipos
  ! ------------------------------------------------------------------
  IMPLICIT none
  INTEGER :: n                  ! número de términos a sumar
  INTEGER :: suma               ! valor de la suma
  INTEGER :: limite = 10000     ! limite superior de la suma
  ! ------------------------------------------------------------------
  ! Bloque de procesamiento
  ! ------------------------------------------------------------------
  suma  = 0
  n     = 0
  DO WHILE (suma <= limite)
     n    = n +1
     suma = suma + n
  ENDDO
  ! ------------------------------------------------------------------
  ! Salida de datos
  ! ------------------------------------------------------------------
  WRITE(*,*) 'n = ', n
  ! ------------------------------------------------------------------
  ! Terminar
  ! ------------------------------------------------------------------
  STOP
END PROGRAM sumar
```

3.4.1. Ejercicios

Ejercicio 3.12 Imprimir una tabla de los cuadrados y cubos de los primeros N números enteros positivos. Ordenar la tabla primero en orden ascendente y luego en orden descendente.

Ejercicio 3.13 Calcular la suma y multiplicación de los N primeros números enteros positivos,

$$\sum_{i=1}^{N} i, \qquad \prod_{i=1}^{N} i.$$

✏ **Inicialización de los acumuladores.**

Siempre recordar inicializar a cero la variable que será utilizada para acumular una suma repetida y a uno la variable que acumulará un producto repetido.

Ejercicio 3.14 Considérese el siguiente conjunto de bucles repetitivos *anidados*.

```
DO i=1,5
     WRITE(*,*) 'Iteración externa = ',i
     DO j=1,4
          WRITE(*,*) 'Iteración interna = ',j
     ENDDO
ENDDO
```

¿Cuantas iteraciones del bucle interno se realizan por cada iteración del bucle externo? ¿Cuántas iteraciones se realizan en total?

Ejercicio 3.15 Tabular la función $f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2}$ para x en el intervalo $[0,1]$ con un paso $h = 0.1$ (*Ayuda:* Notar que si $[a,b]$ es el intervalo bajo consideración, los puntos donde hay que evaluar f están dados por $x_i = a + ih$ con $i = 0,\ldots,N$, siendo $N = (b-a)/h$).

Ejercicio 3.16 Determinar cual es el primer valor de N para el cual la suma de los N primeros números enteros positivos, $\sum_{i=1}^{N} i$, excede a 10 000.

Ejercicio 3.17 Reimplementar el ejercicio anterior con una estructura *repetir hasta que*.

3.5. Implementando lo aprendido.

Los siguientes ejercicios plantean diversos problemas. Diseñar un algoritmo apropiado implementando su pseudocódigo y su diagrama de flujo correspondiente. Luego codificarlo en un programa Fortran.

Ejercicio 3.18 Determinar si un año dado es bisiesto o no. Recordar que un año es bisiesto si es divisible por 4, aunque si es divisible por 100 no es bisiesto, salvo si es divisible por 400. Así, 1988 fue bisiesto, como también lo fue el año 2000, pero no 1800.

Ejercicio 3.19 Dado un conjunto de N números determinar cuantos de ellos son negativos, positivos o cero.

Ejercicio 3.20 Calcular las soluciones de una ecuación cuadrática $ax^2 + bx + c = 0$. Contemplar todas las alternativas posibles (raíces reales distintas, iguales y complejas). Testear el programa para el siguiente conjunto de coeficientes:

a) $a = 2, b = 2, c = -12$ (raíces reales distintas, 2 y -3),
b) $a = 2, b = 4, c = 2$ (raíces reales iguales, -1),
c) $a = 1, b = 0, c = 1$ (raíces complejas conjugadas, i y $-i$).

Ejercicio 3.21 Dado un conjunto de N números reales determinar cual es el máximo, el mínimo y la media aritmética del conjunto.

Ejercicio 3.22 El *máximo común divisor*, mcd, (a,b) de dos número enteros positivos a y b, con $a \geq b > 0$ puede ser calculado por el *algoritmo de Euclides*, según el cual el mcd es igual al último resto no nulo que se obtiene por aplicación sucesiva de la división entera entre el divisor y el resto del paso anterior. Esto es,

$$(a,b) = (b,r_1) = (r_1,r_2) = \cdots = (r_{n-1},r_n) = (r_n,0) = r_n$$

Implementar este algoritmo para calcular el máximo común divisor de dos números enteros *cualesquiera* dados, contemplando la posibilidad de que alguno de ellos, o ambos, sea negativo o cero (*Ayuda*: $(a,b) = (|a|,|b|)$, $(a,0) = |a|$ y $(0,0)$ no está definido). Verificar el programa calculando (25950,1095) = 15, (252,-1324) = 4.

Ejercicio 3.23 La fecha del domingo de Pascua corresponde al primer domingo después de la primera luna llena que sigue al equinocio de primavera en el hemisferio norte. Los siguientes cálculos permiten conocer esta fecha para un año comprendido entre 1900 y 2099:

```
a = MOD(año,19)
b = MOD(año,4)
c = MOD(año,7)
d = MOD(19*a+24,30)
e = MOD(2*b+4*c+6*d+5,7)
n = 22+d+e
```

donde `n` indica el número de días del mes de marzo (o abril si `n` es superior a 31) correspondiente al domingo de Pascua. Realizar un programa que determine esta fecha para un año dado y comprobar que para el 2010, el domingo de Pascua corresponde al 4 de abril.

Ejercicio 3.24 Todo número complejo $z = (x,y) = x+iy$ no nulo admite exactamente n raíces n-ésimas distintas dadas por

$$w_k = \sqrt[n]{\rho}\left[\cos\left(\frac{\theta+2k\pi}{n}\right) + i\ \sin\left(\frac{\theta+2k\pi}{n}\right)\right],$$

donde $k = 0,1,\ldots,n-1$, y

$$\rho = \sqrt{x^2+y^2}, \qquad \tan\theta = y/x.$$

Dado un número complejo z no nulo determinar todas sus raíces n-ésimas (*Ayuda:* para determinar el argumento θ utilizar la función `ATAN2(y,x)`).

Ejercicio 3.25 Calcular el seno de un número x a partir de su serie de Taylor:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Considerar tantos términos como sean necesarios para que el error cometido en la aproximación finita sea menor que cierta tolerancia prescrita (digamos $\leq 10^{-8}$). *Observación:* Notar que dado un término de la serie, el siguiente se obtiene multiplicando por $-x^2$ y dividiendo por el producto de los dos enteros siguientes. Testear el programa tanto con valores pequeños como con valores muy grandes. Comentar los resultados obtenidos.

Ejercicio 3.26 En la ecuación cúbica general de tercer grado

$$x^3 + a_2x^2 + a_1x + a_0 = 0$$

poniendo $x = x' - a_2/3$, obtenemos una ecuación cúbica sin término cuadrático, cuyas raíces son las de la primera, incrementadas en $-a_2/3$. Alcanza, pues, con considerar las ecuaciones cúbicas del tipo

$$x^3 + px + q = 0,$$

donde, para simplificar, supondremos que p y q son números reales. Se sigue entonces del teorema fundamental del algebra que esta ecuación tiene o bien una única raíz real, o bien tres raíces reales (pudiendo coincidir dos de ellas). Tales raíces reales pueden ser calculadas como sigue. Sea

$$\Delta = \frac{q^2}{4} + \frac{p^3}{27},$$

el *discriminante* de la ecuación cúbica. Si $\Delta > 0$, entonces existe una única raíz real, x_1, dada por $x_1 = u_1 + v_1$, donde

$$u_1 = \text{sign}\left(-\frac{q}{2}+\sqrt{\Delta}\right)\left|-\frac{q}{2}+\sqrt{\Delta}\right|^{1/3}, \quad v_1 = \text{sign}\left(-\frac{q}{2}-\sqrt{\Delta}\right)\left|-\frac{q}{2}-\sqrt{\Delta}\right|^{1/3},$$

Si $\Delta = 0$, tendremos tres raíces reales (siendo al menos dos iguales), x_1 , $x_2 = x_3$, dadas por

$$x_1 = 2u_1, \quad x_2 = x_3 = -u_1$$

donde

$$u_1 = \text{sign}\left(-\frac{q}{2}\right)\left|-\frac{q}{2}\right|^{\frac{1}{3}}$$

Finalmente, si $\Delta < 0$, habrá tres raíces reales (distintas) x_1, x_2, x_3, dadas por

$$x_1 = 2\rho^{1/3}\cos\left(\frac{\theta}{3}\right), \quad x_2 = 2\rho^{1/3}\cos\left(\frac{\theta}{3}+\frac{2}{3}\pi\right), \quad x_3 = 2\rho^{1/3}\cos\left(\frac{\theta}{3}+\frac{4}{3}\pi\right)$$

donde

$$\rho = \sqrt{-\frac{p^3}{27}}, \quad \cos\theta = -\frac{q}{2\rho}$$

Teniendo en cuenta lo anterior implemente un algoritmo (y su programa) para calcular las raíces reales de las ecuaciones cúbicas del tipo $x^3 + px + q = 0$. Compruebe el programa con los siguientes casos triviales:

$$x^3 = 0, \quad x^3 - 1 = 0, \quad x^3 - x = 0$$

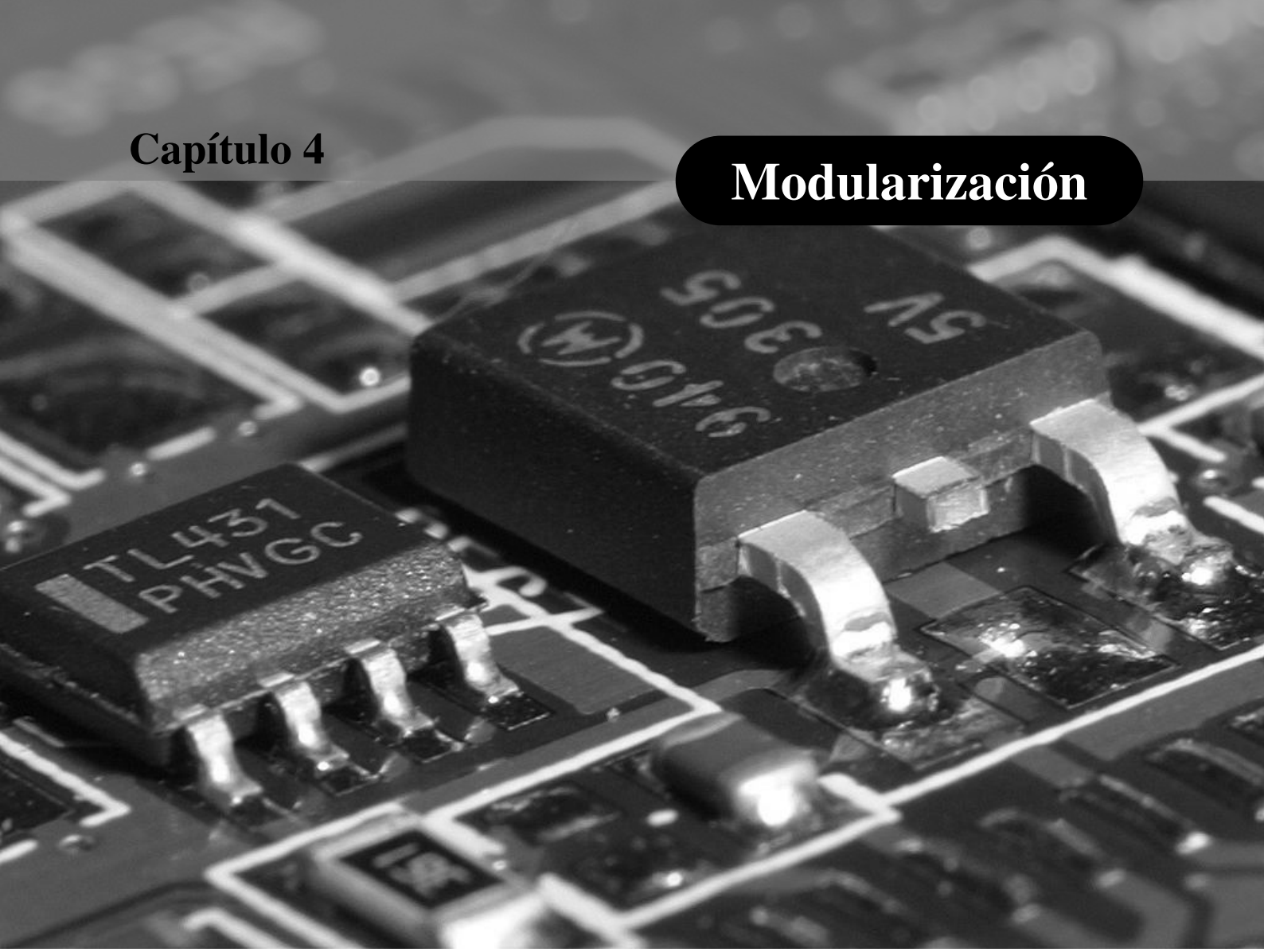
Finalmente, encuentre las raíces reales de la ecuación

$$x^3 - 3.5292x + 2.118176 = 0$$

(*Rta:* Redondeado a cinco decimales, las raíces son $x_1 = 1.43167, x_2 = -2.1272, x_3 = 0.69552$)

Capítulo 4

Modularización



Divide et vinces.
(Divide y vencerás)
– Julio Cesar

4.1. Programación modular

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 4.1 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 4.2 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

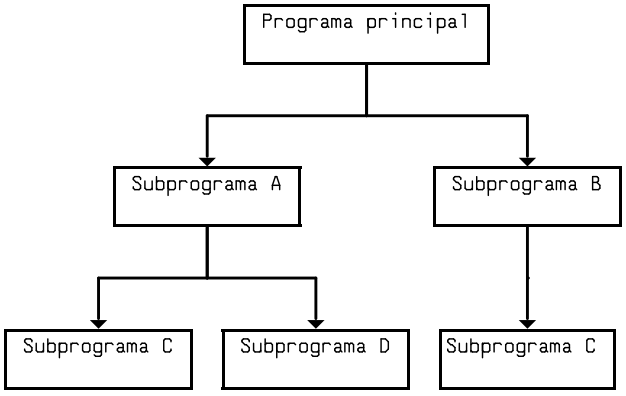


Figura 4.1. Diagrama de estructura de un algoritmo o programa modularizado.

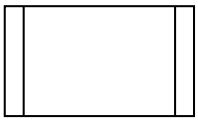


Figura 4.2. Representación de un subprograma en un diagrama de flujo.

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.
- 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*).

✏ **No reinventar la rueda.**

Esta es una regla básica de la programación. Aquí la expresión *reinventar la rueda* se utiliza para referirse a la situación en que se desperdicia tiempo y recursos en la implementación de una solución a un problema que ya ha sido resuelto apropiadamente.

4.2. Funciones y subrutinas.

Fortran provee dos maneras de implementar un subprograma: *funciones* y *subrutinas*. Estos subprogramas pueden ser *intrínsecos*, esto es, provistos por el compilador o *externos*[1]. 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 externos propios. Para ello presentamos, como ejemplo, una versión modularizada del código 1.1para el cálculo del área de un círculo tratado en el capítulo 1.

Código 4.1. Implementación modular del código para calcular el área de un círculo

```
PROGRAM principal
!---------------------------------------
! Declaración de variables
!---------------------------------------
```

[1] Fortran también proporciona subprogramas *internos*, los cuales son procedimientos contenidos dentro de otra unidad de programa, pero ellos no serán considerados aquí.

```
IMPLICIT NONE
REAL :: radio, area
!--------------------------------------
! Declaración de subprogramas utilizados
!--------------------------------------
INTERFACE
  SUBROUTINE leer_radio(radio)
    IMPLICIT NONE
    REAL, INTENT(OUT) :: radio
  END SUBROUTINE leer_radio
  FUNCTION calcular_area(radio)
    IMPLICIT NONE
    REAL :: calcular_area
    REAL, INTENT(IN) :: radio
  END FUNCTION calcular_area
  SUBROUTINE imprimir_area(a)
    IMPLICIT NONE
    REAL, INTENT(IN) :: a
  END SUBROUTINE imprimir_area
END INTERFACE
!--------------------------------------
! Invocar subprogramas
!--------------------------------------
CALL leer_radio(radio)
area = calcular_area(radio)
CALL imprimir_area(area)
!--------------------------------------
! Terminar
!--------------------------------------
STOP
END PROGRAM principal
```

```
SUBROUTINE leer_radio(radio)
!--------------------------------------
! Declaración de argumentos formales
!--------------------------------------
IMPLICIT NONE
REAL, INTENT(OUT) :: radio
!--------------------------------------
! Ingreso de datos
!--------------------------------------
WRITE(*,*) 'Ingrese radio'
READ(*,*) radio
!--------------------------------------
! Retornar
!--------------------------------------
RETURN
END SUBROUTINE leer_radio
```

```
FUNCTION calcular_area(radio)
!--------------------------------------
! Declaración de tipo de la función y
! de los argumentos formales
!--------------------------------------
IMPLICIT NONE
REAL :: calcular_area
REAL, INTENT(IN) :: radio
!--------------------------------------
! Declaración de variables
!--------------------------------------
```

```
  REAL, PARAMETER :: PI = 3.14159
  !--------------------------------------
  ! Cálculo de la función
  !--------------------------------------
  calcular_area = PI*radio**2
  !--------------------------------------
  ! Retornar
  !--------------------------------------
  RETURN
  END FUNCTION calcular_area
```

```
  SUBROUTINE imprimir_area(a)
  !--------------------------------------
  ! Declaración de argumentos formales
  !--------------------------------------
  IMPLICIT NONE
  REAL, INTENT(IN) :: a
  !--------------------------------------
  ! Salida de datos
  !--------------------------------------
  WRITE(*,*) 'Area =', a
  !--------------------------------------
  ! Retornar
  !--------------------------------------
  RETURN
  END SUBROUTINE imprimir_area
```

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:

```
FUNCTION nombre (argumentos)
  declaración de tipo de nombre
  declaraciones de tipo
  sentencias
  nombre = expresión
  RETURN
END FUNCTION nombre
```

```
SUBROUTINE nombre (argumentos)
  declaraciones de tipo
  sentencias
  RETURN
END SUBROUTINE nombre
```

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

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

☞ Cada subprograma es por sí mismo una unidad de programa independiente con sus propias variables, constantes literales y constantes con nombres. 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 PROGRAM|SUBROUTINE|FUNCTION`, la cual indica su fin lógico al compilador.

☞ 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 dentro del cuerpo de la `FUNCTION`. 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.*

☞ **Alcance de los identificadores.** En Fortran 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, constantes con nombres 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. Así, por ejemplo, en nuestro código, la constante con nombre `PI` sólo es conocida 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.

☞ 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`.

☞ 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.

☞ 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 encerrada 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 la distinción entre argumentos formales de entrada, salida y entrada/-salida es especificada a través del atributo `INTENT(`*tipo*`)` en la sentencia de declaración de los mismos, donde *tipo* puede tomar los valores `IN`, `OUT` ó `INOUT`, respectivamente. En particular *en una función los argumentos formales son utilizados solamente como argumentos de entrada*, por lo que los argumentos de la misma siempre deben ser declarados con el atributo `INTENT(IN)`. Nótese que 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. Considerando nuestro ejemplo, vemos que 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`.

☞ Los argumentos que aparecen en la 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*. 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.

✏ **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, 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. Por este motivo los argumentos que actuarán como datos de entrada deben ser declarados en el subprograma con el atributo `INTENT(IN)`, ya que de este modo cualquier intento de modificar su valor dentro del subprograma originará un error de compilación.

☞ **Interfaz implícita y explícita.** De acuerdo a la nota anterior, es claro que *la lista de argumentos actuales en la llamada de un subprograma debe corresponderse con lista de argumentos formales en número, tipo y orden*[2]. Ahora bien, debido a que cada subprograma externo es una entidad completamente independiente de otra (inclusive del programa principal), *el compilador asume que tal correspondencia de argumentos es siempre correcta*. Se dice entonces que los subprogramas externos tienen una *interfaz implícita*. Esto implica que si existe un error de correspondencia en la invocación del subprograma, el programa *aún* compilará, pero generará resultados inconsistentes y erróneos. *Para que el compilador pueda verificar la consistencia de las llamadas a los subprogramas debe hacerse explícita las interfaces de los mismos*. Una manera de proveer una interfaz explícita de un subprograma consiste en utilizar un *bloque de interfaz* en la unidad de programa que efectúa la llamada[3]. La forma de un bloque de interfaz para una subrutina y función externas es

```
INTERFACE
  SUBROUTINE nombre_subrutina(argumentos)
    IMPLICIT NONE
    declaración de argumentos
  END SUBROUTINE nombre_subrutina
  FUNCTION nombre_función(argumentos)
    IMPLICIT NONE
    declaración de nombre_función
    declaración de argumentos
  END FUNCTION nombre_función
END INTERFACE
```

Esto es, se declaran las *cabeceras* (del inglés, *header*) de cada una de las subrutinas y funciones externas que utilizará la unidad de programa en cuestión. Nótese que la cabecera incluye solamente el nombre del subprograma y la declaración de los argumentos formales, por lo que *no* incluye la declaración de variables locales ni instrucciones ejecutables. El bloque de interfaz es colocado en la unidad de programa que invoca al subprograma en la parte reservada para la declaración de tipos. El programa principal de nuestro ejemplo muestra claramente como definir una interfaz explícita para los tres subprogramas utilizados. Nótese que cada interfaz, aún cuando está contenida dentro de una unidad de programa, es una entidad separada de la misma, así que el mismo nombre de variable puede aparecer en la interfaz y la unidad de programa que la incluye sin causar ningún conflicto. Con la información proporcionada por el bloque interfaz, el compilador puede

[2] Fortran permite construir subprogramas con argumentos opcionales y con nombre, pero esta característica avanzada no será considerada aquí.

[3] Una forma alternativa de hacer explícita la interfaz de un subprograma consiste en construir un *módulo de procedimientos* como se verá más adelante.

entonces verificar la consistencia en las llamadas de los subprogramas y en caso de error abortará la compilación indicando el origen del mismo.

☞ 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. En aquellas (raras) circunstancias en que debe asegurarse que una variable local preserve su valor entre sucesivas llamadas al subprograma se utiliza el atributo `SAVE` en la declaración de tipo tal variable. Pero además, *cualquier variable local que es inicializada en su declaración de tipo preserva también su valor entre llamadas*, sin necesidad de especificar el atributo `SAVE` en la misma.

4.2.1. Ejercicios

Ejercicio 4.1 Ampliar el código del ejemplo implementando una función para calcular el perímetro del círculo.

Ejercicio 4.2 Escribir una subrutina que permita el intercambio de dos variables reales. Explicar por qué tal subrutina funciona.

Ejercicio 4.3 En el plano una rotación de ángulo θ alrededor del origen transforma las coordenadas (x,y) de un punto en nuevas coordenadas (x',y') dadas por

$$\begin{cases} x' = x\cos\theta + y\sin\theta, \\ y' = -x\sin\theta + y\cos\theta. \end{cases}$$

Implementar: (*a*) funciones, (*b*) una subrutina, para realizar tal rotación.

Ejercicio 4.4 Identificar los errores cometidos en los siguientes subprogramas o en la invocación de las mismos.

```
PROGRAM main
  IMPLICIT NONE
  REAL :: a,b
  INTERFACE
     SUBROUTINE silly(input,output
         )
       IMPLICIT none
       REAL, INTENT(IN)  :: input
       REAL, INTENT(OUT) :: output
     END SUBROUTINE silly
  END INTERFACE
  !-------------------------
  a = 0.0
  CALL silly(a,b)
  WRITE(*,*) a, b
  !-------------------------
  STOP
END PROGRAM main

SUBROUTINE silly(input,output)
  IMPLICIT NONE
  REAL, INTENT(IN)  :: input
  REAL, INTENT(OUT) :: output
  !-------------------------
  output =  2.0*input
  input  = -1.0
  !-------------------------
  RETURN
END SUBROUTINE silly
```

```
PROGRAM main
  IMPLICIT NONE
  INTERFACE
     SUBROUTINE funny(output)
       IMPLICIT NONE
       REAL, INTENT(OUT) :: output
     END SUBROUTINE funny
  END INTERFACE
  !-------------------------
  CALL funny(1.0)
  !-------------------------
  STOP
END PROGRAM main

SUBROUTINE funny(output)
  IMPLICIT NONE
  REAL, INTENT(OUT) :: output
  !-------------------------
  output = 2.0
  !-------------------------
  RETURN
END SUBROUTINE funny
```

Ejercicio 4.5 El siguiente ejercicio muestra que, sin una interfaz explícita, la invocación a un subprograma con un argumento incorrecto produce resultados indeseables. Considere el siguiente programa:

```
PROGRAM main
  IMPLICIT NONE
  REAL :: x = 1.0
  CALL display(x)
END PROGRAM main

SUBROUTINE display(i)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: i
  WRITE(*,*) i
END SUBROUTINE display
```

¿Compila el programa? ¿Cuál es el resultado obtenido? ¿Que resultado hubiera esperado? Escriba un bloque interfaz para la subrutina en el programa principal y vuelva a compilar el programa. ¿Qué se obtiene?

4.3. 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.f90` mientras que los subprogramas son guardados en los archivos `leer-radio.f90`, `calcular-area.f90`, `imprimir-area.f90` respectivamente, el programa ejecutable `area` puede ser compilado con la siguiente línea de comandos

```
$ gfortran -Wall -o area area.f90 leer-radio.f90 calcular-area.f90 \
           imprimir-area.f90
```

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.f90
$ gfortran -Wall -c leer-radio.f90
$ gfortran -Wall -c calcular-area.f90
$ gfortran -Wall -c imprimir-area.f90
```

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. 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 la utilidad `make`.

4.4. Implementando lo aprendido.

Los siguientes ejercicios plantean diversos problemas. Diseñar un algoritmo apropiado como se indique implementando su pseudocódigo (con su respectivo diagrama de flujo) para luego codificarlo en Fortran.

Ejercicio 4.6 Implementar un subprograma apropiado (función o subrutina) para expresar un ángulo dado en grados, minutos y segundos en radianes y viceversa.

Ejercicio 4.7 Implementar una función para el cálculo del factorial de un número entero positivo, $n!$, el cual es definido en forma recursiva según:

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ n(n-1)! & \text{si } n > 0. \end{cases}$$

Utilizar tal función para determinar el factorial de los 35 primeros enteros positivos.

Observación 1: En vez del procedimiento recursivo calcular el factorial como $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$.

Observación 2: Es probable que deba implementar nuevamente la función para que devuelva un valor real. Explicar por qué esto es así.

Ejercicio 4.8 Dado dos enteros no negativos n y r con $n \geq r$ implementar una función para el cálculo del coeficiente binomial

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}.$$

a) Testee la función calculando $\binom{1}{0}$, $\binom{4}{2}$ y $\binom{25}{11}$.
b) Utilice la función para imprimir las N primeras filas del triángulo de Tartaglia o Pascal.

Observación: En vista de los resultados del ejercicio anterior considere *no* utilizar la función factorial. Para ello note que:

$$\binom{n}{r} = \frac{n(n-1)(n-2)\cdots(n-r+1)}{r!} = \frac{n(n-1)(n-2)\cdots(n-r+1)}{1 \cdot 2 \cdots (r-1) \cdot r}.$$

Ejercicio 4.9 Escribir una función que devuelva un valor lógico (esto es, verdadero o falso) según un número entero positivo dado es primo o no. *Observación*: Recordar que un entero positivo p es primo si es divisible sólo por si mismo y la unidad. El método más simple para determinar si p es primo consiste en verificar si no es divisible por todos los números sucesivos de 2 a $p-1$. Debido a que el único primo par es el 2, se puede mejorar el método separando la verificación de la divisibilidad por 2 y luego, si p no es divisible por 2, testear la divisibilidad por los enteros *impares* existentes entre 3 y $p-1$.

a) Utilice la función para determinar los N primeros números primos.
b) Utilice la función para imprimir los factores primos de un entero positivo dado.
c) Mejore la rapidez de la función notando que podemos terminar el testeo de la divisibilidad no en $p-1$ sino antes, en $[\sqrt{p}]$, ya que si hay un factor mayor que $[\sqrt{p}]$ entonces existe un factor menor que $[\sqrt{p}]$, el cual ya ha sido revisado. Esta modificación ¿afecta a la escritura de los programas anteriores? ¿Y a la ejecución de los mismos?

4.5. Subprogramas como argumentos.

Fortran permite que un subprograma sea pasado a otro subprograma en su lista de argumentos. Para tal efecto, el correspondiente argumento formal es declarado con un bloque de interfaz dentro del subprograma, lo cual permite, entonces, especificar completamente las características del subprograma que será pasado. Nótese que en esta circunstancia, el atributo `INTENT` no tiene relevancia, y por lo tanto no se aplica. Por ejemplo, la siguiente subrutina estima la derivada de cualquier función $f(x)$ suave en un punto $x = a$ haciendo uso de la aproximación $f'(a) \sim [f(a+h) - f(a-h)]/(2h)$ siendo h un paso pequeño dado.

Código 4.2. Subrutina para estimar la derivada primera en un punto

```
SUBROUTINE derivada(f,a,h,df)
  ! -------------------------------
  IMPLICIT NONE
  REAL, INTENT(IN)  :: a  ! Punto a estimar la derivada
  REAL, INTENT(IN)  :: h  ! Paso para la estimación
  REAL, INTENT(OUT) :: df ! Estimación de la derivada
  INTERFACE
     REAL FUNCTION f(x)   ! Funcion a derivar
       IMPLICIT NONE
       REAL, INTENT(IN) :: x
     END FUNCTION f
  END INTERFACE
  ! -------------------------------
  df = (f(a+h)-f(a-h))/(2.0*h)
  RETURN
END SUBROUTINE derivada
```

Así, con ayuda de esta subrutina, podemos estimar, por ejemplo, la derivada del seno y el coseno en $x = \pi/4$ con el siguiente programa.

Código 4.3. Ejemplo del uso de la subrutina derivada

```
PROGRAM main
  ! -------------------------------
  IMPLICIT NONE
  REAL :: a = 0.785398163397448
  REAL :: h = 0.01
  REAL :: df
  INTERFACE
     SUBROUTINE derivada(f,a,h,df)
       IMPLICIT NONE
       REAL, INTENT(IN)  :: a
       REAL, INTENT(IN)  :: h
       REAL, INTENT(OUT) :: df
       INTERFACE
          FUNCTION f(x)
            IMPLICIT NONE
            REAL :: f
            REAL, INTENT(IN) :: x
          END FUNCTION f
       END INTERFACE
     END SUBROUTINE derivada
     FUNCTION f1(x)
       IMPLICIT NONE
       REAL :: f1
       REAL, INTENT(IN) :: x
     END FUNCTION f1
     FUNCTION f2(x)
       IMPLICIT NONE
       REAL :: f2
       REAL, INTENT(IN) :: x
     END FUNCTION f2
  END INTERFACE
  ! -------------------------------
  CALL derivada(f1,a,h,df)
  WRITE(*,*) 'Derivada1 = ', df
  CALL derivada(f2,a,h,df)
  WRITE(*,*) 'Derivada2 = ', df
```

```
    STOP
  END PROGRAM main

  FUNCTION f1(x)
    ! -------------------------------
    IMPLICIT NONE
    REAL :: f1
    REAL, INTENT(IN) :: x
    ! -------------------------------
    f1 = sin(x)
  END FUNCTION f1

  FUNCTION f2(x)
    ! -------------------------------
    IMPLICIT NONE
    REAL :: f2
    REAL, INTENT(IN) :: x
    ! -------------------------------
    f2 = cos(x)
  END FUNCTION f2
```

4.5.1. Ejercicios

Ejercicio 4.10 Escribir una subrutina que tabule una función arbitraria f sobre un conjunto de $(N+1)$ puntos igualmente espaciados del intervalo $[a,b]$, esto es, que imprima los valores de f en los puntos

$$x_i = a + ih, \qquad i = 0,1,\ldots,N,$$

siendo $h = (b-a)/N$ el *paso* de la tabulación y $x_0 = a$, $x_N = b$.

Utilice la subrutina para imprimir la tabla de $f(x) = e^{-x^2}$ y de $f(x) = \cos(x)$ sobre el intervalo $[-1,1]$ en nueve puntos igualmente espaciados.

Ejercicio 4.11 Escribir una subrutina que determine la localización del máximo y mínimo valor de una función arbitraria f sobre un cierto rango $a \leq x \leq b$ evaluando la misma sobre n puntos de dicho rango. La subrutina debe tener como argumentos de entrada: a, b, n y la función f. Los argumentos de salida deben ser: el valor de x donde se alcanza el mínimo (máximo), el valor de f en dicho mínimo (máximo). Utilizar esta subrutina para buscar el máximo y mínimo de $f(x) = x^3 - 5x^2 + 5x + 2$ sobre el rango $-1 \leq x \leq 3$ utilizando $n = 200$ puntos.

4.6. Módulos.

Un módulo es una unidad de programa que permite agrupar subprogramas relacionados (y otros datos) para construir una biblioteca de rutinas que podrá ser reutilizada en cualquier otro programa. La forma más simple de un módulo que contiene sólo subprogramas es:

```
MODULO nombre_del_módulo
  CONTAINS
    subprograma 1
    subprograma 2
    ⋮
    subprograma n
END MODULO nombre_del_módulo
```

donde *subprograma 1*, ..., *subprograma n* son funciones y/o subrutinas. Cada uno de estos subprogramas son llamados *subprogramas* o *procedimientos* del módulo. El nombre del módulo sigue las convenciones usuales

para cualquier identificador en Fortran. Para que los procedimientos de un módulo resulten accesibles a una dada unidad del programa se debe utilizar la instrucción:

```
USE nombre_del_módulo
```

la cual *debe escribirse inmediatamente después de la identificación de la unidad*, inclusive antes de la sentencia `IMPLICIT NONE` que llevan todos nuestros programas.

Por ejemplo, el siguiente módulo contiene dos funciones que permiten convertir una temperatura de la escala Fahrenheit a la escala Celsius y viceversa.

Código 4.4. Módulo para convertir escalas de temperatura

```
MODULE escalas_temperatura
  ! ------------------------------------------------
  ! Modulo para convertir el valor de la temperatura
  ! en diversas escalas:
  !
  ! fahr2cent: Función que convierte de la escala
  !            Fahrenheit a la escala centígrada.
  !
  ! cent2fahr: Función que convierte de la escala
  !            centígrada a la escala Fahrenheit.
  ! ------------------------------------------------
CONTAINS

  FUNCTION fahr2cent(temp)
    ! ----------------------------------------
    IMPLICIT NONE
    REAL :: fahr2cent
    REAL, INTENT(IN) :: temp
    ! ----------------------------------------
    fahr2cent = (temp - 32.0)/1.8
    RETURN
  END FUNCTION fahr2cent

  FUNCTION cent2fahr(temp)
    ! ----------------------------------------
    IMPLICIT NONE
    REAL :: cent2fahr
    REAL, INTENT(IN) :: temp
    ! ----------------------------------------
    cent2fahr = 1.8*temp + 32.0
    RETURN
  END FUNCTION cent2fahr

END MODULE escalas_temperatura
```

Entonces el siguiente programa hace uso de este módulo para convertir una temperatura expresada en grados centígrados, ingresada por el usuario, en la escala Fahrenheit.

Código 4.5. Ejemplo del uso del módulo de escalas de temperatura para convertir de grados Centígrados a Fahrenheit

```
PROGRAM main
  USE escalas_temperatura
  IMPLICIT NONE
  REAL :: temp_cent, temp_fahr
  WRITE(*,*) 'Ingrese temperatura en grados centígrados'
  READ(*,*) temp_cent
```

```
    temp_fahr = cent2fahr(temp_cent)
    WRITE(*,*) 'Grados centígrados = ', temp_cent
    WRITE(*,*) 'Grados Fahrenheit  = ', temp_fahr
    STOP
  END PROGRAM main
```

☞ Como puede verse en el ejemplo, *no* es necesario crear interfaces explícitas para los subprogramas del módulo con un bloque interfaz en la unidad de programa que las utiliza. Esto se debe a que los subprogramas dentro de un módulo, e importados por la sentencia `USE` en otra unidad de programa, tienen una *interfaz explícita*, esto es, todos los detalles de la interfaz del subprograma resultan disponibles al compilador, quien puede, entonces, verificar la consistencia de las llamadas al subprograma.

☞ Existen diferentes maneras de compilar un programa junto con un módulo.

La primera opción es incluir el código fuente del módulo en el archivo del código fuente de la unidad del programa que lo utiliza, justo antes del código de tal unidad de programa obteniendo un único archivo `.f90`, el cual es compilado como es usual. Claramente, esta forma de incluir módulos no es flexible.

Una segunda opción es escribir el código fuente en un archivo (digamos `modulo.f90`) independiente del código fuente del programa que lo utiliza, (guardado, digamos, en el archivo `main.f90`) y compilar ambos en la línea de comandos, *anteponiendo* el archivo del módulo a cualquier otro archivo:

```
$ gfortran -Wall -o ejecutable modulo.f90 main.f90
```

La tercer opción consiste en compilar por separado el módulo y el programa, para luego generar el ejecutable final:

```
$ gfortran -Wall -c modulo.f90
$ gfortran -Wall -c main.f90
$ gfortran -Wall -o ejecutable modulo.o main.o
```

☞ La compilación de un módulo deja, además, como resultado un archivo `.mod`, el cual contiene toda la información relevante para hacer explícita la interfaz de sus subprogramas y es utilizado cada vez que se invoca al mismo con la sentencia `USE`.

☞ En la forma `USE` *nombre_del_módulo*, todos los subprogramas del módulo son importados a la unidad del programa que emplea la sentencia. Si sólo se desea importar algunos subprogramas determinados, entonces la sentencia `USE` puede escribirse en la forma:

```
USE nombre_del_módulo, ONLY: lista de subprogramas del modulo a importar
```

Por ejemplo, en nuestro ejemplo, la sentencia `USE` puede ser reemplazada por `USE escalas_temperatura, ONLY: cent2fhar`.

☞ Un módulo puede contener también datos para ser compartidos con otras unidades de programas. Por ejemplo, en nuestro ejemplo, podemos incorporar las constantes de calor de fusión y evaporación del agua (medidas en caloría por gramo) como sigue:

Código 4.6. Inclusión de variables compartidas en un módulo

```
MODULE escalas_temperatura

  IMPLICIT NONE
  REAL, PARAMETER :: CALOR_DE_FUSION = 79.71
  REAL, PARAMETER :: CALOR_DE_VAPORIZACION = 539.55

CONTAINS
        ....

END MODULE escalas_temperatura
```

Una unidad de programa que haga uso del módulo, vía la sentencia `USE`, podrá disponer de las constantes con nombre `CALOR_DE_FUSION` y `CALOR_DE_VAPORIZACION`. Esta forma de proceder permite crear un repositorio de constantes para todo problema general donde existan constantes "universales" de uso frecuente, eliminando la necesidad de definirlas en cada unidad de programa que las necesiten.

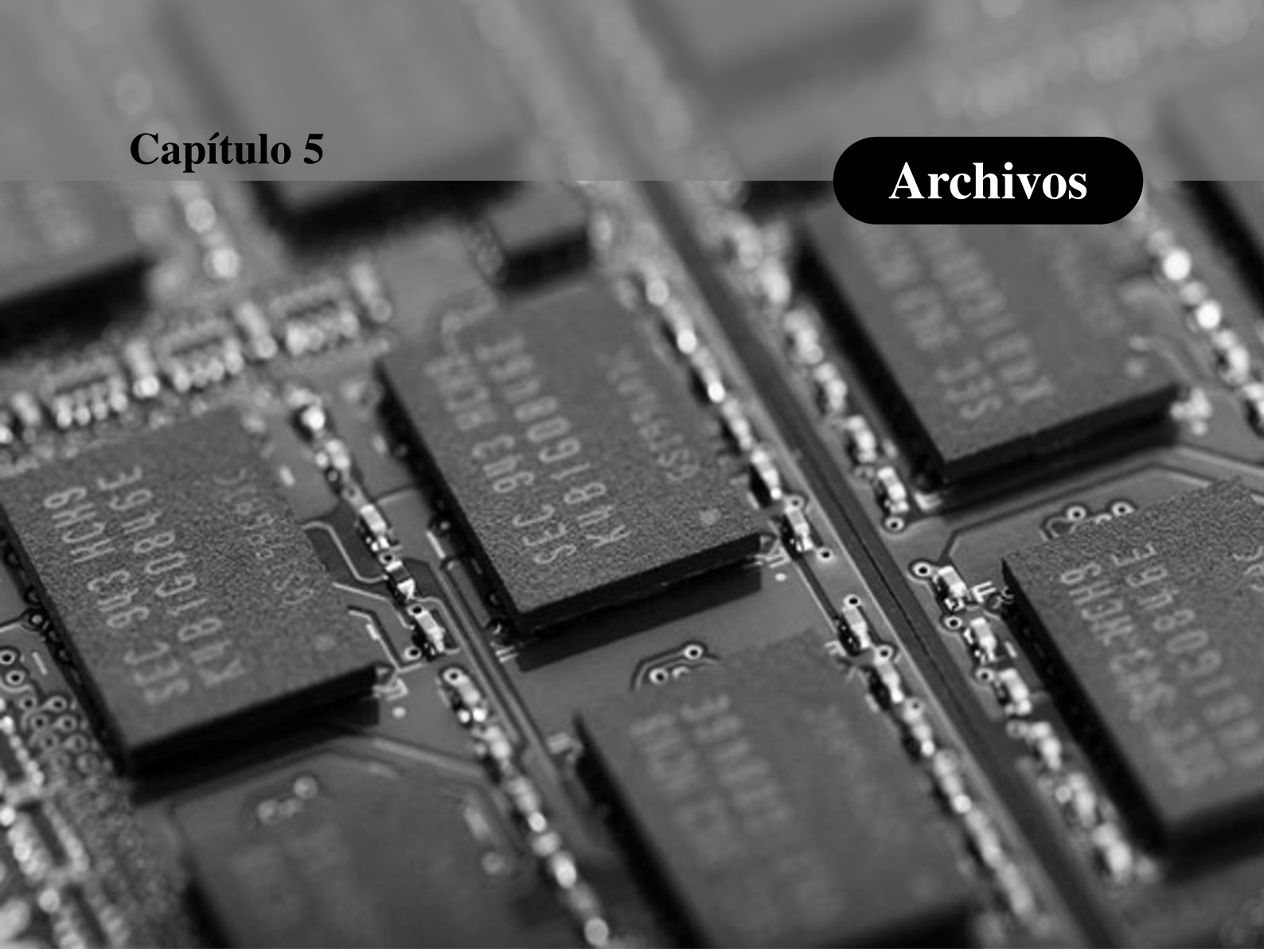
4.6.1. Ejercicios

Ejercicio 4.12 Implementar un módulo con los subprogramas del ejercicio **4.6** que expresan un ángulo dado en radianes en grados, minutos y segundos y viceversa. Escribir un programa principal que haga uso del mismo.

Ejercicio 4.13 Implementar un módulo que permita a cualquier unidad de programa importar los siguientes datos sobre el planeta Tierra: radio medio, radio ecuatorial, volumen, masa, densidad media. Implementar un programa que haga uso de este módulo para calcular el momento de inercia del planeta (*Ayuda*: asuma que el planeta es un cuerpo rígido esférico de radio R y masa M, entonces $I = \frac{2}{5}MR^2$).

Capítulo 5

Archivos



```
Archivo no encontrado... ¿Falsifico? (S/N)
```

5.1. 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 preexistente 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.

Código 5.1. Cálculo del baricentro de un conjunto de puntos en el plano

```
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(UNIT=8,FILE='coordenadas.dat', ACTION='READ')
  ! ------------------------------------------------------
  ! 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
  ! ------------------------------------------------------
  bar_x = bar_x/n
  bar_y = bar_y/n
  ! ------------------------------------------------------
  ! Abrir archivo de salida
  ! ------------------------------------------------------
  OPEN(UNIT=9,FILE='baricentro.sal', ACTION='WRITE')
  ! ------------------------------------------------------
  ! 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 PROGRAM baricentro
```

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

☞ 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.

☞ 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.

☞ 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).

☞ 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.

☞ 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 lee datos del mismo para usarlos (como el archivo

coordenadas.dat en nuestro ejemplo), y es de salida cuando los resultados son escritos en él (como el archivo baricentro.sal). La cláusula ACTION en la sentencia OPEN permite indicar si el archivo será tratado como un archivo de entrada o bien de salida. Específicamente, ACTION='READ' indica que el archivo será abierto para lectura, con lo que cualquier intento de escribir sobre el mismo producirá un error. Esta cláusula es entonces apropiada para un archivo de entrada. Por el contrario, ACTION='WRITE' indica que el archivo será abierto para escritura solamente, con lo que un intento de lectura sobre el mismo conduce a un error. Esta cláusula es apropiada para un archivo de salida.

☞ Por otra parte, es claro 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 (UNIT=número, FILE='nombre del archivo', ACTION='READ', &
         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[1] 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 /= 0) THEN
   WRITE(*,*) 'El archivo de entrada no puede ser leído'
   STOP
ENDIF
```

☞ En el caso de un archivo de salida, éste puede o no existir previamente. Si, en el caso que exista, nos interesa *no* sobreescribir el archivo, entonces la clásula apropiada es STATUS='NEW'. De este modo, si el archivo existe previamente la sentencia OPEN generará un error y con ello evitaremos sobreescribir los datos que contenga el archivo. Si el archivo no existe, entonces la sentencia OPEN lo creará (estando vacío hasta que se escriba algo en él). Nótese que aún en este caso puede un error que impida crear el archivo, por ejemplo, por falta de permisos adecuados. Nuevamente la cláusula IOSTAT nos permite manejar estas situaciones de error.

```
OPEN (UNIT=número, FILE='nombre del archivo', ACTION='WRITE', &
        STATUS='NEW', IOSTAT=variable entera)
IF (variable entera /= 0) THEN
  WRITE(*,*) 'El archivo de salida no puede ser escrito. Posiblemente exista.'
  STOP
ENDIF
```

Si, por el contrario, no nos importa preservar el archivo de salida original en el caso que exista, entonces la clásula apropiada es STATUS='REPLACE'.

```
OPEN (UNIT=número, FILE='nombre del archivo', ACTION='WRITE', &
            STATUS='REPLACE', IOSTAT=variable entera)
IF (variable entera /= 0) THEN
  WRITE(*,*) 'El archivo de salida no puede ser escrito.'
  STOP
ENDIF
```

☞ 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

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

detectar el final del archivo conforme éste se va recorriendo. Esto puede lograrse agregando la cláusula `IOSTAT` 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 >= 0)
 READ(número,*,IOSTAT=io) variables
 IF(io == 0) THEN
   n = n+1
   procesar variables leídas
  ENDIF
ENDDO
```

Aquí `io` es una variable entera para controlar los errores de lectura del archivo mientras que `n` es un contador entero cuyo valor final se corresponde con el número de datos válidos leidos.

5.1.1. Ejercicios

Ejercicio 5.1 Modificar el código 5.1 parametrizando los números de unidades lógicas y permitiendo que el usuario asigne, por teclado, los nombres de los archivos de entrada y salida. (*Ayuda*: los nombres de los archivos pueden ser asignados a variables carácter).

Ejercicio 5.2 Modificar el código anterior para controlar que el archivo de entrada exista previamente y que el archivo de salida puede ser escrito. Detener el programa con un mensaje apropiado si la correspondiente situación no se cumplen.

Ejercicio 5.3 Remover en el programa anterior la lectura del número de puntos N y reescribir la sección de lectura de manera que sea el propio programa determine el número de datos que lee.

5.2. Formatos.

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 5.1 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 las coordenadas con tres decimales en el resultado del programa del código 5.1 podemos escribir

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

☞ 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.

Tabla 5.1. 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 exponencial
ES*w.d*	Dato real en notación científica
A ó A*w*	Dato carácter
'x...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.

☞ 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 $w \geq d + 7$. La especificación de formato ES es similar, sólo que el número no es normalizado, esto es, la parte entera mostrada es un número comprendido entre 1 y 9. Debido a ésto, para un mismo ancho w, el descriptor ES mostrá un dígito significativo más que el descriptor E, debido a que el cero inicial de éste es ahora reemplazado por un dígito significativo.

☞ 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.

☞ 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.

☞ 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.

☞ 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.

☞ 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!

☞ 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.

☞ 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.

5.2.1. Ejercicios

Ejercicio 5.4 Establezca un formato con tres decimales para la tabla que debe crearse en el ejercicio **4.10**.

Ejercicio 5.5 En cierto observatorio meteorológico el promedio de la temperatura exterior en un día es anotada en un archivo día por día con el formato:

$$yyyy{-}mm{-}dd{:}\pm xxx.xx$$

Esto es, la fecha es descrita en el formato año-mes-día, con el año descrito con cuatro dígitos y el mes y el día con dos dígitos cada uno; la temperatura, por su parte, es un dato real con tres dígitos para la parte entera y dos dígitos para la parte decimal. Los dos datos están separados por el signo de dos puntos. Realizar un programa para crear un nuevo archivo en base al anterior pero con la fecha descrita en el formato día/mes/año.

Ejercicio 5.6 Si la especificación de formato no es adecuada para los datos considerados, entonces los mismos pueden ser mostrados incorrectamente. Pruebe los siguientes ejemplos:

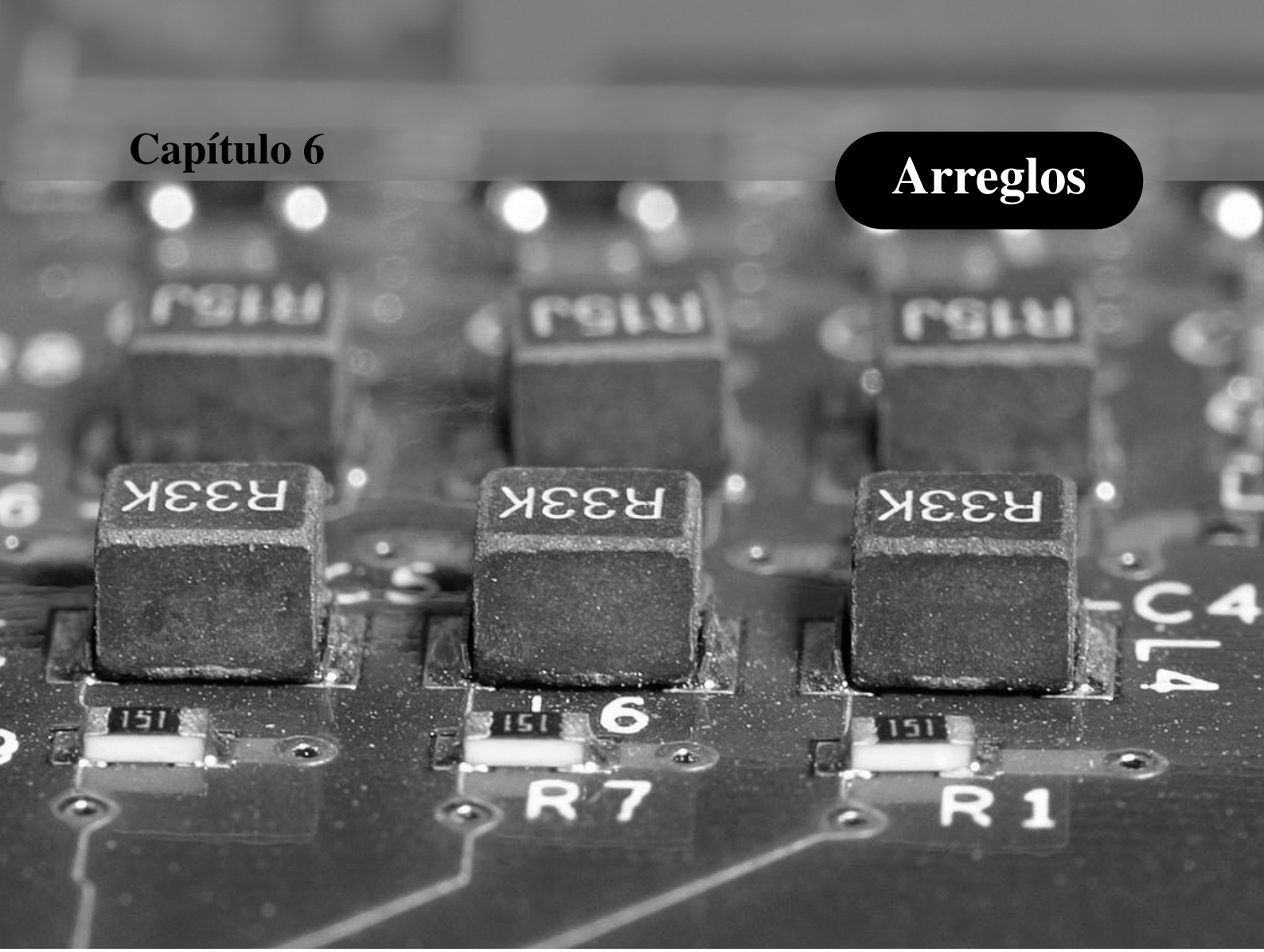
```
WRITE(*,'(I4, 2X, I4)') 12, 12345
WRITE(*,'(F6.2)') 0.12345
WRITE(*,'(F6.2)') 123.45
WRITE(*,'(F6.2)') 12345.0
```

Ancho ajustable en los descriptores I y F

Con el fin de utilizar el menor espacio posible en la salida de datos, los descriptores de formato `I` y `F` permiten especificar un ancho *w* igual a cero. Esto no sólo elimina todo espacio en blanco precedente al dato, sino que también evita el desbordamiento con asteriscos cuando la especificación del ancho del campo es pequeña para el mismo.

Capítulo 6

Arreglos



'Don't Panic'

— Douglas Adams, 'The Hitch-Hiker's Guide to the Galaxy'

6.1. Datos compuestos indexados: arreglos.

Los objetos matemáticos como *vectores*, *matrices* (y con más generalidad, *tensores*) pueden ser almacenados en Fortran en *arreglos* unidimensionales, bidimensionales (y multidimensionales), respectivamente. Un arreglo (*array* en inglés) es una estructura de datos *compuesta* formada por una colección ordenada de elementos del mismo tipo de datos (por lo que se dice que la estructura de datos es *homogénea*) y tal que los elementos se pueden acceder en cualquier orden simplemente indicando la posición que ocupa dentro de la estructura (por lo que se dice que es una estructura *indexada*). Un arreglo, como un todo, se identifica en el programa con un *nombre*, y un elemento particular del mismo es accedido a partir de cierto conjunto de *índices*, los cuales son, en Fortran, de tipo entero. La *dimensión* del arreglo es el número de índices necesarios para especificar un elemento, y su *tamaño* es el número total de elementos que contiene. Antes de ser usado, un arreglo debe ser declarado especificando el tipo de elementos que contendrá y el *rango* de valores que puede tomar cada índice (lo cual define la cantidad de elementos que lo componen). Así, un arreglo unidimensional `V` que contendrá 3 elementos reales y un arreglo bidimensional `A` que contendrá $5 \times 4 = 20$ elementos reales son declarados con la sentencia de declaración de tipo:

```
REAL :: V(3), A(5,4)
```

En el programa, un elemento particular de un arreglo es accedido a través del nombre del arreglo junto con los valores de los índices que especifican su posición. Así `V(1)` indica el primer elemento del arreglo unidimensional `V`, en tanto que `A(2,3)` corresponde al elemento en la fila 2 y la columna 3 de la matriz almacenada en el arreglo bidimensional `A`. Nótese que cada elemento de un arreglo constituye de por sí una variable, la cual puede utilizarse de la misma forma que cualquier variable del mismo tipo es utilizada.

☞ Al igual que las variables ordinarias, los valores de un arreglo debe ser inicializados antes de ser utilizados. Un arreglo puede ser inicializado en *tiempo de ejecución* a través de sentencias de asignación ya sea elemento a elemento a través de bucles `DO`, como ser, por ejemplo:

```
DO i=1,3
  V(i) = REAL(i)
ENDDO

DO i=1,5
  DO j=1,4
    A(i,j) = REAL(i+j)
  ENDDO
ENDDO
```

o bien a través del *constructor* de arreglos `[ ]` implementado en Fortran 2003:

```
V = [1.0,2.0,3.0]
A = RESHAPE([2.0,3.0,4.0,5.0,6.0, &
             3.0,4.0,5.0,6.0,7.0, &
             4.0,5.0,6.0,7.0,8.0, &
             5.0,6.0,7.0,8.0,9.0],[5,4])
```

donde para el arreglo bidimensional se utiliza la función intrínsica `RESHAPE` para construirlo a partir de un arreglo unidimensional cuyas componentes resultan de ordenar *por columna* los valores iniciales de la matriz. Estas construcciones permiten el uso de bucles `DO` implícitos lo cual es especialmente útil para inicializaciones más complicadas o arreglos de muchos elementos, como ser:

```
V = [( REAL(i), i=1,3)]
A = RESHAPE([(( REAL(i+j), i=1,5), j=1,4 )],[5,4])
```

También es posible inicializar todos los elementos de un arreglo a un mismo y único valor con las sentencias:

```
V = 0.0
A = 0.0
```

La inicialización puede realizarse también en *tiempo de compilación* realizando la asignación en la sentencia de declaración de tipo:

```
REAL :: V(3) = [1.0,2.0,3,0]
REAL :: A(5,4) = 0.0
```

Por supuesto, la inicialización también puede realizarse leyendo los datos del arreglo con sentencias `READ`, lo cual nos lleva a la siguiente nota.

☞ La forma más flexible de leer por teclado (o, con el número de unidad apropiado, de un archivo) *n* elementos de un vector para almacenarlo en un arreglo unidimensional `V` consiste en utilizar un bucle `DO` *implícito*:

```
READ(*,*) (V(i), i=1,n)
```

el cual, al involucrar sólo una sentencia de lectura, permite que los elementos del arreglo no necesariamente estén cada uno en una línea, sino que de hecho pueden estar dispuesto de cualquier manera (inclusive todos en la misma línea, o uno por línea, o varios en una línea y el resto en otra, etc.). De manera análoga, una matriz de $n \times m$ puede ser ingresada en la forma usual (esto es, fila por fila) para ser almacenada en un arreglo bidimensional `A` utilizando dos bucles `DO` implícitos:

```
READ(*,*) ((A(i,j),j=1,m),i=1,n)
```

Por otra parte, para el vector el remplazo de la sentencia `READ` por `WRITE` permite escribir los *n* elementos del vector sobre una única línea. Para imprimir una matriz en la forma usual (esto es, fila por fila), en cambio, recurrimos a la combinación de un bucle `DO` usual y uno implícito:

```
DO i=1,n
  WRITE(*,*) (A(i,j), j=1,m)
ENDDO
```

☞ Los arreglos pueden ser utilizados *como un todo* en sentencias de asignación y operaciones aritméticas siempre y cuando todos los arreglos involucrados tengan la misma *forma*, esto es, el mismo número de dimensiones y el mismo número de elementos en cada dimensión. Si esto es así, las correspondientes operaciones serán realizadas elemento a elemento. Por ejemplo, si `X`, `Y`, `V` son arreglos unidimensionales de 3 elementos, la sentencia:

```
V = 2.0*X + Y
```

permite asignar a `V` la combinación lineal resultante de sumar el doble de cada elemento de `X` con el correspondiente elemento de `Y`. Más aún, ciertas funciones intrínsecas de Fortran que son usadas con valores escalares también aceptarán arreglos como argumentos y devolverán un arreglo cuyo resultado será su acción elemento a elemento del arreglo de entrada. Tales funciones son conocidas como *funciones intrínsecas elementales*. Por ejemplo, para nuestro arreglo `V` inicializado como:

```
V = [-3.0,2,0,-1.0]
```

la función `ABS(V)` devolverá el arreglo `[3.0,2.0,1.0]`.

☞ Además de poder utilizar los elementos de un arreglo o, bajo las circunstancias indicadas, el arreglo como un todo en una sentencia, es posible utilizar secciones de los mismos. Por ejemplo, para nuestros arreglos `V` y `A`:

```
V(1:2) ó V(:2)  es el subarreglo de V de elementos V(1) y V(2),
V(1:3:2) ó V(::2)  es el subarreglo de V de elementos con índice impar: V(1) y V(3),
A(1:5,2) ó A(:,2)  es la segunda columna de la matriz contenida en el arreglo A,
A(3,1:4) ó A(3,:)  es la tercer fila de la matriz contenida en A.
```

☞ Un error muy común al trabajar con arreglos es hacer que el programa intente acceder a elementos que están fuera de los límites del mismo. Es *nuestra* responsabilidad asegurarnos que ésto no ocurra ya que tales errores *no* son detectados en tiempo de compilación. Un error de este tipo puede producir, durante la ejecución del programa, que el mismo se aborte con una condición de error denominada *violación de segmento* (*segmentation fault* en inglés) o bien que acceda a la posición de memoria que correspondería al elemento como si éste estuviera asignado. En este último caso, el programa no abortará pero, puesto que tal posición de memoria puede estar asignada para otro propósito, el programa conducirá a resultados erróneos.

El compilador `gfortan` permite detectar, *en tiempo de ejecución*, el intento de acceso fuera de límites de un arreglo si el código es compilado con la opción `-fbounds-check`. Sin embargo, el ejecutable originado correrá mucho más lento ya que la tal comprobación demanda mucho tiempo de cómputo. Por éste motivo *esta opción sólo debe utilizarse durante la fase de prueba del programa* con el propósito de detectar posibles errores.

✏ **Violación de segmento.**

La violación de segmento (*segmentation fault*) es una condición de error que ocurre cuando un programa intenta acceder a una posición de memoria que no le corresponde o cuando intenta acceder en una manera no permitida (por ejemplo, cuando se trata de escribir una posición de memoria que es de sólo lectura). Cuando ésta situación se produce el sistema operativo aborta el programa. Esto garantiza que ningún programa incorrecto (o malicioso) puede sobreescribir la memoria de otro programa en ejecución o del sistema operativo.

La posibilidad de operar con el arreglo como un todo, junto con las funciones intrínsecas específicas proporcionada por el lenguaje para manipular arreglos, permite que los algoritmos del álgebra lineal puedan ser implementados en Fortran en forma muy compacta. Por ejemplo, consideremos el cálculo de las normas vectoriales $\|x\|_\infty$, $\|x\|_1$ y $\|x\|_2$ para un vector $x = (x_1, x_2, \ldots, x_n)$ de $\mathbb{R}^n$. Por definición:

$$\|x\|_\infty = \max_{1 \le i \le n} \{|x_1|, |x_2|, \ldots, |x_n|\}, \qquad \|x\|_1 = \sum_{i=1}^{N} |x_i|, \qquad \|x\|_2 = \sqrt{\sum_{i=1}^{N} x_i^2}.$$

Asumamos que el vector x está almacenado en un arreglo unidimensional `x` de tamaño `n`. Una primera implementación de la norma infinito utilizaría un bucle explícito para recorrer el arreglo en busca de su máximo:

```
norm_inf = 0.0
do i=1,n
    norm_inf = MAX(norm_inf,ABS(x(i))
enddo
```

Sin embargo, a través de las funciones `ABS` y `MAXVAL`, donde esta última devuelve el máximo valor de un arreglo dado como argumento, la explicitación del bucle ya no resulta necesaria:

```
norm_inf = MAXVAL(ABS(x))
```

Similarmente, la implementación de la norma 1 a través de un bucle explícito

```
norm_1 = 0.0
do i=1,n
    norm_1 = norm_1 + ABS(x(i))
enddo
```

puede ser reimplementada con ayuda de la función `SUM`, la cual devuelve la suma de todos los elementos del arreglo pasado como argumento de la misma, en la linea:

```
norm_1 = SUM(ABS(x))
```

Finalmente, la norma 2 puede ser implementada a través de la función `DOT_PRODUCT`, la cual calcula el *producto interno euclideano* entre dos vectores, considerando la raíz cuadrada del producto interno consigo mismo:

```
norm_2 = SQRT(DOT_PRODUCT(x,x))
```

En Fortran 2008 se incorporó la función implícita `NORM2` con lo cual el cálculo de la norma 2 simplemente se reduce a invocar esta función [1]:

```
norm_2 = NORM2(x)
```

[1] La función `NORM2` está disponible en el compilador `gfortran` a partir de su versión 4.6.

De manera similar podemos considerar el cómputo de las normas matriciales $\|A\|_\infty$ y $\|A\|_1$ de una matriz cuadrada real $A = (a_{ij})$ de $n \times n$, siendo

$$\|A\|_\infty = \max_{1\le i\le n} \sum_{j=1}^{N} |a_{ij}|, \qquad \|A\|_1 = \max_{1\le j\le n} \sum_{i=1}^{N} |a_{ij}|.$$

Asumiendo que la matriz A se encuentra almacenada en una arreglo bidimensional `A` de tamaño `n` × `n`, tales cantidades pueden implementarse como

```
norm_inf = MAXVAL(SUM(ABS(A),2))
norm_1   = MAXVAL(SUM(ABS(A),1))
```

donde hemos aprovechado la posibilidad que tiene la función intrínseca `SUM` de indicar como segundo argumento la dimensión sobre la cual realizar la suma de los elementos.

6.1.1. Ejercicios

Ejercicio 6.1 Sea A una matriz de $n \times p$ y B una matriz de $p \times m$ elementos reales, la *multiplicación matricial* de A por B es la matriz C de $n \times m$ elementos c_{ij} dados por:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

Así, por ejemplo,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 15 & 14 \end{bmatrix}.$$

a) Implementar un programa Fortran para la multiplicación de dos matrices de acuerdo con la definición anterior.

b) Simplificar el algoritmo notando que el elemento c_{ij} puede considerarse como el producto interno de la fila i-ésima de A por la columna j-ésima de B.

c) Finalmente utilizar la función intrínsica `MATMUL(A,B)` que Fortran posee para la multiplicación matricial.

6.2. Asignacion estática y dinámica de memoria.

En la asignación de tipo dada para los arreglos `V` y `A` de la sección anterior, sus tamaños ha sido predefinidos en tiempo de compilación y por lo tanto no varían durante la ejecución del programa. Se dice, por lo tanto, que la asignación de memoria para los arreglos es *estática*. Nótese que si el tamaño de los arreglos con los que se trabajará no es conocido *a priori*, esto obliga a definir los arreglos de tamaños suficientemente grande como para contenerlos. La manera más flexible de proceder consiste en asignar el rango de sus índices a través de parámetros enteros. Así el siguiente fragmento de código define un arreglo unidimensional `V` que puede contener un vector de a lo más `NMAX` elementos reales, y un arreglo bidimensional `A` que puede contener una matriz real de a lo más `NMAX` filas y `MMAX` columnas, donde fijamos, para nuestros propósitos `NMAX` = `MMAX` = 6.

```
INTEGER, PARAMETER :: NMAX = 6
INTEGER, PARAMETER :: MMAX = 6
REAL :: V(NMAX), A(NMAX,MMAX)
```

De este modo el vector de 3 elementos está contenido en el subarreglo `V(1:3)` de `V`, mientras que la matriz de 5×4 está almacenada en el subarreglo `A(1:5,1:4)`. Si, posteriormente, el problema requiere de arreglos de mayor tamaño que los máximos asignados, simplemente ajustamos los valores de los parámetros y recompilamos el programa.

Como una alternativa más eficiente en el uso de la memoria disponible en el sistema, Fortran 90 dispone también de un esquema conocido como *asignación dinámica de memoria*, donde el tamaño de los arreglos son asignados durante la *ejecución* del programa. Para utilizar esta característica los arreglos deben ser declarados con el atributo ALLOCATABLE. En nuestro ejemplo:

```
REAL, ALLOCATABLE :: V(:), A(:,:)
```

Esta instrucción indica al compilador que los objetos V y A son un arreglo unidimensional y bidimensional, respectivamente, cuyos tamaños serán especificados *cuando el programa esté en ejecución*. Esto es efectuado a través de la sentencia ALLOCATE, como ser, para nuestro ejemplo:

```
ALLOCATE(V(3),STAT=variable entera)
ALLOCATE(A(5,4),STAT=variable entera)
```

Si el valor devuelto en la variable entera asignada por la cláusula STAT es igual a cero, entonces la asignación se ha realizado sin problemas y a partir de este momento podemos trabajar con los mismos. Si, en cambio, el valor devuelto es no nulo, ha ocurrido un error en la asignación de memoria con lo que debemos proceder a tratar esta situación de error (usualmente abortamos el programa). Una vez que los arreglos asignados no sean más necesarios, debemos *liberar* la memoria utilizada con la sentencia DEALLOCATE:

```
DEALLOCATE(V)
DEALLOCATE(A)
```

Para ejemplificar los puntos anteriores consideremos la implementación de un programa que lea por teclado una matriz (de dimensiones especificadas previamente también por teclado) y a continuación imprimir la misma y su transpuesta. Una implementación con asignación estática de memoria es dada en el siguiente código.

Código 6.1. Lectura e impresión de una matriz y su transpuesta (versión estática)

```
PROGRAM implementacion_estatica
  IMPLICIT NONE
  INTEGER, PARAMETER :: NMAX = 10
  INTEGER, PARAMETER :: MMAX = 10
  REAL :: A(NMAX,MMAX)
  INTEGER :: n, m, i, j

  READ(*,*) n,m
  READ(*,*) ((A(i,j),j=1,m),i=1,n)

  DO i=1,n
     WRITE(*,*) (A(i,j), j=1,m)
  ENDDO

  WRITE(*,*)

  DO i=1,m
     WRITE(*,*) (A(j,i), j=1,n)
  ENDDO

END PROGRAM implementacion_estatica
```

En tanto que el siguiente código hace uso de la asignación dinámica de memoria.

Código 6.2. Lectura e impresión de una matriz y su transpuesta (versión dinámica)

```
PROGRAM implementacion_dinamica
  IMPLICIT NONE
```

```
  REAL, ALLOCATABLE :: A(:,:)
  INTEGER :: n, m, i, j, ok

  READ(*,*) n,m
  ALLOCATE(A(n,m), STAT=ok)
  IF (ok /= 0) THEN
     WRITE(*,*) 'Error de asignación de memoria'
     STOP
  ENDIF
  READ(*,*) ((A(i,j),j=1,m),i=1,n)

  DO i=1,n
     WRITE(*,*)  (A(i,j), j=1,m)
  ENDDO

  WRITE(*,*)

  DO i=1,m
     WRITE(*,*)  (A(j,i), j=1,n)
  ENDDO

  DEALLOCATE(A)

END PROGRAM implementacion_dinamica
```

6.3. Arreglos en subprogramas.

Los arreglos pueden ser pasados a subrutinas o funciones de diversas maneras, aunque la forma más flexible consiste es aquella que declara los argumentos que recibirán arreglos en *forma asumida* indicando simplemente los tamaños de cada dimensión del arreglo mediante el caracter `:`. Dentro de la subrutina, el tamaño del arreglo a lo largo de cada dimensión puede obtenerse vía la función intrínseca `SIZE`. En virtud de ésto los arreglos pasados deben ser del tamaño realmente utilizado, lo cual no es un problema para un arreglo declarado dinámicamente, pero para el caso de la asignación estática, debe pasarse el subarreglo que contenga efectivamente los datos y no el arreglo de trabajo total. Además subprogramas escritos de este modo *requieren una interfaz explícita* ya sea alojándolos dentro de un módulo o escribiendo la interfaz vía la instrucción `INTERFACE` en el programa que realiza la llamada.

Para ejemplificar, consideremos la implementación de una función para calcular la *traza* de una matriz A de $n \times n$ elementos reales, esto es, $\mathrm{tr}A = \sum_{i=1}^{n} a_{ii}$. Dicha función, que sólo requiere como argumento de entrada la matriz cuya traza quiere calcularse, la implementamos dentro de un módulo de manera de hacer explícita su interfaz.

Código 6.3. Función para calcular la traza de una matriz

```
MODULE op_matrix
  CONTAINS
    FUNCTION tr(A)
      IMPLICIT NONE
      REAL :: tr
      REAL, INTENT(IN) :: A(:,:)
      INTEGER :: i
      tr = SUM([(A(i,i),i=1,SIZE(A,1))])
    END FUNCTION tr
END MODULE op_matrix
```

Asumiendo, por simplicidad, que la dimensión n de la matriz y ella misma es ingresada fila por fila desde el teclado, el siguiente programa implementa el uso de la función `tr` que hemos construido para una matriz

almacenada en un arreglo dinámico.

Código 6.4. Calculo de la traza de una matriz (versión dinámica)

```
PROGRAM calculo_traza
  USE op_matrix
  IMPLICIT NONE
  REAL, ALLOCATABLE :: A(:,:)
  INTEGER :: n,i,j, err

  READ(*,*) n
  ALLOCATE(A(n,n),STAT=err)
  IF ( err /= 0 ) THEN
     WRITE(*,*) "Error de memoria."
     STOP
  ENDIF
  READ(*,*) ((A(i,j),j=1,n),i=1,n)

  WRITE(*,*) tr(A)

END PROGRAM calculo_traza
```

Si, en cambio, la matriz es almacenada en un arreglo estático, el código adecuado es el siguiente.

Código 6.5. Calculo de la traza de una matriz (versión estática)

```
PROGRAM calculo_traza
  USE op_matrix
  IMPLICIT NONE
  INTEGER, PARAMETER :: NMAX = 10
  REAL :: A(NMAX,NMAX)
  INTEGER :: n,i,j

  READ(*,*) n
  READ(*,*) ((A(i,j),j=1,n),i=1,n)

  WRITE(*,*) tr(A(1:n,1:n))

END PROGRAM calculo_traza
```

6.3.1. Ejercicios

Ejercicio 6.2 Implementar como una subrutina de Fortran el *método (modificado) de ortogonalización de Gram-Schmidt* en $\mathbb{R}^n$, esto es, determinar para el subespacio generado por un conjunto de m vectores linealmente independientes $\{v_1, v_2, \ldots, v_m\}$ de n elementos reales ($m \leq n$) una base ortonormal $\{w_1, w_2, \ldots, w_m\}$ de dicho subespacio, según el siguiente algoritmo:

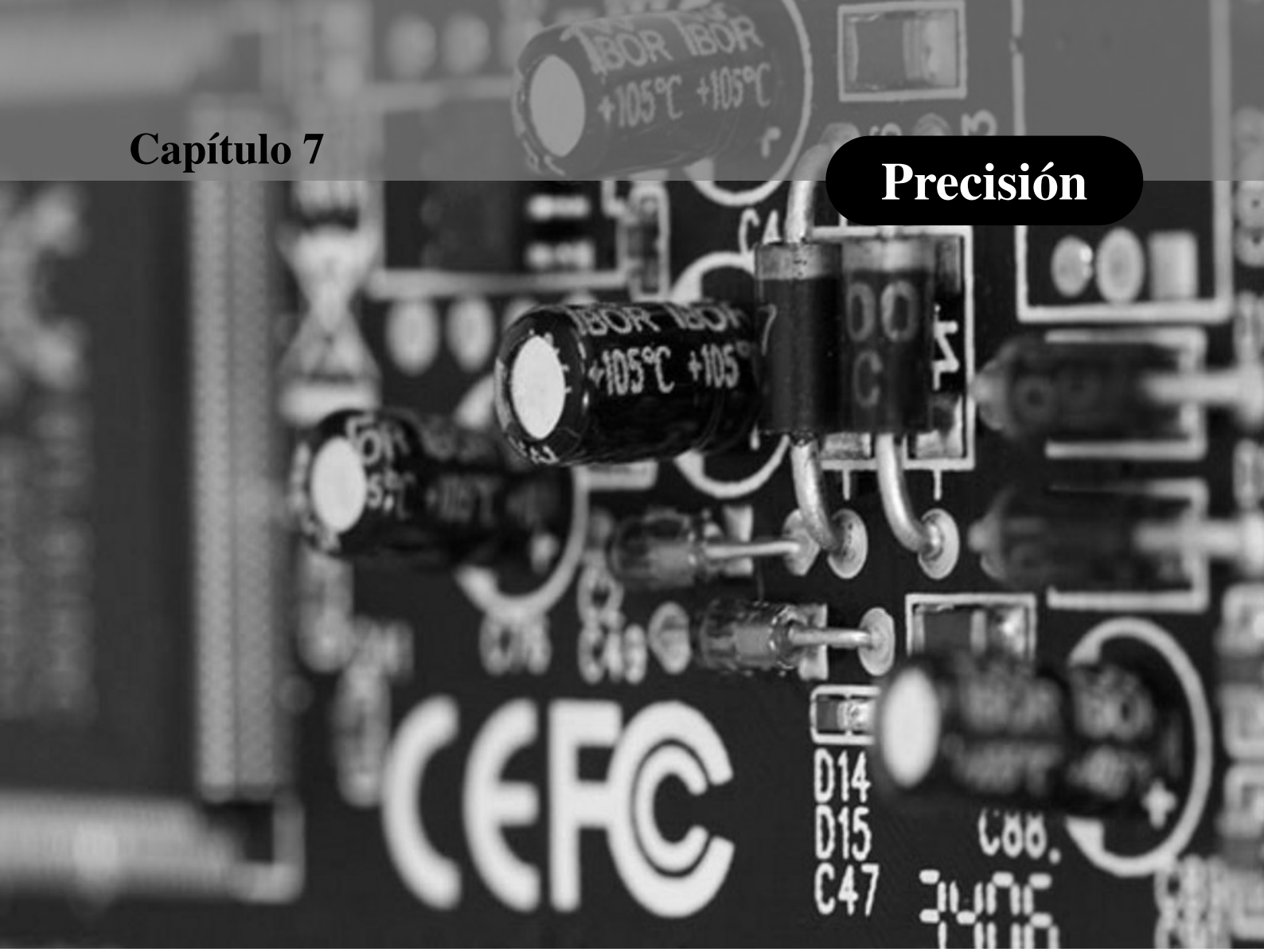
Dados $v_1, v_2, \ldots, v_m$
Para $j = 1, 2, \ldots, m$
  Tomar $w_j = v_j$
  Para $i = 1, 2 \ldots, j-1$
    Calcular $r_{ij} = \langle w_i | w_j \rangle$
    Tomar $w_j = w_j - r_{ij} w_i$
  Calcular $r_{jj} = \|w_j\|_2$
  Tomar $w_j = \dfrac{w_j}{r_{jj}}$

Ayuda: Para la implementación disponga los m vectores v_j (cada uno de n componentes) como las columnas de una matriz A de $n \times m$. De esta manera la subrutina recibe como argumento de entrada dicha matriz y reescribe,

en su salida, las columnas de dicha matriz con los vectores ortonormalizados buscados. Aplique la subrutina a los siguientes vectores de $\mathbb{R}^4$: $v_1 = (1,0,0,-1)$, $v_2 = (1,2,0,-1)$, $v_3 = (3,1,1,-1)$.

Capítulo 7

Precisión



Sólo hay 10 tipos de personas:
Las que saben binario y las que no.

7.1. Representación de punto flotante.

Como ya hemos mencionado, en las computadoras personales actuales, un dato numérico de tipo `REAL` tiene una precisión de unos seis a siete dígitos significativos y puede representar números en el rango de 10^{-38} a 10^{38}. Sin embargo, muchas aplicaciones científicas requieren un mayor número de dígitos significativos y un rango más amplio. Por ello, Fortran dispone de una extensión del tipo de dato real que permite una precisión entre 15 y 16 dígitos significativos con un rango de 10^{-308} a 10^{308}. Una comprensión cabal de estas dos clases para los tipos de dato real nos obliga a discutir la representación interna de los números reales en una computadora, la denominada *representación de punto flotante*.[1]

Debido a su naturaleza finita, una computadora (y de hecho todo dispositivo de cálculo) puede representar a los números reales sólo con cierta *precisión* y dentro de cierto *rango*. Por precisión entendemos el número de dígitos significativos que puede ser preservado en la representación de un número, mientras que el rango indica la diferencia entre la magnitud de los mayores y menores números que pueden ser representados. Por otra parte, virtualmente todas las computadoras actuales almacenan y operan internamente con representaciones

[1] Este capítulo se basa en el apunte *Representación de los números en la computadora* de mi autoría, en el cual se discute con mayor profundidad la anatomía de la representación interna de los números reales y enteros y las consecuencias de cara al usuario de dicha representación.

de los números no en el sistema decimal que utilizamos cotidianamente, sino en el denominado *sistema binario*. Si bien la conversión se realiza internamente y por lo tanto estamos tentados a asumir que el hecho de que la computadora opere de este modo es irrelevante para el usuario, tal representación deja sus huellas en un hecho tan simple como que, mientras el número real 0.5 puede ser representado exactamente, no sucede lo mismo con el número real 0.1.

Cotidianamente para representar los números utilizamos un *sistema posicional* de *base* 10: el *sistema decimal*. En este sistema los números son representados usando diez diferentes caracteres, llamados *dígitos decimales*, a saber, $0,1,2,3,4,5,6,7,8,9$. La magnitud con la que un dígito a_i contribuye al valor del número depende de su posición en el número de manera tal que la representación decimal

$$(-1)^s(a_n a_{n-1}\cdots a_1 a_0.a_{-1}a_{-2}\cdots)$$

corresponde al número

$$(-1)^s(a_n 10^n + a_{n-1}10^{n-1} + \cdots + a_1 10^1 + a_0 10^0 + a_{-1}10^{-1} + a_{-2}10^{-2} + \ldots),$$

donde s depende del signo del número ($s = 0$ si el número es positivo y $s = 1$ si es negativo). De manera análoga se puede concebir otros sistemas posicionales con una base distinta de 10. En principio, cualquier número natural $\beta \geq 2$ puede ser utilizado como base. Entonces, fijada una base, todo número real admite una *representación posicional* en la base β de la forma

$$(-1)^s(a_n \beta^n + a_{n-1}\beta^{n-1} + \cdots + a_1 \beta^1 + a_0 \beta^0 + a_{-1}\beta^{-1} + a_{-2}\beta^{-2} + \ldots),$$

donde los coeficientes a_i son los "dígitos" en el sistema con base β, esto es, enteros positivos tales que $0 \leq a_i \leq \beta - 1$. Los coeficientes $a_{i\geq 0}$ se consideran como los dígitos de la *parte entera*, en tanto que los $a_{i<0}$, son los dígitos de la *parte fraccionaria*. Si, como en el caso decimal, utilizamos un punto para separar tales partes, el número es representado en la base β como

$$(-1)^s(a_n a_{n-1}\cdots a_1 a_0.a_{-1}a_{-2}\cdots)_\beta,$$

donde hemos utilizado el subíndice β para evitar cualquier ambigüedad con la base escogida. Aunque cualquier número natural $\beta \geq 2$ define un sistema posicional, en el ámbito computacional sólo son de interés los sistemas *decimal* ($\beta = 10$), *binario* ($\beta = 2$), *octal* ($\beta = 8$) y *hexadecimal* ($\beta = 16$). El sistema binario consta sólo de los dígitos 0 y 1, llamados *bits* (del inglés *binary digits*). Por su parte, el sistema octal usa dígitos del 0 al 7, en tanto que el sistema hexadecimal usa los dígitos del 0 al 9 y las letras A, B, C, D, E, F[2].

Por otra parte, cuando se quiere representar números reales sobre un amplio rango de valores recurrimos a la *notación científica*. En dicha notación sólo unos pocos dígitos no necesarios, por ejemplo, 976 000 000 000 000 se representa como 9.76×10^{14} y 0.0000000000000976 como 9.76×10^{-14}. Aquí el punto decimal se mueve dinámicamente a una posición conveniente y se utiliza el exponente de 10 para registrar la posición del punto decimal. En particular, todo número real no nulo puede ser escrito en forma única en la notación científica *normalizada*

$$(-1)^s\, 0.a_1 a_2 a_3 \cdots a_t a_{t+1} a_{t+2} \cdots \times 10^e,$$

siendo el dígito $a_1 \neq 0$. De manera análoga, todo número real no nulo puede representarse en forma única, respecto la base β, en la *forma de punto flotante normalizada*:

$$(-1)^s\, 0.a_1 a_2 a_3 \cdots a_t a_{t+1} a_{t+2} \cdots \times \beta^e,$$

donde los "dígitos" a_i respecto de la base β son enteros positivos tales que $1 \leq a_1 \leq \beta - 1$, $0 \leq a_i \leq \beta - 1$ para $i = 1,2,\ldots$ y constituyen la parte fraccional o *mantisa* del número, en tanto que e, el cual es un número entero llamado el *exponente*, indica la posición del punto correspondiente a la base β. El problema que se presenta ahora es que en todo dispositivo de cálculo, como una computadora o calculadora, el número de dígitos posibles para representar la mantisa es *finito*, digamos t dígitos en la base β, y el exponente puede variar sólo dentro

[2]Para sistemas con base $\beta > 10$ es usual reemplazar los dígitos $10,11,\ldots,\beta-1$ por las letras A, B, C, ...

de un rango finito $L \leq e \leq U$ (con $L < 0$ y $U > 0$). Esto implica que sólo un conjunto *finito* de números reales pueden ser representados, los cuales tienen la forma

$$(-1)^s 0.a_1 a_2 a_3 \cdots a_t \times \beta^e.$$

Tales números se denominan *números de punto flotante de precisión t en la base* β *y rango* (L,U). Al conjunto de los mismos lo denotaremos por $\mathbb{F}(\beta,t,L,U)$.

☞ El número de elementos del conjunto $\mathbb{F}$, esto es, la cantidad de números de puntos flotantes de $\mathbb{F}$, es

$$2(\beta-1)\beta^{t-1}(U-L+1).$$

☞ Debido a la normalización el cero *no* puede ser representado como un número de punto flotante y por lo tanto está excluido del conjunto $\mathbb{F}$.

☞ Si $x \in \mathbb{F}$ entonces su opuesto $-x \in \mathbb{F}$.

☞ El conjunto $\mathbb{F}$ está acotado tanto superior como inferiormente:

$$x_{\text{mín}} = \beta^{L-1} \leq |x| \leq x_{\text{máx}} = \beta^U(1-\beta^{-t}),$$

donde $x_{\text{mín}}$ y $x_{\text{máx}}$ son el menor y mayor número de punto flotante positivo representable, respectivamente.

☞ Se sigue de lo anterior que en la recta de los números reales hay cinco regiones excluidas para los números de $\mathbb{F}$:

- Los números negativos menores que $-x_{\text{máx}}$, región denominada *desbordamiento* (*overflow*) *negativo*.
- Los números negativos mayores que $-x_{\text{mín}}$, denominada *desbordamiento a cero* (*underflow*) *negativo*.
- El cero.
- Los números positivos menores que $x_{\text{mín}}$, denominada *desbordamiento a cero* (*underflow*) *positivo*.
- Los números positivos mayores que $x_{\text{máx}}$, denominada *desbordamiento* (*overflow*) *positivo*.

☞ Los números de punto flotante no están igualmente espaciados sobre la recta real, sino que están más próximos cerca del origen y más separados a medida que nos alejamos de él.

☞ Una cantidad de gran importancia es el denominado *epsilon de la máquina* $\varepsilon_M = \beta^{1-t}$ el cual representa la distancia entre el número 1 y el número de punto flotante siguiente más próximo.

El hecho de que sólo el subconjunto $\mathbb{F}$ de los números reales es representable en una computadora implica que dado cualquier número real x, para ser representado, debe ser aproximado por un número de punto flotante de tal conjunto, al que denotaremos por $fl(x)$. La manera usual de proceder consiste en aplicar el *redondeo simétrico* a t dígitos a la mantisa de la representación de punto flotante normalizada (infinita) de x. Esto es, a partir de

$$x = (-1)^s 0.a_1 a_2 \ldots a_t a_{t+1} a_{t+2} \cdots \times \beta^e,$$

si el exponente e está dentro del rango $-L \leq e \leq U$, obtenemos $fl(x)$ como

$$fl(x) = (-1)^s 0.a_1 a_2 \ldots \widetilde{a}_t \times \beta^e, \qquad \widetilde{a}_t = \begin{cases} a_t & \text{si } a_{t+1} < \beta/2 \\ a_t + 1 & \text{si } a_{t+1} \geq \beta/2. \end{cases}$$

El error que resulta de reeemplazar un número real por su forma de punto flotante se denomina *error de redondeo*. Una estimación del mismo está dado en el siguiente resultado: *Todo número real x dentro del rango de los números de punto flotante puede ser representado con un error relativo que no excede la unidad de redondeo* **u**:

$$\frac{|x-fl(x)|}{|x|} \leq \mathbf{u} = \frac{1}{2}\,\varepsilon_M.$$

7.1.1. Ejercicios

Ejercicio 7.1 Mostrar que $(13.25)_{10} = (1101.01)_2 = (15.2)_8 = (\mathrm{D}.4)_{16}$.

Ejercicio 7.2 Considere el conjunto de números de punto flotante $\mathbb{F}(2,3,-1,2)$.

a) Determinar $x_{\text{mín}}$, $x_{\text{máx}}$, ε_M y el número de elementos de $\mathbb{F}$.
b) Determinar los números de punto flotante positivos del conjunto $\mathbb{F}$.
c) Graficar sobre la recta real los números de puntos flotantes determinados en el punto anterior.

Ejercicio 7.3 Determinar la representación de punto flotante decimal ($\beta = 10$) de 5 dígitos de π.

7.2. Números de punto flotante de simple y doble precisión

Con el fin de evitar la proliferación de diversos sistemas de puntos flotantes incompatibles entre sí a fines de la década de 1980 se desarrolló la norma o estandar IEEE754/IEC559[3] el cual es implementado en todas las computadoras actuales. Esta norma define dos formatos para la implementación de números de punto flotante en la computadora, ambos con base $\beta = 2$, pero con diferente precisión y rango, a saber:

- *precisión simple*: $\mathbb{F}(2,24,-125,128)$,
- *precisión doble*: $\mathbb{F}(2,53,-1021,1024)$.

Para el formato de precisión simple tenemos que

$$\begin{aligned} x_{\text{mín}} &= 2^{-126} \sim 10^{-38}, \\ x_{\text{máx}} &= 2^{128}(1-2^{-24}) \sim 10^{38} \\ \varepsilon_M &= 2^{-23} \sim 10^{-7}, \\ \mathbf{u} &= \frac{1}{2}\varepsilon_M = 2^{-24} \sim 6 \cdot 10^{-8}. \end{aligned}$$

lo cual implica unos 7 dígitos *decimales* significativos con un rango entre 10^{-38} y 10^{38}. En tanto que para el formato de precisión doble tenemos que

$$\begin{aligned} x_{\text{mín}} &= 2^{-1022} \sim 10^{-308}, \\ x_{\text{máx}} &= 2^{1024}(1-2^{-53}) \sim 10^{308}, \\ \varepsilon_M &= 2^{-52} \sim 10^{-16}, \\ \mathbf{u} &= \frac{1}{2}\varepsilon_M = 2^{-53} \sim 10^{-16}. \end{aligned}$$

lo cual implica unos 16 dígitos *decimales* significativos con un rango entre 10^{-308} y 10^{308}.

7.3. Implementación en Fortran de los números de punto flotante de simple y doble precisión.

Todos los compiladores Fortran admiten, al menos, dos *clases* para los tipos de datos reales: el primero, simple precisión y, el segundo tipo, doble precisión tal como se especifica en la norma IEEE754/IEC559. Para declarar el tipo de clase de una variable real se utiliza la siguiente sintaxis:

```
REAL(KIND=número de clase) ::  nombre de variable
```

[3] IEEE = Institute of Electrical and Electronics Engineeers, IEC = International Electronical Commission.

donde el *número de clase* es un número entero que identifica la clase de real a utilizar. Este número, para el compilador `gfortran`, es 4 en el caso de simple precisión y 8 para doble precisión. Si no se especifica la clase, entonces se utiliza la clase por omisión, la cual es la simple precisión.

☞ Dado que *el número de clase es dependiente del compilador* es recomendable asignar los mismos a constantes con nombres y utilizar éstas en todas las declaraciones de tipo. Esto permite asegurar la portabilidad del programa entre distintas implementaciones cambiando simplemente el valor de las mismas.

```
INTEGER, PARAMETER :: SP = 4    ! Valor de la clase de simple precisión
INTEGER, PARAMETER :: DP = 8    ! Valor de la clase de doble  precisión
...
REAL(KIND=SP) :: variables
REAL(KIND=DP) :: variables
```

☞ Constantes reales en el código son declaradas de una dada clase agregando a las mismas el guión bajo seguida del número de clase. Por ejemplo:

```
34.0             ! Real de clase por omisión
34.0_SP ! Real de clase SP
124.5678_DP ! Real de clase DP
```

Es importante comprender que, por ejemplo, `0.1_SP` y `0.1_DP` son números de punto flotante *distintos*, siendo el último guardado internamente con un número mayor de dígitos binarios.

☞ Una manera alternativa de especificar la clase de los tipos de datos reales y que resulta *independiente del compilador y procesador utilizado* consiste en seleccionar el número de clase vía la función intrínseca `SELECT_REAL_KIND`. Esta función selecciona automáticamente la clase del tipo real al especificar la precisión y rango de los números de punto flotante que se quiere utilizar. Para simple y doble precisión la asignación apropiada es como sigue:

```
INTEGER, PARAMETER :: SP = SELECTED_REAL_KIND(6,37)   ! simple precisión
INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND(15,307) ! doble  precisión
```

Una manera simple y eficiente de escribir un programa que pueda ser compilado con variables reales ya sea de una clase u otra según se requiera, consiste en utilizar un módulo para definir la precisión de los tipos reales y luego en el programa invocarlo especificando el tipo de clase vía un *alias* como ser `WP` (por *working precisión*, precisión de trabajo), la cual es utilizada para declarar los tipos de datos reales (variables y constantes). Específicamente, definimos el módulo `precision` como sigue:

Código 7.1. Módulo de precisión simple y doble

```
MODULE precision
    ! -----------------------------------------------------
    ! SP : simple precision de la norma IEEE 754
    ! DP : doble  precision de la norma IEEE 754
    !
    ! Uso: USE precision, WP => SP ó USE precision, WP => DP
    ! -----------------------------------------------------
    INTEGER, PARAMETER :: SP = SELECTED_REAL_KIND(6,37)
    INTEGER, PARAMETER :: DP = SELECTED_REAL_KIND(15,307)
END MODULE precision
```

Entonces podemos escribir un programa que se compile ya sea con reales de simple o doble precisión escogiendo apropiadamente la sentencia que importa el módulo. Por ejemplo:

Código 7.2. Uso del módulo de precisión

```
PROGRAM main
  USE precision, WP => SP   ! ó WP => DP
  IMPLICIT NONE
  REAL(KIND=WP) :: a

  a = 1.0_WP/3.0_WP
  WRITE (*,*) a

END PROGRAM main
```

> **¿Cuándo utilizar doble precisión?**
>
> La respuesta corta es *siempre*. Para las aplicaciones científicas la precisión de los resultados es generalmente crucial, con lo cual debe utilizarse la mejor representación de punto flotante disponible en la computadora. Nótese además que aún cuando los números de punto flotante de doble precisión utilizan el doble de dígitos binarios que los de simple precisión, en las computadoras personales (PC) el procesador matemático realiza internamente los cálculos con 80 bits independientemente de la precisión de los datos a ser procesados. Por lo tanto, la diferencia de velocidad entre cálculos en doble y simple precisión en una PC es ínfima.

7.3.1. Ejercicios

Ejercicio 7.4 Fortran dispone de un conjunto de funciones intrínsecas para determinar las propiedades de la representación de punto flotante implementada en una clase de tipo real. Utilizando el siguiente programa verifique que en una computadora personal la clase asignada a los datos de tipo real son efecticamente los números de punto flotante de precisión simple y doble de la norma IEEE754.

```
PROGRAM machar
  USE precision, WP => SP  ! ó WP => DP
  IMPLICIT NONE
  INTEGER :: i
  REAL(KIND=WP) :: x
  WRITE(*,*) ' base  = ', RADIX(i)
  WRITE(*,*) ' t     = ', DIGITS(x)
  WRITE(*,*) ' L     = ', MINEXPONENT(x)
  WRITE(*,*) ' U     = ', MAXEXPONENT(x)
  WRITE(*,*) ' x_max = ', HUGE(x)
  WRITE(*,*) ' x_min = ', TINY(x)
  WRITE(*,*) ' eps_M = ', EPSILON(x)
  STOP
END PROGRAM machar
```

7.4. Números especiales.

La condición de normalización sobre la mantisa de los números de punto flotante impide la representación del cero, por lo tanto debe disponerse de una representación separada del mismo. Por otra parte, en la aritmética de punto flotante pueden presentarse las tres siguientes condiciones excepcionales:

- una operación puede conducir a un resultado fuera del rango representable, ya sea porque $|x| > x_{\text{máx}}$ (*overflow*) o porque $|x| < x_{\text{mín}}$ (*underflow*),
- el cálculo puede ser una operación matemática indefinida, tal como la división por cero, o
- el cálculo corresponde a una operación matemática ilegal, por ejemplo 0/0 ó $\sqrt{-1}$.

Antes de la implementación de la norma IEEE754, frente a tales situaciones excepcionales las computadoras abortaban el cálculo y detenían el programa. Por el contrario, la norma IEEE754 define una aritmética *cerrada* en $\mathbb{F}$ introduciendo ciertos números especiales. De esta manera, con la implementación de la norma IEEE754 en las computadoras actuales, cuando un cálculo intermedio conduce a una de las situaciones excepcionales el resultado es asignado al número especial apropiado y los cálculos continúan. Así, la norma permite una *aritmética de no detención*.

- **Ceros.** En la norma IEEE754 el cero es representado por un número de punto flotante con una mantisa nula y exponente $e = L-1$, pero, como ninguna condición es impuesta sobre el signo, existen dos ceros: $+0$ y -0 (con la salvedad de que en una comparación se consideran iguales en vez de $-0 < +0$). Un cero con signo es útil en determinadas situaciones, pero en la mayoría de las aplicaciones el cero del signo es invisible.
- **Infinitos.** Cuando un cálculo produce un desbordamiento (*overflow*) positivo el resultado es asignado al número especial denominado *infinito positivo*, codificado como `+Infinity`[4]. De la misma manera, el cálculo de un desbordamiento negativo es asignado al número especial *infinito negativo*, codificado como `-Infinity`. Los infinitos permiten considerar también el caso excepcional de la división de un número no nulo por cero: el resultado es asignado al infinito del signo apropiado. Los infinitos son representados en el estandar por los números de punto flotante con mantisa nula y exponente $e = U+1$ con el correspondiente signo.
- **Números denormalizados.** Tradicionalmente si una operación producía un valor de magnitud menor que $x_{\text{mín}}$ (desbordamiento a cero, o *underflow*), el resultado era asignado a cero. Ahora bien, la distancia entre cero y $x_{\text{mín}} = \beta^{L-1}$ (el menor número de punto flotante positivo representable) es mucho mayor que la distancia entre este número y el siguiente por lo que la asignación a cero de una condición de *underflow* produce errores de redondeo excepcionalmente grandes. Para cubrir esta distancia y reducir así el efecto de desbordamiento a cero a un nivel comparable con el redondeo de los números de punto flotante se implementa el *desbordamiento a cero gradual* (*gradual underflow*) introduciendo los *números de punto flotante denormalizados*. Los números denormalizados son obtenidos removiendo en la representación de punto flotante la condición de que a_1 sea no nulo *solo* para los números que corresponden al mínimo exponente $e = L$. De esta manera la unicidad de la representación es mantenida y ahora es posible disponer de números de punto flotante en el intervalo $(-\beta^{L-1}, \beta^{L-1})$. La magnitud del más pequeño de estos números denormalizados es igual a β^{L-t}. De este modo, cuando el resultado de una operación tiene magnitud menor que $x_{\text{mín}}$ el mismo es asignado al correspondiente número de punto flotante denormalizado más próximo. En el estandar, los números denormalizados son representados como números de punto flotante con mantisa no nula y exponente $e = L-1$.
- **NaN.** Operaciones matemáticamente ilegales, como $0/0$ ó $\sqrt{x}$ para $x < 0$, son asignadas al número especial denominado *Not a Number* (no es un número), codificado como `NaN`. En el estandar un `NaN` es representado por un número de punto flotante con mantisa no nula y exponente $e = U+1$ (puesto que la mantisa no está especificada no existe un único `NaN`, sino un conjunto finito de ellos los cuales pueden utilizarse para especificar situaciones de excepción particulares).

Las operaciones aritméticas que involucran a los números especiales están definidas de manera de obtener resultados razonables, tales como

$$
\begin{gathered}
(\pm\texttt{Infinity}) + (+1) = \pm\texttt{Infinity} \\
(\pm\texttt{Infinity}) \cdot (-1) = \mp\texttt{Infinity} \\
(\pm\texttt{Infinity}) + (\pm\texttt{Infinity}) = \pm\texttt{Infinity} \\
(\pm\texttt{Infinity}) + (\mp\texttt{Infinity}) = \texttt{NaN} \\
1/(\pm 0) = \pm\texttt{Infinity} \qquad 1/(\pm\texttt{Infinity}) = \pm 0 \\
0/0 = \texttt{NaN} \qquad (\pm\texttt{Infinity})/(\pm\texttt{Infinity}) = \texttt{NaN} \\
0 \cdot (\pm\texttt{Infinity}) = \texttt{NaN}
\end{gathered}
$$

[4] Nótese que "infinito" no significa necesariamente que el resultado sea realmente ∞, sino que significa "demasiado grande para representar".

Por otra parte un `NaN` se propaga agresivamente a través de las operaciones aritméticas: *en cualquier operación donde un* `NaN` *participe como operando el resultado será un* `NaN`.

✏ **Anulando la aritmética de no detención de la norma IEEE754.**

La filosofía detrás de la aritmética de no detención de la norma IEEE754 es que el sistema de punto flotante extendido simplifica la programación en algunos casos, en particular cuando los cálculos involucran puntos singulares. Sin embargo, muchos usuarios la encuentran confusa y prefieren que los cálculos sean abortados con un apropiado mensaje de error. Para anular el comportamiento del estandar en las situaciones excepcionales se puede utilizar la opción `-ffpe-trap=invalid,zero,overflow,underflow,denormal` en la compilación del programa con el compilador `gfortran`.

7.4.1. Ejercicios

Ejercicio 7.5 Determinar los números de punto flotante denormalizados positivos asociados al conjunto $\mathbb{F}(2,3,-1,2)$.

Ejercicio 7.6 Compilar y ejecutar el siguiente programa para mostrar la acción de los números especiales.

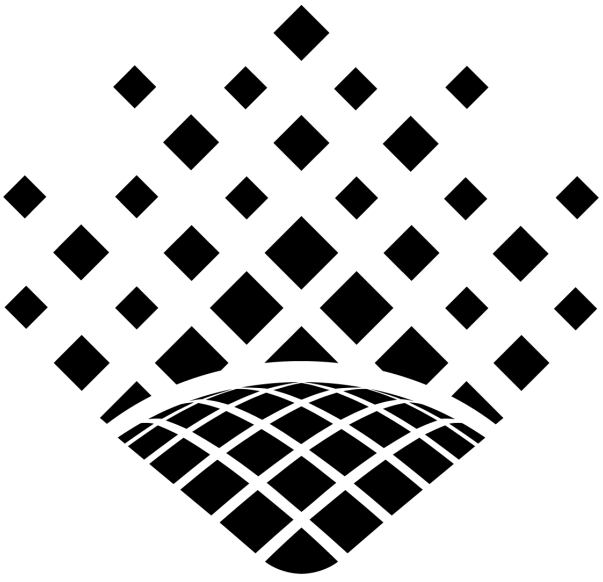
```
PROGRAM excepciones
IMPLICIT NONE
INTEGER :: i
REAL :: x,x_max,x_min,x_min_den,cero

x_max     = HUGE(x)
x_min     = TINY(x)
x_min_den = REAL(RADIX(i))**(MINEXPONENT(x)-DIGITS(x))
cero      = 0.0

WRITE (*,*) 'Desbordamiento                =', 2.0*x_max
WRITE (*,*) 'Desbordamiento gradual a cero =', x_min/2.0
WRITE (*,*) 'Menor numero denormalizado    =', x_min_den
WRITE (*,*) 'Desbordamiento a cero         =', x_min_den/2.0
WRITE (*,*) 'Division por 0                =', 1.0/cero
WRITE (*,*) '0/0                           =', cero/cero
WRITE (*,*) 'NaN + 1                       =', cero/cero + 1.0

STOP
END PROGRAM excepciones
```

Elementos de programación Fortran. Pablo J. Santamaría. Una introducción a la programación en Fortran 95/2003, haciendo énfasis en la escritura de código estructurado y modular.



Facultad de Ciencias
Astronómicas
y Geofísicas
UNIVERSIDAD NACIONAL DE LA PLATA