Programando en `Fortran`

Hans Cristian Muller Santa Cruz

2007

Índice general

Índice de figuras

Índice de cuadros

Parte I

Fortran

Capítulo I

Conceptos Básicos

I.1. Programas

En este primer capítulo se abordará aquellos conceptos básicos relacionados a un programa, de manera a uniformizar la terminología utilizada, profundizar y aclarar conceptos claves y tener los elementos mínimos para concebir y crear programas útiles a nuestros fines. Comenzamos con el primer concepto.

Un **programa** es un conjunto, una colección o una serie de instrucciones realizadas computacionalmente, cada una en un tiempo finito de duración, utilizadas para introducir, procesar o presentar datos.

Ahora bien, para que una computadora o varias computadoras puedan ejecutar el programa, éstas requieren que las instrucciones estén escritas en un **código ejecutable** o **código binario** o **lenguaje máquina**, a través del cual las computadoras sean capaces de llevar a cabo las instrucciones.

El código ejecutable, binario o lenguaje máquina, tal como indica el nombre, está formado por instrucciones que contienen ceros y unos, que en teoría una persona formada podría interpretarlos y escribirlos, pero que en la práctica es casi imposible, por no decir imposible. Por lo tanto, la persona, que desea crear un programa, escribe las instrucciones utilizando un **lenguaje de programación**, lo que da como resultado el **código fuente** del programa, que debe ser traducido en código ejecutable, para que los dispositivos computacionales puedan ejecutar el programa.

I.1.1. Lenguajes de Programación

Clarificando, un **lenguaje de programación** es un lenguaje que puede ser utilizado para controlar el comportamiento de una máquina, particularmente una computadora. Consiste en un conjunto de reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos, respectivamente. El código fuente producido al utilizar un lenguaje de programación, para crear un programa, se lo guarda en archivos de tipo texto, que en el caso de `FORTRAN`, tienen la extensión: `.f90`, `.f`, `.for`, *etc*, y en el lenguaje de Programación `C`, la extensión `.c`.

Aunque muchas veces se usa lenguaje de programación y lenguaje informático como si fuesen sinónimos, no tiene por qué ser así, ya que los lenguajes informáticos engloban a los lenguajes de programación y a otros más, como, por ejemplo, el HTML.

Un lenguaje de programación permite a un programador especificar de manera precisa: sobre qué datos una computadora debe operar, cómo deben ser estos almacenados y transmitidos y qué acciones debe tomar bajo una variada gama de circunstancias. Todo esto, a través de un lenguaje que intenta estar relativamente próximo al lenguaje humano o natural.

Los procesadores usados en las computadoras son capaces de entender y actuar según lo indican programas escritos en un lenguaje fijo llamado lenguaje de máquina. Todo programa escrito en otro lenguaje puede ser ejecutado de dos maneras:

1. Mediante un programa que va adaptando las instrucciones conforme son encontradas. A este proceso se lo llama **interpetar** y a los programas que lo hacen se los conoce como **intérpretes**.

2. Traduciendo este programa al programa equivalente escrito en lenguaje de máquina. A ese proceso se lo llama **compilar** y al traductor se lo conoce como **compilador**.

I.1.2. Compilación

Debido a que este texto está destinado al lenguaje de programación `FORTRAN`, se precisará cuál es el proceso de compilación, que permite convertir un programa en código fuente a código máquina.

Un compilador es un programa que, a su vez, traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, generando un programa equivalente. Usualmente el segundo lenguaje es código máquina, pero también puede ser simplemente texto. Este proceso de traducción se conoce como compilación.

La razón principal para querer usar un compilador es querer traducir un programa de un lenguaje de alto nivel, a otro lenguaje de nivel inferior (típicamente lenguaje máquina). De esta manera un programador puede diseñar un programa en un lenguaje mucho más cercano a como piensa un ser humano, para luego compilarlo a un programa más manejable por una computadora.

En la figura I.1.1, extraída de **Wikipedia** (http://es.wikipedia.org/wiki/Compilador), se tiene un esquéma de cómo es el proceso de compilación de un programa.

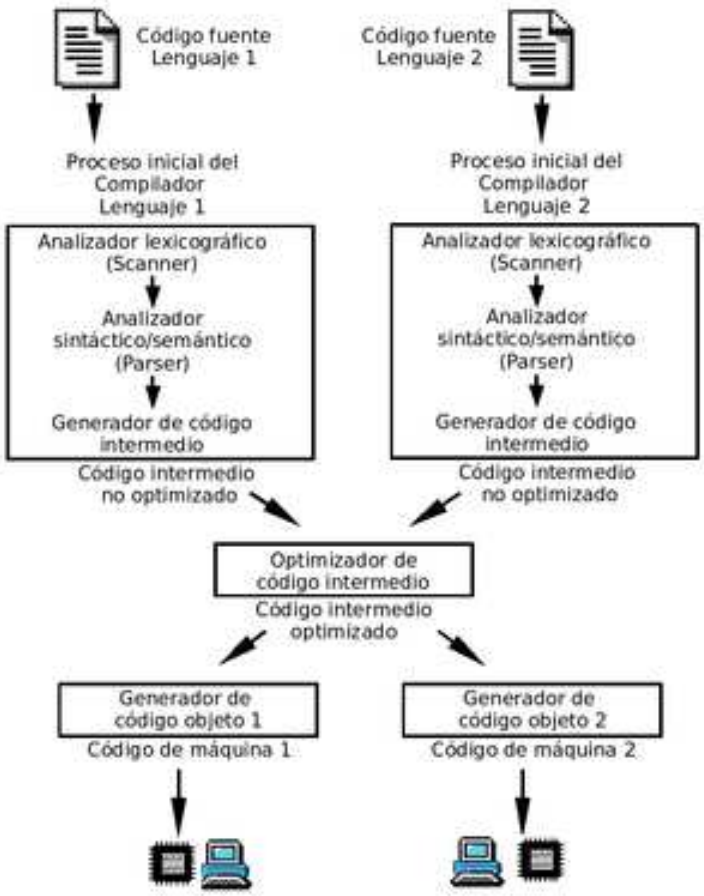


Figura I.1.1: Diagrama en bloques de la operación de un compilador

Tal como puede observarse en la figura I.1.1, el proceso de compilación puede desglozarse en dos subprocesos:

- El primero en el cual el compilador verifica que el código fuente esté bien escrito, y al decir "bien escrito" el código fuente debe cumplir las reglas sintácticas y semánticas del lenguaje de programación, sin preocuparse sobre la pertinencia del conjunto de instrucciones del programa. Resultado de este subproceso, si es que no hay errores, es un código intermedio, llamado **código objeto**, manifestado, para `FORTRAN` y `C`, en archivos de extension `.o`.

- El segundo subproceso, consiste en enlazar (*link* en inglés) el código objeto producido en el primer subproceso, eventualmente con otros códigos que se encuentran en archivos biblioteca, cuya extensión usual son `.a` o `.so`. Luego optimizar este código y convertirlo finalmente en código ejecutable, manifestado en archivos ejecutables.

I.1.3. Estructura de un Programa

Los programas elaborados utilizando un lenguaje de programación de alto nivel, como es el caso del `FORTRAN`, están estructurados lógicamente de una manera jerarquica en unidades programáticas. De esta manera, todo programa suceptible de ser ejecutado por la computadora o computadoras, debe tener una unidad principal, llamada **programa principal**, que contiene las instrucciones que definirán el objetivo del programa, pudiendo recurrir a otras unidades programáticas o subprográmas de nivel jerarquico inferior, para poder realizar algunas de las instrucciones del programa principal.

Los subprogramas de nivel jerárquico inferior son, en consecuencia, son grupos de instrucciones, que agrupadas constituyen una instrucción del programa principal y aplicando este principio de jerarquización, un subprograma puede recurrir a otros subprogramas de nivel jerárquico de nivel más inferior, lo que da la estructura jerárquica esquematizada en la figura I.1.2.

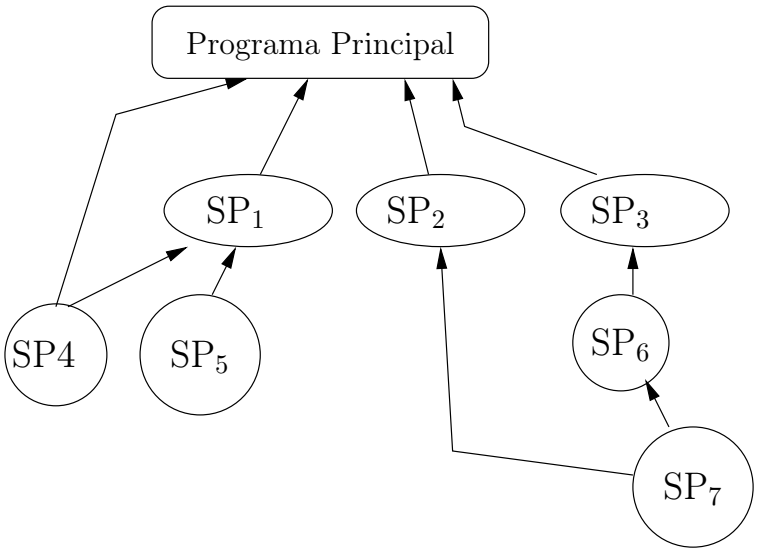


Figura I.1.2: Esquema de la Estructura de un Programa

En consecuencia, el nivel de jerarquía puede ser definido como sigue: el nivel más alto corresponde al primer nivel, el programa principal es de primer nivel o nivel 1, el nivel de un subprograma corresponde al nivel inmediatamente inferior del subprograma de nivel más inferior que sirve. A manera de ilustración, en la figura I.1.2, el programa principal es de nivel 1, SP_1, SP_2 y SP_3 son de nivel 2, SP_4, SP_5 y SP_6 de nivel 3, y por último SP_7 de nivel 4.

Asimismo, la figura I.1.2 permite vislumbrar una relación entre subprogramas: $\mathrm{SP}_k \prec \mathrm{SP}_l$, si existen subprogramas $\mathrm{SP}_{k+1}, \mathrm{SP}_{k+2}, \ldots, \mathrm{SP}_{l-1}$, tales que SP_{k+1} es utilizado por SP_k, SP_{k+2} es utilizado por SP_{k+1} y así sucesivamente hasta SP_l es utilizado por SP_{l-1}. $\mathrm{SP}_i \succ \mathrm{SP}_j$, si $\mathrm{SP}_i \prec \mathrm{SP}_j$.

A partir de la relación definida entre subprogramas, la primera regla que se tiene es que no puede existir dos subprogramas SP_k y SP_l, tales que $\mathrm{SP}_k \prec \mathrm{SP}_l$ y $\mathrm{SP}_l \prec \mathrm{SP}_k$ al mismo tiempo; aun si el proceso de compilación culmina sin errores, con un programa ejecutable, al momento de ejecutar el programa se verán los errores.

También, otra regla que debe respetarse es que un subprograma no puede utilizarse a si mismo, excepto en alguna situación especial de recursividad que permite `Fortran 90`, que será abordado más adelante.

I.2. `Fortran`

`Fortran` (o más bien `FORTRAN` hasta principios de los años 90) es un lenguaje de programación desarrollado en los años 50 y activamente utilizado desde entonces. Acrónimo de "Formula Translation".

Fortran se utiliza principalmente en aplicaciones científicas y análisis numérico. Desde 1958 ha pasado por varias versiones, entre las que destacan `FORTRAN II`, `FORTRAN IV`, `FORTRAN 77`, `Fortran 90`, `Fortran 95` y `Fortran 2003`. Si bien el lenguaje era inicialmente un lenguaje imperativo, las últimas versiones incluyen elementos de la programación orientada a objetos.

El primer compilador de `FORTRAN` se desarrolló para una IBM 704 entre 1954 y 1957 por la empresa IBM, por un grupo liderado por John W. Backus. En la época se consideró imprescindible que los programas escritos en `FORTRAN` corrieran a velocidad comparable a la del lenguaje ensamblador; de otra forma, nadie lo tomaría en cuenta.

El lenguaje ha sido ampliamente adoptado por la comunidad científica para escribir aplicaciones con cómputos intensivos. La inclusión en el lenguaje de la aritmética de números complejos amplió la gama de aplicaciones para las cuales el lenguaje se adapta especialmente y muchas técnicas de compilación de lenguajes han sido creadas para mejorar la calidad del código generado por los compiladores de `Fortran`.

Capítulo II

Comenzando a programar en Fortran

Si bien, este texto está orientado a la versión 90 y posteriores de Fortran, se desarrollará algunos conceptos de las versiones anteriores, en particular FORTRAN 77, ya que muchos de los programas disponibles en la literatura y las bibliotecas de programas están escritos en esta versión.

II.1. Formatos de Escritura de un Programa en Fortran

Para desarrollar un programa en Fortran, el primer paso es obviamente, escribir el código fuente, utilizando un editor de texto disponible en la computadora y guardándolo en archivos texto de extensión .f, .f90, .f95, .for o de extensión que soporte el compilador y el sistema operativo de la computadora en la que se trabaja.

En computadoras que funcionan bajo una versión de LINUX, lo usual que se utiliza, como editor de texto, es **emacs**, debido a las facilidades que ofrece al editar un programa en Fortran. *emacs* puede ser llamado a partir de un *shell* o terminal, o bien utilizando el menú del escritorio de la computadora. Por ejemplo, a partir de la terminal, si se quiere editar el programa nombre.f90, se lo hace de la manera siguiente:

```
home]$ emacs nombre.f90
```

Obteniendo una ventana, mostrada en la figura II.1.1.

Se aconseja utilizar archivos diferentes para guardar código fuente de unidades programáticas diferentes: uno para el programa principal y un archivo para cada subprograma.

II.1.1. Caracteres permitidos en Fortran

El código fuente se escribe utilizando caracteres ASCI; es decir:

- **Caracteres alfanuméricos:** A-Z, a-z, 0-9, _,
- **Otros:**= + * / ( ) , "'.: ; ! &% <>$?
- **Espaciamiento:** ␣

Por defecto, Fortran no distinge mayúsculas y minúsculas, esto significa que el caracter a y el caracter A es lo mismo en Fortran.

Asimismo, los caracteres propios del español, como á, é, ñ y otros no pueden ser utilizados al escribir las instrucciones en Fortran; excepto en cadenas de caracteres, en las cuales hay diferencia entre minúscula y mayúscula, lo que será visto más adelante.

II.1.2. Formato Libre

Aplicable para las versiones de Fortran a partir de la 90, los archivos, por defecto, tienen la extensión .f90 o eventualmente .f95, bajo las siguientes reglas:

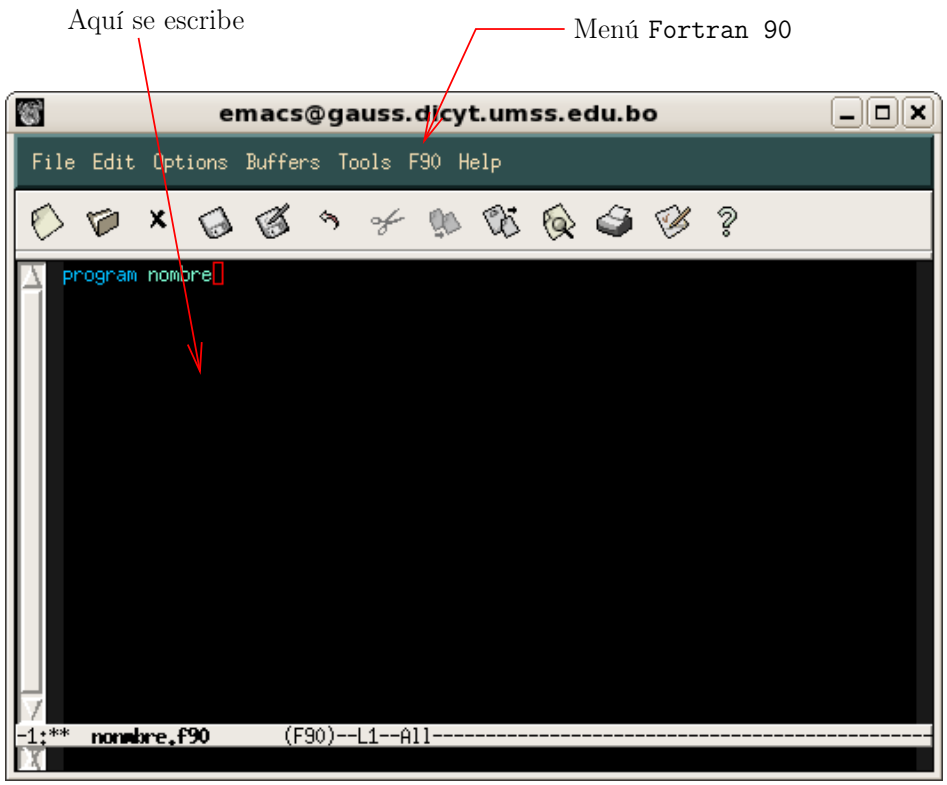


Figura II.1.1: Editor de texto *emacs*

1. Máximo 132 caracteres, incluídos espacios, por línea de texto.
2. El caracter `!` significa que el resto de la línea es comentario y no se toma en cuenta en el proceso de compilación.

```
<instrucción ! comentario de la instrucción>
```

3. El `;` separa dos instrucciones sobre la misma línea

```
<instrucción; <instruccción; <instruccción>
```

4. A falta de espacio en una linea, las instrucciones pueden ser continuadas en la siguiente línea, utilizando `&` al final de la línea que debe ser continuada y `&` al inicio de la siguiente. Esta acción puede reproducirse en cuantas líneas sea necesaria.

```
<instrucción> &
  & <continuación> &
  & <continuación>
<otra instrucción>
```

5. Los objetos del programa, como unidades prográmaticas, variables deben ser nombradas, utilizando una composición de 1 a 31 caracteres alfanuméricos, siendo el primer caracter una letra. Los nombres que se les dan a los objetos, deben ser diferentes a los nombres reservados a las instrucciones.

```
program Main ! program es una palabra reservada para identificar
             ! programas principales, Main es el nombre asignado
             ! a este programa principal.
```

6. Espacios pueden ser introducidos entre los elementos del lenguaje.

Cada unidad programática debe ser escrita bajo la siguiente estructura:

```
<Identificación unidad Programática >
  :
  : ! Instrucciones no ejecutables
  :
  :
  : ! Instrucciones ejecutables
  :
end <Unidad Programática>
```

donde:

- Instrucciones no ejecutables son declaraciones de los objetos que van a ser utilizados en el programa. Esta información es utilizada únicamente durante la compilación.
- Instrucciones ejecutables, como operaciones matemáticas, lectura y escritura de datos en la pantalla y archivos, llamadas a otros subprogramas, etc.
- Comentarios y líneas en blanco, no son tomadas en cuenta por el compilador.
- En un mismo archivo pueden escribirse los códigos de diferentes subprogramas.

II.1.3. Formato Fijo

Aplicable por defecto a las unidades programáticas escritas en archivos de extensión `.f` y `.for`. Formato desarrollado para los programas escritos para las versiones de `Fortran` anteriores a la `90`. Las reglas de este formato son básicamente las siguientes:

1. Máximo 72 caracteres, incluídos espacios, por línea
2. Líneas que comienzan con el caracter `C` es linea de comentario.

   ```
   C ----- Comentario de lo que se hará o se ha hecho.
   ```

3. Máximo una instrucción por línea, que comienza a partir del séptimo caracter; es decir los primeros seis caracteres de la línea son espacios.

   ```
   C2345678901
         <Instrucción>
   ```

4. Si una instrucción no puede ser completada en una línea, esta puede ser continuada en la siguiente línea, colocando un caracter no alfanumérico permitido en el sexto caracter, dejando en blanco los primeros cinco caracteres. Esto puede repetirse hasta un número máximo de líneas permitidas por el compilador a utilizarse.

   ```
   C2345678901
         <Instrucción>
        *<Continuación de la Instrucción>
   ```

5. Los primeros 5 caracteres de una línea de instrucción están destinados, en caso de necesidad a una etiqueta (*label* en inglés), que es un número positivo diferente a cero, con un máximo 5 dígitos.

   ```
   C2345678901
   12345 <Instrucción>
        *<Continuación de la Instrucción>
   ```

6. Los objetos del programa, como unidades prográmaticas, variables deben ser nombradas, utilizando una composición de 1 a 6 caracteres alfanuméricos, siendo el primer caracter una letra. Los nombres que se les dan a los objetos, deben ser diferentes a los nombres reservados a las instrucciones.

```
C234567
C ----- IDENTIFICACION DE UNA FUNCION
      REAL*8 FUNCTION FIBO
C --- REAL*8 y FUNCTION SON INSTRUCCIONES, FIBO ES EL NOMBRE DE UNA
C     FUNCION.
```

7. Espacios pueden ser introducidos entre los elementos del lenguaje.

El programa principal debe ser escrito bajo la siguiente estructura:

```
C234567
C ----- INSTRUCCIONES NO EJECUTABLES
      :
C ----- INSTRUCCIONES EJECUTABLES
      :
      END
```

y los subprogramas de nivel jerarquico inferior:

```
C234567
      <IDENTIFICACION DEL SUBPROGRAMA>
C ----- INSTRUCCIONES NO EJECUTABLES
      :
C ----- INSTRUCCIONES EJECUTABLES
      :
      END
```

donde:

- Instrucciones no ejecutables son declaraciones de los objetos que van a ser utilizados en el programa. Esta información es utilizada únicamente durante la compilación.
- Instrucciones ejecutables, como operaciones matemáticas, lectura y escritura de datos en la pantalla y archivos, llamadas a otros subprogramas, etc.
- Comentarios y líneas en blanco, no son tomadas en cuenta por el compilador.
- En un mismo archivo pueden escribirse los códigos de diferentes subprogramas.

La diferencia entre el programa principal y los subprogramas de nivel jerárquico inferior es el programa principal no requiere identificación, mientras que los subprogramas si lo requieren.

Para finalizar esta sección, existen otras reglas de formato que serán tratadas más adelante.

II.2. Compilando en `Fortran`

Una vez, que se ha escrito el código fuente de los programas y subprogramas, el siguiente paso es compilar para tener una versión ejecutable del o los programas ejecutables.

Tal como se explicó antes, un compilador es un programa que puede ser no comercial, en tal caso es gratuíto y está sujeto a ciertas condiciones de uso, o bien comercial y en tal caso tener un costo monetario. Las computadoras que funcionan bajo una distribución de LINUX, normalmente están provistos de un compilador GNU, que puede ser `g77` para `FORTRAN 77` o bien `gfortran` para `Fortran 90` y `95`, y `gcc` para `C`. Asimismo INTEL tiene a disposición de las versiones de uso no comercial de compiladores para `Fortran` y `C`: `ifort` e `icc` respectivamente. Para conocer las características y facilidades que ofrece cada compilador, se aconseja leer la documentación de éstos, que normalmente se accede a través de los utilitarios `man` o `info`.

```
home]$ man gfortran
```

En la figura II.2.2, se muestra la documentación accedida a través de `man`. También es posible acceder a otro tipo de documentación en formato `pdf` o en INTERNET. Para explicar el uso de un compilador para

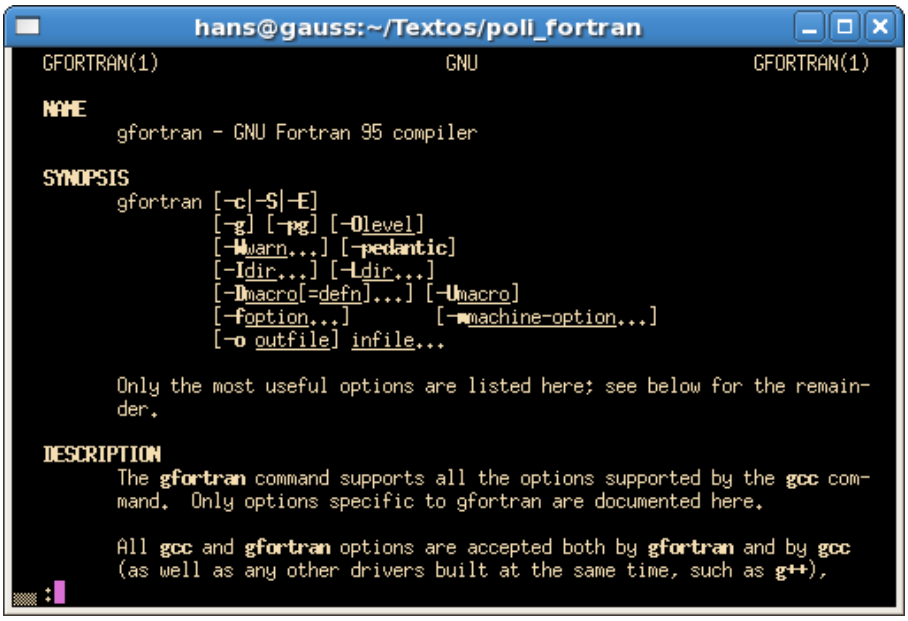


Figura II.2.2: Documentación de `gfortran` por medio de *man*.

`Fortran`, se asumirá las siguientes convenciones de uso:

- `fcomp` denotará el comando que llama a un compilador `Fortran`, como `gfortran` o `ifort`
- `arch1`, `arch2`, ... denotarán los archivos, que si contienen código fuente, tendrán la terminación `.f90`, `.f` u otra extensión de fortran; si contienen código objeto, tendrán la extensión `.o` y se trata de código almacenado en bibliotecas de programas, tendrán la extensión `.a`, en el caso de bibliotecas estáticas, `.so`, en el caso de bibliotecas dinámicas. Por consiguiente, para este primer abordaje, los archivos que serán manipulados serán aquéllos que contienen código fuente, código objeto, archivos de biblioteca, además de los archivos ejecutables.
- Lo que se encuentra dentro de corchetes, [ ], no es obligatorio y puede obviarse.

En consecuencia, el uso del compilador `Fortran` elegido tendrá la siguiente sintáxis.

```
home]$ fcomp [opciones] arch1 [arch2] [...] [archN] ...
```

donde opciones son ninguna o más opciones del compilador.

Puesto que, ya sabemos con que tipos de archivo trabajará el compilador `Fortran`, un paso inicial, antes de compilar, debiera ser crear condiciones de trabajo adecuadas. Por lo tanto, se sugiere crear un directorio de "trabajo", que para este texto lo llamamos `fortran`. Sobre este directorio de trabajo, luego se crea los subdirectorios: `lib`, `include`, `subrutinas`, `ejemplos`, `modulos`, `tmp` y los que sean necesarios.

Así el subdirectorio `subrutinas` tendrá los archivos del código fuente de los subprogramas que se vayan desarrollando, el subdirectorio `lib` los archivos de las bibliotecas de programas que se vayan creando, `ejemplos` el código de los subprogramas principales más trascendentes y que merecen ser guardados, `tmp` los archivos succeptibles de ser eliminados y la finalidad de los otros subdirectorios serán vistas, conforme se avance.

II.2.1. Creando Ejecutables

Para crear el código ejecutable de un programa se requiere:

1. De **manera obligatoria** el código, ya sea fuente, ya sea objeto, del subprograma principal, contenido en el `arch1`. No puede haber más de un subprograma principal.

2. En los casos que se necesite, el código, ya sea fuente, ya sea objeto, de los subprogramas de nivel jerárquico inferior del subprograma principal que no estén contenidos en archivos de biblioteca.

3. En los casos que se necesite, los archivos de bibliotecas que contienen código necesario, tanto para el subprograma principal, como para los subprogramas de nivel jerárquico inferior.

4. En los casos que se necesite, otros archivos necesarios.

5. No utilizar la opción de compilación `-c`, que será abordada cuando se quiera crear código objeto.

6. Los archivos necesarios deben ser accesibles por el compilador, esto significa que deben estar en el directorio donde se está compilando o en caso contrario el archivo debe incluir su ubicación; por ejemplo, el archivo de biblioteca `libX11.so` (en caso que sea necesario), será utilizado con su ubicación y nombre `/usr/lib/libX11.so`.

El nombre por defecto del archivo ejecutable es `a.out`, a menos que se utilice la opción de compilación `-o` seguida del nombre designado para el ejecutable, así por ejemplo:

```
fortran]$ fcomp [....] arch1 [....] -o nombre_ejecutable
```

dará el archivo ejecutable `nombre_ejecutable`.

Los compiladores `Fortran`, permiten acceder a los archivos de biblioteca de una manera más simple, evitando llamar al archivo de biblioteca a través de su ubicación y nombre completo, como se ha dado en la regla 6. Estas son las reglas para una utilización más simple de las bibliotecas de programa:

(a) Los archivos de biblioteca con subfijo **lib**, como `libnombre.so` (biblioteca compartida) o `libnombre.a` (biblioteca estática) pueden llamarse a través de la opción de compilación `-l`, de la manera siguiente: `-lnombre`. **Importante**: entre la opción `-l` y `nombre` no hay espacio.

(b) Cuando se utiliza la opción `-l`, el compilador busca los archivos en los directorios `/usr/local/lib` y `/usr/lib`. Si el archivo no se encuentra en estos directorios, la ubicación se la da a través del comando de compilación `-L` seguida de la ubicación y nombre del directorio `dir`. En el orden de instrucciones de compilación primera se utiliza `-L` y luego `-l`. Asi por ejemplo, en el caso que se esté en el directorio `fortran`:

```
fortran]$ fcomp [...] arch1 [...] -L dir -lnombre [...]
```

(c) Cuando para una biblioteca de programas, existe una versión compartida[1] (`.so`) y una versión estática (`.a`), el compilador utiliza, por defecto, la versión compartida. En el caso en que se desee la versión estática, se utiliza la opción de compilación `-static`:

```
ortran]$ fcomp [...] arch1 [...] -L dir -lnombre [...] -static [...]
```

Importante: Cuando se utiliza la opción `-static`, se aplica a todos los archivos de biblioteca que tienen ambas versiones.

[1] Cuando se utiliza una biblioteca compartida, el ejecutable utiliza durante la ejecución el código de la biblioteca; mientras que, en el caso de las bibliotecas estáticas, se incorpora código de las bibliotecas al ejecutable. Por consiguiente, si por algún motivo no se puede acceder al código de la biblioteca compartida, el ejecutable no funciona.

II.2.2. Creando código objeto y bibliotecas de programas

El código objeto se lo crea a partir de código fuente utilizando la opción de compilación `-c`. A diferencia de la creación de ejecutables, **no se requiere**:

- otros código objeto (`.o`),
- otros archivos de biblioteca de programas, las opciones de compilación `-L` y `-l` no se utilizan.

Solamente se requiere al menos un archivo de código fuente `.f90` o `.f`, eventualmente otro tipo de archivos, lo que será tratado más adelante.

El resultado por defecto, si no se utiliza la opción `-o`, serán los mismos archivos compilados, pero con extensión `.o`. Por ejemplo,

```
fortran]$ fcomp -c [...]arch1.f90 arch2.f ... archN.f90
```

dará como resultado los archivos:

```
fortran]$ ls
arch1.o  arch2.o .... archN.o
```

y eventualmente algunos archivos de extensión `.mod`.

Utilizando la opción `-o` seguida de `nombre.o` dará un solo archivo de código objeto, precisamente `nombre.o`.

```
fortran]$ fcomp -c [...]arch1.f90 arch2.f ... archN.f90 -o nombre.o
```

La utilización de la opción `-o` no es muy usada, la ventaja de su uso es que se optiene un solo archivo para todo el código objeto, la desventaja es que puede crear conflictos con otros archivos que contengan código objeto, o bien alguna biblioteca de programas, al momento de crear el código ejecutable.

En los casos en que subprogramas de orden jerárquico inferior que son o pueden ser utilizados por otros subprogramas principales, lo práctico es crear bibliotecas de programas, cuyo almacenaje es simple y seguro, el acceso a estas no presenta inconvenientes, basados los siguientes principios:

- Una biblioteca de programas contiene subprogramas agrupados en una área espécifica de aplicación; por ejemplo subprogramas relacionados a la solución de sistemas lineales, darán un biblioteca sobre solución de sistemas lineales.
- Para poder utilizar la opción de compilación `-l`, tendrán el prefijo `lib`, de esta manera el nombre sera `libnombre.a` o `libnombre.so` en caso que sea compartida.
- Las bibliotecas no contienen código de subprogramas principales.

Para crear una biblioteca estática, lo primero que hay que hacer es crear el código objeto de los subprogramas y luego utilizar el comando, en LINUX, `ar` con la opción `cr`. La sintaxis es la siguiente:

```
fortran]$ ar cr libnombre.a arch1.o arch2.o .... archN.o
```

Luego guardar el archivo `libnombre.a` en uno de los directorios `lib`. Mayores detalles sobre `ar`, referirse a su documentación.

La construcción y aplicación de bibliotecas compartidas no será abordada en este texto por el grado de complejidad que presentan.

Capítulo III

Instrucciones Básicas

Tal como se explicó en los dos capítulos precedentes, lo que va permitir que un programa sea ejecutado por la computadora, es la elaboración de la unidad programática principal, es decir el subprograma principal. De ahí, tenemos el primer grupo de instrucciones, que servirán para identificar y delimitar un subprograma principal.

Instrucción III.1 *El par de instrucciones* ***program*** *y* ***end program*** *se utilizan al inicio del subprograma principal y al final respectivamente, bajo la siguiente syntaxis*

```
program <nombre del programa>
  :
  : ! instrucciones no ejecutables
  :
  : ! instrucciones ejecutables
  :
end program <nombre del programa>
```

donde <nombre del programa> es una cadena de caracteres alfanuméricos, incluído _, de uno a 23 caracteres, sin espacios, que comienza con una letra.

III.1. Variables y Tipos de Dato

Ya sabemos, identificar y delimitar un subprograma principal, el siguiente paso es trabajar con una clase de objetos que manejará el programa, las variables. Una **variable** es un objeto que representa un dato de un **tipo de dato**, suceptible de modificarse y nombrado por una cadena de caracteres, incluído _, de uno a 23 caracteres, sin espacios, que comienza con una letra. Por ejemplo:

```
pi
dos_pi ! en lugar de 2pi
valor2
```

Importante, para evitarse problemas, una variable debe tener un nombre diferente a las instrucciones del lenguaje y obviamente a la de las otros objetos utilizados.

Los tipos de dato básicos en `Fortran` son los siguientes:

1. **character**: cadenas de uno o varios caracteres.
2. **integer**: números enteros, que pueden tomar todos los valores positivos o negativos entre límites que dependen de la computadora y el compilador.
3. **logical**: valores lógicos o booleanos, que toman solamente uno de los dos valores, `.false.` (falso) o `.true.` (verdadero).

4. **real**: números reales que pueden tomar valores positivos o negativos entre límites que dependen de la computadora y el compilador.

5. **complex**: números complejos compuestos de una parte real y una parte imaginaria, ambas partes de tipo `real`.

Ejemplos

```
'fortran'         ! character ! cadena de caracteres: fortran
'hola como estás' ! character ! cadena de caracteres: hola como estás
123               ! integer
-251              ! integer
.true.            ! logical
123.              ! real
1.23e06           ! real (significa 1.23 por 10^6)
-1.3e-07          ! real
(-1.,2.e-3)       ! complex
```

Los valores o datos, dados en el ejemplo, se llaman constantes.

Importante.- Una cadena de caracteres está delimitada por una comilla (') al principio y al final de la cadena. Si en la cadena de caracteres, se tiene una comilla (') como caracter, se delimita la cadena con doble comilla ("), por ejemplo:

```
''Maria's Pencil'' ! es la cadena de caracteres: Maria's Pencil
```

Los límites de representación de los tipos entero y real son los siguientes:

- Enteros:
$$-2{,}147{,}483{,}648 \leq i \leq 2{,}147{,}483{,}647$$

- Reales Simple Precisión:
$$1,2 \times 10^{-38} \leq |x| \leq 3,4 \times 10^{38}$$
7 cifras significativas.

- Reales Doble Precisión:
$$2,2 \times 10^{-308} \leq |x| \leq 1,8 \times 10^{308}$$
16 cifras significativas

III.1.1. Los tipos implícitos

Si en un código aparecen nombres que no han sido definidos por instrucciones de especificación, el tipo de la variable depende de su primera letra; así:

- `i`, `j`, `k`, `l`, `m`, `n` representan variables de tipo entero.
- Las otras letras variables de tipo real simple precision.

El carácter implícito de las variables de las variables puede ser modificado, a través de la instrucción **implicit** cuya sintaxis es la siguiente:

```
implicit <tipo> (<lista_1>,<lista_2>,...,<lista_k>)
```

donde lista_j es un caracter c_1 o un intervalo de caracteres c_1-c_2, instrucción que se coloca antes de las instrucciones declarativas de variables.

Ejemplo.- Se desea que todas las variables que comienzan con la letra l, las letras entre la u y w, así como la letra z, sean de tipo `logical`. La instrucción será

```
program prueba
  implicit logical (l,u-w,z)
  :
  : instrucciones no ejecutables
  :
  : instrucciones ejecutables
  :
end program prueba
```

La utilización de tipos implícitos puede dar lugar a errores no detectados por el compilador y que son dificiles de detectarlos. Por consiguiente, mejor es evitar, en lo posible, la regla de la definición implícita y definir explícitamente todas las variables a ser utilizadas, colocando la instrucción de especificación

```
implicit none
```

antes de especificar y declarar cada una de las variables del programa, con lo que cualquier variable que no haya sido declarada será detectada y dará error en la compilación del código fuente.

III.1.2. Declaración de Variables

Toda variable debe ser declarada de manera obligatoria, cuando no corresponda a un tipo implícito o bien cuando se haya recurrido a la instrucción `implicit none`.

Instrucción III.2 *La declaración de una variable o más variables del mismo tipo se hace siguiendo la instrucción de especificación, bajo la siguiente sintaxis*

```
<tipo> [,<atributo(s)>] [::] <variable(s)> [=<valor>]
```

Si en la instrucción de especificación aparece `<atributo(s)>`*, o* `=<valor>`*, el uso de* `::` *es obligatorio. Los atributos posibles son:*

```
parameter, save, intent, pointer, target, allocatable, dimension,
public, private, external, intrinsic, optional.
```

Ejemplos.- A continuación veremos algunas declaraciones de variables, utilizando la sintaxis de la instrucción de especificación dada más arriba.

```
real a
real :: b,c,d ! Especificaci'on de varias variables
integer :: a_int=51 ! Especificacion e inicializaci'on de la variable a_int
integer, parameter :: maxint=1000 ! Variable constante
integer, parameter :: imax=3*maxint
real, dimension(10) :: vector1, vector2
logical :: test=.false.
real :: ax, bx=1.0
complex :: z=(0.,1.)
```

Cuando se inicializa una variable con un valor dado, este puede ser modificado durante la ejecución del programa, excepto cuando el atributo de declaración es `parameter`; llamadas posteriores de la variable tendrá asignado no necesariamente el mismo valor de inicialización

III.1.3. Declaración de constantes

Cuando se quiere o requiere que una variable tome únicamente un valor durante la ejecución del programa y no pueda ser suceptible de cambio alguno, se utiliza el atributo `parameter` y la o las variables especificadas con el valor asignado. Por ejemplo

```
integer, parameter :: nmax=100, imax=3*nmax
```

Viendo el ejemplo, la segunda variable `imax` ha sido declarada en función de la primera variable `nmax`. Esto es posible, siempre y cuando las variables que definan una variable con atributo `parameter`, deben tener el atributo `parameter`, estar definidas y las operaciones, que intervienen en la expresión de definición, ser elementales, por ejemplo aritméticas.

La especificación de una variable constante es aconsejada si: la variable no cambia de valor durante la ejecución del programa, para hacer más lisible el código (por ejemplo `dos_pi` y sobre todo para poder modificar más facilmente el código fuente.

III.1.4. Declaración de cadenas de caracteres

Instrucción III.3 *La declaración de una variable de tipo* `character` *se la realiza siguiendo la siguiente sintaxis*

```
character[(len=<longitud>)][,<atributo(s)>][::]<variable(s)>[=<valor>]
```

donde <longitud> representa el número de caracteres de la cadena, incluyendo espacios.

Ejemplos

1. Si no se especifica la longitud, la cadena tiene un solo caracter.

   ```
   character inicial
   ```

2. Si no se conoce la longitud de la cadena de caracteres, se puede utilizar el comodín `*`, para que la longitud sea autoajustable.

   ```
   character(len=*), parameter::ciudad='Cochabamba', pueblo='Sacaba'
   ```

   Observamos que la cadena ciudad tiene 10 caracteres, mientras que la cadena pueblo tiene 6 caracteres.

III.1.5. Tipos de Reales

La mayoría de los compiladores `Fortran` trabajan con dos tipos de real: el primero, simple precisión y el segundo tipo, doble precisión. Existen algunos compiladores, como `ifort`, que trabajan con cuadruple precisión. .

Instrucción III.4 *La sintaxis para definir el tipo de* `real` *es la siguiente:*

```
real(kind=<np>)
```

donde <np> es un entero, que toma los siguientes valores:

- $np = 4$ *para simple precisión,*
- $np = 8$ *para doble precisión,*
- $np = 16$ *para cuadruple precisión.*

Si no se explicita la clase de `real`*, el defecto es simple precisión.*

De esta manera, las siguientes declaraciones son correctas:

```
real(kind=4)::x=1.e0
real(kind=8)::y=-2.d-5
real(kind=16)::qx=-3.q-100
```

Como puede observarse la letra `e` es el identificador de notación científica para simple precisión, `d` para doble precisión y `q` para cuadruple precisión.

Existe otra manera de identificar y establecer la clase de real de un número dado. La clase se la denota, agregando _ y el número de clase `np` detrás el valor del número, así por ejemplo:

```
1.23_4      ! significa que 1.23 es de simple precision
1.23e15_8   ! es el numero de doble precision 1.23d15
1.33e-100_16! es el numero de cuadruple precision 1.33q-100
```

Muchas veces, con la finalidad de facilitar la conversión de un programa en otra clase de real, es conveniente predefinir la clase `kind`, como por ejemplo:

```
integer, parameter::np=8
real(kind=np)::r=1.e-10_np
```

De esta manera, si se quiere cambiar la clase de real, solamente hay que cambiar el valor de la constante `np`.

III.1.6. Tipos de Complejo

Tal como se vió al inicio de esta sección III.1. `complex` es un tipo derivado del tipo `real`; consiguientemente hay `complex` simple precisión, `complex` doble precisión y algunos compiladores acceptan `complex` cuadruple precisión. Las declaraciones, utilizando `kind`, serán por lo tanto:

```
integer, parameter:: np=8
real(kind=4)::c=(1.e0,0.)
real(kind=8)::z=(-2.d-5,1.d0)
real(kind=16)::qz=(-3.q100,-1.q-10)
real(kind=np)::z_np=(1._np,-1._np)
```

Por lo tanto, las observaciones, hechas para `real`, son válidas para `complex`.

III.2. Operaciones elementales

Una de las diferencias radicales, entre el lenguaje de programación `Fortran` y el lenguaje matemático, está en el uso que se da al símbolo "=". Mientras que en la Matemática, su significado es la igualdad, en `Fortran` su significado es la asignación de valor. Esta diferencia de usos, muchas veces puede causar confusión, motivo por el cual en Matemática se utiliza el símbolo ":=" para la asignación de valores y lenguajes de programación como el PASCAL lo han adoptado; sin embargo, `Fortran` no lo ha hecho, por lo que hay que tener cuidado con el uso de eso símbolo elemental.

Instrucción III.5 *La **asignación** de valor a una variable, tiene la siguiente sintaxis*

```
<variable> = <expresión>
```

donde <expresión> es un conjunto de variables y operaciones válidas, cuyo resultado es un valor del mismo tipo de la <variable> o en la ocurrencia, el valor resultante es convertible al tipo de la <variable> por la computadora y el compilador.

Ejemplo

```
real(kind=8)::x=20._8; y
y=20.d0*y*y-40._8
```

Remarcamos que una variable definida sin valor inicial tiene un valor indeterminado al momento de iniciar la ejecución.

III.2.1. Operaciones aritméticas

`Fortran` permite las operaciones aritméticas sobre valores de los tipos `integer`,`real` y `complex`. Consiguientemente, se tiene: la adición (`+`), la sustracción (`-`), la multiplicación (`*`), la división (`/`), la potenciación (`**`) y el cambio de signo u opuesto (`-`). La **regla básica** es que dos operandos del mismo tipo y misma clase de tipo producen un resultado del mismo tipo y misma clase de tipo.

Operaciones aritméticas en el tipo `integer`

Puesto que el tipo `integer` maneja números enteros $n \in \mathbb{Z}$ dentro de un rango $-\texttt{NMAX} \leq n \leq \texttt{NMAX}$, denotamos por $\odot$ la operación aritmética del tipo `integer` que corresponde a la operación aritmética $\circ$ en $\mathbb{Z}$, para ilustrar cómo están definidas las operaciones aritméticas en el tipo `integer`.

$$n \odot m = \begin{cases} n \circ m & \text{si} \quad -\texttt{NMAX} \leq n \circ m \\ \textit{overflow} & \text{sino.} \end{cases} \tag{III.2.1}$$

donde *overflow* es un error que da la computadora, durante la ejecución del programa.

La **división** en el tipo `integer` es la división con resto; es decir

$$\texttt{m/n} = \texttt{l} \iff |\texttt{m}| = |\texttt{l}| \cdot |\texttt{n}| + r, \quad 0 \leq r < |\texttt{n}|\,. \tag{III.2.2}$$

Operaciones aritméticas en los tipos `real` y `complex`

De la misma manera que el tipo `integer`, los valores que manejan los tipos `real` y `complex` están dentro los límites de representabilidad, dados al inicio de la sección III.1. Denotando $\odot$ la operación punto flotante correspondiente a la operación $\circ$ y *rnd* el redondeo del número real o complejo al número del tipo `real` o `complex`, se tiene:

$$x \odot y = \begin{cases} rnd(x \circ y) & \text{si} \quad l_{inf} \leq |x \circ y| \leq l_sup \\ \textit{overflow} & \text{si} \quad |x \circ y| > l_{sup} \\ \textit{underflow} & \text{si} \quad 0 < |x \circ y| < l_{imf} \\ 0 & \text{si} \quad x \circ y = 0. \end{cases} \tag{III.2.3}$$

donde *overflow* da error y *underflow* es tratado por defecto como valor nulo en la mayoría de los compiladores, en caso contrario da error.

Potenciación

`Fortran` trata de diferentes formas la potenciación (`x**y!` x^y) dependiendo del tipo al que pertenece el exponente y. En el caso que el exponente sea entero, se tiene la siguiente regla de definición,

$$\texttt{x**n} = \begin{cases} \underbrace{\texttt{x}*\texttt{x}*\cdots*\texttt{x}}_{\texttt{n}} & \text{si} \quad \texttt{n} > 0 \\ \frac{1}{\texttt{x**(-n)}} & \text{si} \quad \texttt{n} < 0, \\ 1 & \text{si} \quad \texttt{n} = 0 \end{cases} \tag{III.2.4}$$

En el caso que la base x y el exponente y sean de tipo real, se tiene

$$\texttt{x}**\texttt{y} = \exp(y \log(x)), \tag{III.2.5}$$

lo que significa que cuando el exponente es de tipo `real` y la base es de tipo realreal, la base `x` necesariamente debe ser estrictamente positiva. En el caso `complex` es análogo a (III.2.5), con la diferencia que intervienen log y exp complejos.

Conversión de tipos

En caso de operaciones aritméticas entre objetos de tipo o clases de tipo diferentes, la computadora convierte los operandos a valores del mismo tipo y misma clase de tipo, siguiendo las siguientes reglas:

- Los enteros son convertidos en reales o complejos.
- Los reales son convertidos en complejos.
- Los reales o complejos son convertidos en la clase (`kind`) más alto
- En la asignación de valores (`=`), la parte derecha es evaluada en el tipo y clase que corresponde, luego es convertida al tipo y clase de la variable del lado izquierdo.

Así, por ejemplo:

```
integer::n,m
real :: a,b
real(kind=8)::x,y
complex :: c
complex(kind=8):: z
:
a=(x*(n**c))/z
n=a+z
```

Para asignar un valor a la variable `a`, siguiendo el orden establecido por los paréntesis, la computadora convierte `n` en tipo `complex`, evalua `(n**c)` en `complex`; luego convierte `x` en `complex`, evalua `x*(n**c)` en `complex`; después convierte `x*(n**c)` en `complex(kind=8)` y evalua `(x*(n**c))/z` en `complex(kind=8)`; por último, para asignar el valor a la variable `a`, convierte `(x*(n**c))/z` en `real`. Dejamos como ejercicio, el flujo de conversión para la expresión `n=a+z`.

Denotando por `n` una variable entera, `a` una variable `real`, `x` una variable `real(kind=8)`, `c` una variable `complex` y `z` una variable `complex(kind=8)`, tenemos en cuadro (III.1), las conversiones más importantes, dejamos como ejercicio, completar las otras conversiones.

Conversión	**Mecanismo de Conversión**
`x=n`	$x = n$
`x=a`	$x = a$
`n=x`	$n = \begin{cases} m & \text{si} \quad m \leq x < m+1 \text{ y } x \geq 0, \\ -m & \text{si} \quad m \leq -x < m+1 \text{ y } x < 0. \end{cases}$
`a=x`	$a = \text{round}(x)$
`a=c`	$a = \Re(z)$
`z=x`	$z = (x, 0)$

Cuadro III.1: Conversiones de tipo más significativas

Orden de operaciones

Si varias operaciones aritméticas están contenidas en una expresión, la computadora efectúa, en el orden:

1. las potencias,
2. las multiplicaciones y divisiones, yendo de izquierda a derecha,
3. los cambios de signo,
4. las adiciones y sustracciones, yendo de izquierda a derecha.

En caso de duda, utilizar **paréntesis**

III.2.2. Operaciones de comparación

Instrucción III.6 *La instrucción u operación de comparación, tiene la siguiente sintaxis:*

```
<expresión 1> <operador de comparación> <expresión 2>
```

Formato f90	Formato f77	
==	.eq.	es igual a
/=	.ne.	no es igual a
>	.gt.	es estrictamente mayor a
>=	.ge.	es mayor o igual a
<	.gt.	es estrictamente menor a
<=	.ge.	es menor o igual a

Cuadro III.2: Operadores de Comparación

donde ¡expresión 1¿y ¡expresión 2¿son expresiones de tipo numérico (`integer`, `real`, `complex`) y tienen como resultado un valor de tipo lógico (`.true.`, o `.false.`). El listado de los operadores de comparación se encuentra en el cuadro III.2.

Para expresiones de tipo `complex` como operandos, solamente son válidos los operadores de comparación `==` y `/=`.

Las operaciones de comparación se evaluan después de evaluar las expresiones aritméticas o numéricas.

Ejemplo.- El código siguiente ilustra la utilización de un operador de comparación.

```
logical :: l
real :: a,b,c
integer :: n,m
:
l=n**m<a+b**c
```

Las instrucciones listadas, tienen como flujo de conversión y orden de operaciones: la computdora evalua la expresión `n**m`, la convierte en tipo `real`, evalua la expresión `a*b**c` que es de tipo `real`, luego aplica el operador de comparación <, cuyo resultado es de tipo `logical`

III.2.3. Operaciones lógicas

Instrucción III.7 *Una instrucción u operación lógica tiene la siguiente sintáxis:*

`<expresión 1> <operador de comparación> <expresión 2>`

donde <expresión 1¿y <expresión 2> son de tipo `logical` y el resultado tiene como valor, un valor de tipo `logical`. El detalle de los operadores lógicos, que funciónan como operadores lógicos matemáticos, está en la tabla III.3

Operador	**Significado**
.not.	No (operador unario, un solo operando)
.and.	y
.or.	o
.eqv.	equivalente
.neqv.	no equivalente

Cuadro III.3: Operadores lógicos

Las operaciones lógicas se evaluan después de las operaciones de comparación, de izquierda a derecha. En caso de duda utilizar paréntesis, lo que es aconsejable.

Ejemplo.- La expresión listada es un silogismo, dejamos como ejercicio su verificación.

```
logical:: p1,p2,q
q=(.not.(p1.or.p2).eqv.((.not.p).and.(.not.q))
```

III.2.4. Funciones intrínsicas

`Fortran`, como lenguaje de programación tiene disponibles un conjunto de funciones predefinidas, independientemente del compilador que se utilice. Estas funciones se las conoce como funciones intrínsicas y el listado completo de estas funciones, al igual que sus características se encuentran en la documentación de los compiladores y otras fuentes de información, que pueden ser accedidas a través de Internet.

Sin embargo, por el uso frecuente que se da a cierto número de funciones, conocidas como elementales y para facilitar su utilización, daremos a conocer algunas de ellas.

Instrucción III.8 *La sintaxis de una función intrínsica monoargumental es*

```
<función>(<argumento>)
```

donde <función> es el nombre de la función intrínsica, <argumento> es una expresión de un determinado tipo y clase de tipo. El resultado de la función usualmente corresponde al tipo y clase de tipo de la expresión .

En el cuadro III.4, tenemos el listado de las funciones intrínsicas más utilizadas.

Función	Argumento	Resultado	Descripción
`abs`	`real, complex, integer`	`real, integer`	$\|x\|$, $\|z\|$, $\|n\|$
`sqrt`	`real, complex`	`real, complex`	$(\sqrt{x},\ x \geq 0)$, $(\sqrt{z},\ z \in \mathbb{C})$
`int`	`real`	`integer`	Parte entera de una real x.
`fraccion`	`real`	`real`	Parte fraccional de un real x
`real`	`complex`	`real`	$\Re z,\ z \in \mathbb{C}$
`aimag`	`complex`	`real`	$\Im z,\ z \in \mathbb{C}$
`conjg`	`complex`	`complex`	$\bar{z},\ z \in \mathbb{C}$
`cos`	`real complex`	`real complex`	$(\cos x,\ x \in \mathbb{R})$, $(\cos z,\ z \in \mathbb{C})$
`sin`	`real complex`	`real complex`	$(\sin x,\ x \in \mathbb{R})$, $(\sin z,\ z \in \mathbb{C})$
`tan`	`real complex`	`real complex`	$(\tan x,\ x \in \mathbb{R})$, $(\tan z,\ z \in \mathbb{C})$
`acos`	`real complex`	`real complex`	$(\arccos x,\ x \in \mathbb{R})$, $(\arccos z,\ z \in \mathbb{C})$
`asin`	`real complex`	`real complex`	$(\arcsin x,\ x \in \mathbb{R})$, $(\arcsin z,\ z \in \mathbb{C})$
`atan`	`real complex`	`real complex`	$(\arctan x,\ x \in \mathbb{R})$, $(\arctan z,\ z \in \mathbb{C})$
`exp`	`real complex`	`real complex`	$(\exp x,\ x \in \mathbb{R})$, $(\exp z,\ z \in \mathbb{C})$
`log`	`real complex`	`real complex`	$(\log x,\ x > 0)$, $(\log z,\ z \in \mathbb{C},\ z \neq 0)$
`log10`	`real`	`real`	$(\log_{10} x,\ x > 0)$

Cuadro III.4: Algunas Funciones Intrínsicas

Ejemplo A continuación, código fuente ilustrando la utilización de funciones intrínsicas:

```
x=cos(y)
z2=log(sqrt(z)+(1._8,-1._8)
n=int(x**2-y)
```

Las funciones intrínsicas no siempre son monoargumentales, pueden no tener argumento o al contrario tener varios argumentos, de las cuales se destacan dos funciones biargumentales. La instrucción

```
z=(x,y)
```

para asignar un valor a la variable `z` de tipo `complex`, a partir de las variables `x` e `y` de tipo `real` no son válidas; excepto en el caso que `x` e `y` hayan sido definidas como variables constantes, utilizando el atributo `parameter`. La instrucción de asignación de valores es `cmplx`, cuya sintaxis es

```
cmplx(<expresión 1>,<expresión 2>)
```

donde <expresión 1> y <expresión 2> son de tipo `real`. Ilustramos su utilización,

```
z=cmplx(x,x**2-y**2)
```

La otra función biargumental es la función `mod`, cuya sintaxis es

```
mod(<expresión 1>,<expresión 2>)
```

donde <expresión 1> y <expresión 2> son de tipo `integer` o `real`, cuyo resultado es un valor de tipo `integer` o `real` que corresponde al resto de una división con resto. Esta función se aplica sobre todo a expresiones de tipo `integer` y en menor medida a expresiones de tipo `real`. Se tiene la identidad

$$n = n/m + \texttt{mod}(n, m), \tag{III.2.6}$$

y como ejemplo las siguientes instrucciones

```
k=mod(n,2)
mod(n.2)==0.or.mod(n,2)==1
```

III.2.5. Operaciones con caracteres

Comenzamos con la instrucción de asignación, recordando la instrucción III.5, para el tipo `character`, se tiene

```
<variable>=<expresión>
```

donde <variable> es de tipo `character[(len=n)]` y <expresión> es de tipo `character` de longitud `len=m`, con n y m enteros positivos. La asignación se la realiza de la siguiente manera:

- Si $n \leq m$, se asigna a la <variable> los n primeros caracteres, incluídos espacios, de la <expresión>, de izquierda a derecha, elimando o troncando los $m - n$ caracteres restantes de la expresión
  Ejemplo.- Las instrucciones siguientes:

  ```
  character(len=5)::ciudad
  :
  ciudad='Cochabamba'
  ```

  darán como resultado `ciudad='Cocha'`.

- $n > m$, se asigna a la <variable> de izquierda a derecha la cadena de caracteres de la <expresión>, completando los últimos $n - m$ caracteres de la derecha con espacios.
  Ejemplo.- Las instrucciones siguientes:

  ```
  character(len=8)::pueblo
  :
  pueblo='Sacaba'
  ```

  darán como resultado `pueblo='Sacaba␣␣'`.

Operadores binarios

Para las cadenas de caracteres existen los siguienes operadores binarios:

- **Concatenación.-** Dado por el operador `//`. Por ejemplo, la expresión

  ```
  'Cochabamba'//' es una linda ciudad.'
  ```

  da la cadena de caracteres `'Cochabamba es una linda ciudad.'`.

- **Comparación** Dados por los operadores `==` y `/=`, cuyo uso es similar al caso de las expresiones numéricas.

Partes de cadenas de caracteres

Instrucción III.9 *Si <expresión> representa una cadena de caracteres* `'c1 ... ck ... cl, ... cn'` *de n caracteres, con* $1 \leq k \leq l \leq n$*, las instrucciones siguientes dan:*

```
<expresión>(k:l) ! 'c_k...c_l'
<expresión>(:l)  ! 'c_1...c_k...c_l'
<expresión>(k:)  ! 'c_k...c_l...c_n'
```

Así, por ejemplo

```
character(len=10)::ciudad='Cochabamba'
character(len=5):: city,town
town='Cochabamba' (6:) ! town='bamba'
city=ciudad (:,5)      ! city='Cocha'
```

Otras operaciones sobre cadenas de caracteres

Aparte de los operadores binarios de concatenación y comparación, y asignación de partes de cadenas de caracteres, `Fortran` tiene a disposición algunas instrucciones o funciónes útiles para expresiones de tipo `character` cuya sintaxis es la siguiente:

```
<instrucción>(<expresión>)
```

donde <expresión> es una expresión cuyo valor es de tipo `character` e <instrucción> corresponde a una de las instrucciones de la tabla III.5

Instrucción	Resultado	Descripción
`len`	`integer`	Da la longitud de la cadena de caracteres.
`trim`	`character`	Suprime los espacios del final de la cadena.
`adjustl`	`character`	Si hay espacios al inicio de la cadena, los suprime desplazando el resto de la cadena a la izquierda

Cuadro III.5: Instrucciones sobre cadenas de caracteres

Además la instrucción `index`, cuya sintaxis es

```
index(<expresión 1>,<expresión 2>)
```

donde <expresión 1> y <expresión 2> son cadenas de caracteres, proporciona como resultado un valor de tipo `integer`, que indica la primera posición de la <expresión 2> en la <expresión 1>, en el caso en que la <expresión 2> sea un pedazo de la <expresión 1>; sino el valor será `0`. Por ejemplo

```
index('Cochabamba','ba') ! dara 6
```

III.3. Instrucciones básicas de lectura y escritura de datos

Por lo avanzado hasta aquí, en principio, ya tenemos casi todos los elementos para elaborar un programa, en todo caso uno que contenga únicamente un subprograma principal. Sabemos declarar variables de los tipos provistos por `Fortran`, escribir instrucciones de ejecución donde intervienen operaciones y funciones intrínsicas, introducir datos, por medio de la inicialización de variables. Pero todavía no sabemos cómo presentar los resultados del programa e introducir datos de otra manera que no sea en el código del programa.

La forma más sencilla de presentar datos es por medio de la pantalla, mediante texto y la forma de introducir datos, aparte del programa mismo, es por medio del teclado, mediante texto. Existen otros medios como archivos de texto u de otro tipo, que serán abordados en el siguiente capítulo.

III.3.1. Escritura sobre la Pantalla

Un programa escrito en `Fortran`, para escribir sobre la pantalla de la computadora, utiliza dos instrucciónes que son equivalentes: `print` y `write`, cuyas sintaxis son las siguientes:

```
print*,<expresión 1>,...,<expresión n>
write(*,*)<expresión 1>,...,<expresión n>
write(6,*)<expresión 1>,...,<expresión n>
```

donde `print` se escribe, seguido de `*` y una coma, luego el listado de expresiones a escribir, cada una separada por coma. `write` se escribe seguido de `(*,*)` o `(6,*)`, luego el listado de expresiones a escribir, cada una separada por coma. Cada expresión debe ser de un tipo: `integer`, `real`, `complex`,`character` o `logical`.

Ejemplo.- A manera de ilustración, el siguiente programa, presenta sus resultados, mostrados en la Figura III.3.1

```
program prueba
  character(len=10)::ciudad='Cochabamba'
  real(kind=8)::x=1.d0, y=-1.d0
  complex (kind=4)::z=(-.5,2.)
  logical::tautologia=.true., falacia=.false.
  print*,'x=',x,'y=',y,'z=',z
  write(*,*)'tautologia siempre es',tautologia,' falacia siempre es',falacia
  write(6,*)'El nombre de la ciudad es ',ciudad
end program prueba
```

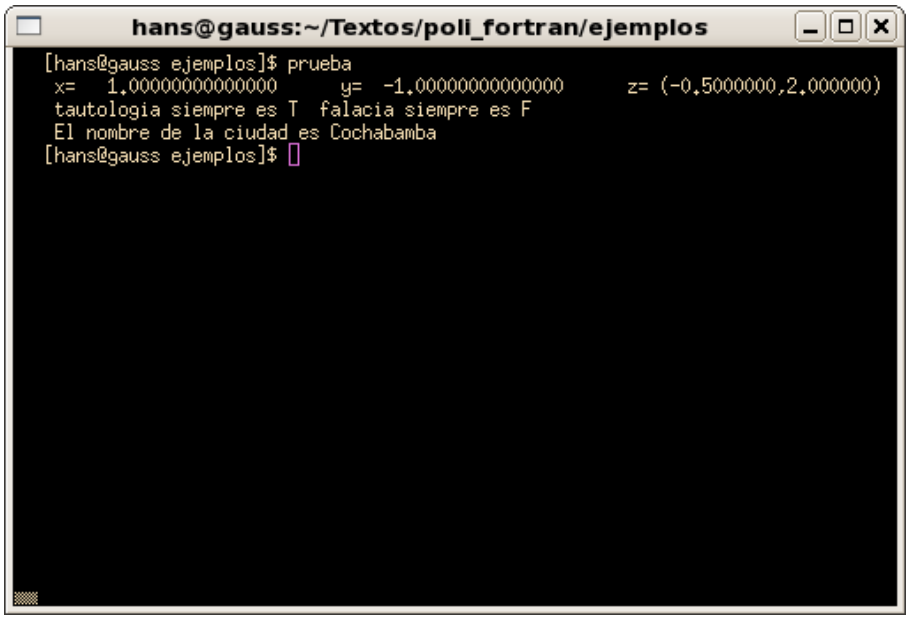


Figura III.3.1: Escritura de Resultados en la Pantalla

El caracter `*` que sigue a la instrucción `print` o el segundo caracter `*` que sigue a la instrucción `write`, hace que la computadora formatee automáticamente los datos, las datos numéricos son presentados con todos sus decimales. En el capítulo siguiente, veremos como uno puede formatear la presentación de los datos.

III.3.2. Lectura de Datos utilizando el Teclado

Un programa escrito en `Fortran`, para introducir datos, desde el teclado de la computadora, utiliza la instruccción `read`, cuyas sintaxis son las siguientes:

```
read*,<variable 1>,...,<variable n>
read(*,*)<variable 1>,...,<variable n>
read(5,*)<variable 1>,...,<variable n>
```

donde `read` se escribe, seguido de `*` y una coma o bien seguido de `(*,*)` o `(5,*)`, luego el listado de variables, separadas por comas, en las que se va asignar los datos introducidos desde el teclado. Necesariamente el listado debe contener exclusivamente variables, ningún otro tipo de expresión. Cada variable debe ser de un tipo: `integer`, `real`, `complex`,`character` o `logical`.

Los datos se introducen separados por espacios o comas. Para las cadenas de caracteres escribirlas delimitadas por comillas o dobles comillas. De esta manera el código fuente del siguiente programa

```
program prueba
  character(25)::ciudad
  logical::proposicion
  real(kind=8)::x,y
  complex (kind=4)::z
  print*,'Introduzca el nombre de la ciudad'
  read*,ciudad
  write(*,*)'Usted ha escrito: ',ciudad
  print*,'Reintroduzca nuevamente el nombre de la ciudad delimitada'
  read(*,*)ciudad
  write(6,*)'Usted ha escrito: ',ciudad
  write(*,*)'Introduzca los otros datos'
  read(5,*)x,y,z,proposicion
  print*,'Usted ha introducido',x,y,z,proposicion
end program prueba
```

producirá la siguente secuencia en la pantalla:

```
 [hans@gauss ejemplos]$ prueba
 Introduzca el nombre de la ciudad
```

```
 [hans@gauss ejemplos]$ prueba
 Introduzca el nombre de la ciudad
Santa Cruz de la Sierra
 Usted ha escrito: Santa
 Reintroduzca nuevamente el nombre de la ciudad delimitada
```

```
 [hans@gauss ejemplos]$ prueba
 Introduzca el nombre de la ciudad
Santa Cruz de la Sierra
 Usted ha escrito: Santa
 Reintroduzca nuevamente el nombre de la ciudad delimitada
'Santa Cruz de la Sierra'
 Usted ha escrito: Santa Cruz de la Sierra
 Introduzca los otros datos
```

```
 [hans@gauss ejemplos]$ prueba
 Introduzca el nombre de la ciudad
Santa Cruz de la Sierra
 Usted ha escrito: Santa
 Reintroduzca nuevamente el nombre de la ciudad delimitada
'Santa Cruz de la Sierra'
 Usted ha escrito: Santa Cruz de la Sierra
 Introduzca los otros datos
1.,3.d0 (3.,3.e-4) .true.
 Usted ha introducido   1.00000000000000        3.00000000000000
 (3.000000,3.0000001E-04) T
[hans@gauss ejemplos]$
```

III.4. Tipos derivados

Hasta la versión 77 de `Fortran`, solamente se podía trabajar con expresiones de tipos intrínsicos (`integer`, `real`, `logical` y `character`); pero a partir de la versión 90 es posible trabajar con expresiones de tipos derivados de datos, creados por el programador, a partir de los tipos intrínsicos. En esta sección se abordará la creación de nuevos tipos de datos, así como la manipulación básica de las expresiones resultantes.

Comencemos con la creación de un tipo derivado, ésta se la realiza siguiendo el siguiente grupo de instrucciones:

Instrucción III.10 *La sintaxis para declarar un nuevo grupo es:*

```
type <Nombre Nuevo tipo}
   <Instrucción declaración variable(s)>
   <Instrucción declaración variable(s)>
    :
   <Instrucción declaración variable(s)>
end type
```

donde <Instrucción declaración variable(s)> corresponde a variables de tipo ya especificadas (intrínsicas o de tipos derivados definidos antes).

Ilustramos la declaración de un nuevo tipo, a través del siguiente ejemplo:

```
type atomo
   character(len=2)::simbolo
   integer::Z ! n'umero at'omico
   real::A ! peso at'omico
end type
```

de esta manera, todo dato del tipo `atomo` estará identificado por tres variables componentes: símbolo (tipo `character`), su número atómico (`integer`) y su peso atómico (tipo `real`).

Ahora veamos, cómo se declara una variable de un tipo derivado y cómo se le asigna valores.

Instrucción III.11 *La sintaxis para declarar una o más variables de un tipo derivado es la siguiente:*

```
type(<nombre tipo>)[,<atributos>]::<variable(s)>[=<valor>]
```

separando las variables con coma (,), en el caso que haya más de una variable.
La asignación de valores se la realiza de manera global o por componentes, siguiendo la sintaxis siguiente:

```
<variable>=(<valor 1>,...,<valor n>) ! asignación globale
<variable>%<variable k>=<valor k> ! asignación por componente
```

donde <valor 1>,...,<valor n> son los valores de las <variable 1>,...<variable n> variables componentes del tipo derivado en el orden establecido, al momento de declarar el tipo derivado.

Siguiendo con el ejemplo del tipo `atomo`, las siguientes líneas de código muestran la declaración de variables y las diferentes formas de asignar valores.

```
program demo
  type atomo
     character(len=2)::simbolo
     integer::Z ! n'umero at'omico
     real::A ! peso at'omico
  end type
  type(atomo)::helio,nitrogeno,oxigeno,litio
  type(atomo),parameter::hidrogeno=atomo('H',1,1.)
  nitrogeno=atomo('N',7,14.)
  helio%simbolo='He'
  helio%Z=2
  helio%A=4.
  print*,'Introduzca globalmente los datos del Oxigeno'
  read*,oxigeno
  print*,'Usted ha introducido'
  print*,'Oxigeno =',oxigeno
  print*,'Introduzca por componentes los datos del litio'
  read*,litio%simbolo
  print*,'Usted ha introducido'
  print*,"S'imbolo del Litio ",litio%simbolo
end program demo
```

cuya ejecución produce en la pantalla

```
[hans@gauss ejemplos]$ demo
 Introduzca globalmente los datos del Oxigeno
0 8 16.
 Usted ha introducido
 Oxigeno =0            8   16.00000
 Introduzca por componentes los datos del litio
Li
 Usted ha introducido
 S'imbolo del Litio Li
[hans@gauss ejemplos]$ demo
 Introduzca globalmente los datos del Oxigeno
'O',8,16.
 Usted ha introducido
 Oxigeno =O            8   16.00000
 Introduzca por componentes los datos del litio
Li
 Usted ha introducido
 S'imbolo del Litio Li
[hans@gauss ejemplos]$
```

La declaración global de una variable de tipo derivado puede ser bastante complicada, dependiendo la complejidad del tipo, por lo que usualmente se utiliza la declaración por componentes.

Más adelante, en capítulos posteriores, se verá cómo se puede crear operaciones, conversiones a otros tipos.

III.5. Ejercicios

1. Completar el código de los ejemplos de este capítulo, de manera que se pueda comprobar lo afirmado por el texto.

2. Escribir un programa que determine las dos raices de la ecuación $x^2 + 2ax + b = 0$, introduciendo por el teclado los valores de a y b.

3. Crear un tipo `alumno` y asignar valores de manera global y por componentes.

Capítulo IV

Lectura y Escritura de Datos

Uno de los aspectos más importantes es la introducción de datos y la presentación de datos. Tal como se explicó en el capítulo precedente, la introducción de datos, se puede hacer inicializando variables con datos, o se puede hacer a través de procesos de lectura de datos. La presentación de datos se la hace únicamente por medio de procesos de escritura.

En este capítulo se abordará exclusivamente las diferentes varientes que ofrece `Fortran` para la lectura y escritura de datos. Comencemos, por lo tanto, a describir los diferentes formatos de representación de datos, tanto para la escritura, como la lectura.

IV.1. Formatos de representación de datos

Al hablar de formato de datos, se debe comenzar diciendo que existen datos formateados y datos no formateados. Cuando uno se refiere a datos formateados, está tratando con datos representados por caractéres de texto; es decir, cadenas de caracteres y una persona con formación suficientes es capaz de interpretarlos. En cambio, para los datos no formateados, la representación de éstos se la hace a través de código binario o código máquina, representación que es dificilmente interpretable por un ser humano.

Los datos formateados, por razones obvias, deben ser almacenados en archivos, a los cuales la computadora accede para leerlos o bien para escribir datos. Estos procesos de lectura y escritura de datos no formateados serán abordados en la siguiente sección; en consecuencia, en esta sección, se tratará únicamente los datos formateados.

IV.1.1. Formato automático

En la sección III.3 del capítulo precedente, se dieron los elementos primarios para la lectura y la escritura de datos, utilizando las instrucciones `read`, `print` y `write`. . La sintáxis establecida para estas instrucciones, en sus diferentes variantes, era:

`print` * `,` < expresión 1 > [, < expresión 2 >, . . . < expresión n >]
`read` * `,` < variable 1 > [, < variable 2 >, . . . < variable n >]
`write`(*, *) < expresión 1 > [, < expresión 2 >, . . . < expresión n >]
`read`(*, *) < variable 1 > [, < variable 2 >, . . . < variable n >
`write`(6, *) < expresión 1 > [, < expresión 2 >, . . . < expresión n >]
`read`(5, *) < variable 1 > [, < variable 2 >, . . . < variable n >

El asterisco * resaltado indica que el formato es automático y ésta establecido por las características propias del `Fortran`. El programador o el usuario solo debe saber que al instroducir los datos, ya sea a través del teclado o un archivo de texto, los datos requeridos por la instrucción `read` deben estar separados por espacios o comas, los datos de tipo `character` delimitados por comillas o dobles comillas.

Para la escritura de datos, la computadora proporcionará los datos de tipo númerico de manera completa, separados por espacios; los datos de tipo `logical` utilizando los caracteres `T` y `F`, representaciones de los

valores `.true.` y `.false.` respectivamente, separados por espacios y las cadenas de caracteres serán escritas de acuerdo a la lóngitud de la expresión de tipo `character` sin tipos de separación.

Así por ejemplo, el siguiente código:

```
program formato_automatico
  integer::n=103,m=-23
  real::a=1./3.
  real(kind=8)::x=1.d0/3.d0
  complex::z=(1.,-1.)
  logical:: p1=.true.,p2=.false.
  character(len=8)::pueblo='Sacaba'
  character ::no='N',si='S'
  write(*,*)'Escritura de todos los valores'
  print*,n,m,a,x,z,p1,p2,pueblo,no,si
  write(*,*)"Variables de tipo num'erico"
  write(*,*) n,m,a,x,z
  write(*,*)"Variables de tipo l'ogico"
  write(*,*) p1,p2
  write(*,*) "Variables de tipo character"
  write(*,*) no,si,pueblo
end program formato_automatico
```

producirá en la pantalla la siguiente escritura:

```
[hans@gauss ejemplos]$ formato_automatico
 Escritura de todos los valores
         103         -23  0.3333333      0.333333333333333
 (1.000000,-1.000000) T F Sacaba  NS
 Variables de tipo num'erico
         103         -23  0.3333333      0.333333333333333
 (1.000000,-1.000000)
 Variables de tipo l'ogico
 T F
 Variables de tipo character
 NSSacaba
[hans@gauss ejemplos]$
```

Para otros ejemplos, referirse a la sección III.3

IV.1.2. Formatos preestablecidos

Si bien, una de las ventajas de utilizar el formato automático, tanto en la lectura, como en la escritura de datos, es que el programador y el usuario del programa solo deben preocuparse de escribir bien las instrucciones del programa e introducir correctamente los datos, separados por comas o espacios. Sin embargo, esta aparente facilidad, tiene algunas consecuencias de orden práctico; como por ejemplo:

- Dificultad para confeccionar cuadros o tablas de resultados presentables.
- Dificultad para reutilizar datos obtenidos, como datos de entrada.
- Confusiones con cadenas de caracteres
- Otras que uno puede encontrar.

Especificadores de Campo y Formato

Antes de dar formato a los datos que serán leídos o escritos por la computadora, se describirá los especificadores de campo y formato más utilizados, dejando el resto para su búsqueda en la documentación disponible.

La unidad de registro, tanto de escritura, como de lectura es la **línea de texto**, sea en la pantalla o en los archivos de tipo texto. Una línea de texto, a su vez, está subvidida en celdas y cada celda está destinada a albergar un caracter de texto. La ubicación de una celda en una línea define una columna, si es que se ve la pantalla o el archivo como una página de texto.

Varias celdas contiguas definen un campo de escritura o lectura y su ubicación estará dada por los campos que le preceden o por un especificador de formato. El primer campo comienza en la primera columna, a menos que se utilize un especificador de formato para cambiar su ubicación.

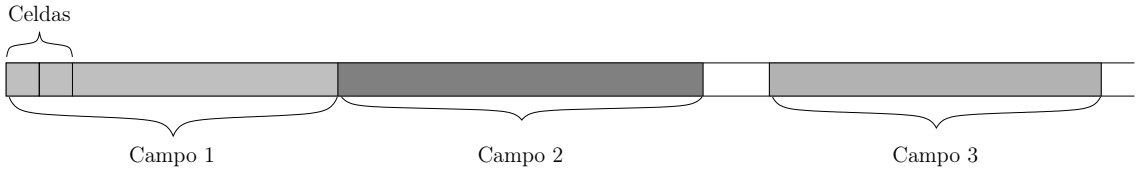


Figura IV.1.1: Campos de Escritura y Lectura en una Línea

El especificador para datos de tipo `integer` es `I`***n***, donde la letra `I` identifica el tipo `integer` y ***n*** es el número de caracteres de la cadena, que será utilizada para representar un valor de tipo `integer` en el campo asignado. El valor será representado en notación usual decimal, siguiendo las siguientes reglas:

1. Si el valor es negativo, el signo `-` irá a la izquierda del primer dígito sin espacio de separación.
2. En caso que la representación utilize varios dígitos, estos ván juntos, sin espacios de separación y el primer dígito a la izquierda es diferente de `0`
3. El dígito correspondiente a las unidades está ubicado en el extremo derecho de la cadena de caracteres utilizada.
4. Los caracteres a la izquierda del primer dígito o eventualmente el signo `-` son espacios.

De acuerdo a lo que establece el especificador `I`***n***, habría que preguntarse, qué sucede si el valor entero requiere más caracteres que los proporcionados por el especificador. En el caso de la escritura, la computadora escribirá `*****` en el campo asignado y en el caso de la lectura, la computadora leerá los dígitos que se encuentran en el campo, ignorando o asignando a otro campo, los que se encuentran fuera.

El especificador de formato punto fijo, para valores de tipo `real`, es `F`***n***`.`***d***, donde ***n*** es la longitud del campo, ***d*** es el número de dígitos asignados a la parte fraccionaria del valor `real`. El valor será representado en notación decimal siguiendo las siguientes reglas:

1. La representación del valor tiene justificación derecha.
2. La parte fraccionaria del valor `real` ocupan los últimos ***d*** caractéres del campo y no hay espacios, solamente dígitos (0-9), el caracter anterior a la parte fraccionaria corresponde al caracter punto "`.`"
3. Los dígitos de la parte entera del valor van a la izquierda del punto, sin espacios de separación. En caso que la parte entera sea diferente de 0, el primer dígito es diferente de 0; en el caso que la parte entera sea 0, el primer dígito es `0` o ningún caracter, dependiendo del espacio disponible del campo o del compilador.
4. En caso que el número sea negativo, el caracter `-` va a la izquierda de la parte entera o el punto (si la parte entera es 0), sin espacios
5. Los caracteres a la izquierda del primer dígito, eventualmente el punto o el `-` son espacios.

Ejemplos Las representaciones de 1/30 1/3, 10/3 y −10/3, utilizando el especificador `F10.5` serán respectivamente:

```
!234567890
    .03333  ! 1/30 en F10.5
    .33333  ! 1/3 en F10.5
   3.33333  ! 10/3 en F10.5
  -3.33333  ! -10/3
```

En caso que el número de caracteres asignados al campo `F`*`n`*`.`*`d`* sea insuficiente, para representar un valor `real`, el computador escribirá caracteres `*` en el campo y leerá únicamente lo que se encuentra en el campo.

El especificador de formato punto flotante, para valores de tipo `real` (simple precisión), es `E`*`n`*`.`*`d`*, donde *`n`* es la longitud del campo, *`d`* es el número de dígitos asignados a la parte fraccionaria de la representación en punto flotante. El valor será representado en notación decimal siguiendo las siguientes reglas:

1. La representación del valor tiene justificación derecha.
2. Las últimas cuatro plazas del campo están reservadas para el exponente, comienza con la letra `E`, seguido del signo `+` o del signo `-` y dos dígitos (`0-9`); es decir, por ejemplo `E+09`.
3. La parte fraccionaria de la representación en punto flotante del valor `real` ocupan los *`d`* caractéres del campo antés de la parte asignada al exponente sin espacios. Los caracteres asignados para la parte fraccionaria son únicamente dígitos (`0-9`).
4. El caracter punto "`.`" va a la izquierda de la parte decimal, sin espacios.
5. La parte entera está representada por `0` o níngun caracter y va a la izquierda del punto, sin espacios de separación.
6. En caso que el número sea negativo, el caracter `-` va a la izquierda de la parte entera o el punto sin espacios
7. Los caracteres a la izquierda del primer dígito, eventualmente el punto o el `-` son espacios.

Ejemplos Las representaciones de 1/30 1/3, 10/3 y −10/3, utilizando el especificador `E10.4` serán respectivamente:

```
!234567890
0.3333E-01  ! 1/30 en E10.4
0.3333E-00  ! 1/3 en F10.4
0.3333E+01  ! 10/3 en F10.4
-.3333E+01  ! -10/3
```

En caso que el número de caracteres asignados al campo `E`*`n`*`.`*`d`* sea insuficiente, para representar un valor `real`, el computador escribirá caracteres `*` en el campo y leerá únicamente lo que se encuentra en el campo.

El especificador de formato punto flotante, para valores de tipo `real` (simple precisión), es `E`*`n`*`.`*`d`*, donde *`n`* es la longitud del campo, *`d`* es el número de dígitos asignados a la parte fraccionaria de la representación en punto flotante. El valor será representado en notación decimal siguiendo las siguientes reglas:

1. La representación del valor tiene justificación derecha.
2. Las últimas cuatro plazas del campo están reservadas para el exponente, comienza con la letra `E`, seguido del signo `+` o del signo `-` y dos dígitos (`0-9`); es decir, por ejemplo `E+09`.
3. La parte fraccionaria de la representación en punto flotante del valor `real` ocupan los *`d`* caractéres del campo antés de la parte asignada al exponente sin espacios. Los caracteres asignados para la parte fraccionaria son únicamente dígitos (`0-9`).
4. El caracter punto "`.`" va a la izquierda de la parte decimal, sin espacios.

5. La parte entera está representada por `0` o níngun caracter y va a la izquierda del punto, sin espacios de separación.

6. En caso que el número sea negativo, el caracter `-` va a la izquierda de la parte entera o el punto sin espacios

7. Los caracteres a la izquierda del primer dígito, eventualmente el punto o el `-` son espacios.

Ejemplos Las representaciones de 1/30 1/3, 10/3 y $-10/3$, utilizando el especificador `E10.4` serán respectivamente:

```
!234567890
0.3333E-01  ! 1/30 en E10.4
0.3333E-00  ! 1/3 en F10.4
0.3333E+01  ! 10/3 en F10.4
-.3333E+01  ! -10/3
```

En caso que el número de caracteres asignados al campo `E`*`n`*`.`*`d`* sea insuficiente, para representar un valor `real`, el computador escribirá caracteres `*` en el campo y leerá únicamente lo que se encuentra en el campo.

El especificador de formato punto flotante, para valores de tipo `real` doble precisión, es `D`*`n`*`.`*`d`*, donde *`n`* es la longitud del campo, *`d`* es el número de dígitos asignados a la parte fraccionaria de la representación en punto flotante. Las reglas son similares al especificador `E`*`n`*`.`*`d`*, ver la documentación del compilador, para más detalles.

Un especificador para valores de tipo `real`, es el especificador `g`*`n`*`.`*`d`*, donde *`n`* es la longitud del campo, *`d`* es el número de dígitos asignados a la parte fraccionaria de la representación en punto flotante. Este especificador elige automáticamente la utilización del especificador `E`*`n`*`.`*`d`* o del especificador `D`*`n`*`.`*`d`* o del especificador de punto flotante `F`*`n`*`.`*`d`*. En el caso que utilice el formato de punto flotante, la justificación es por la izquierda y utiliza *`d`* dígitos que incluyen la parte entera y fraccionaria del valor. La utilización de este especificador permite la escritura de resultados más comprensibles. Así por ejemplo; los valores 10/3 y 1/30, tendrán como representación, utilizando el especificador `g.10.5`:

```
!234567890
3.3333
.33333E-01
```

Para la lectura y escritura de cadenas de caracteres, se tiene los especificadores `A` y `A`*`d`*. El especificador `A` utiliza la longitud de la cadena de la expresión de tipo `character` que va a ser representada; mientras que en el especificador `A`*`d`* la longitud de la cadena es *`d`* y su justificación es por la izquierda.

Los especificadores `A` y `A`*`d`* se utiliza normalmente para expresiones de tipo `character` que pueden tomar diferentes valores en la ejecución del programa. Para la escritura exclusivamente (no para la lectura) se puede crear un campo, asignando una cadena de caracteres preestablecida, siendo la longitud del campo, la longitud de la cadena. La manera de crear el campo es introducir la cadena delimitada por comillas o doble comillas, como por ejemplo: `Él valor de x es =’`.

Los valores resultantes de expresiones de tipo `logical` pueden ser escritos y leídos, utilizando los especificadores `L` y `L`*`d`*. El especificador `L` utiliza un campo de dos caracteres y el especificador `L`*`d`* utiliza un campo de *`d`* caracteres. Para ambos especificadores la justificación es por derecha. La escritura dará los valores `T` y `F`. Para la lectura se puede utilizar `T` y `F` o bien `.true.` y `.false.` si la longitud del campo lo permite.

Aparte de los especificadores de formato, para representar valores de diferente tipo, existen especificadores de control, de los cuales detallamos los más utilizados.

El especificador `X` desplaza de un espacio el siguiente campo de lectura o escritura, consiguientemente el campo creado tiene el caracter “␣”.

El especificador `T`*`n`* desplaza a la columna *`n`* el siguiente campo de lectura o escritura; se lo conoce como el especificador tabulador.

Cuando se ejecuta una instrucción de lectura (`read`) o de escritura (`print` o `write`), una vez concluída la instrucción la computadora pasa a la siguiente línea; para evitar el cambio de línea se tiene a disposición el especificador `$` que mantiene la línea después del último campo.

De la misma forma, el especificador `/` permite cambiar a la línea siguiente.

En el cuadro IV.1, se tiene una síntesis de los especificadores descritos más arriba.

Esp.	Nombre	Acción
I*n*	Entero	Asigna un campo de *n* caractereres para representar un entero en notación decimal, justificado por derecha.
F*n*.*d*	Punto Fijo	Asigna un campo de *n* caractereres para representar un real en formato punto fijo, con *d* dígitos para la parte fraccionaria, justificado por derecha.
E*n*.*d*	Punto Flotante	Asigna un campo de *n* caractereres para representar un real en formato punto flotante, con *d* dígitos para la parte fraccionaria, justificado por derecha.
D*n*.*d*	Punto Flotante doble precisión	Lo mismo que especificador E, pero en doble precisión.
g*n*.*d*	Real	Asigna un campo de *n*caracteres, con *d* dígitos; dependiendo del valor: en punto fijo, justificación a la izquierda o punto flotante, justificación a la derecha.
A*n*	Texto	Asigna un campo de longitud *n*, para expresiones de tipo `character`.
A	Texto	Asigna un campo de longitud, la longitud de la expresión de tipo `character`.
'...'	Texto Fijo	Asigna el campo de longitud la cadena de caracteres y escribe en el campo la cadena.
L*n*	Lógico	Asigna un campo de *n* caracteres para valores lógicos, justificación a la derecha
L	Lógico	Asigna un campo de 2 caracteres para valores lógicos, justificación por derecha.
X	Espacio	Desplaza el siguiente campo de un espacio.
T*n*	Tabulador	El siguiente campo comienza en la columna *n*
$	Mantención de línea	No cambia la linea al final una instrucción de escritura o lectura.
/	Cambio de línea	Cambia de línea.

Cuadro IV.1: Principales Especificadores de Formato y Control

Asignación de Campos de Escritura y Lectura

Al inicio de esta sección, se manifestó que la unidad de escritura o lectura es la línea, tanto cuando se trata de la pantalla o de archivos de tipo texto. Ahora veamos, la manera de asignar los campos de lectura o escritura, utilizando los especificadores de formato y control de campos. La sintaxis de asignación es la siguiente:

```
(<especificador 1>[,<especificador 2>,...<especificador n>)
```

donde <`especificador k`> es un especificador simple, como los descritos más arriba, o es un especificador compuesto, obtenido a partir de especificadores simples. Las reglas de asignación de campos es la siguiente:

1. El primer campo comienza en la primera columna de la línea, a menos que se utilice el especificador tabulador T*n*, que en ese caso envía el inicio a la columna *n* de la línea.

2. Los otros campos empiezan en el caracter siguiente a la finalización del campo que le precede, a menos que antes esté el especificador tabulador, o bien el especificador /, que en ese caso envía al campo a la primera columna de la siguiente línea.

3. En caso que se utilice como último especificador $ en la anterior instrucción de escritura (`write` o `print`) o lectura (`read`) el primer campo empieza en el caracter siguiente del último campo de la anterior instrucción; a menos que se utilice el especificador tabulador T*n*.

4. En caso de necesidad, se puede repetir un especificador simple o compuesto, utilizando un factor de repetición *n*, con la siguiente sintaxis

   ```
   n<especificador>
   ```

como por ejemplo, *n*`Id`, que significa que el campo especificado por `Id` se repite *n* veces de manera seguida en la línea de escritura o lectura.

5. Un especificador compuesto se obtiene agrupando varios especificadores, de la siguiente manera

   `(<especificador 1>,...,<especificador n>)`

   como por ejemplo: `(X,'x=',g10.5)`

6. La instrucción de asignación de campos de escritura y lectura es una cadena de caracteres que se inicia delimitando con `(` y se termina deliminando con `)`

7. Para la instrucción de lectura `read`, la instrucción de asignación no puede contener especificadores de texto fijo `'...'`.

Habiendo concluido este trabajo arduo de describir especificadores y establecer la instrucción de asignación de campos, estamos en capacidad de escribir instrucciónes de escritura y lectura con formato preestablecido en el código fuente. Tenemos varias posibilidades para hacerlo, siguiendo las siguientes sintaxis:

```
read '<instr. de asign.>',<variable 1>[,<variable 2>,...,<variable n>]
read <expr. de tipo char>,<variable 1>[,<variable 2>,...,<variable n>]
read <etiqueta>,<variable 1>[,<variable 2>,...,<variable n>]
read(*,'<instr. de asign.>')<variable 1>[,<variable 2>,...,<variable n>]
read(*,<expr. de tipo char>)<variable 1>[,<variable 2>,...,<variable n>]
read(*,<etiqueta>)<variable 1>[,<variable 2>,...,<variable n>]

print '<instr. de asign.>',<expres. 1>[,<expres. 2>,...,<expres. n>]
print <expr. de tipo char>,<expres. 1>[,<expres. 2>,...,<expres. n>]
print <etiqueta>,<expres. 1>[,<expres. 2>,...,<expres. n>]

write(*,'<instr. de asign.>')<expres. 1>[,<expres. 2>,...,<expres. n>]
write(*,<expr. de tipo char>)<expres. 1>[,<expres. 2>,...,<expres. n>]
write(*,<etiqueta>)<expres. 1>[,<expres. 2>,...,<expres. n>]
```

donde <`etiqueta`> es un valor `integer` que puede ser de `1` a `99999`, <`expr. de tipo char`> tiene como valor una instrucción de asignación: `'(.....)'`. Cuando se utiliza <`etiqueta`> en la instrucción de escritura o lectura, la instrucción de asignación se la escribe en otra línea del código fuente, de la siguiente manera,

```
<etiqueta> format<instr. de asign.>
```

Por ejemplo,

```
     write(*,100)x,ciudad,n
100  format(X,g10.5,X,A10,X,I3)
```

Por último, una regla, no de las menos importantes, que se debe seguir, para no tener problemas: Debe existir concordancia de orden y tipo, entre el listado de variables o expresiones de las instrucciones de lectura y escritura con los especificadores de formato de campos de escritura y lectura (nos olvidamos de los especificadores de control). Pueden sobrar especificadores de lectura y escritura, pero nunca faltar.

A continuación el código de los ejemplos de esta sección.

```
program demo2
  character(len=15)::instruccion='(X,L1,L2)'
  logical::p1=.true.,p2=.false.
  n=100
  write(6,*)'!234567890123456789012345678901234567890'
  write(6,'(X,g10.5,g10.5,F10.5)') 100./3.,1./30.,10./3.
  write(6,'(X,L,/,X,L)')p1,p2
```

```
  read instruccion,p1,p2
100 format(X,L1,L1)
  print 100,p1,p2
end program demo2
```

IV.2. Lectura y Escritura de Datos en Archivos

Hasta aquí, hemos visto las instrucciones de lectura y escritura, utilizando las insterfaces por defecto: el teclado para la lectura y la pantalla para la escritura. Las interfaces están interconectadas a la computadora por **unidades lógicas** y cada unidad lógica está identificada por un número entero, comprendido entre el 0 y 2,147,483,647. En general la unidad 5 está preconectada al teclado, la unidad 6 a la pantalla y la unidad 0 está reservada al *kernel* del sistema operativo. Consiguientemente, para poder escribir o leer datos en un archivo, éste debe estar asociado a una unidad lógica; obviamente, diferente a la 5, 6 y 0.

Los archivos a los que accede y utiliza el programa son de tipo texto, si los datos son formateados, ya sea en formato automático o preestablecido; para este tipo de archivos, las extensiones `.txt`, `.dat`, `.out`, *etc* son utilizadas. Los archivos son de tipo binario (lenguaje máquina) si los datos son guardados sin formato.

Se puede acceder a un archivo de dos formas: la primera, de manera secuencial, en la cual se deben leer todos los registros anteriores al que se desea acceder; la segunda forma de acceso directo, en el cual la instrucción de escritura/lectura indica el número de registro que se desea tratar.

Comencemos con lo primero.

IV.2.1. Apertura de Archivos

Para la apertura de un archivo, existe una manera corta, cuya sintaxis es

```
open(<n>,file='<archivo>')
```

o bien

```
open(<n>,file=<nombre_archivo>)
```

donde <n> es una expresión de tipo `integer`, cuyo vor está dentro el rango permitido, diferente a 0, 5 y 6. <archivo> es el archivo, con su ubicación, que sea desea abrir y <nombre_archivo> es una expresión de tipo `character`, cuyo valor da el archivo (con su ubicación) que se desea abrir.
Ejemplo.- Código para abrir un archivo de la manera corta.

```
  :
  character(len=20)::archivo='/tmp/datos.dat'
  :
  open(10,file='informe.out')
  :
  open(101,file=archivo)
  :
```

La otra forma de abrir un archivo, consiste en utilizar la instrucción `open` con sus opciones; la sintaxis es:

```
open([unit=]<n>,file=<archivo>[,<opción>=<valor>])
```

siendo las opciones más utiles: `status`, `action`, `position`, `iostat`, `err`, `form`, `access` y `recl`.

A continuación, describiremos más o menos en detalle, cada una de las opciones mencionadas.

unit

La opción `unit`, cuya sintaxis es

```
[unit=]<unidad>
```

donde <unidad> es una expresión de tipo `integer`, cuyo valor se encuentra dentro el rango permitido y es diferente de 0, 5 y 6, sirve para identificar la unidad lógica que identificará el archivo que será abierto. Generalmente, la opción `unit` no se la coloca explícitamente en la instrucción `open`, colocando <unidad> antes que las otras opciones; si no es el caso, `unit` debe ir obligatoriamente.

file

La opción `file` es una opción obligatoria, que normalmente va despueés de la opción `unit`, su sintaxis es

```
file=<nombre_archivo>
```

donde <`nombre_archivo`> es una expresión de tipo `character`, cuyo valor indica el nombre y ubicación del archivo que se desea abrir.

status

La sintaxis de esta opción optativa es

```
status=<valor>
```

donde <`valor`> es una expresión de tipo `character`, cuyos valores son:

- `'unknown'` que es el defecto. Si no existe lo crea y si ya existe lo mantiene.
- `'old'`, el archivo ya existe y en caso que no existiese la computadora establece error.
- `'new'`, el archivo no existe y lo creará. En caso que exista lo borra y vuelve a crearlo.
- `'scratch'`, es un archivo temporal que es creado para el uso del programa, una vez cerrado se destruye. En este caso no es necesario incluir la opción `file`. La sintaxis de la instrucción `open` será por consiguiente:

  ```
  open([unit=]<unidad>,status='scratch'[,<otras opciones>]
  ```

- `'replace'`, el archivo ya existe y será remplazado por uno nuevo.

action

La sintaxis de esta opción optativa es

```
action=<valor>
```

donde <`valor`> es una expresión de tipo `character`, cuyos valores son:

- `'readwrite'`, la opción por defecto, el archivo puede ser leído y escrito.
- `'read'`, el archivo es solo de lectura.
- `'write'`, el archivo es solo de escritura.

position

La sintaxis de esta opción optativa es

```
position=<valor>
```

donde <`valor`> es una expresión de tipo `character`, cuyos valores son:

- `'asis'`. Una vez abierto, si la primera instrucción con el archivo es de lectura, la posición del archivo es la primera línea del archivo y si es de escritura la posición es una nueva línea posterior a la última. Es la opción por defecto.
- `'rewind'`, la posición es la primera línea.
- `ápend'`, la posición es en la línea posterior a la última línea.

iostat

La opción `iostat` permite al programa continuar, en caso de error durante la apertura del archivo. La sintaxis de esta opción optativa es

```
iostat=<ierr>
```

donde <`ierr`> es una variable de tipo `integer`. Una vez concluída la instrucción `open`, si <`ierr`> vale `0`, significa que la apertura se ha desarrollado sin problemas, otros valores significa que la apertura del archivo ha tenido problemas. La documentación de cada compilador tiene los detalles para uno de los valore posibles de <`ierr`>.

err

La opción `err` permite al programa continuar, en caso de error durante la apertura del archivo. La sintaxis de esta opción optativa es

```
err=<etiqueta>
```

donde <`etiqueta`> es entero de `1` a `99999`. Si hay error durante la apertura del archivo, el programa va a la instrucción etiquetada. Por ejemplo:

```
   :
   open(100,file='datos.dat',iostat=ierr,err=23)
   :
23 print*,'Hay un error con el archivo datos.dat con el error',ierr
   :
```

form

La sintaxis de esta opción optativa es

```
form=<valor>
```

donde <`valor`> es una expresión de tipo `character`, cuyos valores son:

- `'formatted'`. La opción por defecto, si el archivo es de tipo texto; es el caso de los archivos que pueden ser manipulados por editores de texto, como *emacs*
- `'unformatted'`, si la representación de los datos del archivo se la realiza en lenguaje máquina o binario.

access

La sintaxis de esta opción optativa es

```
access=<valor>
```

donde <`valor`> es una expresión de tipo `character`, cuyos valores son:

- `'sequential'`. La opción por defecto, la computadora lee y escribe los datos, línea tras línea.
- `'direct'` la instrucción de lectura/escritura indica el número de registro a tratar. Los registros deben tener el mismo tamaño y los datos son sin formato; es decir, en lenguaje máquina.

recl

Esta opción es obligatoria, cuando el acceso es directo; es decir, cuando `access='direct'` tiene lugar. La sintaxis es

```
recl=<n>
```

donde <`n`> es la talla del registro. Normalmente <`n`> toma los valores 1, 2, 4, 8 o 16 que es el número de *bytes* que ocupa cada registro.

IV.2.2. Lectura y Escritura en Archivos

Para la lectura y escritura de datos, de la misma manera que para la instrucción `open`, existe una forma corta y otra con opciones. La forma corta, tiene como sintaxis, para la escritura y la lectura en archivos de tipo texto (formateados):

```
write(<unidad>,<formato>) <expr. 1>[,<expr. 2>,...,<expr. n>]
read(<unidad>,<formato>) <variable 1>,[<variable 2>,...,<variable n>]
```

donde <`unidad`> es la únidad lógica que identifica al archivo, <`formato`> puede ser: `*` para formato automático, una instrucción de asignación de campos o una etiqueta, descritas en la sección precedente.

Para datos sin formato; es decir archivos en lenguaje máquina, la sintaxis es:

```
write(<unidad>) <expr. 1>[,<expr. 2>,...,<expr. n>]
read(<unidad>) <variable 1>,[<variable 2>,...,<variable n>]
```

Ejemplo.- Ilustramos el uso de la forma corta, tanto en su variante de datos formateados, como de datos no formateados.

```
:
open(10,file='datos.dat')
open(20,file='resultados',form='unformatted')

write(10,'(2X,I4,X,2(g12.5,X))') n,x1,x2
read(20) y,n,ciudad
:
read(10,*) n,x1,x2
:
write(20)'Cochabamba',y,x2
```

El uso de las instrucciones `read` y `write`, puede realizarse utilizando opciones, la sintaxis de esta forma es:

```
read([unit=]<unidad>[,[fmt=]<formato>,<otras opciones>) <lista variables>
write([unit=]<unidad>[,fmt=<formato>,<otras opciones>) <lista expresiones>
```

donde la opción `unit` ha sido tratada en la subsección referente a la instrucción `open`. La opción `fmt` es la opción relacionada al formato de lectura y escritura de datos, donde <formato> ha sido tratado más arriba, tanto al inicio de esta subsección, como en una anterior.

Las otras opciones serán abordadas de acuerdo al uso que se les da y sobre todo al contexto en que intervienen.

Control de error de lectura

Puede suceder, muchas veces, que los datos del archivo estén dañados, no correspondan al tipo de la variable que se va asignar el valor leído o que simplemente, ya no haya datos que leer. Las opciones `iostat` y `err` pueden utilizarse en el proceso de lectura y la sintaxis es la misma que para la instrucción `open`. Es decir:

```
iostat=<ierr>
err=<etiqueta>
```

Si la instrucción `read` se ha llevado a cabo sin problemas, la variable de tipo `integer`, <`ierr`> toma el valor `0`, sino otro valor. En el caso que la instrucción `read` no se haya podido llevar sin errores, el programa va a la instrucción etiquetada con <`etiqueta`>. Como ejemplo de código, se tiene:

```
   read(11,400,iostat=ierr,err=15)n,x1,ciudad
   :
15 ciudad='Cochabamba'
   :
400 format(I4,X,f15.5,X,a10)
   :
```

Lectura y Escritura en acceso directo

Cuando el acceso al archivo es directo, opción `access='direct'` de la instrucción `open` los datos son sin formato y las instrucciones de lectura y escritura se realizan con la opción `rec` siguiendo la siguiente sintaxis:

```
read([unit=]<unidad>,rec=<registro>) <variable>
write([unit=]<unidad>,rec=<registro>) <expresión>
```

donde <`registro`> es una expresión de tipo `integer`, cuyo valor es el número de registro que se desea acceder para la lectura o escritura. Un solo dato por lectura o escritura; ademas, todos los datos deben tener la misma talla en *bytes*, dada por la opción `recl` de la instrucción `open`. Por consiguiente, se debe seguir el modelo, dado por el siguiente código:

```
real(kind=8)::x,y
:
open(50,file='datos.dat',access='direct',recl=8)
:
read(50,rec=25) x
:
write(50,rec=n) y
:
```

Lectura y escritura sin cambio de línea

En la primera sección de este capítulo, se vio que el especificador de formato `$`, colocado al final de la instrucción de asignación de campos de escritura y lectura, tenía como efecto la mantención de línea. Esto se puede puede hacer, sin necesidad de este especificador, por medio de la opción `advance=ño'`, cuya utilización es posible, si el valor de la opción `fmt` es diferente a `*`. Como ejemplo de código, se tiene:

```
write(14,'(3x,2(g12.4,x))',advance='no') x, y
```

que es equivalente a la instrucción

```
write(14,'(3x,2(g12.4,x),$)') x,y
```

Finalmente, existen otras formas de lectura y escritura en archivos, que se deja a las necesidades e inquietudes personales, como la utilización del `namelist`, etc.

IV.2.3. Otras instrucciones

Se ha abordado lo sustancial en lo que se refiere a la escritura y lectura de datos en archivos, queda pendiente describir algunas instrucciones, cuyo uso puede ser útil y necesario.

Cierre de archivos

Cuando se ha concluído de trabajar con un archivo o se requiere manipular el archivo con otras opciones, con las que se abrio inicialmente, es conveniente cerrar el archivo. Esto se consigue con la instrucción `close`, cuya forma corta es

```
close(<unidad>)
```

o bien, a través de la forma mas completa:

```
close([unit=]<unidad>[,status=<valor>)
```

<valor> puede tomar `'keep'` que es el valor por defecto, guardando el archivo al cerrar. El otro valor es `'delete'`, que borra el archivo al cerrarse.

Remontar al inicio de un archivo

Uno puede retornar al inicio de un archivo, utilizando la instrucción `rewind`, cuya sintaxis es:

```
rewind([unit=]<unidad>)
```

Remontar de una fila o registro

Es posible remontar de una fila en el archivo, utilizando la instrucción `backspace`, cuya sintaxis es

```
backspace([unit=]<unidad>)
```

IV.3. Ejercicios

1. Completar en programas, los códigos de los ejemplos y contrastar los resultados, con lo expuesto en este capítulo.

Capítulo V

Tableros, Operaciones con Tableros

Los tableros que son arreglos de valores de un determinado tipo, engloban las nociones de vector (tablero unidimensional), matriz (tablero bidimensional), grilla (tablero tridimensional), etc. Se los puede generar de manera estática o de manera dinámica.

Los tableros pueden ser tratados por `Fortran`, como listas de valores para ciertas funciones y como matrices, en el sentido matemático, por otras.

V.1. Nociones sobre Tableros

Para poder comprender, cómo `Fortran` maneja los tableros, debemos abordar desde el punto de vista de una lista de valores indexados; por consiguiente:

1. Un tablero unidimensional es una lista indexada de valores a un índice:

   $$x_{ni},\ x_{ni+1}, \ldots, x_i, \ldots, x_{ns-1},\ x_{ns}$$

   Se observa que el índice i que identifica la ubicación del valor en la lista está acotado por ni y ns. Normalmente la cota inferior ni vale 1; sin embargo, nada impide que tome otros valores, la única condición es que $ns \geq ni$.

2. Un tablero bidimensional es una lista indexada de valores a dos índices:

   $$\begin{array}{ccccccc} x_{ni,mi} & x_{ni,mi+1} & \cdots & x_{ni,j} & \cdots & x_{ni,ms} \\ x_{ni+1,mi} & x_{ni+1,mi+1} & & & \cdots & x_{ni+1,ms} \\ \vdots & & & & & \\ x_{i,mi} & & & x_{i,j} & & \\ \vdots & & & & & \\ x_{ns,mi} & & & x_{ns,j} & & x_{ns,ms} \end{array}$$

   Los índices i y j están acotados por ni y ns, mi y ms respectivamente. Tradicionalmente el índice i identifica el número de fila y el índice j identifica el número de columna de la ubicación de un valor en la lista. Al igual que en el tablero unidimensional, los valores de las cotas inferiores ni y mi valen 1.

3. Un tablero tridimensional será una lista indexada a tres índices i, j y k, donde los índices estan acotados por una cota inferior y una cota superior como en el tablero unidimensional y en el tablero bidimensional.

4. Un tablero n-dimensional sera una lista indexada a n índices i_1, i_2,...,i_n, donde los índices están acotados por una cota inferior ni_k y una cota superior ns_k.

Ahora bien, la computadora solo maneja listas indexadas a un indice, por lo que `Fortran`, en el proceso de compilación, convierte los tableros multidimensionales en tableros unidimensionales, cuyo índice está acotado inferiormente por 1. Esta hecho es fundamental y debe ser tomado en cuenta, cuando se utiliza tableros multidimensionales en los programas.

Conversión de tableros multidimensionales a unidimensionales

El primer paso en la conversión, inclusive para tableros unidimensionales, es la transformación de la cota inferior ni_k a 1; ésto significa una transformación del índice i_k en el índice j_k, dada por:

$$j_k = i_k + 1 - ni_k, \quad ms_k = ns_k + 1 - ni_k. \tag{V.1.1}$$

donde ms_k es la cota superior del índice j_k.

Con esta transformación inicial podemos suponer que las cotas inferiores de los índices i_1, i_2,...,i_n es 1 y ns_k. Continuamos con la conversión, describiendo su algoritmo:

P1.- Si $n > 1$, convertimos el tablero de índices $i_1, i_2, \ldots, i_{n-1}, i_n$ en un tablero de índices $i_1, i_2, \ldots, i_{n-2}, j_{n-1}$, a través de la transformación:

$$j_{n-1} = ns_n(i_n - 1) + i_{n-1}, \quad 1 \leq i_{n-1} \leq ns_{n-1},\ 1 \leq i_n \leq; ns_n. \tag{V.1.2}$$

lo que significa que $1 \leq j_{n-1} \leq ns_{n-1} \cdot ns_n$.

P2.- Continuar con el paso 1, hasta obtener un tablero unidimensional.

En el caso de un tablero bidimensional, una visualización gráfica de la conversión, ilustrada por la figura V.1.1, muestra que la conversión se la hace por columnas; es decir, se coloca los valores de la primera columna, luego de la segunda columna y así sucesivamente.

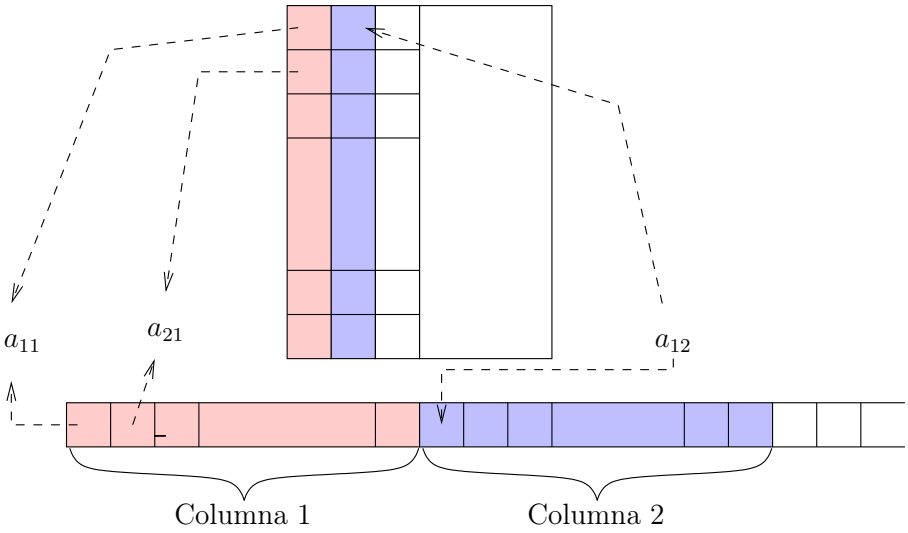


Figura V.1.1: Conversión de Tableros

La forma de conversión que realiza `Fortran`, hay que tenerla encuenta cuando se recurre a subprogramas hechos en otros lenguajes de programación, como `C`, ya que muchos de estos lenguajes convierten los tableros a tableros unidimensionales a través de las filas y no columnas como en `Fortran`.

V.2. Declaración de Tableros

Comencemos con la forma más facil de declarar un tablero, por la forma estática, definiendo el número de índices y el rango de los índices, al momento de declararlo. La notación o simbología, en `Fortran` del rango es,

$$\texttt{ni} : \texttt{ns} \quad \text{que significa } \texttt{ni} \leq i \leq \texttt{ns}. \tag{V.2.3}$$

El número de índices y su posición, se obtiene colocando los rangos en orden, de izquierda a derecha, separados por una coma. Así; por ejemplo,

$$\texttt{ni}_1 : \texttt{ns}_1, \texttt{ni}_2 : \texttt{ns}_2, \ldots, \texttt{ni}_n : \texttt{ns}_n \tag{V.2.4}$$

significa que el tablero tiene `n` índices con sus respectivos rangos. Cuando un `ni=1`, no se puede indicar el rango del índice, solamente colocando `ns`, en lugar de `1:ns`; el compilador asumirá que el rango es `1:ns`.

La declaración de un tablero, cuyos valores son de un tipo dado, puede realizarse, utilizando el atributo `dimension` o directamente sobre la variable, de acuerdo a las siguientes sintaxis:

```
<tipo>,dimension(<decl. índices>)[,<otros atrib.>]::<lista variables>
<tipo>[,<otros atrib.>]::<variable>(<decl. índices>)[,<otras varia>]
```

donde, <tipo> es uno de los tipos intrínsicos (`integer`, `real`, `character`, `complex`, `logical`) o un tipo derivado; <decl. índices> está dado por (V.2.4) y donde cada $\texttt{ni}_k$ y $\texttt{ns}_k$ es un valor constante de tipo `integer` o una variable constante de tipo `integer`, declarada antes, con el atributo `parameter`. De esta manera, se tiene como, las siguientes declaraciones válidas:

```
integer,parameter::ni1=-2,ns1=3,ni2=4,ns2=10
type carrera
 character(len=15)::nombre
 integer::nivel
 real(kind=4)::promedio
end type
real(kind=8)::vector(20),escalar
complex(kind=8),dimension(ni1:ns1,ni2:ns2)::matriz
type(carrera),dimension(273,2:ns2)::matematicas
```

Una vez declarado un tablero, los elementos o componentes del tablero son identificados por sus índices, tal como ilustran los siguientes ejemplos, de los tableros dados por el código de más arriba.

```
vector(10)
vector(i)
matriz(i,j)
matriz(10,i*5-j)
matematicas(20,(n2-ns1)/2)
```

Por lo observado, puede utilizarse como índices: valores enteros fijos, variables de tipo `integer` o expresiones de tipo `integer`. Solamente hay que tener cuidado que el número de indices esté en concordancia con la declaración del tablero y que los valores de los indices estén dentro el rango de cada uno de los índices del tablero.

V.2.1. Asignación de valores a tableros

Definir el contenido de un tablero; es decir, dar valores a cada uno de sus elementos o componentes, puede realizarse componente por componente, o bien de manera global.

La asignación de valores, componente por componente de un tablero de un tipo dado, se basa en el principio que cada componente es una variable (independiente) del mismo tipo del tablero, por lo tanto la asignación se la hace de la misma manera que para una variable no tablero, ya vista en un capítulo anterior. Así:

```
real(kind=8),dimension(3)::vector
real(kind=4)::matriz(2,2)
:
:
vector(1)=1.d0; vector(2)=2.d0; vector(3)=3.d0
matriz(1,1)=1.; matriz(2,1)=0.
matriz(1,2)=0.; matriz(2,2)=vector(2)*vector(3)
```

es un ejemplo de definir el contenido asignando valores, componente por componente.

Ahora definamos el contenido de un tablero de manera global. Por lo tanto, comencemos con tableros unidimensionales (de un solo índice); para tal efecto, puede introducirse la totalidad de las componentes, utilizando los delimitadores `(/` y `/)`, por medio de la sintaxis:

```
<tablero>=(/<listado de (ns-ni)+1 expresiones>/)
```

donde <tablero> es de `dimension(ni:ns)`, las expresiones del listado son del mismo tipo que el tablero y están separadas por comas. El listado puede darse de manera explícita, como por ejemplo:

```
integer,dimension(5)=numero, numero2
:
numero=(/1,2,3,4,5/)
numero2=(/1,j,k,4,m/)
```

O bien, utilizando constructores de tablero (*array constructor*) que son instrucciones de la forma:

```
(<expr. 1>[,<expr. 2>,...,<expr. k>],<variable>=<np>,<nf>[,<nc>])
```

donde <variable> es de tipo `integer`, que toma los valores enteros `np`, `np+nc`, `np+nc+nc`, ...,`np+nc+···+nc` comprendidos entre `np` y `nf`; si se omite `nc`, toma el valor de `1`. Ilustramos con los siguientes ejemplos:

```
(0.,(-1.)**{k},k=0,2) ! da 0.,1.,0.,-1.,0.,1.
(100.,i=10,9)         ! no da nada.
(100.,i=10,9,-1)      ! da 100,100
(5*i,i=1,5,2)         ! da 5,15,25
```

El o los constructores de tablero se insertan en los lugares que se quiere asignar los valores, dentro los delimitadores de tablero unidimensional `(/` y `/)`, teniendo cuidado que el número de valores introducidos en el tablero corresponda al asignado en la declaración. Asi:

```
real(kind=8),dimension(8)::vector1,vector2
:
vector1=(/1.d0,((-1.d0)**i,i=0,5),-1.d0/)
vector2=(/(cos(2*pi*k/(17.d0)),k=0,7)/)
```

son asignaciones de contenido correctas, utilizando constructores de tableros.

También es posible inicializar un tablero (unidimensional) al momento de declarar:

```
integer,dimension(-3,3)::v=(/((-1)**k,k=-3,3)/)
```

Ahora pasemos con los tableros multidimensionales. Se lo puede hacer a partir de un tablero unidimensional, por medio de la instrucción `reshape`

reshape

El objetivo es convertir un tablero unidemensional `v(ns)` a un tablero multidimensional $\texttt{T(ns}_1\texttt{,...,ns}_m$. Para tal efecto, se utiliza la instrucción `reshape`, cuya sintaxis es:

```
T=reshape( v,(/ns1,ns2,...,nsm/)[,order=(/i1,i2,...,im/),pad=(/<lista>/)])
```

donde `nsk` es el rango del índice `k`, la opción `order` indica el orden, en el cual se tiene que llenar el tablero `T`, la opción `pad` los valores que se deben asignar a las variables que no han sido asignadas con los valores de `v`. Cuando no se utiliza la opción `orden`, la conversión se la hace siguiendo el orden preestablecido por `Fortran`.

Ilustremos el uso de `reshape` y sus opciones por medio de los siguientes ejemplos:
Consideramos la instrucción siguiente:

```
matriz=reshape((/1,2,3,4,5,6/),(/3,2/))
```

da como resultado

$$\texttt{matriz} = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Como puede observarse, el primer argumento contiene la lista de los valores a utilizar. El segundo argumento la forma del tablero, en este caso una matriz.

Utilizando la opción `order`, se puede modificar el orden de conversión.

```
matriz=reshape((/1,2,3,4,5,6/),(/3,2/),order=(/2,1/))
```

da la matriz:

$$\texttt{matriz} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}.$$

Podemos precisar los valores "por defecto" que deben ser utilizados para el llenado, en caso que la lista de valores del primer argumento no llenen la totalidad de los valores del tablero multidimensional.

```
matriz=reshape((/1,2,3/),(/3,2/),order=(/2,1/),pad=(/0/))
```

da

$$\texttt{matriz} = \begin{pmatrix} 1 & 2 \\ 3 & 0 \\ 0 & 0 \end{pmatrix}.$$

y

```
matriz=reshape((/1,2,3/),(/3,2/),order=(/2,1/),pad=(/0,1/))
```

da

$$\texttt{matriz} = \begin{pmatrix} 1 & 2 \\ 3 & 0 \\ 1 & 0 \end{pmatrix}.$$

Para concluir los ejemplos de la instrucción `reshape`, veamos una forma de definir la matriz identidad `nid` $\times$ `nid`, como parámetro.

```
  :
  integer,parameter::nid=10, np=8
  real(kind=np),dimension(nid*nid),parameter::idaux=(/1._np,&
               &((0._np,i=1,nid),1._np,j=1,nid-1)/)
  real(kind=np),dimension(nid,nid),parameter::Id=reshape(idaux,(/nid,nid/))
   :
```

Este es un lindo ejemplo de utilización de constructores de tableros y la función `reshape`.

V.2.2. Declaración Dinámica de Tableros

`Fortran 90` y versiones posteriores permiten crear tableros, asignando memoria de manera dinámica, durante la ejecución de un programa. Existen tres momentos para la creación de un tablero de estas características: un primer momento de notificación, durante las declaraciones de variables; un segundo momento de creación propiamente dicho del tablero y un tercer momento de destrucción del tablero.

La notificación, se hace asignando un tipo al tablero, una forma (cuantos índices tendrá el tablero) a través de la opción `dimension` y utilizando la opción `allocatable`., de acuerdo al siguiente código ilustrativo:

```
real,dimension(:),allocatable:: vector
complex,dimension(:,:),allocatable::matriz
character(len=8),dimension(:,:,:),allocatable::grilla
```

La creación del tablero con el rango de índices asignado, se hace a través de la instrucción `allocate`, de la misma forma, a las siguientes líneas de instrucción:

```
:
n=10
:
allocate(vector(3:n),matrix(1:n,0:n-1),grilla(3,4,2))
```

Remarcamos que `allocate` puede crear varios tableros al mismo tiempo, lo único que hay tener cuidado los tableros estén separados por "comas" y que estén en concordancia con lo declarado, a través de la opción `allocatable`.

Por último la destrucción o anulación del tablero, se la hace utilizando la instrucción `deallocate`, siguiendo el ejemplo:

```
deallocate(vector,matriz)
deallocate(grilla)
```

Un tablero, por ejemplo `matriz`, puede ser creado y destruído las veces que uno quiera.

Uno de los inconvenientes, y no de los menores, cuando se crea dinámicamente tableros es la asignación de memoria por parte de la computadora. Muchas veces, dependiendo los otros usos que se esté haciendo de la computadora o las condiciones de configuración, hacen que el programa tenga problemas; por lo tanto, se puede crear un tablero, agregando la opción `stat` a la instrucción `allocate`. Por ejemplo, la instrucción:

```
allocate(vector(1:n),stat=ierr)
```

dara `ierr=0`, si la creación se ha desarrollado sin problemas, otro valor sino.

V.3. Asignación en tableros

Tal como se vio en la sección precedente, se puede asignar valores a un tablero elemento por elemento, de manera global y veremos también que es posible asignar valores por bloques o secciones de un tablero. Veamos las dos primeras formas, mediante los ejemplos ilustrativos:

```
real,dimension(3,2)::matriz1
real,dimension(-1:1,0:1)::matriz2
:
matriz1(1,1)=1.  ! asignacion a un solo elemento
matriz1=1.       ! asignacion a todo el tablero con el valor 1.
matriz2=matriz(2,2) ! asignacion de todo el tablero matriz2 con
                    ! el valor de matriz(2,2).
matriz2=matriz1  ! asignacion, copiando los valores de la matriz1
```

Para la ultima instrucción de asignación, es necesario que ambos tableros sean compatibles; es decir misma forma.

V.3.1. Secciones de Tableros

Se puede identificar bloques o secciones de tableros, utilizando indexaciones de la forma

```
<inicio>:<final>:<incremento>
```

Así

```
real,dimension(1:3)::vector
:
print*,vector(1:3:2) ! imprime: vector(1) y vector(3).
```

Si <`inicio`> se omite, la cota inferior del índice es tomada. Si <`final`> se omite, la cota superior del índice es tomada. Si el <`incremento`> se omite, vale `1`. Estos argumentos pueden tomar valores negativos. Continuamos con líneas de código, a manera de ilustración:

```
real,dimension(1:3)::vector
real,dimension(1:3,1:3)::matriz
:
write(*,*)vector(1:2) ! los dos primeros elementos
write(*,*)matriz(1:1,1:3) ! la primera fila
write(*,*)matriz(1:3:2,1:3) ! la primera y tercera fila
write(*,*)matriz(3:1:-2,1:3) ! la tercera y primera fila
write(*,*)matriz(:2,:2) ! El primer bloque 2x2
write(*,*)matriz(:,1)  ! La primera columna
write(*,*)matriz(::2,1) ! El primer y tercer coeficiente de la
                        ! primera columna.
```

Hay que remarcar que `matriz(:,1)` se convierte en un tablero de rango `1`. Colocar una constante como índice, tiene como efecto reducir el rango de `1` del tablero considerado.

V.3.2. Expresiones de asignación y operaciones aritméticas

La primera regla que se debe seguir rigurosamente es que las expresiones entre tableros deben implicar tableros de la misma forma. La segunda regla es que los elementos de los tableros están relacionados por la relación de orden preestablecida, sobre los índices, por `Fortran`, por ejemplo en tableros matrices, primero son las columnas y luego las filas.

Las operaciones aritméticas y funciones intrínsicas, como `cos`, `exp`, operan elemento por elemento. Así por ejemplo, para dos tableros de la misma forma `A` y `B`, `A*B`, da un tablero de la misma forma `C`, cuyos elementos son:

$$c_{i_1,\ldots,i_m} = a_{i_1,\ldots,i_m} * bi_1,\ldots,i_m \tag{V.3.5}$$

remarcando que la multiplicación `*` no es la multiplicación de matrices u otras formas de multiplicación definidas en tableros.

Para ilustrar, veamos el siguiente código, a manera de ejemplo.

```
real,dimension(1:3)::vector
real,dimension(1:3,1:3)::matriz1
real,dimension(0:2,-1:1)::matriz2
:
matriz1(:,1)=matriz2(0,:)+vector ! El resultado va a la columna 1
                                 ! de la matriz1 de la suma de
                                 ! fila 0 con el vector
matriz1=matriz1*matriz2  ! Los productos son calculados
                         ! individualmente
matriz1=cos(matriz2)
matriz=exp(matriz2)
matriz2(:,0)=sqrt(vector)
```

Vectores sub-indices

Aparte de utilizar indicadores de bloques o secciones de tableros, también es posible identificar los elementos de un tablero, a través de un vector índice, cuyos elementos son de tipo `integer`. Ilusstramos el uso de estos vectores sub-indices, con el siguiente código:

```
integer,dimension(1:3)::indice::(/2,4,6/)
integer,dimension(1:10)::permutacion::(/3,4,5,6,7,8,9,10,1,2/)
real,dimension(1:10,1:10)::Matriz
:
print*,Matriz(5,indice) ! escribe los elementos (5,2), (5,4) y (5,6)
print*,Matriz(indice,indice) ! los elementos en el orden (2,2),(4,2), (6,2)
                             ! (2,4), (4,4), (6,4), (2,6), (4,6) y (6,6)
```

```
matriz(:,1)=matriz(permutacion,1) ! permutacion de los elementos
                                  ! de la primera columna
matriz(indice,5)=matriz(1:5:2,6) ! Se asigna a matriz(2,5) el
                                 ! valor de matriz(1,6),
                                 ! a  matriz(4,5) el valor matriz(3,6)
                                 ! y a matriz(6,5) el valor matriz(5,6)
matriz(2,indice)=(/1.,-1.,0./)
matriz((/4,6,1/),(/2,4/)=0.
```

V.4. Instrucciones y operaciones exclusivas a Tableros

Hasta aquí, las asignaciones, operaciones e instrucciones descritas han involucrado los elementos de los tableros de manera individual, uno por uno. En esta sección se abordará las instrucciones y operaciones sobre tableros, que toman en cuenta la estructura de los tableros.

V.4.1. Instrucciones de Control

`fortran` permite que la asignación de valores en un tablero, se realice solamente en aquellos elementos del tablero, que cumplen una determinada condición. La instrucción `where` permite esta acción. La sintaxis de la instrucción es la siguiente:

```
where(<tablero control>) <tablero>=<expresion>
```

donde <`tablero control`> es un tablero, cuyos elementos son de tipo `logical`, <`tablero`> es un tablero de la misma forma que el tablero de tipo `logical`. El efecto de la instrucción `where` se aplica únicamente a aquellos elementos del <`tablero`> correspondientes a los elementos cuyo valor es `.true.` del tablero de control.

Veamos la acción de `where` con el siguiente ejemplo: Sea `A` un tablero de 2×2, dado por:

$$\mathtt{A} = \begin{pmatrix} 100. & 10. \\ 1. & 0. \end{pmatrix}$$

luego, como código:

```
:
real,dimension(2,2)::A,B
:
where(A>0) B=log10(A)
:
```

dará como resultado

$$\mathtt{B} = \begin{pmatrix} 2. & 1. \\ 0. & 0. \end{pmatrix}$$

También es posible, realizar asignaciones de valores, en aquellos elementos correspondientes a los elementos, cuyo valor es `.false.`, del tablero de control, utilizando el par de instrucciones `where` y `else where`. La sintaxis es:

```
where(<tablero control>)
     <bloque de instrucciones>
else where
     <bloque de instrucciones>
end where
```

Así, continuando con el ejemplo precedente, se puede realizar las siguientes instrucciones:

```
:
real,dimension(2,2)::A,B
:
B=A
:
where(A>0)
    B=log10(A)
else where
    B=-100.
end where
:
```

El resultado será:

$$\texttt{B} = \begin{pmatrix} 2. & 1. \\ 0. & -100. \end{pmatrix}$$

Es importante no olvidar, que el tablero de control de la instrucción `where` y todos los tableros involucrados tengan la misma forma.

Así mismo existen otras instrucciones de control, que serán útiles en el siguiente capítulo. Las describimos a continuación. Consideramos un tablero de control `Tcontrol`, cuyos valores son de tipo `logical`. Las siguientes funciones (de control) dan, como resultado:

```
all(Tcontrol) ! valor .true. si todos todos elementos de Tcontrol
              ! tienen valor .true.. Sino el valor es .false.
              !
any(Tcontrol) ! valor .true. si al menos un elemento de Tcontrol
              ! tiene valor .true. . Caso contrario el valor es
              ! .false.
count(Tcontrol)! valor integer indicando el numero de elementos de
               ! Tcontrol, cuyos valores son .true.
```

Supongamos que el número de índices de `Tcontrol` sea `m`; es decir, `Tcontrol` sea de la forma:

```
logical,dimension(ni1:ns1,ni2:ns2,...,nim:nsm)::Tcontrol
```

Las instrucciones o funciones de control: `all`, `any`, `count`, agregando la opción `dim`, a través de las instrucciones:

```
all(Tcontrol,dim=k)
any(Tcontrol,dim=k)
count(Tcontrol,dim=k)
```

dan como resultados tableros unidimensionales (vectores) de talla `nsk+1-nik`, cuyos elementos corresponden a la instrucciones

```
all(Tcontrol(:,:,..,j,:,...,:))
any(Tcontrol(:,:,..,j,:,...,:))
count(Tcontrol(:,:,..,j,:,...,:))
```

Como ejemplo ilustrativo, consideremos

$$\texttt{Tcontrol} = \begin{pmatrix} \texttt{.true.} & \texttt{.false.} \\ \texttt{.false.} & \texttt{.false.} \\ \texttt{.true.} & \texttt{.true.} \end{pmatrix}.$$

Así:

```
all(Tcontrol,dim=1) ! da (/.false.,.false.,.true./)
all(Tcontrol,dim=2) ! da (/.false.,.false./)
any(Tcontrol,dim=1) ! da (/.true.,.false.,.true./)
any(Tcontrol,dim=2) ! da (/.true.,.true./)
count(Tcontrol,dim=1) ! da (/1,0,2/)
count(Tcontrol,dim=2) ! da (/2,1/)
```

V.4.2. Funciones Intrínsicas

Aparte de la función o instrucción `reshape`, `Fortran`, en sus versiones `90` y posteriores, cuenta con funciones destinadas a tableros, de las cuales se describirán, aquéllas que son de mayor uso.

Para ilustrar estas funciones, consideramos el tablero

$$\mathtt{A} = \begin{pmatrix} 5. & 3. & 1. \\ 8. & 12. & 10. \\ 9. & 11. & 7. \\ 4. & 6. & 2. \end{pmatrix}$$

cuya declaración es:

```
real,dimension(0:3,2:4)::A
```

lbound

Esta función da como resultado un vector de tipo `integer`, cuyos valores son las cotas inferiores que pueden tomar los índices del tablero. Así, por ejemplo:

```
lbound(A) ! da para nuestra matriz A: (/0,2/)
```

Agregando la opción `dim`, se obtiene la cota inferior (valor `integer`) que puede tomar el índice, indicada por la opción `dim`; de esta manera:

```
lbound(A,dim=1) ! da: 0
lbound(A,dim=2) ! da: 2
```

ubound

Esta función da como resultado un vector de tipo `integer`, cuyos valores son las cotas superiores que pueden tomar los índices del tablero. Así, por ejemplo:

```
ubound(A) ! da para nuestra matriz A: (/3,4/)
```

Agregando la opción `dim`, se obtiene la cota superior (valor `integer`) que puede tomar el índice, indicada por la opción `dim`; de esta manera:

```
ubound(A,dim=1) ! da: 3
ubound(A,dim=2) ! da: 4
```

size

Esta función proporciona el número de elementos de un tablero, (valor `integer`). Para nuestra matriz de demostración `A`, se tiene:

```
size(A)! da: 12
```

Agregando la opción `dim`, se tiene la longitud del rango del índice indicado por `dim`. Así:

```
size(A,dim=1) ! da: 4
size(A,dim=2) ! da: 3
```

shape

Proporciona la forma de un tablero, en un vector de tipo `integer`, cuya talla es el número de índices del tablero y donde cada elemento del vector proporciona la longitud del rango de valores del respectivo índice. Siguiendo con el ejemplo, se obtiene

```
shape(A) ! da: (/4,3/)
```

minval, maxval

Funciones aplicables a tableros con valores de tipo numérico. `minval` y `maxval` proporcionan el valor mínimo y el valor máximo, respectivamente, entre todos los valores asignados a los elementos del tablero. De esta manera, en la matriz de ilustración:

```
minval(A) ! da: 12.
maxval(A) ! da: 1.
```

Utilizando la opción `dim`, estas funciones proporcionan un vector cuyos valores son los valores máximo y mínimo, que tienen los elementos del tablero, fijando el valor del índice proporcionado por `dim`. En nuestro ejemplo de ilustración:

```
minval(A,dim=1) ! da: (/1.,8.,7.,2./)
maxval(A,dim=1) ! da: (/5.,12.,11.,6./)
minval(A,dim=2) ! da: (/4.,3.,1./)
maxval(A,dim=2) ! da: (/9.,12.,10./)
```

minloc, maxloc

Funciones aplicables a tableros con valores de tipo numérico. `minloc` y `maxloc` proporcionan los índices del valor mínimo y el valor máximo, respectivamente, entre todos los valores asignados a los elementos del tablero. De esta manera, en la matriz de ilustración:

```
minloc(A) ! da: (/1,3/)
maxloc(A) ! da: (/2,2/)
```

Remarcamos que los resultados de `minloc` y `maxloc` son vectores de tipo `integer`, de talla la dimensión del tablero `A`.

sum, product

Funciones aplicables a tableros con valores de tipo numérico. `product` proporciona el producto de todos los elementos del tablero y `sum` la suma de todos los elementos del tablero. En nuestra matriz ejemplo, se obtiene:

```
product(A)  ! da  4.7900160E+08
sum(A)   ! da 78.00000
```

Estas funciones admiten la opción `dim`, cuyos resultados son tableros de tipo numérico y restringen las operaciones de suma y multiplicación al índice fijado por `dim`. Así:

```
product(A,dim=1) ! da (/1440.000,2376.000,140.0000/)
product(A,dim=2) ! da (/15.00000,960.0000,693.0000,48.00000/)
sum(A,dim=1) ! da  (/26.00000,32.00000,20.00000/)
sum(A,dim=2) ! da (/9.000000,30.00000,27.00000,12.00000/)
```

V.5. Operaciones matriciales

`Fortran`, en sus versiones `90` y posteriores permite realizar operaciones matriciales con matrices (tableros bidimensionales) y vectores (tableros unidimensionales). A continuación detallamos estas operaciones matriciales.

La adición, sustracción, multiplicación por escalar de matrices y vectores, se las trata como tableros, lo que ha sido visto más arriba. La multiplicación de matrices, matrices con vector (columna), se la efectua utilizando la función intrínsica `matmul`, cuyas sintaxis son:

```
matmul(<matriz>,<matriz>) ! da una matriz
matmul(<matriz>,<vector>) ! da un vector
```

Está demás decir, que las tallas de las matrices y vectores involucrados deben ser compatibles con la definición de la multiplicación de matrices y vectores.
Ejemplo.- Consideremos las matrices:

$$A = \begin{pmatrix} 1. & 2. & 3. \\ 4. & 5. & 6. \end{pmatrix} \qquad B = \begin{pmatrix} 1. & 2. \\ 3. & 4. \\ 5. & 6. \end{pmatrix}$$

y los vectores:

$$u = \begin{pmatrix} 1. \\ 2. \\ 3. \end{pmatrix} \quad v = \begin{pmatrix} 1. \\ 2. \end{pmatrix}$$

Ahora, ilustremos `mathmul` con líneas de código `Fortran`.

```
program demo8
  real,dimension(3)::u=(/1.,2.,3./),x
  real,dimension(2)::v=(/1.,2./),y
  real,dimension(3,2)::A=reshape((/(1.*i,i=1,6)/),(/3,2/))
  real,dimension(2,3)::B=reshape((/(1.*i,i=1,6)/),(/2,3/))
  real,dimension(3,3)::C
  real,dimension(2,2)::D
  C=matmul(A,B)
  D=matmul(B,A)
  y=matmul(B,u)
  x=matmul(A,v)
  :
```

da como resultados:

$$C = \begin{pmatrix} 9{,}000000 & 19{,}00000 & 29{,}00000 \\ 12{,}00000 & 26{,}00000 & 40{,}00000 \\ 15{,}00000 & 33{,}00000 & 51{,}00000 \end{pmatrix} \qquad D = \begin{pmatrix} 22{,}00000 & 49{,}00000 \\ 28{,}00000 & 64{,}00000 \end{pmatrix}$$

y los vectores:

$$x = \begin{pmatrix} 9{,}000000 \\ 12{,}00000 \\ 15{,}00000 \end{pmatrix} \qquad y = \begin{pmatrix} 22{,}00000 \\ 28{,}00000 \end{pmatrix}.$$

El producto escalar o producto punto de dos vectores de la misma talla, se la realiza aplicando la instrucción o función `dot_product`, cuya sintaxis es:

```
dot_product(<vector>,<vector>)
```

Ilustramos su uso con las siguientes instrucciones:

```
:
real::valor
real,dimension(8)::vector=(/(1.*i,i=1,8)/)
:
valor=dot_product(vector(1:7:2),vector(2:8:2)) ! da valor=100
:
```

También es posible transponer una matriz, mediante la instrucción o función `transpose`, cuya sintaxis es:

```
transpose(<matriz>)
```

Dejamos al lector, construir su ejemplo.

V.6. Lectura y Escritura de Tableros

Como se vio más arriba, la asignación de valores a tableros, ya sea de forma global o por secciones, necesita la utilización de los delimitadores de tablero (unidimensional) "(/" y "/)", separar los valores del tablero mediante comas ",". En el caso de la lectura y escritura de tableros, por medio de archivos, la pantalla y el teclado, no se utiliza los delimitadores "(/" y "/)"; este hecho constituye una diferencia que hay que tomarla siempre en cuenta.

Aparte de la lectura y escritura de manera individual, de cada uno de los elementos de un tablero, se puede efectuar estas operaciones globalmente y por secciones y para tal efecto, `Fortran` interpreta los tableros (multidimensionales), como tableros unidimensionales, de acuerdo a la regla de conversión, explicada al inicio de este capítulo; consiguientemente, `Fortran` escribirá una lista de valores y leerá una lista de valores, cuyos números de valores están definidos por la talla del tablero o la sección del tablero.

Comencemos a ilustrar el uso de las instrucciones de lectura y escritura, con el formato automático "*". Así por ejemplo, el siguiente programa de demostración:

```
program demo9
integer,dimension(3,2)::A
write(*,*)'Introduzca los valores de la matriz A de talla 3x2'
read(*,*)A
write(*,*)'La matriz que usted ha introducido es:'
write(*,*)'Primera Fila',A(1,:)
write(*,*)'Segunda Fila',A(2,:)
write(*,*)'Tercera Fila',A(3,:)
end program demo9
```

Al ejecutarlo, se tiene:

```
[hans@gauss ejemplos]$ demo9
 Introduzca los valores de la matriz A de talla 3x2
1,2,3,4 5 6
 La matriz que usted ha introducido es:
 Primera Fila           1           4
 Segunda Fila           2           5
 Tercera Fila           3           6
[hans@gauss ejemplos]$
```

Tal como puede observarse, los valores a asignarse al tablero `A` pueden estar separados por espacios, o bien por comas.

Para la lectura y escritura con formato preestablecido, la instrucción de asignación de campos de escritura y lectura debe estar en concordancia con la lista de elementos del tablero o la sección del tablero.

V.7. Ejercicios

1. Completar todos los códigos de ejemplos del capítulo en programas.

2. Utilizando un tablero de tipo `character`, escribir un programa que invierta el orden de caracteres de una palabra. Por ejemplo, para "`Cochabamba`", el programa debe escribir "`abmabahcoC`".

Capítulo VI

Estructuras e Instrucciones de Control

Hasta ahora, somos capaces de escribir programas en `Fortran`, como una colección de instrucciones que siguen un orden secuencial de principio a fin, en las que se ejecutan todas las instrucciones una sola vez.

Ahora bien, a menudo se requiere que ciertas instrucciones sean ejecutadas, si se cumple alguna condición y otras veces se requiere repetir un grupo de instrucciones, sin que sea necesario repetir las instrucciones en el código fuente del programa. `Fortran` permite crear estructuras de control para este propósito y en este capítulo desarrollaremos aquellas que son más útiles, dejando las otras a la curiosidad del lector.

VI.1. Instrucción `if`

La instrucción `if` permite someter la ejecución de una o varias instrucciones a una condición:

$$\text{Si } < \text{condición} > \Rightarrow < \text{instrucciones ejecutables} > \tag{VI.1.1}$$

Si no hay más que una sola instrucción ejecutable, se puede utilizar la forma compacta de la instrucción `if`, como en el ejemplo siguiente

```
if(x>=0.) print*,'x es positivo'
```

Si hay más de una instrucción ejecutable, se debe utilizar un bloque `if`, como:

```
if(delta>=0.)then
   delta=sqrt(delta)
   x1=(-b+delta)/(2.*a); x2=(-b-delta)/(2.*a)
end if
```

Importante, se comienza el bloque después de la instrucción `then` y se concluye el bloque con `end if`.

Remarcamos que las instrucciones del bloque `if`, se realizan únicamente si se cumple la condición de la instrucción `if` y luego se continúa con las instrucciones que siguen al bloque; sin embargo, muchas veces se requiere que el programa realize instrucciones alternativas, en caso que no se cumpla la condición de la instrucción `if`. Para este tipo de situaciones, se utiliza la instrucción `else`. Ilustramos su uso, continuando con el código del ejemplo precedente.

```
if(delta>=0.)then
   delta=sqrt(delta)
   x1=(-b+delta)/(2.*a); x2=(-b-delta)/(2.*a)
else
  write(*,*)'El discriminante es negativo'
  stop
end if
```

Observamos que la introducción de la instrucción `else` permite incorporar dos bloques de instrucciones ejecutables: una que se realiza cuando se cumple la condición y la otra cuando no se cumple. Pero también es posible crear varias alternativas de instrucciones, al bloque de instrucciones cuando la condición de la instrucción `if` es cierta, a través de la instrucción `else if`. Así por ejemplo:

```
if(delta>0.)then
   delta=sqrt(delta)
   x1=(-b+delta)/(2.*a); x2=(-b-delta)/(2.*a)
else if(delta==0.) then
   x1=-b/(2.*a); x2=x1
else
   write(*,*)'El discriminante es negativo'
   stop
end if
```

Al igual que la instrucción `if`, si hay una sola instrucción para la condición de `else if`, se puede utilizar la forma compacta.

El bloque correspondiente a la primera condición cierta se ejecuta. Si ninguna condición se cumple, se ejecuta el bloque de instrucciones que sigue la instrucción `else`.

Se puede utilizar las instrucciones `else if`, tanto como se quiera. Además los bloques `if` pueden estar imbricados. Por ejemplo:

```
if(delta>=0.)then
   if(delta>0)then
      delta=sqrt(delta)
      x1=(-b+delta)/(2.*a); x2=(-b-delta)/(2.*a)
    else
      x1=-b/(2.*a); x2=x1
    end if
else
   write(*,*)'El discriminante es negativo'
   stop
end if
```

Para efectos de mayor comprensión en el código fuente, es también posible asociar un nombre a la instrucción `if`. A manera de ilustración:

```
discriminante: if(delta>=0.)then
   delta=sqrt(delta)
   x1=(-b+delta)/(2.*a); x2=(-b-delta)/(2.*a)
else discriminante
  write(*,*)'El discriminante es negativo'
  stop
end if discriminante
```

Finalmente, las formas compactas `elseif` y `endif` pueden ser utilizadas, en lugar de `else if` y `end if`.

VI.2. El ciclo `do`

La instrucción `do` permite repetir la ejecución de un bloque de instrucciones. Esta instrucción admite algunas variantes de uso, que serán detalladas a continuación.

VI.2.1. `do` "infinito"

"infinito" entre comillas, ya que por principio ninguna instrucción puede repetirse una infinidad de veces. Por lo tanto, comencemos con la situación más absurda:

```
do
  print*,'Debo estudiar'
end do
```

Si el programa, que contiene este bloque `do` de instrucciones, llegase a ejecutarse, jamás se detendría y escribiría de manera indefinida `Debo estudiar`. Veamos las formas elegantes de salir de este ciclo. La primera opción es utilizar la instrucción `stop`, que detiene el programa,. Luego la instrucción `exit` permite salir del bloque de instrucciones `do`, continuando en la siguiente instrucción que sigue el `end do` del ciclo. Además se cuenta con la instrucción `cycle`, que obliga al programa regresar al inicio de las instrucciones del bloque `do`. Conforme avancemos ilustraremos su uso.

También es posible asignar un nombre a un ciclo `do`, para fines de mayor comprensión y control del ciclo:

```
absurdo: do
  print*,'Debo estudiar'
end do absurdo
```

Por lo tanto, la estructura completa de un ciclo `do` "infinito", tiene la siguiente forma:

```
[<nombre>:] do
    :
    :
   if(<condición>) cycle
    :
   if(<condición>) exit
end do [<nombre>]
```

Se puede imbricar ciclos `do`. Las instrucciones `cycle` y `exit` se asocian al ciclo más interno en el cual se encuentran estas instrucciones; salvo si, éstas están precedidas por el nombre de un ciclo, en este caso están asociadas al ciclo nombrado. La estructura de dos ciclos imbricados, con instrucciones `cycle exit` es de la forma:

```
ciclo_externo: do
  ! se llega aquí con cycle ciclo_externo
  :
  ciclo_interno: do
     ! se llega aquí con cycle
     :
     if(<condición>) cycle
     if(<condición>) exit
     if(<condición>) cycle ciclo_externo
     if(<condición>) exit ciclo_externo
     :
  end do ciclo_interno
  ! se llega aquí con exit
  :
end do ciclo_externo
! se llega aquí con exit ciclo_externo
:
```

VI.2.2. `do while`

Una manera de limitar el número de iteraciones de un bloque de instrucciones `do`, consiste en utilizar la variante `do while`, cuya estructura es:

```
[<nombre>:] do while(<condición>)
  :
end do [<nombre.]
```

El ciclo continúa realizando mientras la condición sea cierta; eso significa, que para que se ejecute al menos una vez, la condición tiene que ser cierta en el primer ciclo. En esta variante, está permitido utilizar las instrucciones `cycle` y `exit`; como también es posible imbricar otros ciclos `do`.

VI.2.3. `do` con un número fijo de iteraciones

Si se conoce de antemano, el número de iteraciones que se debe realizar, se puede utilizar el ciclo `do`, bajo la siguiente sintaxis:

```
[<nombre>:] do <variable integer>=<inicio>,<final>[,<incremento>]
    :
end do [<nombre>]
```

La <`variable integer`> puede ser utilizada dentro el ciclo, pero de ninguna manera debe ser modificada. Esta variable vale <`inicio`> en la primera iteración, para cada iteración es incrementada de <`incremento`>. El programa sale del ciclo, una vez que el valor <`final`> ha sido depasado Si <`incremento`> no se especifica, vale `1`.

Una condición necesaria para que el ciclo `do`, es que mediante aplicaciones sucesivas del <`incremento`>, la <`variable integer`> pueda alcanzar o depasar <`final`>. Así,por ejemplo:

```
do i=10,10,-1 ! se ejecuta una vez
 :
end do
:
do i=5,1 ! no se ejecuta
 :
end
:
do i=1,3,4 ! se ejecuta una sola vez
 :
end do
:
do i=1,5,-1 ! no se ejecuta
  :
end do
:
```

Finalmente, como corolario, se tiene el `do` implícito, que dicho sea de paso, se lo ha visto en la construcción de tableros; es frecuente utilizarlo en la escritura y lectura de tableros; por ejemplo:

```
:
read*,(a(i,j),j=1,n) ! lee a(i,1),a(i,2),....,a(i,n)
:
write(b(j),j=10,1,-1) ! escribe b(10), b(8), b(6), b(4), b(2)
:
```

VI.3. La estructura `select case`

La estructura `select case` permite elegir un bloque de instrucciones, entre varios bloques, en función del valor de una expresión. La estructura sintáxtica de `select case` es la siguiente:

```
[<nombre>:] select case (<expresión>)
      case (<opción 1>) [<nombre>]
         :
      case (<opción 2>) [<nombre>]
         :
```

```
        :
     case default [<nombre>]
end select [<nombre>]
```

donde <expresión> puede ser una expresión de tipo `character`, `logical` o `integer`; el <nombre> es opcional; <opción k> es un valor, rango de valores (especificados por :) o valores y rangos (separados por comas).

Ejemplos.- Mostramos el

```
character(len=1) :: inicial
:
cronograma: select case(inicial)
  case(:'D') cronograma
    print*,'Lunes'
  case('E':'L') cronograma
    print*,'Martes'
  case('M','N') cronograma
    print*,'Miércoles'
  case('O':'R') cronograma
    print*,'Jueves'
  case('S':) cronograma
    print*,'Viernes'
  case default cronograma
    print*,'Nunca'
end select cronograma
```

Importante.- Hay que evitar cualquier superposición en las alternativas, para la instrucción `case`.

Al igual que las otras estructuras de control, es posible imbricar las estructuras `select case`.

VI.4. La instrucción `go to`

La instrucción `go to` o en forma compacta `goto` es una instrucción, cuyo uso debe evitarse en la medida de lo posible; sin embargo, existen situaciones cuyo uso puede ser de extrema utilidad. Esta instrucción redirige al programa hacia una instrucción etiquetada del código, saltando las instrucciones que se encuentran entre el `go to` y la instrucción etiquetada. La estrura de esta instrucción es la siguiente:

```
  :
 go to <etiqueta>
  :
<etiqueta> .. ! se llega a este lugar
  :
```

donde <etiqueta> es un valor númerico `integer` de 1 a 99999.

VI.5. Ejercicios

1. Completar los ejemplos del capítulo en programas.
2. Utilizando una estructura `do`, realizar la instrucción `a(1:n)=a(2:n+1)`.

Capítulo VII

Procedimientos

Tal como se explicó en el capítulo I, sección I.1.3 los programas escritos en `Fortran` están estructurados en unidades programáticas, en base a niveles jerárquicos. Existen cuatro tipos de unidades programaticas:

- `program` que es la unidad programática principal, sobre la cual hemos ido trabajando en el transcurso de los anteriores capítulos.
- `subroutine` (subrutina) unidad programática que contempla instrucciones ejecutables.
- `function` (función) unidad programática que contempla instrucciones ejecutables
- `module` unidad programática que contempla instrucciones de declaración de variables, inicialización de variables, interfaces entre funciones y subrutinas.

El programa principal, las subrutinas y funciones, son llamadas procedimientos. Un procedimiento es interno si está definida dentro de una unidad programática, la cual puede ser una unidad "hospedante" si se trata de un procedimiento que llama al procedimiento interno, o bien un módulo. Un procedimiento puede ser externo, si no está contenida en ninguna otra unidad programática.

VII.1. Subrutinas

Una subrutina es subprograma de orden jerárquico de nivel inferior al subprograma principal, que tiene como objetivo llevar a cabo instrucciones ejecutables, que serán ejecutadas por un subprograma de nivel jerárquico superior, a través de una instrucción que llame a la subrutina. La estructura sintáxica de una subrutina es:

```
subroutine <nombre>[(<argumentos (ficticios)>)]
 ! instrucciones de declaración de los argumentos (ficticios)
 :
 ! instrucciones de declaración de los objetos locales
 :
 ! instrucciones ejecutables
 :
 end subroutine <nombre>
```

donde los <argumentos (ficticios)>, en el caso que sean utilizados, son los objetos, sobre los cuales la subrutina trabajará preferentemente, están separados por comas y pueden ser variables, funciones, subrutinas, tableros, punteros, procedimientos de módulo. Los objetos locales son de uso exclusivo de la subrutina y no son utilizados por subprogramas de nivel jerárquico superior.

La subrutina es llamada por un subprograma de nivel jerárquico superior, por medio de la instrucción `call`, de la siguiente manera

```
<identificación unidad programática>
 :
 call <nombre>[<argumentos (usados)>]
```

```
  :
end <unidad programática>
```

donde la lista de argumentos (de uso) deben coíncidir con la lista de argumentos (ficticios), en posición, tipo y clase, de acuerdo al ejemplo de la figura VII.1.1.

```
program demo10
  integer,parameter::dp=8
  integer::n
  real(kind=dp),dimension(0:100)::a(0:100)
  real(kind=dp)::x,p,q
  write(*,*)'Introduzca un polinomio de grado menor o igual a 100'
  write(*,*)'Intrudzca el grado del polinomio'
  read(*,*)n
  write(*,*)'Introduzca los coeficientes del polinomio'
  write(*,*)'con grado decreciente'
  read(*,*)a(n:0:-1)
  write(*,*)'Introduzca el punto x donde desea ser evaluado'
  read(*,*)x
  call horner(n,a,x,p,q)
  write(*,*)'El valor del polinomio es p(x)=',p
  write(*,*)"El valor de la derivada es p'(x)=",q
end program demo10
```

```
subroutine horner(n,a,x,p,q)
  integer,parameter::dp=8
  !inputs
  ! n (entero) grado del polinomio a evaluar
  ! a (tablero doble precision) a(0:n),
  !    coeficientes del polinomio
  ! x valor doble precision, donde se desea
  !          evaluar el polinomio
  !outputs
  ! p evaluacion del polinomio en x
  ! q evaluacion del polinomio derivada en x
  !!!!!!!!!!!!!!!
  ! declaraciones !!!!!!!!!!!!!!
  integer::n,i
  real(kind=dp)::x,p,q
  real(kind=dp),dimension(0:n)::a
  p=a(n); q=0.d0
  do i=n-1,0,-1
     q=q*x+p; p=p*x+a(i)
  end do
end subroutine horner
```

Figura VII.1.1: Uso de subrutinas

Remarcamos que los argumentos (de uso) al llamar una subrutina, no necesariamente deben tener los mismos nombres, lo único que se exige es que coíncidan en posición, tipo y clase; por ejemplo, donde corresponde una variable entera, debe ir una expresión entera. Esto se explica porque `fortran` maneja los objetos por **referencia** y no por **valor**, como lo hacen otros lenguajes de programación.

El manejo de objetos por **referencia** significa que el programa, cuando se ejecuta, trabaja únicamente con el argumento (de uso); así por ejemplo, si `i` es la variable de uso y `j` es la variable ficticia de una subrutina, en cuyo código fuente, se tiene la instrucción

$$\texttt{j} = \texttt{j} + 1, \tag{VII.1.1}$$

lo que el programa al ejecutarse, hace realmente es

$$\texttt{i} = \texttt{i} + 1. \tag{VII.1.2}$$

VII.1.1. Declaración de los Argumentos Ficticios

En la estructura de una subrutina, existe un bloque de instrucciones que sirven para declarar los argumentos (ficticios) de una subrutina, (*c.f.* estructura sintáxica de una subrutina). Aúnque dependiendo de la subrutina, la declaración detallada de los argumentos no es imprescindible, el hacerlo constituye una buena práctica, que al final da buenos resultados en la comprensión del programa, la detección de posibles errores. Por consiguiente, el consejo, en la medida de lo posible, es declarar los argumentos (ficticios) de la manera más detalla y precisa posible.

Los argumentos (ficticios) que son variables y tableros deben ser declarados con el atributo `intent`, cuyos valores son:

- `intent(in)`: variables y tableros cuyos valores a la entrada de la subrutina son utilizados por la subrutina. Estos valores son proporcionados por el subprograma de nivel jerárquico superior que llama a la subrutina. Los valores no pueden ser modificados por la subrutina.

- `intent(out)`: variables y tableros, cuyos valores a la entrada de la subrutina no son tomados en cuenta, pero que son modificados por la subrutina y asimilados a la salida por el subprograma que llama a la subrutina.

- `intent(inout)`: variables y tableros, cuyos valores a la entrada de la subrutina son tomados en cuenta y que pueden ser modificados por la subrutina y asimilados por la unidad prográmatica que llama a la subrutina.

Una visualización del atributo `intent` se da en el diagrama de la Figura VII.1.2
Importante.- Puede utilizarse valores constantes, variables y tableros con atributo `parameter`, como argumento de uso, siempre y cuando el atributo del argumento ficticio sea `intent(in)`, caso contrario el programa presentará errores en la ejecución.

La declaración de punteros será tratada en el capítulo correspondiente y se debe utilizar el atributo `pointer`. Si uno de los argumentos ficticios corresponde a una subrutina o una función, la declaración se la hace utilizando una interface, lo que será visto más adelante.

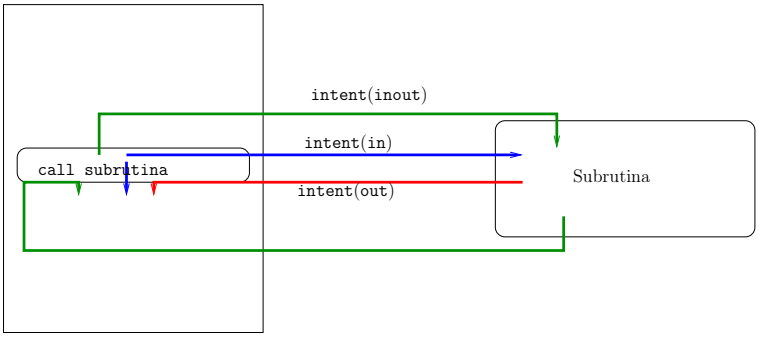


Figura VII.1.2: Diagrama de uso del atributo `intent`

Declaración de Tableros

Continuemos detallando la declaración de tableros, cuando éstos son argumentos de la subrutina. Recalcando nuevamente que `Fortran` trabaja siempre con tableros unidimensionales y eso hay que tenerlo en cuenta, al igual que la forma de conversión de tableros multidimensionales a tableros unidimensionales.

La primera regla que hay seguir, puesto que los tableros serán tratados por referencia, es que la talla del tablero de uso sea mayor o igual que la talla del tablero ficticio. En en el caso que la talla del tablero de uso sea estrictamente mayor al tablero ficticio, la subrutina no manipulará los últimos elementos sobrantes del tablero de uso.

La segunda regla, al momento de programar, es que hay que tener en cuenta, qué elementos del tablero de uso y el tablero ficticio serán manipulados por el programa y cómo estarán relacionados.

Ilustremos estas dos reglas, con una subrutina que cálcula la matriz inversa de una matriz de 2×2. El código de esta subrutina, lo escribimos:

```
subroutine inversa(A,B)
  real(kind=4),intent(in),dimension(2,2)::A
  real(kind=4),intent(out),dimension(2,2)::B
  real(kind=4)::det
  det=a(1,1)*a(2,2)-a(2,1)*a(1,2)
  if(det==0)then
     write(*,*)'La matriz no es inversible'
     stop
  end if
  b(1,1)=a(2,2); b(2,2)=a(1,1); b(2,1)=a(2,1); b(1,2)=a(1,2)
```

```
   b=b/det
 end subroutine inversa
```

luego, algunas instrucciones de la unidad programática que llama a la subrutina.

```
real(kind=4),dimension(10,10)::C,D
:
c(1,1:2)=(/2.,1./); c(2,2:2)=(/1.,1./)
call inversa(C,D)
write(*,*)d(1,1:2)
write(*,*)d(2,1:2)
:
```

Ahora bien, cuando el programa se ejecute, la subrutina utilizará los elementos `c(1:4,1)` del tablero `C` y no los elementos `c(1:2,1:2)`; lo mismo sucederá con el tablero `D`. Por lo tanto, si el programa no se detiene antes, el resultado que será impreso no corresponderá a la inversa de la matriz que estamos buscando. Para evitarse el problema descrito, lo mejor es llamar a la subrutina, introduciendo los tableros `C` y `D` por bloques o secciones, como sigue:

```
call inversa(C(1:2,1:2),D(1:2,1:2)
```

El ejemplo descrito muestra una de las formas de declarar la talla y forma de los tableros: la forma y talla son fijas y no se alteran en utilizaciones posteriores de la subrutina. Esta manera simple de declarar tiene una desventaja, cuando se requiere trabajar con tableros, cuya forma y talla varía de acuerdo a los problemas que debe resolver un programa, por ejemplo hallar la matriz inversa de una matriz de `n` × `n`, donde `n` no se conoce a priori.

Veamos, por lo tanto, otras formas de declarar tableros, sin tener que prefijar las tallas y formas.

Se puede declarar la forma y talla de un tablero ficticio de manera explícita, utilizando otros argumentos (ficticios) de tipo `integer`, como ilustra el siguiente código:

```
subroutine subrutina(n,m,A,b)
  integer::n,m
  real(kind=8),intent(in),dimension(n,0:m)::A
  real(kind=8),intent(inout)::b(0:m)
  :
end subroutine subrutina
```

Nuevamente, está demás recordar que siempre hay que cuidar la correspondencia entre el tablero de uso y el ficticio para evitarse sorpresas desagradables y una forma de hacerlo es transmitir el tablero de uso por la sección que nos interesa.

También es posible, declarar la forma y talla de un tablero ficticio de manera implícita, utilizando "`:`" en la declaración del tablero, como muestra el código de:

```
subroutine prodescalar_matriz(A,B,t)
  real,dimension(:,:),intent(in)::A,B
  real,intent(out)::t
  t=sum(A*B)
end subroutine prodescalar_matriz
```

Pero **ojo**, esta subrutina tiene que ser declarada antes en la unidad programática que la utilice, mediante una instrucción `interface`, sino no funcionará. Veamos con un ejemplo:

```
program demo11
  real::A_punto_B
  real,dimension(10,10)::A,B
  interface
     subroutine prodescalar_matriz(A,B,t)
       real,dimension(:,:),intent(in)::A,B
```

```
      real,intent(out)::t
    end subroutine prodescalar_matriz
  end interface
  A=1.; B=1.
  call prodescalar_matriz(A(1:3,1:3),B(1:5:2,1:3),A_punto_B)
  write(*,*)'El producto escalar de A y B es',A_punto_B
end program demo11
```

La ejecución del programa, da:

```
[hans@gauss ejemplos]$ demo11
 El producto escalar de A y B es   9.000000
```

En las versiones 77 y anteriores de FORTRAN, la forma de declarar un tablero de manera implícita es dejando libre la última dimensión, colocando un "*", como muestra la siguiente declaración:

```
real(kind=8),dimension(*)::vector
```

y que las versiones 90 y posteriores de Fortran la aceptan como valida. Sin embargo, esta manera de declarar impide que el programa recupere la forma y talla del tablero; inclusive declarando la subrutina con una interface. Lo que se hace es introducir en la subrutina la talla y forma a través de argumentos de tipo integer, como en la forma explícita. Consiguientemente, esta forma de declarar tableros, no debiera utilizarse.

Cadenas de caracteres, como argumentos

La declaración de una cadena de caracteres, como argumento ficticio, puede hacerse de muchas formas. Al igual que un tablero, se puede fijar, la longitud de la cadena:

```
subroutine subrutina(ciudad)
   character(len=10)::ciudad
   :
end subroutine subrutina
```

De manera explícita, por medio de un argumento de tipo integer:

```
subroutine subrutina(n,ciudad)
   integer,intent(in)::n
   character(len=n),intent(inout)::ciudad
   :
end subroutine subrutina
```

O bien de manera implícita, utilizando el símbolo "*", como especificador automático de la longitud de la cadena:

```
subroutine subrutina(ciudad)
   character(len=*)intent(inout))::ciudad
   :
   n=len(ciudad) ! dara la longitud de la cadena del argumento de uso.
   :
end subroutine subrutina
```

Procedimientos como argumentos

Fortran permite utilizar como argumentos subrutinas y funciones. Para tal efecto, se utiliza la instrucción bloque interface , cuya estructura sintáxica es la siguiente:

```
interface
  subroutine subrutina1(<argumentos ficticios>)
  : declaración argumentos ficticios
```

```
  end subroutine subrutina1
   :
  subroutine subrutinak(<argumentos ficticios>)
  : declaración argumentos ficticios
  end subroutine subrutinak
  function fcn1(<argumentos ficticios>)
   : declaración argumentos ficticios y función
  end function fcn1
   :
  function fcn1(<argumentos ficticios>)
   : declaración argumentos ficticios y función
  end function fcn1
end interface
```

Se hace la declaración obligatoria de las cabeceras[1] de cada una de las subrutinas y funciones que son argumentos ficticios de la subrutina y de manera optativa, de todas aquellas subrutinas externas que utilizará la subrutina, como en el ejemplo de declaración implícita de la forma y talla de un tablero.
Ejemplo.- Una subrutina que aproxima la derivada de una función en un punto dado.

```
subroutine derivada(f,a,h,dev)
  real(kind=8),intent(in)::a,h
  real(kind=8),intent(out)::dev
  interface
     real(kind=8) function f(x)
       real(kind=8),intent(in)::x
     end function f
  end interface
  dev=.5d0*(f(a+h)-f(a-h))/h
end subroutine derivada
```

Luego, el código de un programa (principal) que utilice esta subrutina:

```
program demo12
  real(kind=8)::a=1.,h=1.d-10,dev
  call derivada(fcn,a,h,dev)
  write(*,*)'La derivada vale',dev
contains
  real(kind=8) function fcn(x)
    real(kind=8)::x
    fcn=x*x+x+1.d0
  end function fcn
end program demo12
```

cuya acción es:

```
[hans@gauss ejemplos]$ demo12
 La derivada vale   3.00000024822111
[hans@gauss ejemplos]$
```

En el ejemplo, la función utilizada es interna, tema que será abordado más adelante.

VII.1.2. Declaración de Objetos Locales

Luego de declarar los argumentos ficticios, en la estructura de una subrutina, corresponde las instrucciones de declaración de los objetos locales, de uso exclusivo de la subrutina, que serán utilizados en las instrucciones ejecutables del procedimiento.

[1] La cabecera incluye solamente el nombre de la subrutina, la declaración de los argumentos ficticios y no así la declaración de los objetos locales de la subrutina, ni instrucciones ejecutables

Las variables, los tableros se las declara de la misma manera que para un subprograma principal, utilizando los atributos que correspondan. Una novedad, para las variables y tableros locales es el atributo `save` , que permite conservar los valores de una llamada a otra, por ejemplo:

```
subroutine subrutina(<argumentos>)
  : ! declaración argumentos
  integer,save::i=0
  :
```

En la primera llamada de la subrutina, la variable `i` se inicializará con `i=0` y en las siguiente llamada tomará el último valor.

Para los otros objetos, como procedimientos, en el caso que sean externos, se puede utilizar interfaces para declararlos, sobre todo cuando la declaración de tableros es implícita

Una diferencia con el subprograma principal, es que en las subrutinas, tanto tableros, como cadenas de caracteres, pueden ser declaradas a partir del valor de un argumento (ficticio) de tipo `integer`, asi por ejemplo:

```
subroutine subrutina(n,<otros argumentos>)
  integer,intent(in)::n
  : ! declaración argumentos
  integer,save::i=0
  real(kind=8),dimension(0:n)::A
  character(len=n)::ciudad
  :
```

En este caso, de declarar un tablero o una cadena de caracteres utilizando el valor de un argumento, la situación es como la de un tablero dinámico, cuya existencia solamente se dará mientras la subrutina corra; no les corresponde el atributo `save`.

Asimismo, las tallas y formas de tableros y cadenas de caracteres no pueden especificarse de manera implícita.

VII.1.3. Subrutinas Internas

Un procedimiento, como las subrutinas, puede ser interno, si hace parte de un subprograma de nivel jerárquico superior y por lo tanto su uso será local (de uso exclusivo del subprograma que contiene al procedimiento interno). O bien externo, si no hace parte de la estructura del código de un subprograma de nivel jerárquico superior y por lo tanto utilizable, en principio, por cualquier subprograma de nivel jerárquico superior.

La estructura sintáxica de una subrutina interna es:

```
<identificación unidad programática>
:
call subrutina[(<argumentos de uso>)]
:
contains
  subroutine subrutina[(<argumentos ficticios>)
  :
  end subroutine subrutina
  :
end <unidad programática>
```

La primera regla con las subrutinas internas es que una subrutina interna tiene acceso a todos los objetos (variables, tableros, procedimientos, etc) de la unidad programática que contiene a la subrutina; a menos que sean declaradas como objetos locales o argumentos ficticios. Por ejemplo:

```
program demo12
 integer::i=10
 call subrutina
```

```
 stop
contains
 subroutine subrutina
   print*,i ! da i=10
 end subroutine subrutina
end program demo12
```

Mientras que:

```
program demo12
 integer::i=10
 call subrutina
 print*,i ! da i=10, e i es una variable global.
 stop
contains
 subroutine subrutina
   integer:: i=3
   print*,i ! da i=3, i es una variable local.
 end subroutine subrutina
end program demo12
```

Los objetos que han sido declarados, como objetos locales dentro la subrutina interna, no pueden ser accedidos por la unidad programática que contiene a la subrutina interna.

Un procedimiento, así como sus procedimientos internos son compilados juntos. El compilador puede decidir de remplazar la llamada del procedimiento por las instrucciones contenidas en este.

Una subrutina puede contener procedimientos internos, a condición que no se trate de una subrutina interna. Suficiente utilizar la estructura `contains`.

En el caso que una subrutina interna, sea utilizada como argumento de uso, ésta no debe ser declarada a través de la estructura `interface`.

Ahora veamos las ventajas y desventajas de utilizar una subrutina interna. Las ventajas son:

- No es necesario contruir una `interface`.
- La subrutina interna tiene acceso a todos los objetos del procedimiento hospedante (salvo utilización de nombre idénticos en los objetos locales). Por consiguiente, no es necesario pasarlos por argumentos o vía módulos.

Las desventajas son:

- Existe un riesgo de modificar accidentalmente los valores de las variables del procedimiento hospedante.
- Una subrutina interna solamente puede ser accedida por la unidad prográmatica que la contiene, por lo tanto no puede ser utilizada por otras.

VII.2. Funciones

Una función es un tipo particular de subrutina, a la que se accede directamente sin utilizar la instrucción `call`. Por consiguiente, todos los aspectos abordados para una subrutina son válidos para las funciones. La función como unidad programática puede ser llamada al interior de una expresión y cuyo resultado es utilizado en esta expresión. Así, por ejemplo

```
:
sumacuadrodos=suma(a,b,c)**2
:
```

utiliza la función `suma` con los argumentos de uso `a, b, c`.

La estructura sintáxica de una función es la siguiente:

```
[<tipo>] function <nombre>[(<argumentos ficticios>)] [result(<resultado>)]
 : ! declaración función (nombre o resultado)
 : ! declaración argumentos ficticios
 : ! declaración objetos locales
 : ! instrucciones ejecutables
end function <nombre>
```

El tipo de resultado de una función puede ser declarado en la identificación de la función y por consiguiente, ya no es necesario declarar al interior de la función. Por ejemplo:

```
real(kind=8) function suma(a,b,c)
  real(kind=8),intent(in)::a,b,c
  suma=a+b+c
end function suma
```

Normalmente, como se ve en el ejemplo, se utiliza como argumento ficticio el nombre de la función y éste debe ser utilizado para definir el valor del resultado. Sin embargo, a través de la instrucción `result`, se puede asignar otro nombre al resultado.

```
real(kind=8) function suma(x,y,z) result(resultado)
  real(kind=8),intent(in)::x,y,z
  resultado=x+y+z
end function suma
```

Al predefinir el tipo de resultado en la identificación de la función uno se limita a utilizar los tipos intrínsicos, sin explotar los atributos de éstos; por ejemplo, obtener como resultado un tablero o bien un resultado de un tipo derivado; lo que es posible si se declara al interior de de la función, de manera similar a la de un argumento ficticio. Veamos algunos ejemplos ilustrativos.
Ejemplos.-

```
 function producto_cruz(a,b) result(vector)
   real(kind=8),dimension(3),intent(in)::a,b
   real(kind=8),dimension(3)::vector
   vector(1)=a(2)*b(3)-a(3)*b(2)
   vector(2)=a(3)*b(1)-a(1)*b(3)
   vector(3)=a(1)*b(2)-a(2)*b(1)
 end function producto_cruz
```

Función que calcula el producto vectorial de dos vectores $\vec{a}$ y $\vec{b}$. Utilizando en el programa de demostración siguiente:

```
program demo14
 real(kind=8),dimension(3)::a=(/1.d0,1.d0,1.d0/),&
                            &b=(/0.d0,1.d0,0.d0/),c
 interface
  function producto_cruz(a,b)
   real(kind=8),dimension(3)::producto_cruz
   real(kind=8),dimension(3),intent(in)::a,b
  end function producto_cruz
 end interface
 c=producto_cruz(a,b)
 print*,c(1:3)
end program demo14
```

da como acción:

```
[hans@gauss ejemplos]$ demo14
  -1.00000000000000       0.000000000000000E+000   1.00000000000000
```

El siguiente ejemplo, una función que utiliza el tipo derivado polinomio

```
type polinomio
  integer::grado
  real(kind=8),dimension(0:100)::coef
end type polinomio
```

para calcular el producto de dos polinomios.

```
program demo15
  type polinomio
     integer::grado
     real(kind=8),dimension(0:100)::coef
  end type polinomio
  type(polinomio)::a,b,c
  a%grado=3; b%grado=2
  a%coef(0:a%grado)=(/(-(-1.d0)**i,i=0,3)/)
  b%coef(0:b%grado)=(/(1.d0,i=0,2)/)
  c=a_por_b(a,b)
  write(*,*)'El grado es',c%grado
  write(*,*)'Los coeficientes en orden ascendente, son:'
  write(*,*)c%coef(0:c%grado)
contains
  function a_por_b(x,y)
    ! no es necesario declarar el tipo, funci'on interna
    type(polinomio)::a_por_b
    type(polinomio),intent(in)::x,y
    integer::i,jx,kx,jy,ky
    a_por_b%grado=x%grado+y%grado
    do i=0,a_por_b%grado
       jx=min(i,x%grado); kx=max(0,i-y%grado)
       jy=min(i,y%grado); ky=max(0,i-x%grado)
       a_por_b%coef(i)=dot_product(x%coef(kx:jx),y%coef(ky:jy:-1))
    end do
  end function a_por_b
end program demo15
```

El ejecutable da:

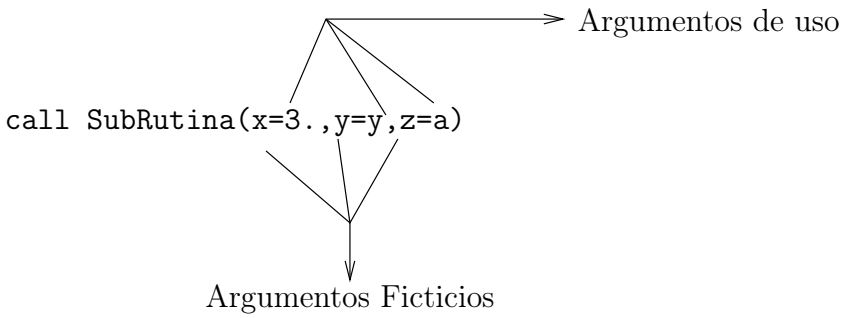
```
[hans@gauss ejemplos]$ demo15
 El grado es           5
 Los coeficientes en orden ascendente, son:
  -1.00000000000000       0.000000000000000E+000  0.000000000000000E+000
  0.000000000000000E+000  0.000000000000000E+000   1.00000000000000
[hans@gauss ejemplos]$
```

VII.3. Misceláneas

Las versiones 90 y posteriores de `Fortran` permiten aprovechar algunas opciones de lenguaje, las cuales serán desarrolladas a continuación.

VII.3.1. Argumentos por nombre

La opción por defecto, cuando se llama a un procedimiento, sea éste una subrutina o sea una función, se lo hace mediante el uso de argumentos posicionales; ésto significa que la lista de argumentos de uso corresponden a la lista de argumentos ficticios en el mismo orden y posición. Sin embargo, existe la posibilidad de recordar la lista de argumentos ficticios cuando se llama al procedimiento, tal como se ilustra:



La ventaja es que se puede modificar el orden de los argumentos, cuando se llama al procedimiento. Esta forma de introducir los argumentos de uso, se la conoce *keyword arguments* (argumentos por nombre).

Se puede mesclar el método de los argumentos posicionales con los argumentos por nombre. La regla es que del momento que un argumento es llamado por su nombre, los siguientes deben serlo igualmente:

```
call SubRutina(3.,y=y,z=a)
```

VII.3.2. Argumentos opcionales

En algunas situaciones es bastante conveniente utilizar argumentos opcionales, de manera que cuando no se utiliza un argumento opcional, el procedimiento realiza una serie de instrucciones por defecto y en el caso que utilise el argumento opcional, las instrucciones se adaptan a la opción.

Los argumentos opcionales son declaradados en el procedimiento (subrutina o función) utilizando el atributo `optional`. y la función intrínsica `present` de tipo `logical` permite detectar si el argumento opcional ha sido utilizado en la llamada del procedimiento. Veamos el uso de `optional` y `present` en el siguiente código.

```
real(kind=8) function raiz(n,x)
  integer,intent(in),optional::n
  real(kind=8),intent(in)::x
  if(present(n))then
     raiz=x**(1.d0/n)
  else
     raiz=sqrt(x)
  end if
end function raiz
```

Aplicado al siguiente ejemplo:

```
program demo17
  real(kind=8)::x
  interface
    real(kind=8) function raiz(n,x)
       integer,intent(in),optional::n
       real(kind=8),intent(in)::x
    end function raiz
  end interface
  print*,'La raiz cuadrada de 9 es:',raiz(x=9.d0)
  print*,'La raiz cubica de 27 es:', raiz(3,x=27.d0)
end program demo17
```

se obtiene:

```
[hans@gauss ejemplos]$ demo17
 La raiz cuadrada de 9 es:   3.00000000000000
 La raiz cubica de 27 es:   3.00000000000000
[hans@gauss ejemplos]$
```

VII.3.3. Recursividad

Un procedimiento recursivo es un procedimiento que puede llamarse a si mismo. Suficiente de declararlo con la palabra clave `recursive` delante de la identificación del procedimiento. Por ejemplo:

```
recursive function factorial(n)
  integer::factorial
  integer,intent(in)
  if(n==0)then
     factorial=1
  else
    factorial=n*factorial(n-1)
  end if
end function factorial
```

La utilización de procedimientos (funciones y subrutinas) recursivas es poco eficiente. Es más conveniente utilizar ciclos.

VII.4. Ejercicios

1. Completar el código de los ejemplos, de manera a obtener programas ejecutables.

Capítulo VIII

Módulos

Los módulos constituyen un tipo particular de unidad programática, cuya utilización se da a partir de las versiones `90` y posteriores de `Fortran`. Un módulo, como unidad programática independiente, puede contener la información o los elementos siguientes:

- Precisión utilizada, si es simple o doble precisión.
- Instrucciones de asignación, para la definición de constantes y variables compartidas.
- Interfaces de procedimientos.
- Procedimientos de módulos.

La estructura sintáxica de un módulo es la siguiente:

```
module <nombre>
  : ! instrucciones de especificación o declaración
 contains
  : ! procedimientos de módulo
end module <nombre>
```

Una unidad programática utiliza la información y los elementos de un módulo predefino, mediante la instrucción `use`, inmediatamente después de la identificación de la unidad:

```
<Identificación unidad programática>
  use <nombre>
  : ! instrucciones de declaración
  : ! instrucciones ejecutables
end <Unidad programática>
```

VIII.1. Programando con Módulos

Hay diferentes maneras de programar y utilizar módulos.

1. Incluir el código fuente del módulo o módulos en el archivo o archivos del código fuente del programa principal y procedimientos externos, justo antes del código de cada uno de los procedimientos.

   ```
   module <nombre módulo>
    :
   end module <nombre módulo>
   :
   <Identificación unidad programática>
    :
   end <Unidad programática>
   ```

Esta forma de incluir módulos se justifica, si su uso será exclusivo para el programa que se está escribiendo, pero no cuando el módulo tiene un uso más generalizado.

2. Editar el código fuente de los módulos en archivos, con extensión `.f90`, guardarlos en el directorio `modulos`, compilarlos con la opción `-c`. Se obtendrá archivos de extensión `.mod` y `.o`. Si el módulo contiene código ejecutable utilizar el archivo de extensión `.o` en la compilación del ejecutable. Luego compilar el programa ejecutable utilizando la opción `-J <dir modulos>` o eventualmente `-I <dir modulos>`

3. Incluir los archivos `.f90` de los módulos, delante de los otros archivos en la instrucción de compilación.

VIII.2. Datos y Objetos Compartidos

Tal como se mencionó más arriba, podemos utilizar un módulo para definir las clases de precisión. Por ejemplo:

```
module precisiones
 integer,parameter::sp=4,dp=8,qp=16,pr=dp
end module precisiones
```

Este módulo de precisiones, lo utilizamos en una unidad prográmatica, como un subprograma principal.

```
program principal
  use precisiones
  real(kind=sp),dimension(500)::xpg,ypg
  real(kind=dp)::x=1.e-7_dp
  :
```

En caso que se requiera cambiar las precisiones, suficiente modificar los valores de `sp`, `dp` y `qp` en el módulo, para cambiar las clases de precisión en el programa entero.

Existen constantes "universales", cuyo uso, es frecuente en la solución de muchos problemas. Se puede crear un módulo o varios módulos, de tal manera que se tenga un repositorio de constantes y no tener que volver a definirlas en las unidades programáticas que uno vaya creando.

```
module constantes
 use precisiones
 real(kind=pr),parameter:: numero_pi=3.141592654_pr,&
     &                     cons_gravedad=9.81,&
     :
end module constantes
```

La primera constación es que un módulo puede utilizar otro módulo; además, el uso de modulos es transitivo; es decir, si un módulo 3 utiliza un módulo 2, que a su vez el módulo 2 utiliza un módulo 1, el módulo 3 utiliza la información y los elementos del módulo 1.

Un módulo puede igualmente contener objetos (variables, tableros, etc) que pueden ser compartidos por diferentes unidades programáticas

```
module mod_pgplot
 !color index
  integer,parameter::red=2,green=3,blue=4,cyan=5,magent=6,yellow=7
  !line style
  integer,parameter::full=1,dashed=2,dotdsh=3,dotted=4,fancy=5
  !character fontn
  integer,parameter::normal=1,roman=2,italic=3,script=4
  !fill-area style
  integer,parameter::solid=1,hollow=2
  integer::axis,just
```

```
  real(kind=4)::xmin,xmax,ymin,ymax
end module mod_pgplot
```

Cuando una unidad programática utiliza un módulo, por defecto, es toda la información y objetos contenidos en el módulo que están involucrados; sin embargo, es posible que solo se requiera parte de la información o bien denotar con otros nombres los objetos contenidos en el módulo, afín de evitar confusiones al momento de escribir el código de las diferentes unidades programáticas. Para tal efecto exite, la instrucción `only`, que permite explicitar que objetos serán utilizados del módulo.

```
program principal
  use constantes, only:numero_pi,cons_gravedad
  :
```

El programa principal solamente utilizará las constantes `pi` y `cons_gravedad`, obviando las otras constantes contenidas en el módulo constantes.

En los ejemplos que preceden, se ha utilizado `cons_gravedad`, para identificar la constante de gravedad en la superficie de la Tierra. Como observarse el nombre utilizado es un poco largo para fines prácticos, en su lugar lo habitual es usar la letra `g`. Esto es posible mediante el símbolo "`=>`", cuyo uso se ilustra en el siguiente código:

```
program principal
  use constantes,g=>cons_gravedad,pi=>numero_pi
  :
```

`cons_gravedad` será utilizado por el programa principal mediante la letra `g`, al igual que `numero_pi` será reconocido por `pi`. El uso de "`=>`" y `only` no se excluyen, pueden utilisarse de manera combinada.

Un módulo puede contener igualmente interfaces para declarar subrutinas y funciones externas, en lugar de hacerlo en las unidades programáticas que utilizarán el procedimiento externo. La forma de hacerlo es idéntica a la planteada en el capítulo precedente; es decir:

```
module mod_pgplot
 !color index
  integer,parameter::red=2,green=3,blue=4,cyan=5,magent=6,yellow=7
  !line style
  integer,parameter::full=1,dashed=2,dotdsh=3,dotted=4,fancy=5
  !character fontn
  integer,parameter::normal=1,roman=2,italic=3,script=4
  !fill-area style
  integer,parameter::solid=1,hollow=2
  integer::axis,just
  real(kind=4)::xmin,xmax,ymin,ymax
  interface
     integer function pgopen(ch)
       character(*)::ch
     end function pgopen
  end interface
end module mod_pgplot
```

Cuando un módulo contiene varios procedimientos declarados en una `interface` es preferible utilizar la opción `only` de `use`, de manera que se referencie a los procedimientos estrictamente necesarios y por consiguiente evitar superposiciones y otras confusiones en el código y así como también evitar que una subrutina llame a su propia interface.

Por defecto, todos los objetos definidos en un módulo son accesibles por los procedimientos que usan el módulo. Estos objetos, tienen implícitamente el atributo `public`.. Ahora bien, se puede dar el atributo `private` a ciertos objetos; estos objetos son de uso exclusivo del módulo y no están definidas en los procedimientos que llaman al módulo.

Se utiliza el atributo `private` para definir otros objetos contenidos en el módulo, cuyos atributos serán `public`. Por ejemplo, para definir la matriz identidad, podemos construir:

```
module matriz_identidad
  integer,private,parameter::NMAX=1000
  real(kind=8),private,dimension(NMAX*NMAX),parameter::Idaux=&
       &             (/1.d0,((0.d0,i=1,NMAX),1,j=1,NMAX-1)/)
  real(kind=8),public,dimension(NMAX,NMAX),parameter::Idmax=&
       &             reshape(Idaux,(/NMAX,NMAX/))
end module matriz_identidad
```

Este módulo proporcionará un tablero bidimensional que corresponde a una matriz identidad "grande", la cual puede ser utilizada por secciones, para representar matrices identidades de diferente talla.

Otra alternativa es definir el atributo `private` al final, como en:

```
module matriz_identidad
  integer,parameter::NMAX=1000
  real(kind=8),dimension(NMAX*NMAX),parameter::Idaux=&
       &             (/1.d0,((0.d0,i=1,NMAX),1,j=1,NMAX-1)/)
  real(kind=8),dimension(NMAX,NMAX),parameter::Idmax=&
       &             reshape(Idaux,(/NMAX,NMAX/))
  private::NMAX, Idaux
end module matriz_identidad
```

Existen otras variantes más para el uso de estos atributos, solo es cuestión de consultar la documentación.

VIII.3. Procedimientos de módulo

Un módulo puede encapsular procedimientos. La ventaja de esta forma de trabajar es que ya no es necesario crear interfaces. Ilustremos su uso mediante el siguiente ejemplo:

```
module formulas_geometricas
  real(kind=8),parameter::pi=3.1415926535897932_8
contains
  function vol_esfera(radio) result(volumen)
    real(kind=8),intent(in)::radio
    real(kind=8)::volumen
    volumen=4.d0*pi*(radio**3)/3.d0
  end function vol_esfera
  function area_esfera(radio) result(area)
    real(kind=8),intent(in)::radio
    real(kind=8)::area
    area=4.d0*pi*(radio**2)
  end function area_esfera
  function vol_cilindro(altura,radio) result(volumen)
    real(kind=8),intent(in)::radio
    real(kind=8)::volumen
    volumen=pi*radio*radio*altura
  end function vol_cilindro
  function area_cilindro(altura,radio) result(area)
    real(kind=8),intent(in)::radio
    real(kind=8)::area
    area=2.d0*pi*(radio*(radio+altura))
  end function area_cilindro
end module formulas_geometricas
```

módulo que guardamos en el archivo `formulas_geometricas.f90`, compilamos con la opción `-c`:

```
[hans@gauss ejemplos]$ f95 -c formulas_geometricas.f90
```

lo que da dos archivos: `formulas_geometricas.mod` y `formulas_geometricas.o`. Podemos incluir el archivo `.o` en una librería de programas o bien guardarlo aparte.
Luego construímos nuestro programa de demostración:

```
program demo19
  use formulas_geometricas, only: volumen=>vol_esfera, area=>area_esfera
  real(kind=8)::radio=3.d0
  print*,'El volumen de una esfera de radio 3 es', volumen(radio)
  print*,'El area de una esfera de radio 3 es', area(radio)
end program demo19
```

Compilamos y ejecutamos:

```
s@gauss ejemplos]$ f95 -o demo19 -J. demo19.f90 formulas_geometricas.o
[hans@gauss ejemplos]$ demo19
 El volumen de una esfera de radio 3 es   113.097335529233
 El area de una esfera de radio 3 es   113.097335529233
[hans@gauss ejemplos]$
```

Como puede observarse, se puede utilizar la opción `only` y el cambio de nombre `=>`, sin ningún problema.
Finalmente un procedimiento de módulo puede contener procedimientos internos.

VIII.4. Interfaces genéricas

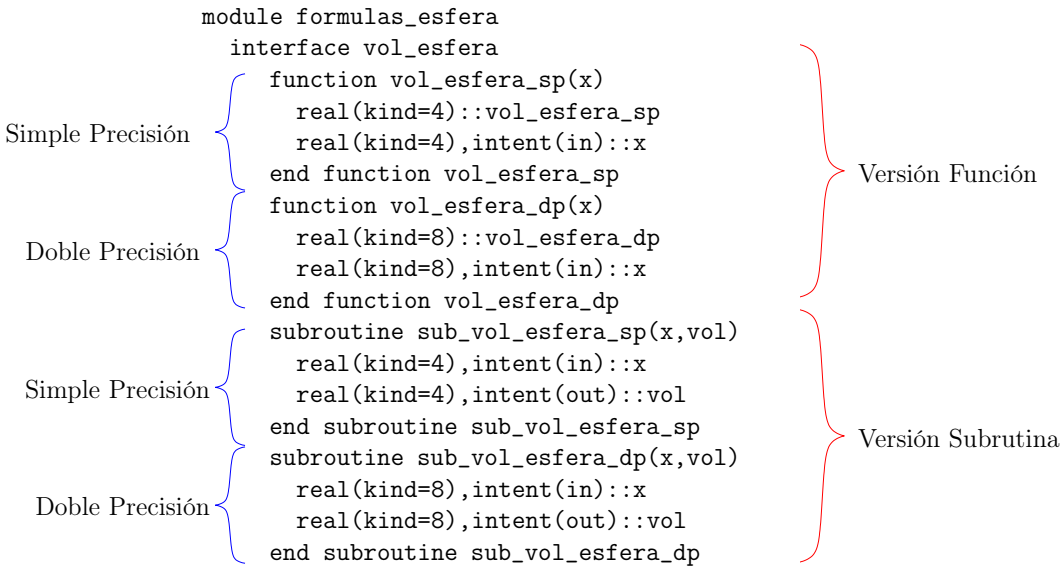
Muchas instrucciones intrínsicas, como `sin(x)`, tienen nombres genéricos. Esto significa que se las puede utilizar con argumentos de tipo y clases diferentes. Por ejemplo, `x` puede estar en simple o en doble precisión o ser un tablero. Todas estas versiones de la función sin están agrupadas bajo un solo nombre.

Es posible crear procedimientos que asocian igualmente, bajo un mismo nombre, diferentes versiones. Un primera forma de trabajar consiste en asociar diferntes versiones a través de una interfaz genérica.

La estructura sintáxica de una interfaz genérica es la siguiente:

```
interface <nombre genérico>
  <identificaci'on procedimiento 1>
   : cabecera
  end <identificaci'on procedimiento 1>
   :
 <identificaci'on procedimiento m>
   : cabecera
  end <identificaci'on procedimiento m>
end interface <nombre genérico>
```

La interfaz genérica puede estar incluida en la parte declarativa de un procedimiento, que utilizará el nombre genérico del procedimiento o bien hacer parte de un módulo. Veamos un ejemplo:




```
module formulas_esfera
  interface vol_esfera
     function vol_esfera_sp(x)
       real(kind=4)::vol_esfera_sp
       real(kind=4),intent(in)::x
     end function vol_esfera_sp
     function vol_esfera_dp(x)
       real(kind=8)::vol_esfera_dp
       real(kind=8),intent(in)::x
     end function vol_esfera_dp
     subroutine sub_vol_esfera_sp(x,vol)
       real(kind=4),intent(in)::x
       real(kind=4),intent(out)::vol
     end subroutine sub_vol_esfera_sp
     subroutine sub_vol_esfera_dp(x,vol)
       real(kind=8),intent(in)::x
       real(kind=8),intent(out)::vol
     end subroutine sub_vol_esfera_dp
  end interface
  interface area_esfera
     function area_esfera_sp(x)
       real(kind=4)::area_esfera_sp
       real(kind=4),intent(in)::x
     end function area_esfera_sp
     function area_esfera_dp(x)
       real(kind=8)::area_esfera_dp
       real(kind=8),intent(in)::x
     end function area_esfera_dp
     subroutine sub_area_esfera_sp(x,area)
       real(kind=4),intent(in)::x
       real(kind=4),intent(out)::area
     end subroutine sub_area_esfera_sp
     subroutine sub_ara_esfera_dp(x,area)
       real(kind=8),intent(in)::x
       real(kind=8),intent(out)::area
     end subroutine sub_ara_esfera_dp
  end interface
end module formulas_esfera
```

Luego ilustramos el uso del modulo `formulas_esfera` en un procedimiento:

```
use formulas_esfera, only:volumen=>vol_esfera
real(kind=8)::radio, vol
:
vol=volumen(radio)     !
call volumen(radio,vol)! dos maneras de utilizar volumen
:
```

Por lo mostrado en el ejemplo, es posible utilizar las opciones `only` y `=>` en interfaces genéricas. También está claro que el código de los procedimientos externos que definen una interface genérica deben existir y hacer parte del proceso de compilación.

Remarcamos que algunos compiladores solo aceptan que la interface genérica contenga exclusivamente funciones o exclusivamente subrutinas; habrá que leer la documentación del compilador.

VIII.4.1. Interfaces genéricas con procedimientos de módulo

Una segunda manera de trabajar con interfaces genéricas consiste en ubicar los diferentes procedimientos en un módulo. Estos procedimientos se convierte en un módulo, luego se asocia las diferentes versiones bajo un nombre común por medio de una interfaz genérica. La estructura sintáxica es la siguiente:

```
module <nombre módulo>
 interface <nombre interfaz 1>
   module procedure <proced. 1,proced. 2,...,proced. k>
 end interface <nombre interfaz 1>
 :
 interface <nombre interfaz m>
   module procedure <proced. 1,proced. 2,...,proced. l>
 end interface <nombre interfaz m>
 contains
   <identificación nombre 1>
    :
   end <identificación nombre 1>
   :
end module <nombre módulo>
```

Ilustremos con las fórmulas geométricas de la esfera.

```
module formulas_esfera
  real(kind=4),private,parameter::pi_4=3.1415926535897932_4
  real(kind=8),private,parameter::pi_8=3.1415926535897932_8
  interface volumen_esfera
     module procedure vol_esfera_sp, vol_esfera_dp,sub_vol_esfera_sp,&
          &           sub_vol_esfera_dp
  end interface volumen_esfera
  interface area_esfera
     module procedure area_esfera_sp, area_esfera_dp,&
          &           sub_area_esfera_sp, sub_area_esfera_dp
  end interface area_esfera
contains
  function vol_esfera_sp(x)
    real(kind=4)::vol_esfera_sp
    real(kind=4),intent(in)::x
    vol_esfera_sp=4.*pi_4*(x**3)/3.
  end function vol_esfera_sp
  function vol_esfera_dp(x)
    real(kind=8)::vol_esfera_dp
    real(kind=8),intent(in)::x
    vol_esfera_dp=4.d0*pi_8*(x**3)/3.d0
  end function vol_esfera_dp
  subroutine sub_vol_esfera_sp(x,vol)
    real(kind=4),intent(in)::x
    real(kind=4),intent(out)::vol
    vol=4.*pi_4*(x**3)/3.
  end subroutine sub_vol_esfera_sp
  subroutine sub_vol_esfera_dp(x,vol)
    real(kind=8),intent(in)::x
    real(kind=8),intent(out)::vol
    vol=4.d0*pi_8*(x**3)/3.d0
  end subroutine sub_vol_esfera_dp
  function area_esfera_sp(x)
```

```
    real(kind=4)::area_esfera_sp
    real(kind=4),intent(in)::x
    area_esfera_sp=4.*(x*x)*pi_4
  end function area_esfera_sp
  function area_esfera_dp(x)
    real(kind=8)::area_esfera_dp
    real(kind=8),intent(in)::x
    area_esfera_dp=4.d0*(x*x)*pi_8
  end function area_esfera_dp
  subroutine sub_area_esfera_sp(x,area)
    real(kind=4),intent(in)::x
    real(kind=4),intent(out)::area
    area=4.*(x*x)*pi_4
  end subroutine sub_area_esfera_sp
  subroutine sub_area_esfera_dp(x,area)
    real(kind=8),intent(in)::x
    real(kind=8),intent(out)::area
    area=4.d0*(x*x)*pi_8
  end subroutine sub_area_esfera_dp
end module formulas_esfera
```

Aplicamos este módulo al programa:

```
program demo19
  use formulas_esfera, only:volumen=>volumen_esfera
  real(kind=8)::radio=3.d0, vol
  print*,'volumen=',volumen(radio)
  call volumen(radio,vol)
  print*,'volumen=',vol
end program demo19
```

compilamos y obtenemos:

```
[hans@gauss ejemplos]$ ifort -o demo19 form_circun.f90 demo19.f90
[hans@gauss ejemplos]$ demo19
 volumen=   113.097335529233
 volumen=   113.097335529233
[hans@gauss ejemplos]$
```

Remarcamos nuevamente, que algunos compiladores solo aceptan que la interface genérica contenga exclusivamente funciones o exclusivamente subrutinas; habrá que leer la documentación del compilador.

VIII.5. Interfaz operador

Recordamos que `+`, `-`, `*`, `**` y `/` son operadores intrínsicos para expresiones de tipo numérico; `< > .ge.` son operadores intrínsicos de comparación; `.and.`, `.or.` son operadores intrínsicos de expresiones de tipo lógico. `Fortran` permite extender las definiciones de operadores existentes a expresiones de otro tipo y también crear nuevos operadores; por ejemplo `.cruz.`, `.convolucion.`, etc. Los nombres de éstos últimos deben obligatoriamente empezar y terminar por un punto.

Estos operadores están asociados con procedimientos, cuyos argumentos (del operador) deben tener el atributo `intent(in)`. Se puede construir operadores de uno o dos argumentos.

Ilustremos la elaboraciñ de una interfaz operador:

```
module operadores_vectoriales
  interface operator(.cruz.)
     module procedure cruz_sp, cruz_dp
  end interface
```

```
  interface operator(.escalar.)
     module procedure escalar_sp, escalar_dp
  end interface
contains
  function cruz_sp(u,v) result(w)
    real(kind=4),intent(in),dimension(3)::u,v
    real(kind=4),dimension(3)::w
    w(1)=u(2)*v(3)-u(3)*v(2)
    w(2)=u(3)*v(1)-u(1)*v(3)
    w(3)=u(1)*v(2)-u(2)*v(1)
  end function cruz_sp
  function cruz_dp(u,v) result(w)
    real(kind=8),intent(in),dimension(3)::u,v
    real(kind=8),dimension(3)::w
    w(1)=u(2)*v(3)-u(3)*v(2)
    w(2)=u(3)*v(1)-u(1)*v(3)
    w(3)=u(1)*v(2)-u(2)*v(1)
  end function cruz_dp
  function escalar_sp(u,v) result(a)
    real(kind=4),intent(in),dimension(3)::u,v
    real(kind=4)::a
    a=dot_product(u,v)
  end function escalar_sp
  function escalar_dp(u,v) result(a)
    real(kind=8),intent(in),dimension(3)::u,v
    real(kind=8)::a
    a=dot_product(u,v)
  end function escalar_dp
end module operadores_vectoriales
```

Luego, en un procedimiento que utilice este módulo las instrucciones:

```
a=b.cruz.c
alpha=a.escalar.b
```

son válidas.

Una aspecto que debe quedar claro, es que se puede extender el campo de definición de los operadores intrínsicos de `Fortran`, pero nunca se puede remplazar lo ya predefinido; por ejemplo modificar la adición en los enteros.

VIII.6. Interfaz de asignación

De la misma manera que en caso de la interfaz operador, se puede prolongar el campo de definición del signo de asignación `=`, crear nuevas instrucciones de acción, siguiendo la misma regla sintáxica (punto al inicio y final del nombre), por ejemplo `.igual.`. Todo ésto se puede hacer por medio de un bloque `interface assignment`, cuya estructura la ilustramos mediante el siguiente ejemplo, que convierte un vector en un escalar y un escalar en un vector:

```
module asignaciones_vectoriales
  interface assignment(=)
     module procedure vector_igual_escalar, escalar_igual_vector
  end interface
contains
  subroutine vector_igual_escalar(vector,escalar)
    real(kind=8),intent(out),dimension(:)::vector
```

```
    real(kind=8),intent(in)::escalar
    vector(1)=escalar
  end subroutine vector_igual_escalar
  subroutine escalar_igual_vector(escalar,vector)
    real(kind=8),intent(in),dimension(:)::vector
    real(kind=8),intent(out)::escalar
    escalar=vector(1)
  end subroutine escalar_igual_vector
end module asignaciones_vectoriales
```

Observamos que se puede utilizar subrutinas a dos argumentos, siendo el primer argumento el lado izquierdo de la instrucción de asignación y el segundo argumento el lado derecho de la instrucción de asignación.

De esta manera, hemos creado un mecanismo de conversión entre objetos de naturaleza diferente.

VIII.7. Ejercicios

1. Completar los ejemplos en programas.
2. Crear un tipo derivado para $\mathbb{F}_7$, luego definir por medio de interfaces las operaciones y asignaciones de este cuerpo.

Capítulo IX

Punteros

Un puntero (*pointer*) es una entidad que se asocia a la dirección de memoria de un objeto destino (*target*), como tal contiene la dirección de memoria y otra información descriptiva. En `Fortran` hay que pensar en los punteros como "alias" en lugar de como posiciones de memoria; su declaración crea y asigna memoria al puntero, pero no crea un objeto destino.

Se puede trabajar con un puntero, como si se trabajase con los objetos destino. Manipulando una sola entidad, el puntero, se puede actuar sobre diferentes objetos en el programa, los destinos.

El puntero y los objetos destino deben estar declarados con el mismo tipo, la misma clase y el mismo rango. Los punteros reciben el atributo `pointer`, los objeto destino el atributo `target`. Consiguientemente, ilustramos una forma de declarar un puntero:

```
real,dimension(:),pointer::lista
real,dimension(:),allocatable,targer::lista1,lista2
```

Cuando un tablero de talla variable, tiene el atributo `pointer`, automáticamente tiene el atributo `allocatable`, motivo por el cual, ya no es necesario colocar este atributo en la declaración.

La instrucción de asociación de un puntero con un objeto destino está dada por los símbolos =>,,

```
lista=>lista1
:
lista=>lista2
```

Para saber si un puntero está asociado, se puede utilizar la función `logica associated`:

```
associated(lista)
```

que da un valor `.true.` si el puntero está asociado y un valor `.false.` si el puntero no esta asociado a ningún objeto destino.

Para anular la asociación entre un puntero y un eventual objeto destino, se puede utilizar la función `nullify`:

```
nullify(lista)
```

`Fortran` permite utilizar un puntero, como un tablero ajustable, al cual se le puede asignar una existencia propia, independientemente de los objetos destino, que podría apuntar.

```
allocate(lista(1:10))
:
deallocate(lista)
```

Por cuestiones de eficacia, puede ser interesante utilizar un puntero, como argumento de un procedimiento. Se transmite de esta manera la dirección de la memoria del objeto, en lugar del objeto mismo.

Parte II

Graficando con PGPLOT

Capítulo X

Biblioteca Gráfica PGPLOT

La biblioteca gráfica `pgplot` es una colección de procedimientos, desarrollada por CalTech (California Institute of Technology), básicamente para `FORTRAN 77`, que permite elaborar gráficos de uso cient'fico con diferentes dispositivos y medios.

Tanto el código fuente de la colección de procedimientos, como la documentación, están disponibles en Internet, en la dirección:

`http://www.astro.caltech.edu/~tjp/pgplot/`

X.1. Instalación de `pgplot`

Existen dos maneras simples de instalar `pgplot`. La primera, válida para computadoras que corren bajo LINUX, (*e.g.*: Fedora(s), RedHat, Mandrake, etc,) consiste en instalar utilizando una versión `rpm` de pgplot, que se puede obtener en la Web. La segunda forma consiste en obtener el código fuente de:

`ftp://ftp.astro.caltech.edu/pub/pgplot/pgplot5.2.tar.gz`

descompactar el archivo pgplot5.2.tar.gz y seguir las instrucciones de instalación.

La primera forma de instalación es la más simple, pero con la desventaja que es una instalación estándar, que no toma en cuenta las necesidades y particularidades del programador.

Una vez instalado `pgplot`, de manera correcta, se debe poder identificar los siguientes archivos y o directorios.

1. `libpgplot.a` versión estática de la biblioteca gráfica. Debe encontrarse, normalmente, en alguno de los siguientes directorios:
   - `/usr/local/lib`
   - `/usr/lib/`
   - `$HOME/lib`
2. `libpgplot.so` versión compartida de la biblioteca gráfica. Su ubicación debiera ser la misma que la versión estática.
3. Un directorio `pgplot`, cuya ubicación exacta debiera ser una de las siguientes alternativas:
   - `/usr/local/pgplot`
   - `/usr/local/lib/pgplot`
   - `/usr/lib/pgplot`
   - `$HOME/lib/pgplot`

   Este directorio debe contener, mínimamente, los siguientes archivos:

a) `pgxwin_server`, programa para desplegar gráficos en la pantalla. Es también posible que este programa se encuentre en `/usr/bin` o `/usr/local/bin`

b) `grfont.dat`, base de datos, para caracteres y símbolos.

c) `rgb.txt`, base de datos, para colores.

Por último, como parte del proceso de instalación, sobre todo en el caso del código fuente, habrá que declarar algunas variables de entorno, especialmente la ubicación del directorio `pgplot`. Esto se hace a través de la instrucción

```
fortran]$ export PGPLOT_DIR=<pgplot>
```

que se la realiza cada vez que se hará uso de la biblioteca `pgplot` o insertando esta instrucción en el archivo `.bashrc`, que permite dar la ubicación del directorio al momento de ingresar a la computadora.

X.2. Dispositivos Gráficos

Para el uso que daremos a la biblioteca gráfica, nos interesa dos tipos de dispositivo gráfico, para los otros ver la documentación de pgplot.

El primer dispositivo, par nuestro interés, es aquel que produce una copia de los gráficos en un archivo. Los gráficos contenidos pueden ser visualizados a través de un utilitario.

El segundo dispositivo, es aquel que produce los gráficos en la pantalla. Su uso puede ser interactivo.

Ahora bien, un dispositivo está identificado por su nombre y por su tipo, ambos separados por el caracter `/`. El nombre del dispositivo es el nombre del archivo, en el caso de que el dispositivo produzca un archivo, o bien el nombre por el cual el dispositivo será reconocido por el sistema operativo. Es conveniente utilizar doble comillas `"` para delimitar el nombre del dispositivo. Si el nombre del dispositivo se omite, un nombre por defecto le será otorgado.

En cuanto se refiere al tipo de dispositivo, en este texto se manipulará:

- Gráficos en archivo. Los tipos son:

  1. `PS` archivo en formato PostScript, orientación paisaje, monocolor.
  2. `VPS` archivo en formato PostScript, orientación retrato, monocolor.
  3. `CPS` archivo en formato PostScript, orientación paisaje, color.
  4. `VCPS` archivo en formato PostScript, orientación retrato, color.

  El nombre por defecto de estos tipos de dispositivo es "`pgpplot.ps`".

- Gráficos en la pantalla. El tipo de dispositivo básico es:
  `XWINDOW`, o mediante las abreviaciones: `XW` o `XWIN`.

Resumiendo, los siguientes ejemplos son especificaciones válidas de dispositivos:

```
/VCPS
elipse.ps/VCPS
/XWIN
1/XWINDOW
```

Para detalles técnicos de los dispositivos, referirse al manual o bien a la dirección:

```
http://www.astro.caltech.edu/~tjp/pgplot/devices.html
```

Uno abre un dispositivo gráfico a través de la función `pgopen` , cuya interfaz de identificación es:

```
interface
  function pgopen(dispositivo)
    integer:: ppopen
    character(*),intent(in)::dispositivo
  end function pgpopen
end interface
```

donde `dispositivo` es la especificación del dispositivo a abrir (nombre + tipo de dispositivo). Aparte de la especificación del dispositivo, `dispositivo` acepta utilizar el signo de interrogación "`?`", como valor de entrada, que permite introducir al usuario, de manera interactiva, la información del dispositivo, durante la ejecución del programa.

Los valores enteros que proporciona `pgopen` son:

- Valores negativos indican error al abrir el dispositivo.
- Valor positivo indica el número que pgplot ha asignado al dispositivo, de utilidad cuando se utiliza diferentes dispositivos. A este número lo identificamos por `id`.

Uno cierra el dispositivo a travez de la subrutina `pgend`, cuya interfaz es

```
interface
  subroutine pgend
  end subroutine pgend
end interface
```

Debe remarcarse que `pgplot` trabaja con el último dispositivo gráfico abierto o seleccionado, esto significa que para cambiar de dispositivo, hay que hacerlo de manera explícita. La subrutina de interfaz:

```
interface
  subroutine pgslct(id)
  integer,intent(in)::id
  end subroutine pgslct
end interface
```

donde `id` es el número de dispositivo proporcionado por `pgopen`, permite cambiar al dispositivo seleccionado por `id`.

X.3. Ventanas y *viewports*

Una vez elegido el dispositivo gráfico, utilizando la función `pgopen`, `pgopen` abre una superficie de graficación *surfaceview*, que en el caso del dispositivo de tipo Postscript es una hoja de 11 por 8 pulgadas (retrato) y 8 por 11 pulgadas (paisaje). En el caso del dispositivo `xwindow` es una ventana que ocupa la pantalla.

Para cambiar las dimensiones de la superficie de dimensión, se utiliza la subrutina `pgpap` inmediatamente después de abrir el dispositivo, o bien, después de cambiar una página. La interfaz de `pgpap` es:

```
interface
  subroutine pgpap(xlen,yprop)
    real,intent(in)::xlen,yprop
  end subroutine pgpap
end interface
```

donde `xlen` es la longitud horizontal de la superficie de graficación, expresada en pulgadas; `yprop` es el factor de proporción, respecto a la longitud horizontal, de la longitud vertical de la superficie de graficación; por ejemplo: `yprop=1.`, dará la longitud vertical igual a la longitud horizontal, `yprop=.5`, dará como resultado, la longitud vertical igual a la mitad de la longitud horizontal.

Se debe observar, que cada vez que se llama a la subrutina `pgpap`, `pgplot` crea una nueva superficie de graficación, con las dimensiones incorporadas en la subrutina.

Por defecto, `pgplot` trabaja con un cuadro o *frame* por superficie de graficación. La subrutina `pgsubp` con interfaz

```
 end interface
   subroutine pgsubp(nfx,nfy)
    integer,intent(in)::nfx,nfy
   end subroutine pgsubp
 end interface
```

permite dividir una superficie en cuadros de igual tamaño, distribuidos en |`nfx`| columnas y `nfy` filas, bajo el siguiente criterio:

- Si `nfx`> 0, el orden de numeración se da por filas.
- Si `nfx`< 0, el orden de numeración se da de acuerdo a las columnas.

Un la figura X.3.1, se ilustra el orden de numeración. Cada vez que se llama a la subrutina `pgsubp` se crea

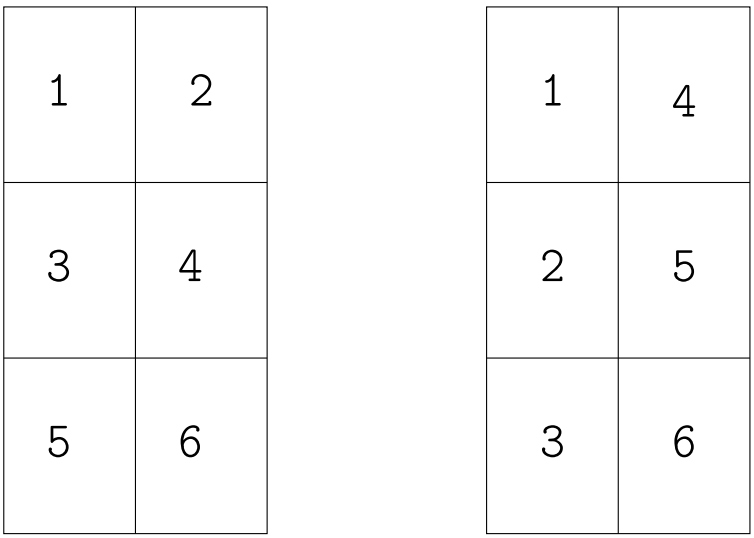


Figura X.3.1: Ilustración de la subrutina `pgsubp`

una superficie de graficación y en caso que se depase el número de cuadros establecido por la subrutina, se crea una nueva superficie de graficación, con las dimensiones establecidas por `pgpap` o en caso contrario, las dimensiones son las por defecto.

Consiguientemente, para efectos prácticos, la unidad de graficación será un cuadro o *frame*, que se inicia llamando a la subrutina `pgpage`

```
interface
  subroutine pgpage
  end subroutine
end interface
```

y obviamente cuando uno desea cambiar de unidad de graficación debe volver a llamar la subrutina `pgpage`.

Ahora bien, `pgplot` destina un sector rectangular dentro la unidad de graficación, llamado *portview*, para la graficación de los diferentes objetos, dejando el espacio libre para anotaciones del gráfico. Por defecto el *portview* está centrado en la unidad de graficación. Ilustramos en la figura X.3.2.

Las dimensiones de un *portview* son físicas; es decir están expresadas en púlgadas y están relacionadas a las dimensiones de la unidad de graficación. Se puede modificar la ubicación en la unidad de graficación y sus dimensiones, por medio de las subrutinas, cuyas interfaces son:

```
interface
 subroutine pgsvp(xrmin,xrmax,yrmin,yrmax)
   real,intent(in)::xrmin,xrmax,yrmin,yrmax
 end subroutine pgsvp
 subroutine pgvsiz(xamin,xamax,yamin,yamax)
    real,intent(in)::xamin,xamax,yamin,yamax
```

```
 end subroutine pgvsiz
 subroutine pgvstd
 end subroutine pgvstd
end interface
```

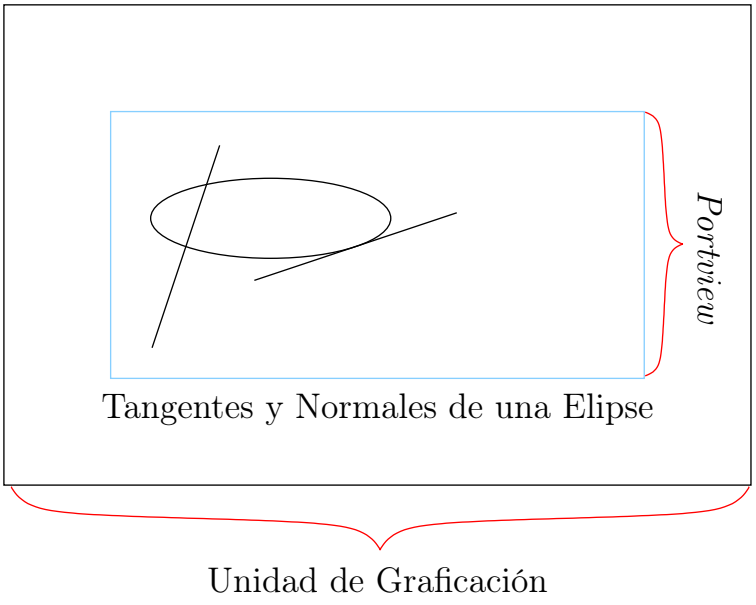
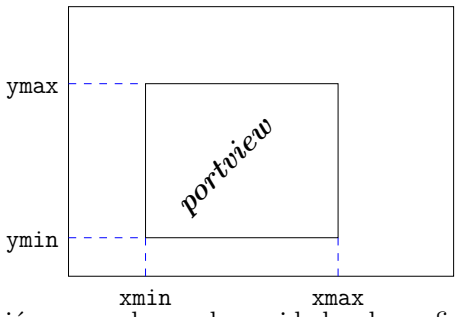


Figura X.3.2: Ilustración de una unidad de graficación

En el caso de la subrutina `pgsvp`, `xrmin`, `xrmax`, `yrmin` y `yrmax` determinan las coordenadas relativas (valores entre `.0` y `1.`) de los vértices del *portview*.
Para `pgvsiz`: `xamin`, `xamax`, `yamin` y `yamax` determinan las coordenadas absolutas (valores en pulgadas) de los vértices del *portview*.
La subrutina `pgvstd` ubica el *portview* en el centro de la unidad de graficación y determina automáticamente las dimensiones del *portview*.



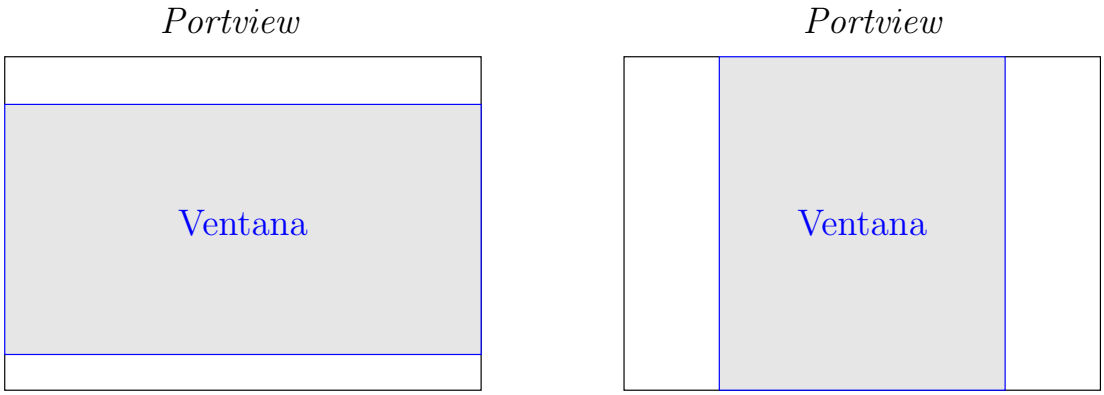
Las escalas utilizadas hasta ahora: desde la superficie de graficación, pasando por las unidades de graficación, hasta el *portview* son escalas físicas, expresadas en pulgadas, que corresponden a las dimensiones o tallas del dispositivo de graficación utilizado. Ahora bien, las escalas utilizadas por los gráficos, no necesariamente, corresponden a las escalas del *portview*, por lo que `pgplot` hace internamente una transformación de coordenadas, de las coordenadas del gráfico a las del dispositivo gráfico, de manera que el programador no se ocupe de hacerlo y lo hace de dos formas. Para tal efecto, introducimos el concepto de ventana, que es la porción rectangular más grande, contenida en el *portview*, en la cual es posible construir gráficos. Veamos las dos formas:

1. La transformación de coordenadas es la composición de traslaciones y homotecias. Esto significa que una unidad de la escala horizontal y vertical de la ventana vale lo mismo en pulgadas en la escala del dispositivo gráfico. La ventana queda centrada, ya sea verticalmente o horizontalmente en el *portview*. La ventana es creada, llamando a la subrutina

   ```
   interface
     subroutine pgwnad(xmin,xmax,ymin,ymax)
   ```

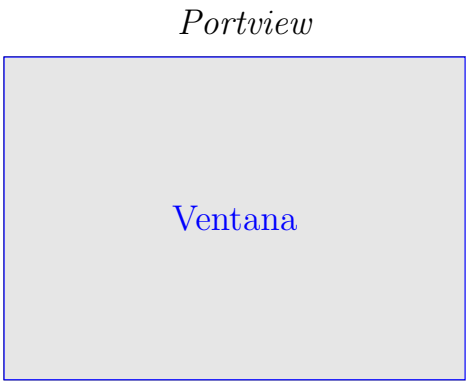
```
  real,intent(in)::xmin,xmax,ymin,ymax
end interface
```

donde `xmin`, `xmax`, `ymin` y `ymax` determinan los rangos horizontal y vertical de la ventana, en la cual están las gráficas a hacerse.



Debe observarse que (`xmin`,`ymin`) son las coordenadas del vértice inferior izquierdo e (`xmax`,`ymax`) las coordenadas del vértice superior izquierdo; puede darse que `xmax`<`xmin` o `ymax`<`ymin`.
Para efectos prácticos, `pgplot` ajusta el *portview* a la ventana, permitiendo utilizar el espacio vacio del *portview* para las anotaciones del gráfico.

2. La segunda forma consiste en ajustar la ventana al *portview*, lo que significa que una unidad de la escala horizontal de la ventana, no necesariamente es igual a una unidad de la escala vertical de la ventana, expresada en pulgadas, en el dispositivo gráfico.



Para la creación de la ventana, se utiliza la subrutina

```
interface
  subroutine pgswin(xmin,xmax,ymin,ymax)
  real,intent(in)::xmin,xmax,ymin,ymax
end interface
```

donde `xmin`, `xmax`, `ymin` y `ymax` determinan los rangos horizontal y vertical de la ventana, en la cual están las gráficas a hacerse. Debe también observarse que (`xmin`,`ymin`) son las coordenadas del vértice inferior izquierdo e (`xmax`,`ymax`) las coordenadas del vértice superior izquierdo; puede darse que `xmax`<`xmin` o `ymax`<`ymin`.

Antes de continuar con la construcción de nuestros gráficos, es importante indicar que tanto el `viewport`, como la ventana deben estar establecidos. Usualmente, primero se define el `viewport` y luego la ventana; sin embargo, es posible definir primero la ventana y luego el `viewport`. El siguiente paso será preparar el entorno de nuestro gráfico, gráficando los ejes, colocando títulos en los espacios libres dejados por el `viewport`. La subrutina `pgbox`, de interfaz

```
interface
  subroutine pgbox(xopt,xtick,nxsub,yopt,ytick,nysub)
    character(len=*),intent(in)::xopt,yopt
    real,intent(in)::xtick,ytick
    integer,intent(in)::nxsub,nysub
  end subroutine pgbox
end interface
```

permite trazar ejes, bordes de recuadro del *portview*, colocar números en los ejes. El significado de cada uno de los argumentos, así como sus posibles valores se los encuentra en la descripción de subrutinas del manual de `pgplot`. Los títulos e información adicional puede insertarse, por ejemplo, llamando a la subrutina `pglab` de interfaz

```
interface
  subroutine pglab(xlabel,ylabel,toplabel)
    character(len=*),intent(in)::xlabel,ylabel,toplabel
end interface
```

donde `xlabel` es la etiqueta del eje x, `ylabel` es la etiqueta del eje y y `toplabel` puede ser el título del gráfico que se coloca en la parte superior del gráfico.

A continuación, veamos un ejemplo ilustrativo de lo abordado hasta aquí.

```
program demo22
  use mod_pgplot
  real(kind=4)::xmin,xmax,ymin,ymax,xrmin,xrmax,yrmin,yrmax
  integer::iostat
  iostat=pgopen('/XWIN') ! apertura del dispositivo de graficaci'on
  call pgpap(3.,.5) ! tamano de la superficie de graficacion
  call pgpage ! inicializacion unidad de graficacion
  xrmin=.2; xrmax=.9; yrmin=.15; yrmax=.8
  call pgsvp(xrmin,xrmax,yrmin,yrmax) ! definicion del portview
  xmin=2.; xmax=-3.; ymin=2.; ymax=-3.
  call pgswin(xmin,xmax,ymin,ymax)  ! definicion de la ventana de graficacion
  call pgbox('AL',0.0,0,'AL',0.0,0) ! ejes logaritmicos
  call pglab('eje x','eje y','Ejemplo Ilustrativo') ! etiquetas
  call pgend
end program pgdemo22
```

En la figura X.3.3 podemos observar el resultado del código ejemplo. Si bien la forma de preparar un gráfico, construyendo una ventana de graficación, sobre todo por la cantidad y variedad de opciones que se debe introducir en la subrutina `pgbox`, el ejercicio era indispensable para introducir los conceptos desarrollados.

Debe observarse, que `pgplot` tiene a disposición una subrutina, de interfaz:

```
interface
  subroutine pgenv(xmin,xmax,ymin,ymax,just,axis)
    real,intent(in)::xmin,xmax,ymin,ymax
    integer,intent(in)::just,axis
  end subroutine pgenv
end interface
```

donde los valores `xmin`, `xmax`, `ymin` y `ymax` determinan la ventana de graficación, como en `pgswin` o `pgwnad`; `just`$\neq$`1` ajusta la ventana de graficación al *portview*, como en `pgswin`, `just=1` ajusta el *portview* a la ventana de graficación, como en `pgwnad`; el valor de `axis` determina el tipo de ejes:

`axis=-2` No dibuja nada.

`axis=-1` Dibuja solamente el recuadro de la ventana de graficación

`axis=0` Dibuja recuadro de la ventana y gradua con sus coordenadas.

`axis=1` Como `axis=0`, con además los ejes x e y.

`axis=2` Como `axis=1`, pero además agrega una grilla de líneas horizontales y verticales.

`axis=10` Dibuja recuadro de la ventana de graficación y el eje x logaritmicamente.

`axis=20` Dibuja recuadro de la ventana de graficación y el eje y logaritmicamente.

`axis=30` Dibuja recuadro de la ventana de graficación y ambos ejes logaritmicamente.

`pgenv`, como subrutina, llama sucesivamente a las subrutinas: `pgpage` (crea una unidad de graficación), `pgsvtd` (crea un estándar *portview*), `pgswin` o `pgwnad` (crea una ventana de graficación), por lo que su uso simplifica la tarea de preparación de graficación, sobre todo para gráficos simples. Las diferentes alternativas de `axis`, la ilustramos en la figura X.3.4

X.4. Primitivas

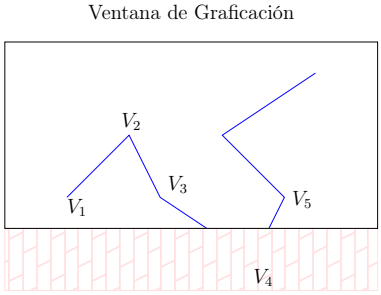
Existe cuatro tipos de primitivas: líneas, texto, relleno de áreas y marcadores gráficos. El principio básico es que `pgplot` gráfica únicamente lo que se encuentra dentro la ventana de graficación y lo que está fuera no lo hace; por lo tanto al programar uno no debe preocuparse por lo que puede suceder fuera de los bordes de la ventana de graficación.

X.4.1. Líneas poligonales

`pgplot` permite el trazado de líneas poligonales a través de la subrutina `pgline`, cuya interfaz es:

```
interface
  subroutine pgline(npol,xpg,ypg)
    integer::intent(in)::npol
    real,dimension(npol)::xpg,ypg
  end subroutine pgline
end interface
```

donde `npol` es el número de puntos de la línea poligonal. `xpg` e `ypg` son vectores que contienen las coordenadas (sistema de coordenadas de la ventana) de los vértices de la línea poligonal; así `xpg(k)` y `ypg(k)` son: la absisa y ordenada del k-simo vértice de la línea poligonal.



`pgplot` traza la línea poligonal, partiendo del primer vértice, va al segundo, luego al tercero y así sucesivamente. Cuando parte de un segmento o la totalidad del segmento se encuentra fuera de la ventana de gráficación, `pgplot` traza el segmento hasta el borde de la ventana. Las líneas poligonales tienen como atributo: color, forma (continua, segmentada, punteada, etc) y espesor, que pueden ser modificadas en el transcurso del programa, para mayores detalles ver el manual de `pgplot`

Figura X.3.3: Uso demostrativo de `pgplot`.

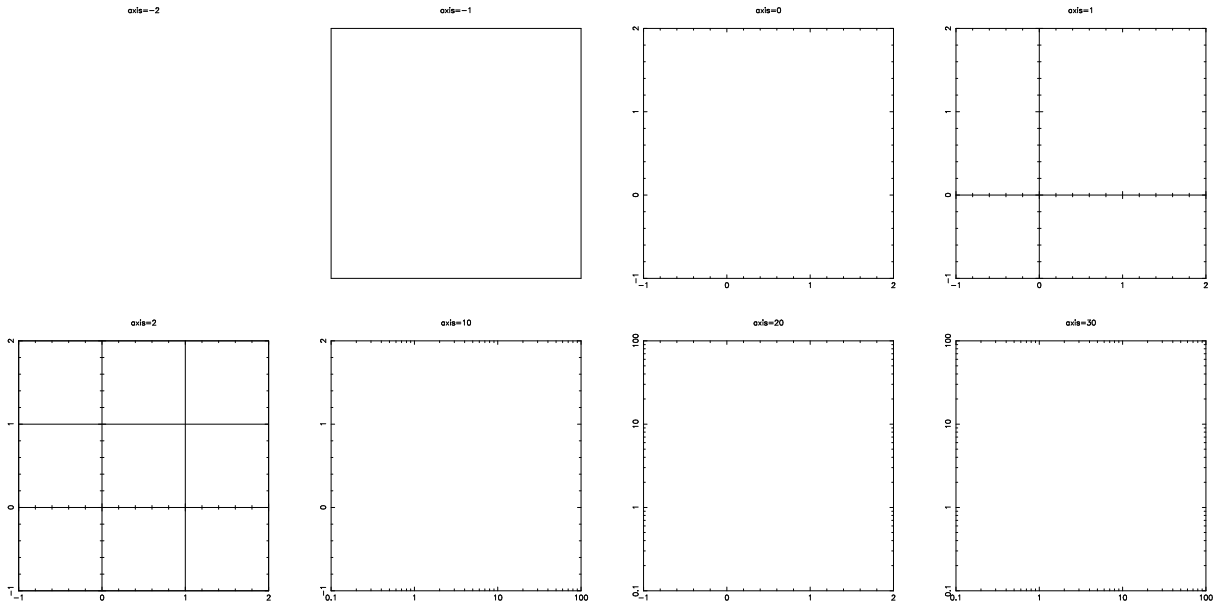


Figura X.3.4: `pgenv` y la opción `axis`

X.4.2. Marcadores Gráficos

Un marcador gráfico es un símbolo, como una cruz, un punto, etc, dibujado en un punto específico de la ventana de graficación.

La subroutina `pgpt`, cuya interfaz es

```
interface
  subroutine pgpt(n,xpg,ypg,nsym)
    integer,intent(in)::n,nsym
    real(kind),dimension(n),intent(in)::xpg,ypg
  end subroutine pgpt
end interface
```

dibuja uno o más marcadores gráficos. `n` es el número de puntos para marcar, los vectores `xpg` e `ypg` contienen las coordenadas, respecto a la ventana de graficación, de los puntos a marcar y `nsym` identifica el símbolo que se utilizará.

Los valores de `nsym` pueden ser:

- `nsym=-1` para dibujar un punto de la talla más pequeña.
- `nsym=0,...,31` para dibujar uno de los símbolos de la Figura 4.1 del Manual de `pgplot`.
- `nsym=-3,...,-8` para dibujar un polígono regular de 3,...,8 lados.
- `nsym=33,...,127`, para dibujar el correspondiente `ASCII` caracter. Para obtener el valor númerico del caracter, se puede utilizar la función `ichar`; por ejemplo:

  ```
  call pgpt(n,xpg,ypg,ichar('H'))
  ```

  dibujará el caracter "H" en los correspondientes puntos.
- `nsym`>`127` dibujará uno de los símbolos Hershey del Apéndice B del Manual de `pgplot`.

X.4.3. Texto

Para insertar texto, dentro la ventana de graficación, `pgplot` tiene a disposición dos subrutinas:

```
interface
  subroutine pgtext(x,y,texto)
    real,intent(in):: x,y
    character(len=*)::texto
  end subroutine pgtext
  subroutine pgtxt(x,y,angulo,texto)
    real,intent(in):: x,y, angulo
    character(len=*)::texto
  end subroutine pgtxt
end interface
```

donde `x`, `y` son las coordenas, respecto a la ventana de graficación', donde se insertará el texto, `texto` es una cadena de caracteres y `angulo` expresa un ángulo en grados. `pgtext` inserta la cadena de caracteres de `texto`, de manera horizantal y partiendo del punto (`x,y`). `pgtxt` escribe el texto con una inclinación de ángulo `angulo` centrado en el punto (`x,y`).

Remarcamos que estas dos subrutinas se utilizan exclusivamente en la ventana de graficación. Para escribir texto en otras regiones de la unidad de graficación, se puede utilizar la subrutina `pgmtxt`, cuyo modo de uso se detalla en el Manual de `pgplot`.

`pgplot` permite una serie de facilidades con la inserción de texto, como, la utilización de caracteres especiales, cambio de fuente, índices, sobre índices, etc; cuyo detalle se encuentra en el Manual de `pgplot`.

X.4.4. Polígonos

Juntamente con la graficación de líneas poligonales, la graficación de polígonos son las primitivas más importantes en la construcción de un buen gráfico. `pgplot` trata los polígonos como superficies, que dependiendo del atributo elegido, éstas son rellenadas, ya sea por líneas, color entero o simplemente dejadas vacias; para tal efecto se tiene las siguientes subrutinas, de interfaz:

```
  subroutine pgpoly(n,xpg,ypg)
     integer::intent(in)::n
     real, intent(in)::xpg,ypg
  end subroutine pgpoly
  subroutine pgrect(x1,x2,y1,y2)
     real,intent(in)::x1,x2,y1,y2
  end subroutine pgrect
  subroutine pgcirc(cx,cy,radio)
     real,intent(in)::cx,cy,radio
  end subroutine pgcirc
```

donde `pgpoly` rellena el polígono de `n` vértices, cuyas coordenas, respecto a la ventana de graficación, están contenidas en los vectores `xpg` e `ypg`; `pgrect` rellena el rectángulo de lados paralelos a los ejes, donde (`x1`,`y1`) y (`x2`,`y2`) son las coordenadas, respecto a la ventana de graficación, de dos vértices diagonalmente opuestos. Por último `pgcirc` rellena el círculo de centro (`cx`,`cy`) y radio `radio`; debe observarse que se está graficando sobre una ventana de graficación y que para obtener un círculo sobre el `portview` las escalas de las coordenadas de la ventana de graficación respecto a las coordenadas del `portview` deben ser las mismas.

X.5. Atributos

Tal como se mencionó en la anterior sección, las primitivas de graficación tienen diferentes atributos, como: color, talla, tipo, etc.

X.5.1. Color

`pgplot` asigna un número a cada color, llamado índice de color; por consiguiente, cualquier cambio del atributo color se lo realiza cambiando de índice color. El índice color de valor 0 es asignado al color de fondo (*background color*), el de valor 1 corresponde al color por defecto, para la graficación de los objetos; para el dispositivo gráfico `/XWINDOW` el color de fondo es negro y el color por defecto para gráficar es blanco, mientras que para los dispositivos gráficos de tipo `PostScript` el color de fondo es blanco y el color por defecto es negro.

Aparte de estos dos colores, si el dispositivo gráfico permite colores no monocronos, que es el caso de `/XWINDOW` y `PostScript` en color, `pgplot` tiene asignado los índices de color 2 a 15 a colores prestablecidos y los índices 16 para adelante para colores a definirse. Mayores detalles de los colores preestablecidos en el Manual de `pgplot`. El cambio de índice de color se realiza, por medio de la subrutina, cuya interfaz es:

```
interface
   subroutine pgsci(indice_color)
      integer::intent(in)::indice_color
   end subroutine pgsci
end interface
```

como ejemplo de utilización:

```
  call pgsci(2) ! cambia al color de  indice 2
  call pgline(npg,xpg,ygp) ! grafica la linea poligonal con el
                           ! color de indice 2
  call pgsci(1)  ! el dispositivo cambia al color de defecto
```

Ahora veamos la forma de asignar un color determinado a un color indice y lo primero que hay que hacer es conocer el rango de índices color que permite el disposito gráfico, esto se logra llamando a la subrutina, de interfaz:

```
interface
  subroutine pgqcol(cii,cis)
     integer,intent(out)::cii,cis
  end subroutine pggcol
end interface
```

donde `cii` es la cota inferior del rango de índices, normalmente vale 0 o 1 y `cif` es la cota superior del rango de índices, que para dispositivos `/XWINDOW` y `PostScript` color vale 255.

Una vez conocido el rango disponible de índices de color, se puede asignar un color a un índice de tres formas:

- Utilizando la representación RGB o (RVA en español: rojo, verde y azul), representación utilizada en los televisores de color, por medio de la subrutina de interfaz:

  ```
  interface
    subroutine pgscr(ic,cr,cg,cb)
      integer,intent(in)::ic
      real,intent(in)::cr,cg,cb
    end subroutine pgscr
  end interface
  ```

  donde `ic` es el índice de color asignado al color, cuya representación RGB está dada por `cr` (rojo), `cg` (verde) y `cb` (azul).

- Utilizando la representation HLS (TLS en español: tonalidad, luminosidad y saturación), representación utilizada en el procesamiento de imágenes, por medio de la subrutina de interfaz:

```
interface
  subroutine pgshls(ic,ch,cl,cs)
    integer,intent(in)::ic
    real,intent(in),ch,cl,cs
  end subroutine pgshls
end interface
```

donde `ic` es el índice de color asignado al color, cuya representación HLS está dada por `ch` (tonalidad), `cl` (luminosidad) y `cs` (saturación).

- Utilizando el nombre del color, presente en el archivo `rgb.txt` del directorio `pgplot`, por medio de la subrutina:

```
interface
  subroutine pgscrn(ic,nombre_color,ier)
    integer,intent(in)::ic
    character(len=*),intent(in)::nombre_color
    integer,intent(out)::ier
  end subroutine pgscrn
end interface
```

donde `ic` es el índice de color asignado al color de nombre `nombre_color`. Si la asignación ha sido correcta, `ier=0` y si ha habido un error, como nombre inexistente, `ier=1`.

X.5.2. Estilos de líneas

`pgplot` tiene a disposición cinco estilos de trazado para las líneas:

Estilo 1 Línea continua,

Estilo 2 Línea segmentada,

Estilo 3 Línea segmentada y punteada,

Estilo 4 Línea punteada,

Estilo 5 Línea segmentada con tres puntos entre segmentos.

El estilo por defecto es el 1 y la forma de cambiar de estilo, es llamando a la subrutina:

```
interface
  subroutine pgsls(ls)
    integer,intent(in)::ls
  end subroutine
end interface
```

donde `ls` es el número de estilo de línea. Este atributo solamente afecta al trazado de líneas poligonales y polígonos y el cambio de estilo debe hacerse antes de la graficación de líneas poligonales y polígonos.

X.5.3. Grosor de líneas

Índice alfabético