PRACTICA 1e

Elementos de programación Fortran: Programación modular.

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

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

En una implementación computacional, cada subproblema es implementado como un *subprograma*. De este modo el programa consta de un *programa principal* (la unidad del programa de nivel más alto) que llama a subprogramas (unidades del programa de nivel más bajo) que a su vez pueden llamar a otros subprogramas. La figura 1 representa esquemáticamente la situación en un diagrama conocido como *diagrama de estructura*. Por otra parte en un diagrama de flujo del algoritmo, un subprograma es representado como se ilustra en la figura 2 el cual permite indicar que la estructura exacta del subprograma será detallada aparte en su respectivo diagrama de flujo.

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

Un diseño modular de los programas provee la siguientes ventajas:

- El programa principal consiste en un resumen de *alto nivel* del programa. Cada detalle es resuelto en los subprogramas.
- Los subprogramas pueden ser planeados, codificados y comprobados independientemente unos de otros.
- Un subprograma puede ser modificado internamente sin afectar al resto de los subprogramas.
- Un subprograma, una vez escrito, pueden ser ejecutado todas las veces que sea necesario a través de una invocación al mismo.
- Una vez que un subprograma se ha escrito y comprobado, se puede utilizar en otro programa (*reusabilidad*).

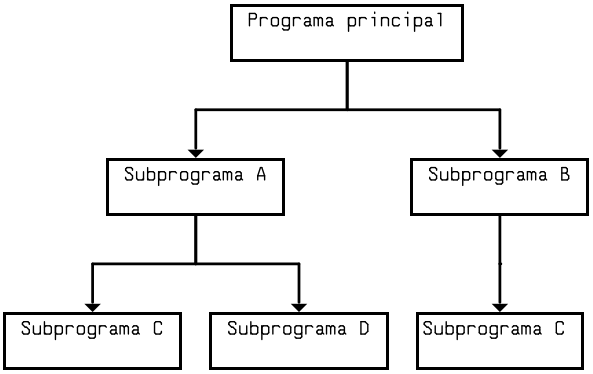

Figura 1. Diagrama de estructura de un algoritmo o programa modularizado.

Figura 2. Representación de un subprograma en un diagrama de flujo.

Funciones y subrutinas. Fortran provee dos maneras de implementar un subprograma: *funciones* y *subrutinas*. Estos subprogramas pueden ser *intrínsecos*, esto es, provistos por el compilador o *definidos por el usuario*. Ya hemos mencionado (y utilizado) las funciones intrínsecas que provee el propio lenguaje. Por otra parte los subprogramas definidos por el usuario pueden ser escritos por el propio usuario o bien formar parte de un paquete o *biblioteca* de subprogramas (*library*, en inglés) desarrollado por terceros. En lo que sigue consideraremos la creación de subprogramas propios. Para ello presentamos, como ejemplo, la implementación de subprogramas apropiados para expresar un ángulo dado en grados, minutos y segundos en radianes y viceversa.

Código 1. Conversión entre el sistema sexagesimal y el sistema circular

```
MODULE trigonometria
  IMPLICIT NONE
  REAL, PARAMETER :: PI = 4.0*ATAN(1.0)

CONTAINS

  REAL FUNCTION grad2rad(grad,min,seg)
    ! --------------------------------
    ! Declaración de argumentos
    ! --------------------------------
    INTEGER, INTENT(IN) :: grad
    INTEGER, INTENT(IN) :: min
    REAL,    INTENT(IN) :: seg
    ! --------------------------------
    grad2rad = grad+min/60.0+seg/3600.0
    grad2rad = grad2rad*(PI/180.0)
    ! --------------------------------
  END FUNCTION grad2rad
```

```
    SUBROUTINE rad2grad(rad,grad,min,seg)
      ! -------------------------------
      ! Declaración de argumentos
      ! -------------------------------
      REAL,    INTENT(IN)  :: rad
      INTEGER, INTENT(OUT) :: grad
      INTEGER, INTENT(OUT) :: min
      REAL,    INTENT(OUT) :: seg
      ! -------------------------------
      ! Variables locales
      ! -------------------------------
      REAL :: ang
      ! -------------------------------
      ang  = (180.0/PI)*rad
      grad = INT(ang)
      ang  = (ang - grad)*60.0
      min  = INT(ang)
      seg  = (ang -min)*60.0
      ! -------------------------------
    END SUBROUTINE rad2grad

  END MODULE trigonometria

  PROGRAM conversion_ang
    USE trigonometria
    IMPLICIT NONE
    INTEGER :: g,m
    REAL    :: s,r
    ! -------------------------------
    ! Leer el ángulo en el sistema
    ! sexagesimal
    ! -------------------------------
    WRITE(*,*) 'Ingrese el ángulo&
     & en grados, minutos y segundos.'
    READ(*,*) g,m,s
    ! -------------------------------
    ! Convertir a radianes
    ! -------------------------------
    r = grad2rad(g,m,s)
    WRITE(*,*) 'Radianes=', r
    ! -------------------------------
    ! Volver al sistema sexagesimal
    ! -------------------------------
    CALL rad2grad(r,g,m,s)
    WRITE(*,*) 'Grados  =',g
    WRITE(*,*) 'Min     =',m
    WRITE(*,*) 'Seg     =',s
    ! -------------------------------
  END PROGRAM conversion_ang
```

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

```
tipo FUNCTION nombre (argumentos)
  declaraciones de los argumentos
  declaraciones de variables locales
  sentencias
  nombre = expresión
END FUNCTION nombre

SUBROUTINE nombre (argumentos)
  declaraciones de los argumentos
  declaraciones de variables locales
  sentencias
END SUBROUTINE nombre
```

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

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

☞ Cada subprograma es por sí mismo una unidad de programa independiente con sus propias variables, constantes literales y constantes con nombres. Por lo tanto cada unidad tiene sus respectivas declaraciones de tipo. Ahora, mientras que el comienzo de un programa principal es declarado con la sentencia `PROGRAM`, el comienzo de una subrutina está dado por la sentencia `SUBROUTINE` y, para una función, por la sentencia `FUNCTION`. Cada una de estas unidades del programa se extiende hasta su respectiva sentencia `END PROGRAM|SUBROUTINE|FUNCTION`, la cual indica su fin lógico al compilador.

☞ Los subprogramas deben tener un nombre que los identifique. El nombre escogido debe seguir las reglas de todo identificador en Fortran. Ahora bien, como una función devuelve un valor *el nombre de una función tiene un tipo de dato asociado* el cual debe, entonces, ser declarado en la sentencia `FUNCTION`. Por el contrario, al nombre de una subrutina no se le puede asignar un valor y por consiguiente *ningún tipo de dato está asociado con el nombre de una subrutina.*

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

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

☞ La forma de compartir información entre una unidad de programa y el subprograma invocado es a través de una *lista de argumentos.* Los argumentos utilizados en la declaración de una subrutina o función son conocidos como *argumentos formales* o *ficticios* y consisten en una lista de nombres simbólicos separados por comas y encerrada entre paréntesis. La lista puede contar con cualquier número de

elementos, inclusive ninguno, pero, por supuesto, no puede repetirse ningún argumento formal. Si no hay ningún argumento entonces los paréntesis pueden ser omitidos en las sentencias de llamada y definición de una subrutina, pero para una función, por el contrario, siempre deben estar presentes. Los argumentos formales pueden ya sea pasar información de la unidad del programa al subprograma (*argumentos de entrada*), del subprograma hacia la unidad de programa (*argumentos de salida*) o bien en ambas direcciones (*argumentos de entrada/salida*). La distinción entre los mismos es especificada a través del atributo `INTENT(`*intención*`)` en la sentencia de declaración del argumento, donde *intención* puede tomar los valores `IN`, `OUT` ó `INOUT`, respectivamente. En particular *en una función los argumentos formales son utilizados solamente como argumentos de entrada*, por lo que los argumentos de la misma siempre deben ser declarados con el atributo `INTENT(IN)`. Nótese que es el nombre de la función en sí mismo el que es utilizado para devolver un valor a la unidad de programa que lo invocó. De este modo, el nombre de una función puede ser utilizada como una variable dentro de la misma y debe ser asignada a un valor antes de que la función devuelva el control.

☞ Los argumentos que aparecen en la invocación de un subprograma son conocidos como *argumentos actuales*. La asociación entre argumentos actuales y formales se realiza cada vez que se invoca el subprograma y de este modo se transfiere la información entre la unidad del programa y el subprograma. *La correspondencia entre los argumentos actuales y los formales se basa en la posición relativa que ocupan en la lista*, no a través de los nombres (esto es, los nombres de variables en los argumentos formales y actuales no deben ser necesariamente los mismos) Además *el tipo de dato de un argumento actual debe coincidir con el tipo de dato del argumento formal correspondiente*. Un argumento formal de salida (y de entrada/salida) es una variable en el subprograma cuyo valor será asignado dentro del mismo y sólo puede corresponderse con un argumento actual que sea una variable (del mismo tipo, por supuesto) en la unidad de programa que invoca al subprograma. Un argumento formal de entrada, por otra parte, es una variable que preserva su valor a través de todo el subprograma y puede corresponderse a un argumento actual de la unidad del programa que pueden ser no sólo una variable sino también a una expresión (incluyendo una constante). Si al argumento actual es una expresión, ésta es evaluada antes de ser transferida.

✏ **Pasaje por referencia.**

En programación existen varias alternativas para implementar la manera en la cual los argumentos actuales y formales son transmitidos y/o devueltos entre las unidades de programa. Fortran, independientemente de si los argumentos son de entrada, salida o entrada/salida, utiliza el *paradigma de pasaje por referencia*. En este método en vez de pasar los valores de los argumentos a la función o subrutina (*pasaje por valor*), se pasa la dirección de memoria de los argumentos. Esto significa que el argumento actual y formal comparten la misma posición de memoria y por lo tanto, cualquier cambio que realice el subprograma en el argumento formal es *inmediatamente* "visto" en el argumento actual. Por este motivo los argumentos que actuarán como datos de entrada deben ser declarados en el subprograma con el atributo `INTENT(IN)`, ya que de este modo cualquier intento de modificar su valor dentro del subprograma originará un error de compilación.

☞ Según la nota anterior, es claro que *la lista de argumentos actuales en la llamada de un subprograma debe corresponderse con la lista de argumentos formales en número, tipo y orden*[1]. Para que el compilador pueda verificar la consistencia de las llamadas a los subprogramas debe hacerse *explícita* las *interfaces* de los mismos, lo cual se logra muy facilmente construyendo un *módulo de procedimientos*. Un *módulo* es una unidad de programa que permite agrupar subprogramas relacionados (y otros datos) para construir una biblioteca de rutinas que podrá ser reutilizada en cualquier otro programa. La forma general de un módulo de procedimientos es:

```
MODULO nombre_del_módulo
  IMPLICIT NONE
CONTAINS
    subprograma 1
    subprograma 2
    ⋮
    subprograma n
END MODULO nombre_del_módulo
```

donde *subprograma 1*, ..., *subprograma n* son funciones y/o subrutinas. Cada uno de estos subprogramas son llamados *subprogramas* o *procedimientos* del módulo. El nombre del módulo sigue las convenciones usuales para cualquier identificador en Fortran. Para que los procedimientos de un módulo resulten accesibles a una dada unidad del programa se debe utilizar la instrucción:

```
USE nombre_del_módulo
```

la cual *debe escribirse inmediatamente después de la identificación de la unidad*, inclusive antes de la sentencia `IMPLICIT NONE`.

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

```
USE nombre_del_módulo, ONLY: subprogramas
```

Por ejemplo, en nuestro código, la sentencia `USE` puede ser reemplazada por `USE trigonometria, ONLY: dec2rad, rad2dec`.

☞ Un módulo puede contener también datos para ser compartidos con otras unidades de programas, ya para los subprogramas dentro del módulo o con la unidad que lo invoca. Estos datos son conocidos como *variables globales*. En nuestro código, por ejemplo, la constante con nombre `PI` está disponible para su uso tanto dentro de los subprogramas del módulo como para el programa principal.

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

☞ Existen diferentes maneras de compilar un programa junto con un módulo. La primera opción, utilizada en nuestro código, es incluir el código fuente del módulo en el mismo archivo del código fuente de la unidad del programa que lo utiliza, justo antes del código de tal unidad de programa. Tenemos así un único archivo `.f90`, el cual es compilado como es usual. Claramente, esta forma de incluir módulos no es flexible. Una segunda opción es escribir el código fuente del módulo en un archivo (digamos `mod.f90`) separado del código fuente del programa que lo utiliza, (guardado, digamos, en el archivo `main.f90`) y compilar ambos en la línea de comandos, *anteponiendo* el archivo del módulo a cualquier otro archivo:

```
$ gfortran -Wall -o exe mod.f90 main.f90
```

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

```
$ gfortran -Wall -c mod.f90
$ gfortran -Wall -c main.f90
$ gfortran -Wall -o exe mod.o main.o
```

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

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

☞ Una manera alternativa, pero menos general, de dar una interfaz explícita a los subprogramas, es definirlos como procedimientos *internos* del programa principal. Esto se logra como se muestra esquemáticamente a continuación

```
PROGRAM principal
  ...
CONTAINS
  SUBROUTINE sub_int()
  ...
  END SUBROUTINE sub_int
  FUNCTION func_int()
  ...
  END FUNCTION func_int
END PROGRAM principal
```

Nótese que los subprogramas internos tienen acceso a todos los datos de la unidad de programa que los contiene.

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

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

Ejercicio 1. En el plano una rotación de ángulo θ alrededor del origen transforma las coordenadas (x, y) de un punto en nuevas coordenadas (x', y') dadas por

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

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

Ejercicio 2. Escribir una subrutina que permita el intercambio de dos variables reales.

Ejercicio 3. Implementar un subprograma apropiado (función o subrutina) para convertir la temperatura de la escala a Fahrenheit a la escala Celsius y viceversa.

Ejercicio 4. Implementar una función para el cálculo del factorial de un número entero positivo, $n!$, el cual es definido en forma recursiva según:

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

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

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

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

Ejercicio 5. Dado dos enteros no negativos n y r con $n \geq r$ implementar una función para el cálculo del coeficiente binomial

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

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

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

$$\binom{n}{r} = \frac{n(n-1)(n-2)\cdots(n-r+1)}{r!} = \left(\left(\left(\left(\frac{n}{1}\right)\frac{(n-1)}{2}\right)\frac{(n-2)}{3}\right)\cdots\frac{(n-r+1)}{r}\right).$$

Ejercicio 6. Escribir una función que devuelva un valor lógico (esto es, verdadero o falso) según un número entero positivo dado es primo o no. *Observación*: Recordar que un entero positivo p es primo si es divisible sólo por si mismo y la unidad. El método más simple para determinar si p es primo consiste en verificar si no es divisible por todos los números sucesivos de 2 a $p-1$. Debido a que el único primo par es el 2, se puede mejorar el método separando la verificación de la divisibilidad por 2 y luego, si p no es divisible por 2, testear la divisibilidad por los enteros *impares* existentes entre 3 y $p-1$.

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

Ejercicio 7. Implementar en Fortran una función para calcular la *traza* de una matriz A de $n \times n$ elementos reales, esto es, $traza(A) = \sum_{i=1}^{n} a_{ii}$. Dicha función sólo debe tener como argumento de entrada la matriz cuya traza quiere calcularse.

Ejercicio 8. Implementar un módulo que permita a cualquier unidad de programa importar los siguientes datos sobre el planeta Tierra: radio medio, radio ecuatorial, volumen, masa, densidad media. Implementar un programa que haga uso de este módulo para calcular el momento de inercia del planeta (*Ayuda*: asuma que el planeta es un cuerpo rígido esférico de radio R y masa M, entonces $I = \frac{2}{5}MR^2$).

Subprogramas como argumentos. Fortran permite que un subprograma sea pasado a otro subprograma en su lista de argumentos. Para tal efecto, el correspondiente argumento formal es declarado a través de un *bloque de interfaz* dentro del subprograma, lo cual permite, entonces, especificar completamente las características del subprograma que será pasado. Nótese que en esta circunstancia, el atributo `INTENT` no tiene relevancia, y por lo tanto no se aplica. La forma de un bloque de interfaz para una subrutina y una función es

```
INTERFACE
  SUBROUTINE nombre_sub(argumentos)
    declaración de argumentos
  END SUBROUTINE nombre_sub
  tipo FUNCTION nombre_func(argumentos)
    declaración de argumentos
  END FUNCTION nombre_func
END INTERFACE
```

Esto es, se declaran las *cabeceras* (del inglés, *header*) de la subrutina o función que utilizará la unidad de programa en cuestión. Nótese que la cabecera incluye solamente el nombre del subprograma y la declaración de los argumentos formales, *no* incluye la declaración de variables locales ni instrucciones ejecutables.

Por ejemplo, la siguiente subrutina estima la derivada de cualquier función $f(x)$ suave en un punto $x = a$ haciendo uso de la aproximación $f'(a) \sim [f(a+h) - f(a-h)]/(2h)$ siendo h un paso pequeño dado.

Código 2. Subrutina para estimar la derivada primera en un punto

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

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

Código 3. Ejemplo del uso de la subrutina derivada

```
PROGRAM main
  USE calculo, ONLY: derivada
  IMPLICIT NONE
  REAL :: a = ATAN(1.0)
  REAL :: h = 0.01
  REAL :: df

  CALL derivada(f1,a,h,df)
  WRITE(*,*) 'Derivada1 = ', df
  CALL derivada(f2,a,h,df)
  WRITE(*,*) 'Derivada2 = ', df
```

```
  CONTAINS
    REAL FUNCTION f1(x)
      REAL, INTENT(IN) :: x
      f1 = sin(x)
    END FUNCTION f1
    REAL FUNCTION f2(x)
      REAL, INTENT(IN) :: x
      f2 = cos(x)
    END FUNCTION f2
  END PROGRAM main
```

Ejercicio 9. Escribir una subrutina que tabule una función arbitraria f sobre un conjunto de $(N+1)$ puntos igualmente espaciados del intervalo $[a, b]$, esto es, que imprima los valores de f en los puntos

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

siendo $h = (b-a)/N$ el *paso* de la tabulación y $x_0 = a$, $x_N = b$. Utilice la subrutina para imprimir la tabla de $f(x) = e^{-x^2}$ y de $f(x) = \cos(x)$ sobre el intervalo $[-1, 1]$ en nueve puntos igualmente espaciados.