Departamento de Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

Apuntes para la asignatura

Informática

Facultad de Ciencias (Matemáticas)

http://www.lcc.uma.es/personal/pepeg/mates

Tema 10. Diseño modular

10.1 Los módulos de Modula-2 2
10.2 Ventajas de la modularización de programas 2
10.3 Módulos de biblioteca 3
10.3.1 Módulos de definición 3
10.3.2 Módulos de implementación 4
10.3.3 Inicialización de un módulo 6
10.3.4 Usos de los módulos de biblioteca 7
10.4. Compilación separada 8

Bibliografía

- Programación 1. José A. Cerrada y Manuel Collado. Universidad Nacional de Educación a Distancia.
- Programming with TopSpeed Modula-2. Barry Cornelius. Addison-Wesley.

Introducción

En este tema estudiamos el sistema de módulos del lenguaje Módula-2. Este mecanismo es esencial para poder escribir programas reales (con cierto tamaño). Además, hace posible la creación de módulos de biblioteca reutilizables.

10.1 Los módulos de Modula-2

En programación, un módulo es un fragmento de un programa que se desarrolla de forma independiente del resto del programa. Esta independencia hace posible un mecanismo de compilación por separado que limita la complejidad del programa que se está desarrollando. Al compilarse el módulo por separado, la persona que lo desarrolla sólo debe preocuparse de él, prescindiendo en parte de cómo se utiliza este módulo dentro del programa. Quien escriba el resto del programa no debe preocuparse de los detalles del módulo sino sólo de cómo utilizarlo.

Los módulos son, probablemente, la característica más innovadora del lenguaje *Modula-2*. Hay cuatro tipos de módulos:

- módulos de programa
- módulos internos
- módulos de definición
- módulos de implementación

Los **módulos de programa** son unidades de programa completas que pueden *importar recursos* (constantes, variables, procedimientos..) de módulos de biblioteca. Son módulos de programa todos los que hemos utilizado hasta el momento.

Los **módulos de definición** e **implementación** son las dos partes que constituyen los módulos de biblioteca.

Los **módulos internos o locales** pueden ser definidos dentro del cuerpo de, bien módulos de programa, bien módulos de implementación, y se usan para ayudar a controlar el ámbito y la visibilidad de los distintos objetos.

Un programa en *Modula-2* consta de un módulo de programa y cualquier número de módulos de biblioteca, de los que el módulo de programa importa entidades (constantes, tipos, variables, funciones y procedimientos). Estos módulos de biblioteca pueden a su vez importar de otros módulos de biblioteca.

Cada sistema *Modula-2* suministra un conjunto de bibliotecas básico. El programador tiene, además, la posibilidad de crear módulos de biblioteca adicionales para usar en uno o más programas.

Los módulos usan **listas de importación** para indicar los recursos externos que van a utilizar.

10.2 Ventajas de la modularización de programas

Los módulos son una herramienta indispensable para crear programas reales cuyo tamaño sea grande. Un programador debe diseñar un programa grande como un conjunto de módulos, relacionados entre sí mediante interfaces definidas apropiadamente. Escribir y depurar el programa es más fácil porque el programador puede trabajar con un módulo cada vez, usando los servicios facilitados por otros módulos pero ignorando los detalles de cómo estos módulos trabajan (*principio de abstracción*). Este tipo de **diseño modular** es particularmente

necesario cuando el programa se está desarrollando entre un conjunto de programadores, que es lo más habitual en cualquier programa de cierta envergadura.

Los programas modulares son más **fáciles de modificar**. Puesto que los detalles de implementación de un módulo se ocultan a los demás, se pueden cambiar detalles de un módulo sin afectar al resto.

La modularización hace los **programas mas portables**. El programador puede ocultar los detalles dependientes de la máquina en un único módulo, de forma que cuando se transporte a otro ordenador, sólo debe preocuparse de reescribir dicho módulo.

Cada módulo se compila por separado. El programa se divide en trozos, que el compilador puede procesar separadamente. De esta forma, un cambio en un módulo sólo requiere volver a compilar dicho módulo, no el programa completo.

El desarrollo de **bibliotecas con código reutilizable** conlleva no sólo un ahorro de trabajo, sino además un aumento de la fiabilidad del programa, pues dichas bibliotecas están más probadas que si la parte de la biblioteca que se usa se codifica de nuevo.

10.3 Módulos de biblioteca

Los módulos de biblioteca se usan para exportar recursos a otros módulos. El programador que utiliza estos módulos sólo necesita conocer cuales son los recursos disponibles pero no cómo están implementados. Un módulo de biblioteca consta de dos partes, la **parte de definición** y la **parte de implementación**. La parte de definición contiene sólo las definiciones de entidades que el módulo exporta; define la interfaz del módulo de biblioteca con el resto del programa, informa al programador de los recursos existentes en la biblioteca. La parte de implementación describe cómo implementar los servicios indicados en la parte de definición. Para resaltar que cada parte es por sí sola una unidad de programa, la primera es llamada DEFINITION MODULE, y la segunda IMPLEMENTATION MODULE. Cada una está contenida en un fichero. En *TopSpeed Modula-2*, el nombre del fichero que contiene la parte de definición debe tener la extensión .DEF, mientras que el fichero que almacena la parte de implementación deberá tener la extensión .MOD.

10.3.1 Módulos de definición

Un módulo de definición contiene *definiciones* de constantes, tipos, variables y subprogramas. Las definiciones son similares a las declaraciones, pero pueden tener menos información.

- **Constantes**

Son idénticas a las declaraciones de constantes normales.

- **Tipos.**

Podemos definir tipos de dos formas. Una posibilidad es usar una declaración de tipo normal:

```
TYPE
  COMPLEJO =  RECORD
                PReal, PImag : REAL;
              END;
```

En este caso se llama **tipo transparente** y la estructura del tipo es visible en los módulos que lo importen. Un módulo que importe el tipo COMPLEJO, sabe que el tipo está definido como un registro con dos campos cuyos nombres son campos `PReal` y `PImag`, y puede acceder a éstos.

Otra alternativa consiste en omitir el signo de igualdad y la especificación del tipo:

```
TYPE
```

```
    COMPLEJO;
```

A un tipo definido de esta forma se le llama **tipo opaco**. Un módulo que lo importe no conoce su estructura. En este caso, sólo se pueden efectuar operaciones de asignación o de comparación de igualdad y desigualdad sobre variables declaradas de dicho tipo opaco, además de las operaciones definidas en el módulo de definición para dicho tipo. La definición completa del tipo opaco debe aparecer en el módulo de implementación y deberá ser un tipo puntero (ver tema 12).

- **Variables**.

Las definiciones de variables son idénticas a las declaraciones de variables normales:

```
VAR
  EsCorrecto : BOOLEAN;
```

- **Subprogramas**.

La definición de un subprograma consta de la cabecera del mismo. La declaración completa debe aparecer en el módulo de implementación.

Un módulo de definición puede incluir también listas de importación; sólo debe importar aquellas entidades externas (normalmente tipos) que se usen en el módulo de definición. Es un error común importar aquí entidades que sólo son necesarias en la parte de implementación.

La forma exacta de escribir un módulo de definición se describe mediantes las siguiente reglas BNF:

Módulo_definición **::=**	Cabecera_definición **{**Definición_de_elementos **}** END Identificador.
Cabecera_definición **::=**	DEFINITION MODULE Identificador;
	{ Lista_importados ; **}**
	[Lista_exportados ;]
Definición_de_elementos**::=**	Declaración_de_constantes **|**
	TYPE **{** lidentificador **[** = Esquema_de_tipo]; **}|**
	Declaración_de_variables **|** Cabecera_subprograma;

10.3.2 Módulos de implementación

Un módulo de implementación tiene la misma sintaxis que un módulo de programa, salvo que comienza con las palabras **IMPLEMENTATION MODULE**, en lugar de **MODULE**.

Al igual que los módulos de programa, los módulos de implementación pueden contener tanto listas de importación como declaraciones. Mediante listas de importación un módulo de implementación debe importar las entidades de otros módulos de biblioteca que utilice.

Las declaraciones sirven para dos propósitos:

1) Especificar los aspectos indefinidos en el módulo de definición.
2) Declarar entidades que sean estrictamente locales al módulo de implementación.

Cualquier entidad declarada en la parte de definición de un módulo de biblioteca es visible en la parte de implementación.

En *TopSpeed Modula-2*, las entidades importadas en la parte de definición son visibles en la parte de implementación automáticamente aunque esto no es un estándar del lenguaje *Modula-2*.

En un módulo de implementación pueden aparecer declaraciones de cinco clases de entidades:

- **Constantes**.

Las constantes definidas en la parte de definición son visibles en la parte de implementación de forma automática. Todas las constantes declaradas en un módulo de implementación son locales a dicho módulo; no son visibles fuera del módulo de implementación.

- **Tipos.**

Los tipos definidos en la parte de definición del módulo son visibles automáticamente en la parte de implementación. Si en la parte de definición aparece la especificación del tipo, no tiene que aparecer una declaración del mismo en la parte de implementación, sin embargo, si se omitió dicha especificación (se definió como tipo opaco), debe aparecer una declaración completa en la parte de implementación. Esta declaración debe ser obligatoriamente de tipo POINTER (que será estudiado en un tema posterior). Pueden aparecer, además, declaraciones de nuevos tipos, en cuyo caso sólo son visibles dentro de la parte de implementación.

Ejemplo. Si el tipo COMPLEJO ha sido definido como opaco en el módulo de definición, habría que declarar:

```
TYPE
  COMPLEJO =  POINTER TO RECORD
                PReal, PImag: REAL
              END;
```

- **Variables.**

Las variables declaradas en la parte de definición no pueden volver a ser declaradas en la parte de implementación puesto que son visibles en ésta automáticamente. Todas las variables declaradas en la parte de implementación son locales a ésta.

- **Subprogramas.**

La parte de implementación de un módulo debe contener una declaración completa de todos los subprogramas cuya cabecera aparece en la parte de definición. Hay que respetar estrictamente la forma en que se definió la cabecera del procedimiento en la parte de definición, es decir, hay que declarar los mismos parámetros y tipos para ellos. Además, pueden declararse otros procedimientos que serán locales al módulo de implementación.

- **Módulos.**

Dentro de un módulo de implementación pueden declararse otros módulos. A estos módulos se los llama *módulos internos*. No los estudiaremos en este curso.

Los módulos de biblioteca pueden tener un cuerpo, cuyas instrucciones se ejecutan al comienzo de la ejecución del programa (sobre esta cuestión insisteremos más adelante).Todas las variables declaradas en el nivel más externo de un módulo de biblioteca retienen sus valores durante todo el tiempo de vida del programa, a diferencia de lo que ocurre con las variables declaradas en un subprograma. Por tanto, sólo es necesario inicializar una vez las variables pertenecientes a un módulo de biblioteca (cuando el programa comienza la ejecución).

En el siguiente ejemplo, se define un módulo de biblioteca con operaciones para la determinación de la posición del máximo y el mínimo elemento de un array de números reales de mil elementos.

MaxMin.Def

```
DEFINITION MODULE MaxMin;
  TYPE (* Tipo Transparente *)
    ARRAYREAL = ARRAY [1..1000] OF REAL;

  PROCEDURE MaxPos(A : ARRAYREAL): CARDINAL;
  PROCEDURE MinPos(A : ARRAYREAL): CARDINAL;
END MaxMin.
```

MaxMin.Mod

```
IMPLEMENTATION MODULE MaxMin;

PROCEDURE MaxPos(A : ARRAYREAL):CARDINAL;
VAR
  Indice : CARDINAL;
  PosMax : [1..1000];
BEGIN
  PosMax := 1;
  FOR Indice := 2 TO 1000 DO
    IF A[Indice] > A[PosMax] THEN
      PosMax := Indice
    END
  END;
  RETURN PosMax
END MaxPos;

PROCEDURE MinPos(A : ARRAYREAL):CARDINAL;
VAR
  Indice : CARDINAL;
  PosMin : [1..1000];
BEGIN
  PosMin := 1;
  FOR Indice := 2 TO 1000 DO
    IF A[Indice] < A[PosMin] THEN
      PosMin := Indice
    END
  END;
  RETURN PosMin
END MinPos;

END MaxMin.
```

10.3.3 Inicialización de un módulo

Los módulos de implementación pueden contener código para su inicialización. Este código es ejecutado una sola vez, antes de ejecutar el código del módulo cliente (aquel que hace uso de él). En el caso de que varios módulos de biblioteca contengan código para su inicialización, cada bloque de código de inicialización es ejecutado sólo una vez en la secuencia dada por la lista de importación.

Como ejemplo, consideremos los siguientes módulos de biblioteca y el siguiente módulo de programa:

```
DEFINITION MODULE biblioteca1;
END biblioteca1.

IMPLEMENTATION MODULE biblioteca1;
  FROM IO IMPORT WrStr, WrLn;
```

```
BEGIN
  WrStr("Dentro de Biblioteca 1");
  WrLn;
END biblioteca1.

DEFINITION MODULE biblioteca2;
END biblioteca2.

IMPLEMENTATION MODULE biblioteca2;
  FROM IO IMPORT WrStr, WrLn;
BEGIN
  WrStr("Dentro de Biblioteca 2");
  WrLn;
END biblioteca2.

DEFINITION MODULE biblioteca3;
END biblioteca3.

IMPLEMENTATION MODULE biblioteca3;
  FROM IO IMPORT WrStr, WrLn;
BEGIN
  WrStr("Dentro de Biblioteca 3");
  WrLn;
END biblioteca3.

MODULE principal;
  FROM IO IMPORT WrLn, WrStr;
  IMPORT Biblioteca3;
  IMPORT Biblioteca1;
  IMPORT Biblioteca2;
BEGIN
  WrLn;
  WrStr("Dentro del modulo principal");
END principal.
```

La sálida por pantalla que resulta al ejecutar el módulo principal es:

```
Dentro de Biblioteca 3
Dentro de Biblioteca 1
Dentro de Biblioteca 2
Dentro del modulo principal.
```

10.3.4 Usos de los módulos de biblioteca

El concepto de módulo está muy ligado al concepto de abstracción. Un módulo debe definir un elemento abstracto o varios relacionados entre sí. El diseño del módulo debe permitir su uso de modo que se sepa qué hace cada parte sin que sea necesario saber cómo está implementada.

Se pueden construir módulos de biblioteca con las siguientes finalidades:

1) **Agrupar una cierta cantidad de identificadores de constantes relacionadas**. Por ejemplo podríamos construir un módulo de biblioteca que reuniera las principales constantes matemática (π, e,...). En este caso, solo aparecen definiciones en el módulo de defínción de la biblioteca. El módulo de implementación queda vacío.

2) **Agrupar subprogramas relacionados**. Por ejemplo la biblioteca MATHLIB agrupa las funciones matemáticas de uso más frecuente (seno, coseno, raíz cuadrada...). En este caso, sólo aparecen definiciones de subprogramas en el módulo de definición. Las correspondientes implementaciones aparecen en el módulo de implementación.

3) **Definir un tipo abstracto de datos**. Se utiliza la biblioteca para definir un nuevo tipo y las operaciones que se pueden realizar con él. Ejemplo: biblioteca FIO. En este caso la

interfaz del módulo equivale a la especificación del tipo abstracto de datos, y la realización del módulo establece su representación interna. En el módulo de definición aparece la definición de tipo y los subprogramas asociados.

4) **Ocultar una única variable de un tipo abstracto (dato encapsulado).** El módulo constituye una cápsula que protege al dato de manera que sólo se puede acceder a él usando las operaciones de la interfaz. En el módulo de definición sólo aparecen definiciones de subprogramas. En el módulo de implementación hay una variable global (el dato encapsulado). Los subprogramas acceden al dato encapsulado mediante efectos laterales. Este es uno de los pocos casos en los que es lícito usar efectos laterales, ya que éstos quedan aislados a los subprogramas del módulo.

El siguiente ejemplo muestra un módulo en el que se define un dato encapsulado:

```
DEFINITION MODULE Autor;
  PROCEDURE Leer;
  PROCEDURE Escribir;
END Autor.

IMPLEMENTATION MODULE Autor;
IMPORT IO;
VAR autor:ARRAY [0..20] OF CHAR;
PROCEDURE Leer;
BEGIN
  IO.RdStr(autor)
END Leer;

PROCEDURE Escribir;
BEGIN
  IO.WrStr(autor)
END Escribir;

END Autor.

MODULE Principal;
IMPORT Autor;
BEGIN
  Autor.Leer;
  Autor.Escribir
END Principal.
```

10.4. Compilación separada

Modula-2 permite compilar módulos de programa y de biblioteca de forma separada. El compilador chequea que cada entidad importada de un módulo de biblioteca sea usada de forma consistente con su definición en el módulo (para ello usa la información contenida en la parte de definición del módulo de biblioteca).

Una de las ventajas proporcionadas por la compilación separada es que cuando cambian algunos de los módulos de un programa, sólo es necesario recompilar los módulos afectados, y no todos los módulos del programa.

Muchos sistemas *Modula-2* requieren que se compilen los módulos de definición; sin embargo, en *TopSpeed* sólo necesitan ser compilados los módulos de implementación y de programa. La compilación de cada fichero con extensión .MOD genera un fichero con el mismo nombre y extensión .OBJ. Los ficheros .DEF no producen ficheros .OBJ ya que no son compilados.

Si un programa consta de varios ficheros con extensión .MOD (un módulo de programa y varios módulos de implementación) se puede seguir cualquier orden a la hora de compilar los ficheros siempre que se respete la siguiente norma:

El módulo de definición de una librería debe existir antes de que pueda compilarse su correspondiente módulo de implementación o cualquier otro módulo que importe de la librería.

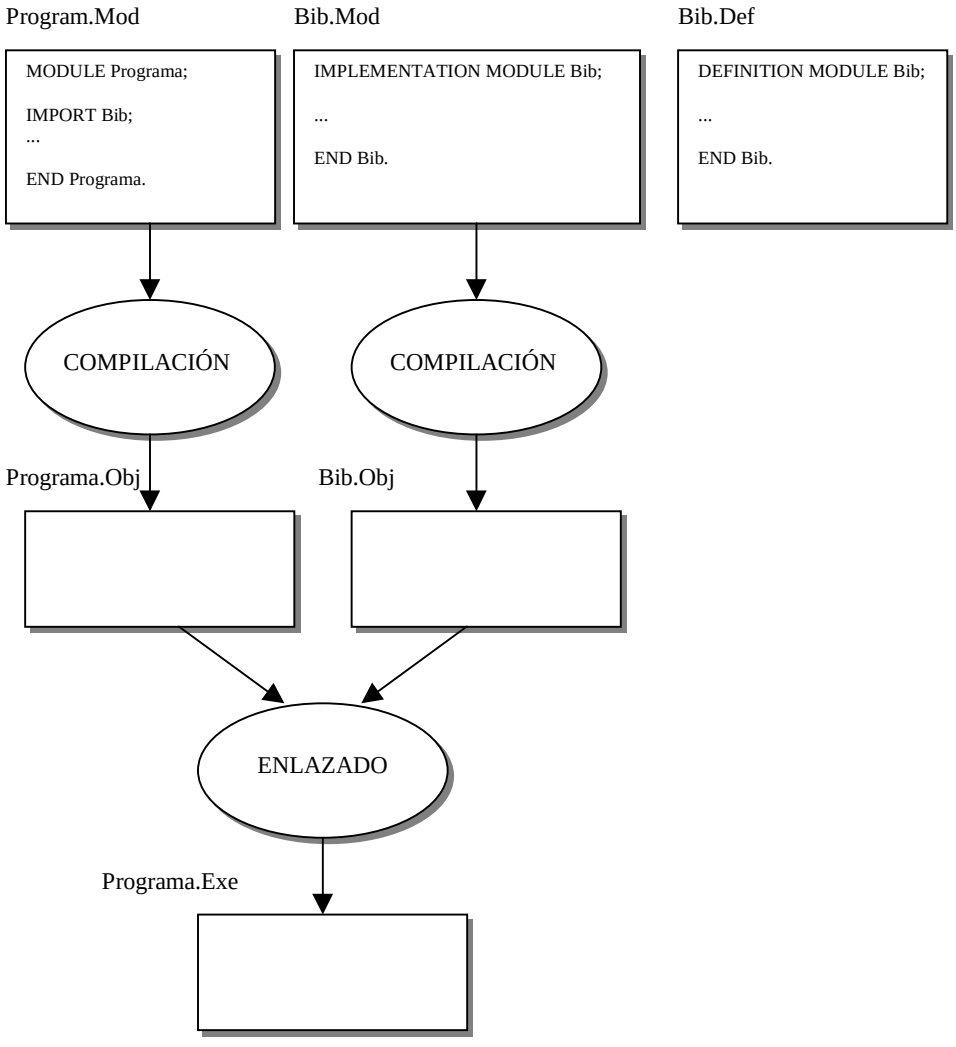
Una vez generados todos los ficheros .OBJ, hay que enlazarlos. El proceso de enlazado genera un fichero ejecutable con el nombre del módulo de programa y con extensión .EXE.

Si se modifica uno de los ficheros que forman el programa, no es necesario volver a compilar todos los ficheros de nuevo. Las normas son:

- Si un módulo de programa o un módulo implementación es modificado, sólo él debe ser recompilado. A continuación basta con repetir el proceso de enlazado para obtener un nuevo fichero ejecutable actualizado.
- Si un módulo de definición es modificado, su correspondiente módulo de implementación y todos los módulos que importen la biblioteca deben ser recompilados. Es necesario también repetir el proceso de enlazado.

En *TopSpeed Modula-2* todo este proceso está automatizado mediante la opción **Make** del menú. Cuando esta opción es seleccionada, el sistema recompila los módulos necesarios y, además, realiza el proceso de enlazado.

El siguiente ejemplo muestra el proceso de compilación y enlazado para un programa formado por un módulo de programa (Programa.Mod) y un módulo de biblioteca (Bib.Mod y Bib.Def):



Relación de Problemas (Tema 10)

1. Escribe una biblioteca `MatConst` que exporte las siguientes constantes matemáticas:

`PI`: relación entre el radio y la circunferencia de un círculo.

`E`: el valor de la función exponencial en el punto 1.0.

`LN10`: valor de la función logaritmo neperiano en el punto 10.

2. Diseña una biblioteca que exporte el tipo de dato `COMPLEJO` y operaciones para construir, sumar, restar, multiplicar, dividir, leer de teclado y escribir por pantalla de valores de dicho tipo. El módulo de definición será el siguiente:

```
DEFINITION MODULE Complejo;

TYPE
  COMPLEJO =  RECORD
                PReal  : LONGREAL;
                PImag  : LONGREAL
              END;
PROCEDURE RdComplejo () : COMPLEJO;
PROCEDURE WrComplejo (C : COMPLEJO);
PROCEDURE AsignaRectangular (Real, Imag: LONGREAL)  : COMPLEJO;
PROCEDURE AsignaPolar (Modulo, Argumento: LONGREAL) : COMPLEJO;
PROCEDURE Suma       (C1, C2 : COMPLEJO) : COMPLEJO;
PROCEDURE Resta      (C1, C2 : COMPLEJO) : COMPLEJO;
PROCEDURE Producto   (C1, C2 : COMPLEJO) : COMPLEJO;
PROCEDURE Division   (C1, C2 : COMPLEJO) : COMPLEJO;
PROCEDURE SonIguales (C1, C2 : COMPLEJO) : BOOLEAN;

END Complejo.
```

La función `AsignaRectangular` toma dos números reales, correspondientes a la parte real e imaginaria de un número complejo, y devuelve un número complejo con dichas componentes. La función `AsignaPolar` es similar, pero construye el número a partir de su módulo y argumento. La función `SonIguales` devuelve `TRUE` si sus dos argumentos corresponden al mismo valor complejo.

Prueba la biblioteca construida con el siguiente programa:

```
MODULE Ej1002;
IMPORT Complejo;
FROM MatConst IMPORT PI;
FROM IO IMPORT WrLn;
VAR
  a, b, c, Suma, Resta, Producto, Division : Complejo.COMPLEJO;

BEGIN
  a := Complejo.AsignaRectangular(4.0, 5.0);
  b := Complejo.AsignaPolar(1.0, PI);
  c := Complejo.RdComplejo();
  Complejo.WrComplejo(a); WrLn;
  Complejo.WrComplejo(b); WrLn;
  Complejo.WrComplejo(c); WrLn;
  Suma     := Complejo.Suma(a, c);
  Resta    := Complejo.Resta(a, c);
  Producto := Complejo.Producto(a, b);
  Division := Complejo.Division(c, b);
  Complejo.WrComplejo(Suma);     WrLn;
  Complejo.WrComplejo(Resta);    WrLn;
  Complejo.WrComplejo(Producto); WrLn;
```

```
   Complejo.WrComplejo(Division)
END Ej1002.
```

3. Se desea escribir una función que devuelva un número real aleatorio entre 0 y 1 cada vez que se llame.

Aunque no es fácil generar mediante un programa de ordenador números realmente aleatorios, existen técnicas para la generación de números pseudo-aleatorios. Una de estas técnicas consiste en utilizar los valores obtenidos al ejecutar repetidamente la sentencia:

Semilla := (a * Semilla + c) **MOD** m;

Donde `a`, `c` y `m` son valores enteros. No es fácil escoger valores apropiados para `a`, `c` y `m`, pero se puede demostrar que si el valor inicial de `Semilla` es menor que `m` y se cumplen las siguientes igualdades:

$$a = 4c + 1$$

$$c = 2x + 1$$

$$m = 2^y$$

donde *x* e *y* son cualquier valor entero, los valores asignados a `Semilla` comienzan a repetirse cíclicamente tras `m` valores y todos los valores en el rango [0..m-1] aparecen exactamente una vez en cada ciclo.

Por ejemplo, si escogemos `y` = 11 y `x` = 3, tenemos que `m` = 2048, `c` = 7 y `a` = 29. Al repetir la sentencia

Semilla := (29 * Semilla + 7) **MOD** 2048

cada uno de los valores asignados a `Semilla` será un número pseudo-aleatorio en el rango [0..2047]. Para obtener números aleatorios entre 0 y 1 basta con dividir los sucesivos valores de `Semilla` por 2047.

Escribir un módulo de biblioteca para la generación de números pseudo aleatorios según el siguiente módulo de definición y utilizando el método explicado:

```
DEFINITION MODULE NumAleat;

   PROCEDURE EstablecerSemilla (NuevaSemilla : CARDINAL);
   PROCEDURE NumeroAleatorio () : REAL;

END NumAleat.
```

La función `NumeroAleatorio` devuelve un número real entre 0 y 1 cada vez que se llama. En el módulo de implementación se definirá una variable global `Semilla` que se actualizará, según la fórmula vista, con cada llamada a la función `NumeroAleatorio`. La función `EstablecerSemilla` permite cambiar el valor inicial de la semilla.

NOTA: Obsérvese que tanto `NumeroAleatorio` como `EstablecerSemilla` acceden a la variable `Semilla` mediante un efecto lateral. Una variable global encapsulada dentro de un módulo de biblioteca es una de las pocas circunstancias en las cuales es lícito el acceso a una variable mediante un efecto lateral.

4. Desarrolla una biblioteca que permita trabajar con cadenas de caracteres implementadas según el criterio habitual en *Modula-2*. Las cadenas se representan mediante arrays de caracteres, con terminador `CHR(0)`, si la longitud de la cadena es menor al tamaño del array. La biblioteca se llamará `Cadenas` y debe construirse de manera que el programa adjunto funcione correctamente, por lo que debe definir las siguientes operaciones:

- Una función llamada `Longitud` que toma como parámetro una cadena de caracteres, y

devuelve su longitud (número de caracteres de la cadena que contiene).

- Una función llamada `Buscar`, que toma como parámetros dos cadenas de caracteres y permite buscar la primera cadena dentro de la segunda. Para ello, devuelve la posición de la primera ocurrencia de la primera cadena en la segunda (o el valor –1 si no aparece).
- Un procedimiento `Sustituir`, que toma como parámetros dos cadenas de caracteres y sustituye la primera ocurrencia de la primera cadena en la segunda (si está) por asteriscos.
- Un procedimiento llamado `Borrar`, que toma como parámetros dos cadenas de caracteres y elimina de la primera cadena todas las ocurrencias de la segunda cadena.
- Un procedimiento llamado `WrCadena`, que toma como parámetro una cadena de caracteres, y escribe por pantalla los caracteres que la forman (no se puede usar `IO.WrStr`).
- Un procedimiento llamado `RdCadena`, que lee de teclado una cadena de caracteres y la almacena en la variable que toma como parámetro. (no se puede usar `IO.RdStr`).
- Un procedimiento llamado `Insertar`, que toma tres parámetros, el primero una cadena de caracteres, el segundo un valor de tipo CARDINAL, y el tercero una cadena de caracteres, y realiza la inserción de la primera cadena en la segunda a partir de la posición indicada por el segundo parámetro. En caso de que el segundo parámetro haga referencia a una posición no válida, el procedimiento no hace nada.

NOTA: Los argumentos que representan cadenas de caracteres se pasan como arrays abiertos. La primera letra dentro de una cadena tiene como posición asociada el valor cero.

```
MODULE Ej1004;
FROM Cadenas IMPORT RdCadena, WrCadena, Longitud, Buscar,
           Sustituir, Insertar, Borrar;
FROM IO IMPORT RdLn, RdChar, WrLn, WrInt;
VAR
  C1   : ARRAY [1..20] OF CHAR;
  C2   : ARRAY [1..10] OF CHAR;
  Fin  : CHAR;
  Pos  : INTEGER;
BEGIN
  REPEAT
    WrCadena("Introduzca una cadena de caracteres: "); WrLn;
    RdCadena(C1);
    WrCadena("Se ha leido la cadena");
    WrCadena(C1); WrLn;
    WrCadena("La longitud de la cadena leida es: ");
    WrCard(Longitud(C1), 0); WrLn;
    WrCadena("Introduzca una cadena a buscar:");
    RdCadena(C2);
    Pos := Buscar(C2,C1);
    IF Pos = -1 THEN
      WrCadena("El patron NO aparece en la cadena");
    ELSE
      WrCadena("Encontrado en la posición:");
      WrInt(Pos,0)
    END;
    WrLn;
    Sustituir(C2,C1);
    WrCadena("Tras Sustituir, la cadena inicial queda:");
    WrCadena(C1); WrLn;
    Borrar(C1,"ABC");
    WrCadena("Despues de borrar ABC, la cadena queda:");
    WrCadena(C1); WrLn;
```

```
      Insertar("DDD", 3, C1);
      WrCadena("Despues de insertar DDD, la cadena queda:");
      WrCadena(C1); WrLn;
      WrCadena("Desea repetir el proceso (S/N)?: ");
      Fin := RdChar(); RdLn;
   UNTIL CAP(Fin) = 'N'
END Ej1004.
```

5. Diseña una biblioteca `Punto` que exporte el tipo de datos `PUNTO`, que representa un punto del plano bidimensional. La biblioteca debe exportar, además, operaciones para

- asignar valor a un punto,
- escribir un punto en pantalla
- leer un punto de teclado
- comparar dos puntos
- desplazar un punto según la dirección determinada por el origen de coordenadas y otro punto especificado como parámetro
- girar un cierto ángulo expresado en radianes el vector determinado por el origen y otro punto (especificado como parámetro) en el sentido contrario a las agujas del reloj

NOTA: Representar el punto bidimensional mediante un tipo registro con campos las componentes `x` e `y` del punto.

6. Sabemos que un conjunto es una colección desordenada de elementos en la que no existen elementos duplicados. En programación, los elementos de un conjunto tienen todos el mismo tipo al que se le llama tipo base del conjunto. En *TopSpeed Modula-2*, el tipo base de un conjunto puede ser CARDINAL, SHORTCARD, CHAR, un tipo enumerado, o un subrango de cualquiera de estos tipos. Esta restricción limita bastante el uso de los tipos conjuntos. Se desea desarrollar un programa que trabaje con conjuntos de puntos del plano bidimensional, para lo cual es necesario disponer conjuntos con dicho tipo base.

Diseña el tipo de datos `CONJUNTOPUNTOS` (conjunto de puntos del plano bidimensional), construyendo para ello una biblioteca denominada `ConjPunt` que ofrezca, además del tipo `CONJUNTOPUNTOS`, las siguientes operaciones:

`Vacio`: devuelve un conjunto vacío de este tipo.

`Pertenece(p,c)`: devuelve `TRUE` si el punto `p` es un elemento del conjunto `c`, o `FALSE` en otro caso.

`Incluye(p,c)`: incluye el punto `p` en el conjunto `c`. El subprograma debe comprobar previamente si el elemento ya estaba, en cuyo caso no hace nada.

`Excluye(p,c)`: excluye el punto `p` del conjunto `c`. Si el punto no está en el conjunto se escribirá en pantalla un mensaje indicándolo y no se modificará el conjunto `c`.

`WrConjunto(c)`: muestra por pantalla todos los elementos del conjunto `c`.

`Union(c1,c2)`: devuelve un nuevo conjunto que es la unión de los conjuntos `c1` y `c2`. Los conjuntos `c1` y `c2` no se modifican.

`Interseccion(c1,c2)`: devuelve un nuevo conjunto que es la intersección de los conjuntos `c1` y `c2`. Los conjuntos `c1` y `c2` no se modifican.

`Diferencia(c1,c2)`: devuelve un nuevo conjunto que es la diferencia `c1-c2`. Los conjuntos `c1` y `c2` no se modifican.

`Incluido(c1,c2)`: devuelve `TRUE` si el conjunto `c1` está incluido en el conjunto `c2`.

`SonIguales(c1,c2)`: devuelve `TRUE` si el conjunto `c1` coincide con el conjunto `c2`.

Téngase en cuenta que el orden de los elementos no es relevante para determinar la igualdad de los dos conjuntos.

NOTA: Para simplificar, consideraremos que el cardinal máximo de cualquier variable del tipo `CONJUNTOPUNTOS` será 20 elementos. El tipo `PUNTO` debe importarse de la biblioteca desarrollada en el problema anterior.

Escribe un programa de prueba para la biblioteca desarrollada.

Relación de Problemas complementarios (Tema 10)

7. Escribe una biblioteca, similar a la del ejercicio 2, en la que se defina el tipo `RACIONAL` y operaciones para asignar, operar, comparar, leer del teclado y escribir por pantalla valores de dicho tipo. Prueba la biblioteca con un programa que la use. La función de comparación devolverá un valor en el rango [-1..1] (-1 si el primer argumento es menor al segundo, 0 si son iguales y +1 si el primero es mayor al segundo).

8. Escribe una biblioteca que exporte las funciones trigonométricas `Seno` y `Coseno`. Para evaluarlas suma las siguientes series hasta que los términos obtenidos sean menores a una milésima.

$$\text{Seno}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$\text{Coseno}(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Dado que estas series convergen cuando $x \to 0$, deberán sumarse para valores de x en el rango $0 \le x \le \pi/2$. Si los argumentos no se encuentran en este rango, se deberán utilizar las siguientes igualdades:

$$\begin{aligned}
\text{Seno}\ (x) &\equiv \text{Seno}\ (\pi - x) && si\ \pi/2 \le x \le \pi \\
\text{Seno}\ (x) &\equiv -\text{Seno}\ (x - \pi) && si\ \pi \le x \le 3\pi/2 \\
\text{Seno}\ (x) &\equiv -\text{Seno}\ (2\pi - x) && si\ 3\pi/2 \le x \le 2\pi \\
\text{Coseno}\ (x) &\equiv -\text{Coseno}\ (\pi - x) && si\ \pi/2 \le x \le \pi \\
\text{Coseno}\ (x) &\equiv -\text{Coseno}\ (x - \pi) && si\ \pi \le x \le 3\pi/2 \\
\text{Coseno}\ (x) &\equiv \text{Coseno}\ (2\pi - x) && si\ 3\pi/2 \le x \le 2\pi
\end{aligned}$$

9. Escribe una biblioteca estadística en la cual se definan las siguientes operaciones, que toman como argumento un array abierto de valores reales:

`Maximo`: devuelve la medida de valor máximo del array que toma como argumento.

`Minimo`: devuelve la medida de valor mínimo del array que toma como argumento.

`Media`: devuelve la media de los valores del array que toma como argumento.

`DesviacionTipica`: devuelve la desviación típica de los valores del array que toma como argumento.

`Varianza`: devuelve la varianza de los valores del array que toma como argumento.

`Mediana`: devuelve la mediana de los valores del array que toma como argumento.

`Moda`: devuelve la moda de los valores del array que toma como argumento. Si ésta no es única escribirá un mensaje de aviso en pantalla y devolverá la menor de las modas.

Para este ejercicio calcularemos la mediana como:

- el valor que ocupa la posición (N+1) /2 en el array una vez ordenado si el número de datos (N) es impar,
- la media aritmética de los valores que ocupan las posiciones N/2 y (N+1)/2 en el array una vez ordenado si el número de datos es par .