PROGRAMACIÓN FUNCIONAL EN C#

DEFINICIÓN

- La programación funcional es un paradigma de programación declarativa basado en el uso de funciones matemáticas, en contraste con la programación imperativa, que enfatiza los cambios de estado mediante la mutación de variables.

The “Big Picture”

- Podemos decir que se basa en los siguientes pilares:
  - Lambda-Cálculo o (Cálculo Lambda).
  - Clausuras (Closures).
  - Teoría de categorías y en específico patrones mónadicos basados en funtores.
  - Recursividad.

LAMBDA-CÁLCULO

Definición

- Uno de los pilares de la programación funcional.
- Formulado por el matemático-lógico norteamericano Alonzo Church consiste en "...un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión…".
- Se basa en una definición de función alternativa a la tradicional que dentro del contexto de las ciencias de computación denominaremos…

  Expresión Lambda.

BASES DEL LAMBDA-CÁLCULO - I

Definición De Expresión Lambda

- En matemática tradicional las funciones se representan con un nombre f o g
  f(x)=x+2 Un parámetro.
  pow(x,y)=x^y Dos parámetros.

- En el lambda-cálculo esta misma función se expresaría sin nombre nombre solo los parámetros que entran y la expresión que la representa.
  λx.x+2 Un parámetro (definición).
  (λx.x+2) 10 = 12 Un parámetro (aplicación de una valor).
  λ(x,y). x^y Dos parámetros.

- En ciencias de computación se expresará a través de una función anónima o expresión lambda.
  x → x+2 Un parámetro (definición).
  (x → x+2)(10) = 12 Un parámetro (aplicación de un valor).
  (x,y) → x^y Dos parámetros.

EXPRESIONES LAMBDA EN C#

Usos

- Una expresión lambda en C# es un método anónimo (sin nombre) que se ha de declarar/definir y asignar inmediatamente, sobre una instancia de un delgado.

Las expresiones lambda pueden ser usadas en estos escenarios:

- Argumentos de otras funciones.
- Tipo de retorno de una función.
- Expresiones LINQ.
- Asignación a instancias de delegados genéricos.

EXPRESIONES LAMBDA EN C#

Declaración

- La representación general de una expresión lambda sigue este formato

```
(parámetros) => sentencia o {bloque con return si función}
```

- Descripción puntual:
  - **(parámetros)** Lista de parámetros separada por comas.
    - Se debe dejar los paréntesis vacíos () si no hay parámetros.
    - Si solo hay un parámetro los paréntesis se pueden obviar.
  - **=>** Este es el operador que se encarga de mapear los parámetros expresión al conjunto de sentencias de bloque.
  - Puesto que debe asignarse a un delegado los tipos de los parámetros de pueden deducir a partir de la declaración del mismo.

```
// x es double y retorna double
Func<double, double> cubo = (x) => x * x * x;

// x e y son enteros y retorna entero
Func<int, int, int> suma = (x, y) => x + y;
```

CLAUSURAS EN C# - I

Variables Externas

- En el cuerpo de expresión de una expresión lambda podemos incluir referencias a variables locales y los parámetros de un método.
- En el contexto de una expresión lambda, a estas referencias se les conoce como variables externas.

```
int factor = 3;
Func<int, int> producto = n => n * factor;
Console.WriteLine(producto(5)); // Mostrará 15
```

CLAUSURAS EN C# - II

Variables Capturadas

- A cualquier variable a la que se referencie en el cuerpo de una expresión lambda, se le conoce como variable capturada (e.g., **factor**).
- A las expresiones lambda que están integradas por variables capturadas se les llama clausuras.
- Estas se evalúan en el momento en que se llama a la expresión Lambda.

```
int factor = 3;

Func<int, int> producto = n => n * factor;

factor = 10;

Console.WriteLine(producto(5)); // Mostrará 50
```

CLAUSURAS EN C# - III

Vida Y Ámbito De Una Variable Capturada

- La referencia y el valor a una variable capturada, está disponible mientras el ámbito de la expresión lambda esté al alcance de su ejecución.
- A pesar de encontrarnos fuera el ámbito de su definición, se guarda una referencia y siempre estará accesible para la expresión lambda.

```
static Func<int> Contador() {
    int contador = 0;
    return () => (++contador); // Contador es una clausura.
}

static void Main() {
     Func<int> Contador1 = Contador();
     Func<int> Contador2 = Contador();

     Console.WriteLine(Contador1());
     Console.WriteLine(Contador2());
     Console.WriteLine(Contador1());
     Console.WriteLine(Contador2());
}
```

CLAUSURAS EN C# - IV

Reescribiendo El Cálculo De Potencias Con Clausuras

```
static Func<double, double> Potencia(double y)
{
    return x => Math.Pow(x, y);        // Tendremos una clausura de y
}

static readonly Func<double, double> Cuadrado = Potencia(2);

static readonly Func<double, double> Cubo = Potencia(3);

static void Main()
{
    Console.WriteLine(Cuadrado(10));  // Calcularía 10²
    Console.WriteLine(Cubo(10));      // Calcularía 10³
}
```

FUNCIONES DE UTILIDAD SOBRE OBJETOS ENUMERABLES

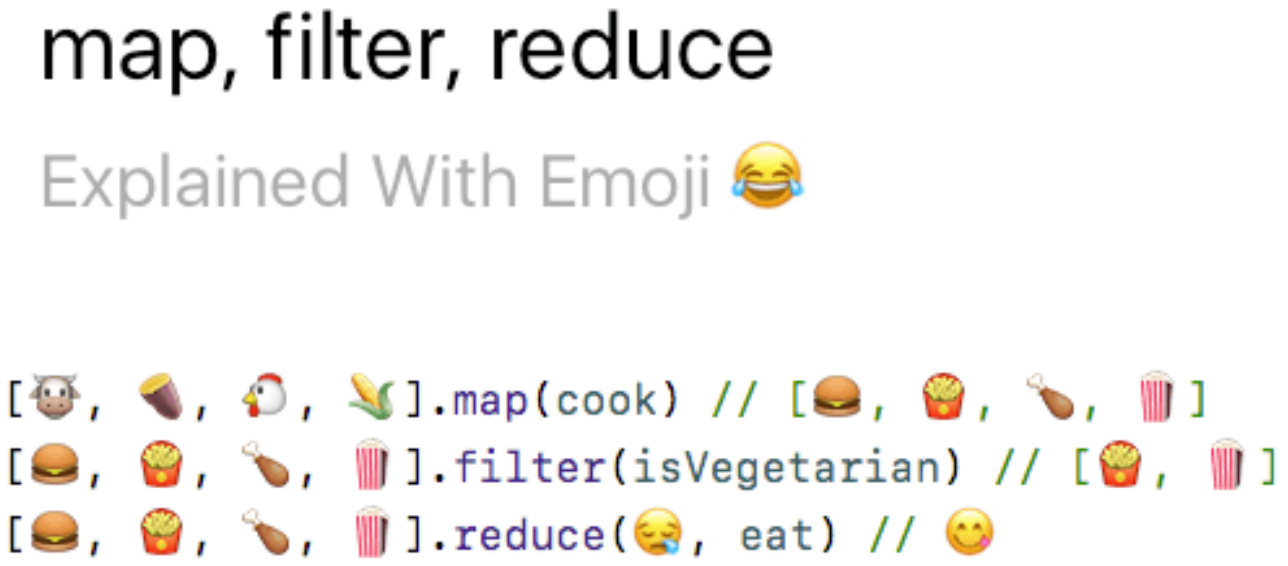
Métodos Extensores De Enumerable En System.Linq

- Dispondremos de una gran cantidad de a aplicar a secuencias los cuales están definidos en : https://docs.microsoft.com/es-es/dotnet/api/system.linq.enumerable?view=net-5.0
- Entre ellos podremos destacar:
  - Select, Where, Agreggate y GroupBy que veremos más adelante.
  - Distinct, ElementAt, Join, GroupJoin, OrderBy, Reverse, SelectMany

PATRÓN MAP-FILTER-REDUCE

Introducción

- Es un patrón de diseño utilizado en numerosos lenguajes tan populares como Javascriprt, (ES6), Java 8, Python, Scala, Swift, etc...
- Se trata de operaciones sobre secuencias de elementos.



PATRÓN MAP-FILTER-REDUCE

Map

- Aplica una función única a cada elemento de la secuencia (o enumeración) y devuelve una nueva secuencia que contiene los resultados, en el mismo orden. Es decir: (E → F) × Sec<E> → Sec<F>
- En C# lo aplicaremos a través del método extensor de IEnumerable<T> Select definido en System.Linq y que me devolverá otro objeto IEnumerable<T> con la función de transformación o "mapeo" aplicada.

```
// Dada la siguiente secuencia de números reales
List<double> reales = new List<double> { 1.3, 3.4, 4.5, 5.6, 8.7 };

// Aplicamos la función ToInt32 para trasnformar la secuencia a enteros.
List<int> enteros = reales.Select(Convert.ToInt32).ToList();
enteros.ForEach(n => Console.Write($"{n} "));
```

PATRÓN MAP-FILTER-REDUCE

Filter

- La operación de secuencia importante es filter, comprueba cada elemento con un predicado unario y solo aquellos elementos que cumplen el predicado se delvolverán en una nueva secuencia filtrada.
  Es decir: (E → bool) × Sec<E> → Sec<E>
- En C# lo aplicaremos a través del método extensor de IEnumerable<T> Where definido en System.Linq y que me devolverá otro objeto IEnumerable<T> con la función de predicado aplicada.

```
// Dada la siguiente secuencia de números
List<int> números = new List<int> { 1, 3, 4, 5, 8, 10 };

// Filtramos aquellos que sean pares.
Func<int, bool> esPar = n => n % 2 == 0;
List<int> pares = números.Where(esPar).ToList();
pares.ForEach(n => Console.Write($"{n} "));
```

PATRÓN MAP-FILTER-REDUCE

Reduce

- Reduce combina los elementos de la secuencia en conjunto, utilizando una función binaria. Es decir: (F × E → F) × Sec<E> × F → F
- Además, de la función y la secuencia, también toma un valor inicial que inicializa la reducción, y que termina siendo el valor de retorno si está vacía.
- En C# lo aplicaremos a través del método extensor de IEnumerable<T> Aggregate definido en System.Linq y que me devolverá un objeto con el resultado de la combinación o agregación de elementos.
- Aunque, dispondremos de muchas funciones de agregación ya implementadas como Sum, Max, Min, Average, o Count.

```
// Dada la siguiente secuencia de números
List<int> números = new List<int> { 10, 3, 4, 5, 8, 2 };

// Cuenta los números mayores o iguales a 5.
Func<int, int, int> cuentaAprobados = (c, n) => n >= 5 ? c + 1 : c;
int mayores = números.Aggregate(0, cuentaAprobados);
Console.WriteLine($"Suma: {mayores}");
```

PATRÓN MAP-FILTER-REDUCE

Combinación De Las Tres Operaciones

- Podremos combinar las tres operaciones.

Ejemplo

- Imaginemos que queremos saber el número de aprobados de una lista de notas con decimales.
- Una posible implementación utilizando una escritura funcional con el patrón M-F-R podría ser.

```
List<double> notas = new List<double> { 1.3, 3.4, 4.6, 5.6, 8.7 };

int aprobados = notas.Select((Func<double, double>)Math.Round)
                     .Where(n => n >= 5)
                     .Aggregate(0d, (c, n) => c + 1,
                                c => Convert.ToInt32(c));
Console.WriteLine($"Aprobados: {aprobados}");
```

- Map (Select) → Redondeará las notas.
- Filter (Where) → Se quedará solo con los aprobados.
- Reduce (Aggregate) → Los contará y tranformará a entero el resultado.

OTRAS FUNCIONES DE UTILIDAD PARA SECUENCIAS

OrderBy/OrderByDescending

- Ordena de manera ascendente/descendente los elementos de una secuencia en función de una clave.
- Los tipos deben ser comparables. Si no lo son, necesitará de un objeto que implemente el interfaz IComparer<T> entre dos claves.

Distinct

- Elimina valores repetidos en una secuencia. El tipo de datos de la secuencia deberá implementar Iequatable o deberemos pasarle un objeto que implemente un interfaz de comparación.

```
var notas = new double[] { 1, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7 };

int notaMayor = notas.Select(n => Convert.ToInt32(Math.Round(n)))
                     .Distinct()
                     .OrderByDescending(n => n)
                     .First();

Console.WriteLine($"La nota mayor es {notaMayor}");
```

OTRAS FUNCIONES DE UTILIDAD PARA SECUENCIAS

GroupBy - I

- Hay muchas sobrecargas pero nosotros usaremos esta:

```
IEnumerable<TResult> GroupBy(
            Func<TSource, TKey> keySelector,
            Func<TKey, IEnumerable<TSource>, TResult> resultSelector);
```

- Función del selector de claves por la que agruparé la secuencia:
  - Recibe TSource es el tipo de la secuencia a procesar.
  - Devuelve TKey la clave resultado del mapeo de TSource por que que voy a agrupar.
- Función que proyecta los elemento de cada grupo con su clave:
  - Recibe Tkey con la clave por la que agrupo y la agrupación de los objetos TSource para esa clave.
  - Devuelve TResult con lo que quiero producir en la nueva secuencia resultado de la agrupación.

OTRAS FUNCIONES DE UTILIDAD PARA SECUENCIAS

GroupBy – II

- Por ejemplo, vamos a agrupar por la parte entera de las notas, para obtener una gráfica, a partir de una secuencia de las mismas.

```
var notas = new double[] {1, 3.4, 4.3, 4.6, 4.3, 7.2, 7.6, 5.6, 8.7};
notas.GroupBy(
            // Función selectora:
            //   TSource son los doubles de la secuencia de entrada.
            //   TKey será un entero resultado de quedarnos
            //        con la parte entera de la nota.
            // Todos los elementos de la lista de entrada que produzcan
            // la misma clave, estarán en el mismo grupo.

            n => Convert.ToInt32(Math.Floor(n)),

            // TResult será una nueva secuencia con la parte entera de
            // la nota y la cuenta de elementos agrupados por la misma.

            (pe, ln) => new { Nota = pe, Cantidad = ln.Count() })

        .OrderBy(d => d.Nota)
        .ToList()
        .ForEach(r => Console.Write($"Con {r.Nota} -> {r.Cantidad}\n"));
```

LINQ

LINQ – I (USO DE TIPOS ANÓNIMOS)

Uso

- Los tipos anónimos suelen usarse en la cláusula select de una expresión de consulta LINQ, para devolver un subconjunto de las propiedades de cada objeto en la secuencia de origen.
- Pueden actuar sobre cualquier objeto que implemente IEnumerable<T>
- El lenguaje de consultas será muy similar a SQL.
- Por ejemplo para la colección de la página anterior vamos a obtener nombre y código de país de aquellas persona que sean cubanas.

```
var cubanos = from h in Personas
              where h.CodigoPais == "CU" orderby h.Nombre
              select new {
                 Nombre = h.Nombre.ToUpper(),
                 CodigoPais = h.CodigoPais
              };

foreach (var tupla in cubanos)
              Console.WriteLine(tupla);
```

LINQ - II

¿Qué Hace La Consulta?

```
var cubanos = from h in Personas
              where h.CodigoPais == "CU"
              orderby h.Nombre
              select new {
                  Nombre = h.Nombre.ToUpper(),
                  CodigoPais = h.CodigoPais
              };
```

- El compilador de C# reescribe el la consulta de LINQ aplicando programación funcional de la siguiente forma...

```
var cubanos = Personas
                .Where(h => h.CodigoPais == "CU")
                .OrderBy(h => h.Nombre)
                .Select(h => new {
                    Nombre = h.Nombre.ToUpper(),
                    CodigoPais = h.CodigoPais
                });
```

LINQ - III

¿Funciona También Con Arrays ?

- La respuesta es Sí.
  - Supongamos el típico código que hacíamos hace unos meses para obtener de un array todos los números impares ordenados.

```
int[] enteros = new int[] { 2, 4, 6, 13, 7, 12, 9, 11, 66 };
```

  - Con programación imperativa podría plantearse así:

```
List<int> impares1 = new List<int>();
foreach (int e in enteros)
    if (e % 2 != 0) impares1.Add(e);
impares1.Sort();
foreach (int i in impares1) Console.Write($"{i} ");
```

  - Con Programación Funcional podría plantearse así:

```
enteros.Where(e => e % 2 != 0).OrderBy(e => e).
Select(e => e)).ToList().ForEach(i => Console.Write($"{i} "));
```

  - Con LINQ podría plantearse así:

```
List<int> impares2 = new List<int>(
    from e in enteros where e % 2 != 0 orderby e select e);
impares2.ForEach(i => Console.Write($"{i} "));
```

RECURSIVIDAD – I

Definiciones

- **Llamada recursiva**: se produce cuando una se llama a si misma para obtener el valor a retornar.
- **Recursividad**: la capacidad de un módulo de llamarse a si mismo.
- **Algoritmo recursivo**: un algoritmo resuelto mediante recursividad.

Cualquier ALGORITMO RECURSIVO tiene un equivalente ITERATIVO.

Ventajas

- Existen problemas cuya solución natural es claramente recursiva con lo que el algoritmo resulta muy simple y claro frente a la solución iterativa.

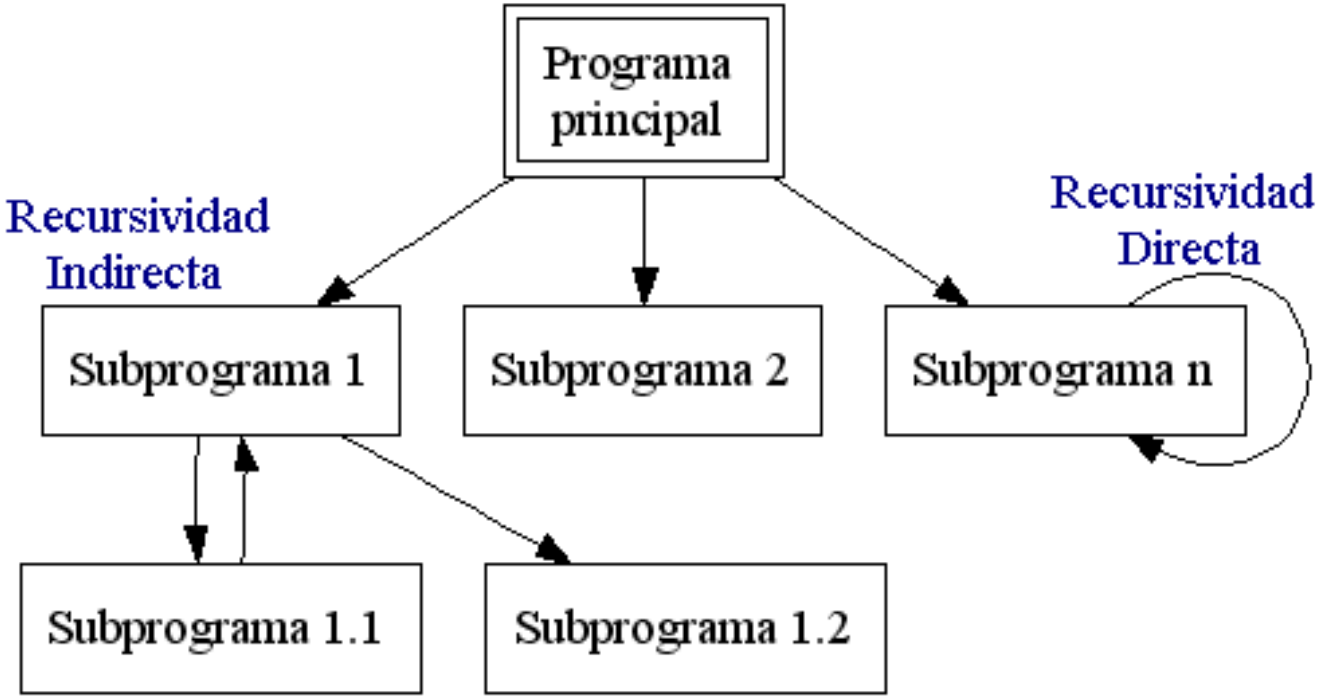
Desventajas

- Generalmente es menos eficiente que la solución iterativa. Aunque se puede aliviar con una técnica denominada MEMOIZATION.
- Consume muchos más recursos.
- Puede llevar al desbordamiento de la pila de llamadas o (Stack Overflow).

RECURSIVIDAD – II

Tipos De Recursividad

- **Recursividad directa**: se produce cuando un subprograma o módulo se llama a si mismo.
- **Recursividad indirecta**: se produce cuando un subprograma o módulo A llama a otro B que acaba llamando de nuevo a B. (NO USARLA)



RECURSIVIDAD – III

Para Diseñar Un Programa Recursivo Necesitaremos...

CASO GENERAL

- Se definirá en función de versiones menores de si mismo.
  - Pueden darse varios casos generales.
  - Todos tenderán a un caso base.
- Deberemos confiar en que la llamada recursiva va a hacer su trabajo y devolver el resultado correcto.

CASO BASE

- Condición de salida de la recursividad, donde el caso general ya no se puede dividir y donde dejamos de hacer llamadas recursivas.
  - Pueden darse varios casos base.
  - Si no existe el caso base no saldríamos de la recursividad y se produciría un Stack Overflow.

RECURSIVIDAD - IV

DESDE EL PUNTO DE VISTA DEL LENGUAJE C#

- Tendremos una pila de llamadas al mismo método.
- En cada llamada se apilan los objetos locales y parámetros del método llamador.
- Al recuperar el control, el valor de los objetos apilados es recuperado.
- Es interesante usar el depurador para ver como funcionan este tipo de programas.

RECURSIVIDAD - V

Ejemplo1

- Función recursiva para calcular el factorial de un número entero.
- El factorial se simboliza como n!, se lee como "n factorial", y la definición es: *n! = n * (n-1) * (n-2) * ... * 1*
- No se puede calcular el factorial de números negativos, y el factorial de 0 es 1, de modo que una función bien hecha para cálculo de factoriales debería incluir un control para esos casos.

Pasos Para La Resolución

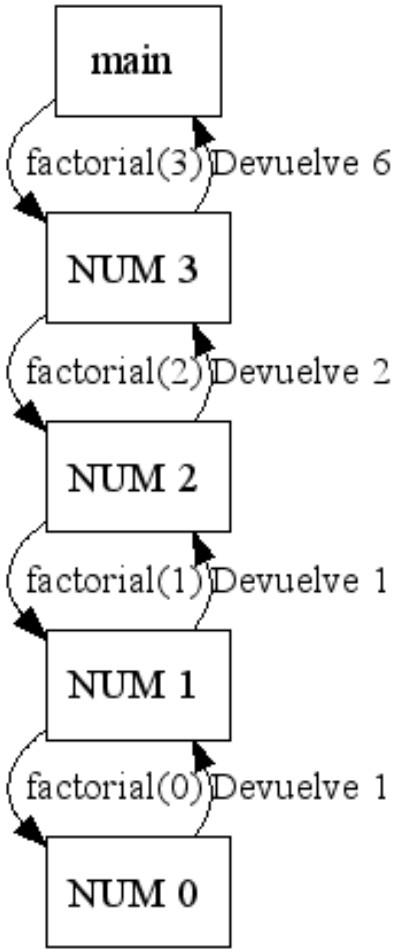
1. La función debería tener un interfaz descriptivo, tal y como lo tiene la iterariva, esto es: ulong Factorial(ulong n)
2. Identificaremos el caso base, en este caso es si n=0 → !n = 1
3. Identificaremos el caso general, esto es, cómo descomponer el problema de tal manera que tienda al caso base. En este caso será n! = n * (n-1)!

RECURSIVIDAD - V

Solución

```
static ulong Factorial(ulong n)
{
    ulong valor;

    if (n == 0)
        // Caso Base
        valor = 1;
    else
        // Caso General
        valor = n * Factorial(n - 1);
    return valor;
}
```



- La variable local valor y el parámetro formal n será diferente en cada llamada y se liberarán de memoria en cuanto se desapilen la llamadas a la función.

RECURSIVIDAD – VI

Ejemplo2

Vamos ha hacer una función recursiva que calcule la potencia de un numero:

- Interfaz: Debería ser el normal para la iterativa, esto es,
  $b^e \rightarrow$ double Pow(double b, int e)
- Caso Base: $b^0 \rightarrow 1$
- Caso General: $b^e \rightarrow b \times b^{e-1}$

```
static double Pow(double b, int e)
{
    double p = 0d;

    if (e <= 0)
        // Caso Base
        p = 1d;
    else
        // Caso General
        p = b * Pow(b, e - 1);

    return p;
}
```

RECURSIVIDAD CON EXPRESIONES LAMBDA C# - I

Para expresar la función factorial anterior con expresiones lambda haríamos algo cómo…

```
Func<int, int> factorial = n => (n == 0) ? 1 : n * factorial(n - 1);
```

Pero como aún no hemos terminado de ejecutar la instrucción donde definimos el delegado factorial nos dice que aún no existe.

Existen varias soluciones al problema elegantes desde el punto de vista funcional y matemático. Pero una solución muy simple sería…

Definir primero el identificador que referencia al delegado asignándole null.

```
Func<int, int> factorial = null;
```

Cómo ya está definido el identificador y es una referencia a un delegado. Podremos reasignarlo haciendo una clausura dentro del mismo la referencia.

```
factorial = n => (n == 0) ? 1 : n * factorial(n - 1);
```

RECURSIVIDAD CON EXPRESIONES LAMBDA C# - III

¿Serías Capaz De Hacerlo Con Esta Función Recursiva?

```
static string Binario(int numDecimal) {
    string cadenaBinaria;

    switch (numDecimal)     {
      case 0:
          cadenaBinaria = "0";
          break;

      case 1:
          cadenaBinaria = "1";
          break;

      default:
      {
          cadenaBinaria = Binario(numDecimal / 2) + $"{numDecimal % 2}";
          break;
      }
    }
    return cadenaBinaria;
}
```

Piensa bien los operadores y las expresiones a utilizar para quede matemáticamente sintetizada o “elegante”.

RECURSIVIDAD CON EXPRESIONES LAMBDA C# - IV

Solución

```
Func<int, string> binario = null;

binario = n =>
n > 1 ? binario(n / 2) + $"{n % 2}" : (n == 0 ? "0" : "1");

// La siguiente expresión mostrará los 16 primeros números en binario.
Enumerable.Range(0, 16).Select(i => binario(i)).
ToList().ForEach(b => Console.WriteLine(b));
```