Tema 6

Descargar estos apuntes

Índice

Expresiones Regulares en C#

Para probar las expresiones regulares '*on-line*' podemos usar la siguiente url regexr.com

Definición de expresión regular

Una expresión regular, a menudo llamada también **patrón**, es una expresión representada por **caracteres** y **metacaracteres** que describe un **conjunto de cadenas** sin enumerar sus elementos.
A través de la expresión regular es posible interpretar rápidamente vastas cantidades de texto y ejecutar varias operaciones como:

- Buscar patrones.
- Eliminar texto.
- Extraer patrones.

Caracteres

Será todo lo que no sean metacaracteres. Coinciden en la expresión con ellos mismos.

Metacaracteres

Son caracteres que tienen cierto significado especial '*', '?', etc.

```
root# ls *.odt
```

En el enlace del tema podrás ver la especificación oficial del la sintaxis soportada por C# para expresiones regulares. Nosotros en este tema hemos hecho un **'*resumen*'** con las características más comunes de las mismas y que podrás encontrar en la mayoría de lenguajes.

Clases de carácter

Son conjuntos de caracteres que encontrarán una correspondencia si uno de los caracteres incluido en el conjunto coincide.

Metacaracter	Descripción
`.`	Coincide con cualquier caracter menos \n. Para poner '.' usaremos \.
`[aeiou]`	Coincide con cualquier caracter dentro del grupo
`[^aeiou]`	Coincide con cualquier caracter que no pertenezca al grupo
`[0-9]`	Coincide con cualquier caracter dentro del rango indicado
`[0-9A-Z]`	Coincide con cualquier caracter dentro del rango indicado, con espacio
`\w`	Coincide con cualquier caracter de palabra `[a-zA-Z0-9]`
`\W`	Coincide con cualquier caracter de no-palabra
`\s`	Coincide con cualquier caracter de espacio
`\S`	Coincide con cualquier caracter de no-espacio
`\uNNNN`	Coincide con cualquier caracter Unicode. Por ejemplo, `\u00A9` significa ©
`\d`	Coincide con cualquier dígito. Otra manera es usar `[0-9]`
`\D`	Coincide con cualquier no-dígito. Otra manera es usar `[^0-9]`

Alternancia de carácter

La obtendremos con los corchetes `[<caracter1><caracter2>...]`

> **Nota:** Los metacaracteres en el corchete no será interpretados como tales sino como los caracteres tal cual `*, $, +, ?, |` … salvo el caso de algunas clases de carácter como `\d, \D, \w, \W, \s, \S` y por tanto la barra de escape `\` deberemos escaparla si queremos que se interpreta tal cual.

Alternancia de expresiones

Nos sirve para indicar una de varias opciones entre varias expresiones regulares, separadas por el metacaracter barra vertical `|` (`Alt + 124` o `Alt Gr + 4 y espacio`).

> **Ejemplos:**
>
> 1. `este|oeste|norte|sur` → Permitirá encontrar cualquiera de los nombres de los puntos cardinales.
> 2. `\b(mi|[ts]u)\b` → Permitirá encontrar pronombres posesivos.

Aserciones atómicas de ancho cero

No hacen avanzar el motor a través de la cadena de consumo. Simplemente producen coincidencia o error en función de la posición actual en la cadena.

Metacaracter	Descripción
Especifica que la coincidencia debe producirse al principio de la cadena o de la línea.	
`$`	Especifica que la coincidencia debe producirse al final de la cadena, antes de `\n` o al final de la línea.
`\A`	La coincidencia se debe producir al principio de la cadena.
`\Z`	La coincidencia se debe producir al final de la cadena.
`\b`	La coincidencia debe producirse en límites de palabras separadas por caracteres no alfanumérico (letras y números).
`\B`	Especifica que la coincidencia **no** se debe producir en un límite `\b`.

Ejemplos:

1. `^\d$` → Permite asegurar de que la cadena a verificar representa un único dígito, sin nada antes y después.
2. `^\d\d\/\d\d\/\d\d\d\d$` → Permite validar un formato de fecha, aunque no permite verificar si es una fecha válida, ya que 99/99/9999 también sería válido en este formato.
   Nota: Aunque C# no lo necesita, es conveniente escapar la barra `"/"` ya que en otros lenguajes cómo JavaScript es necesario por tanto pondremos `@"\/"`.

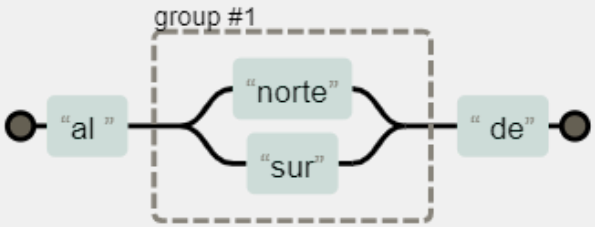
Construcciones de agrupamiento o grupos

Las construcciones de agrupamiento **permiten capturar grupos de sub-expresiones**.

Metacaracter	Descripción
`(e)`	Captura la subcadena coincidente o grupo sin captura. Las capturas que utilizan `()` se numeran automáticamente en función del orden del paréntesis de apertura, comenzando por el número uno. La primera captura, la captura de elemento número cero, es el texto que coincide con el modelo completo de la expresión regular.
`(?<nombre>e)`	Captura la subcadena coincidente dentro de un nombre de grupo o nombre de número. La cadena que se utiliza para nombre no debe contener ningún signo de puntuación y no puede comenzar por un número.

Ejemplos:

1. `al (norte|sur) de` → Permite crear un grupo sin captura para norte y sur representando las cadenas *…al **norte** de…* y *…al **sur** de…*



2. `^(?<dia>\d\d)\/(?<mes>\d\d)\/(?<año>\d\d\d\d)$` → Busca la coincidencia exacta con una fecha. Pero **además tendremos 3 grupos etiquetados** con las sub-expresiones encontradas en dia, mes y año.

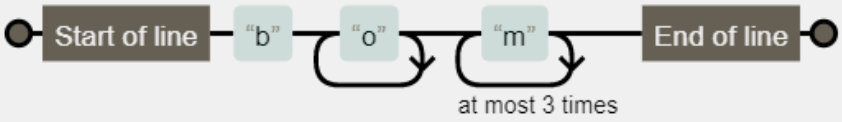
Cuantificadores

Los cuantificadores agregan datos de cantidades opcionales a una expresión regular. Una expresión de cuantificador se aplica al carácter, grupo o clase de caracteres que lo precede inmediatamente.

Metacaracter	Descripción
`e*`	Especifica 0 o más coincidencias
`e+`	Especifica 1 o más coincidencias
`e?`	Especifica 0 o 1 coincidencias
`e{n}`	Especifica exactamente n coincidencias
`e{n,}`	Especifica n coincidencias como mínimo.
`e{n,m}`	Especifica n coincidencias como mínimo y m como máximo.

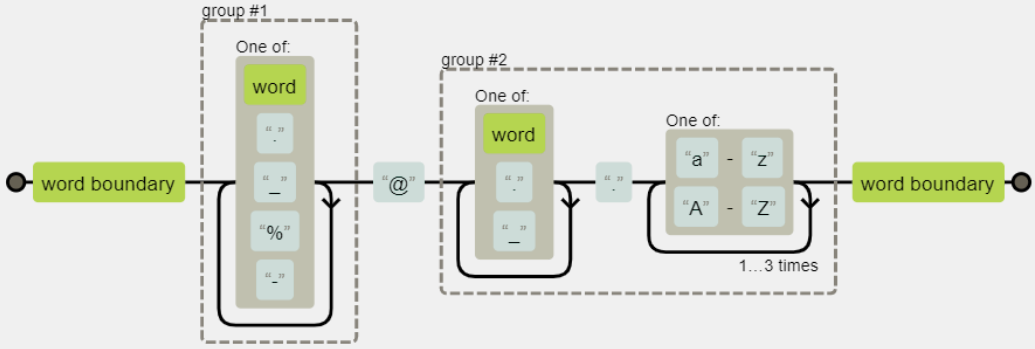
Ejemplos:

1. `^bo+m{1,4}$` → Onomatopeya de explosión.



2. `^\d{1,2}\/\d{1,2}\/\d{4}$` → Fecha.

3. `\b(?<usuario>[\w._%-]+)@(?<dominio>[\w._]+\.[a-zA-Z]{2,4})\b` → Correo electrónico (Simplificado).



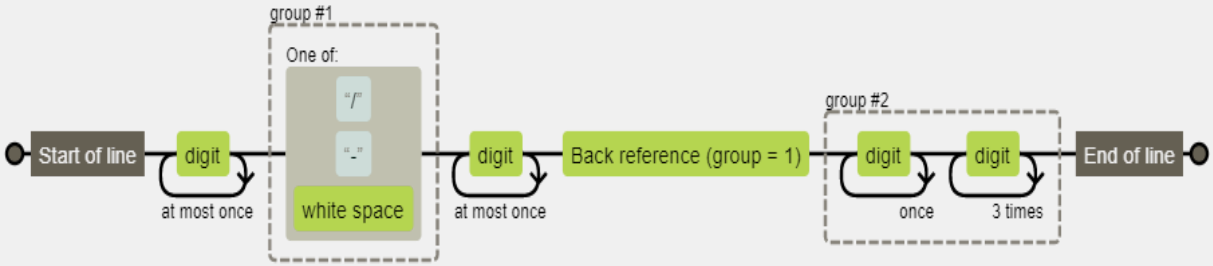
Construcciones de referencia inversa

Una construcción referencia inversa permite identificar una sub-expresión coincidente previamente más adelante en la misma expresión regular.

Metacaracter	Descripción
`\1`	Referencia inversa. Coincide con el valor de una sub-expresión numerada 1… n.
`\k<nombre>`	Referencia inversa. Coincide con el valor de una sub-expresión con nombre.

Ejemplos:

1. `(\w)\1` → Devolverá '*aa*' en '***aa****ron*'
2. `(?<letra>\w)\k<letra>` → Devolverá '*aa*' en '***aa****ron*'
3. Su pongamos esta expresión para detectar fechas:
   `^\d{1,2}[/-\s]\d{1,2}[/-\s](\d{2}\d{4})$` Si nos fijamos la separación entre los valores de día, mes y año pueden ser diferentes por lo que podrían ser válidas entradas como: **22/03-2021** y **3-12 2021** pero... **¿Cómo hacernos para asegurarnos que el separador que se ponga entre mes y año sea el mismo que se consumió entre día y mes?**
   - Lo que quiero ver si se repite o no más adelante, lo pondré en un grupo.
   - Heré referencia al grupo con los ejemplos que hemos visto más adelante.
     Tendremos pues la espresión `^\d{1,2}([/-\s])\d{1,2}\1(\d{2}\d{4})$`



Uso de expresiones regulares en el lenguaje

Utilizaremos la clase de las BCL `Regex` definida en el espacio de nombres `System.Text.RegularExpressions`, que contendrá métodos estáticos para usarlas.

A la hora de usar el patrón podemos indicar mediante una serie de flags definidos en el enumerado **RegexOptions**, cómo se va a comportar el motor de expresiones regulares.

> **Ejemplo 1:**
>
> Comprobación símple de la e.r. para comprobar un e-mail anterior.

```
static void Main()
{
    Console.Write("Introduce una e-mail: ");
    string correo = Console.ReadLine();

    // Las opciones del enumerado para el motor de e.r. serán un grupo de flags.
    RegexOptions opciones = RegexOptions.Compiled | RegexOptions.IgnoreCase;
    string patron = @"^[\w._%-]+@[\w.-]+\.[a-zA-Z]{2,4}$";

    bool correoValido = Regex.IsMatch(correo, patron, opciones);
    Console.WriteLine($"El correo {correo} {(!correoValido ? "no " : "")}es válido.");
}
```

También también podemos instanciar un objeto de la clase RegEx por si nos interesara aplicar diferentes operaciones sobre el mismo patrón.

> **Ejemplo 2:**
>
> Igual que el **ejemplo 1** pero rehusando un objeto instanciado de `RegEx` .

```
static void Main()
{
    bool correoValido;
    Regex patron = new Regex(@"^[\w._%-]+@[\w.-]+\.[a-zA-Z]{2,6}$",
                             RegexOptions.Compiled | RegexOptions.IgnoreCase);
    do
    {
        Console.Write("Introduce una dirección de correo: ");
        string correo = Console.ReadLine();
        correoValido = patron.IsMatch(correo);
        if (!correoValido)
            Console.WriteLine($"{correo} no es un correo válido.");
    } while (!correoValido);
}
```

Métodos de utilidad sobre cadenas usando ER

Destacaremos un par de ellos, tendremos la posibilidad de usarlos tanto sobre un objeto instanciado o como métodos de clase estáticos (nosotros vamos a ver estos últimos).

- `static string Replace(string cadenaConsumo, string patrón, string reemplazo, RegexOptions opciones)`
  Reemplaza un patrón en la cadena de consumo por una cadena de reemplazo.

```
string texto1 = "Esta   es una    cadena    con " +
                "un  número de     espacios en blanco indeterminado.";
Console.WriteLine(texto1);
string texto2 = Regex.Replace(texto1, @"\s+", " ");
Console.WriteLine(texto2);
```

- `static string[] Split(string cadenaConsumo, string patrón, RegexOptions opciones)`
  Trocea una cadena por un patrón.

```
string texto = "Esta   es una    cadena    con " +
               "un  número de     espacios en blanco indeterminado.";
string[] palabras = Regex.Split(texto, @"\s+");
Console.WriteLine(string.Join("\n", palabras));
```

Gestión de grupos con o sin etiqueta

Utilizaremos la clase Match que además de contener información de los grupos, me ayudará a gestionar **coincidencias**.

- Como ha sucedido hasta ahora, obtendremos un objeto Match a partir de una instancia de objeto `RegEx` o directamente a través de un método de clase de RegEx:
  - `Match objRegex.Match(string cadenaDeEntrada);`
  - `static Match RegEx.Match(string cadenaDeEntrada, string patrón);`
- Una vez tenemos un objeto `Match` ...
  - Dispondremos de la propiedad `bool objMatch.Success` que medirá si se encontró una coincidéncia o no (del grupo principal).
  - Dispondremos de la propiedad `bool objMatch.Groups` que podremos indizar a través de un entero o de una cadena con la etiqueta del grupo, y nos devolverá el grupo correspondiente.
  - Dispondremos del método de instancia `Match objMatch.NextMatch()` que continuará buscando coincidencias de la ER que generó el objeto match al que se le aplico.
- Una vez tenemos un objeto `Group` indizado...
  - Dispondremos de la propiedad `string objGroup.Value` que me devolverá un string con la coincidéncia encontrada.

Ejemplo:

Veamos su funcionamiento a través de un programa de ejemplo en el que vamos a pedir una fecha con formato **`dd/mm/aaaa`** y además de ver si es correcta la entrada, vamos a mostrar los valores de día, més y año introducidos.

```
public static void Main()
{
    Console.Write("Introduce una fecha dd/mm/aaaa: ");
    string fecha = Console.ReadLine();
    Regex patronFecha = new Regex(@"^(?<Dia>\d{2})\/(?<Mes>\d{2})\/(?<Año>\d{4})$");

    // Busco la primera y única coincidencia en la cadena de consumo fecha.
    Match coincidencia = patronFecha.Match(fecha);

    if (coincidencia.Success)
    {
        // Sabemos con 'seguridad' que en los grupos hay valores numéricos
        // En este ejemplo como los grupos estaban etiquetados, puedo indexar
        // utilizando el nombre de la etiqueta del grupo.
        ushort dia = ushort.Parse(coincidencia.Groups["Dia"].Value);
        ushort mes = ushort.Parse(coincidencia.Groups["Mes"].Value);
        ushort año = ushort.Parse(coincidencia.Groups["Año"].Value);
        Console.WriteLine($"Dia = {dia}, Mes = {mes} y Año = {año}");
    }
    else
        Console.WriteLine("No ha introducido un formato de fecha válido.");
}
```

Buscar una ocurrencia concreta con un patrón

Vamos a modificar ligéramente el ejemplo anterior, para que a partir de la cadena **"*El 11/9/2001 fueron derribadas las torres gemelas de NY y el 20/7/1969 llegó el hombre a la luna*"** me diga si en la misma se encuentra la fecha de la llegada del hombre a la luna con formato `dd/mm/aaaa` pero esta vez `dd` y `mm` pueden estar formados también por una solo dígito y **los grupos NO estarán etiquetados**.

```
    static void Main()
    {
        string texto  = "El 11/9/2001 fueron derribadas las torres gemelas de NY y el 20/7/1969 llegó
        string patron = @"(\d{1,2})\/(\d{1,2})\/(\d{4})";
        Match coincidencia    = Regex.Match(texto, patron);
        bool  fechaEncontrada = false;

        // Mientras no encuentre lo que busco y haya coincidencias con el patrón que busco en la caden
9       while (!fechaEncontrada && coincidencia.Success)
        {
            string fecha = coincidencia.Value;  // Equivale a coincidencia.Groups[0].Value
            // Cada paréntesis en la expresión para definir un grupo se numerarán de izquierda a derec
            // empezando por 1 ya que el índice 0 es toda la expresión.
            // Puede ser una opción si tenemos pocos grupos pero no anidados, aunque será menos legibl
            // que la anterior.
16          ushort dia = ushort.Parse(coincidencia.Groups[1].Value);
            ushort mes = ushort.Parse(coincidencia.Groups[2].Value);
18          ushort año = ushort.Parse(coincidencia.Groups[3].Value);

            fechaEncontrada = dia == 20 && mes == 7 && año == 1969;
            // NextMatch busca la siguiente coincidéncia si la hay a partir de la la última que encont
22          coincidencia = coincidencia.NextMatch();
        }
        Console.WriteLine($"La fecha de la llegada a la luna {(fechaEncontrada?"no se":"se")} encuentr
    }
```

Buscar todas las ocurrencias de un patrón

- Usaremos el método `Regex.Matches(cadena, patrón, ...)` que me devolverá una colección indexable y recorrible con `foreach` con todas las coincidencias.
- Este caso será útil, si queremos obtener **todas** las coincidencias, y no una en concreto que puede que la encontremos justo al principio, con lo cual esta opción sería más costosa.

```
    ...
2       foreach(Match coincidencia in Regex.Matches(texto, patron))
        {
            string fecha = coincidencia.Value; // Equivale a coincidencia.Groups[0].Value
            ushort dia = ushort.Parse(coincidencia.Groups[1].Value);
            ushort mes = ushort.Parse(coincidencia.Groups[2].Value);
            ushort año = ushort.Parse(coincidencia.Groups[3].Value);
            Console.WriteLine($"Fecha encontrada {fecha} Dia = {dia}, Mes = {mes} y Año = {año}");
        }
    }
```

🎓 Caso de estudio

Supongamos que queremos hacer un método de utilidad que nos diga si una **IP** de entrada es correcta. Además, usaremos ese método como apoyo a otro que lea una IP desde teclado y filtre solo valores válidos.

Nota: Una **IP** esta formada por **cuatro bytes** separados por punto de `0` a `255` por ejemplo `10.0.2.254`

Una primera aproximación, podría ser dar como válidas tres secuencias de **1 a 3 dígitos** separadas por un `.` con nada antes `^` y después `$`. Por tanto el patrón podría quedar algo así:
`^\d{1,3}\.\d{1,3}\.\d{1,3}.\d{1,3}$` .
Fíjate, que hemos escapado el caracter `'.'` por ser un metacaracter de las clase de caracter y queremos se interprete de forma literal.
A primera vista, se repiten tres partes en la expresión compuestas por el caracter `'.'` seguido de 3 dígitos, por lo que podríamos simplificar la expresión así:
`^\d{1,3}(\.\d{1,3}){3}$`
Fíjate, para indicar que lo que se repite 3 veces es eso hemos tenido que agruparlo en una sub-expresión.
Una vez hemos decidido la expresión regular nuestro código podría quedar así.

```
    static class Programa
    {
        static bool IPCorrecta(string entrada)
        {
5           const string patronIP = @"^\d{1,3}(\.\d{1,3}){3}$";
            return Regex.IsMatch(entrada, patronIP);
        }

        static string LeeIP(string mensaje)
        {
            bool válida;
            string ip;
            do
            {
                Console.Write($"{mensaje}: ");
                ip = Console.ReadLine();
                válida = IPCorrecta(ip);
                if (!válida)
                    Console.WriteLine($"IP {ip} no es válida.");
            } while (!válida);

            return ip;
        }

        static void Main()
        {
            string ip = LeeIP("Introduce la IP");
            Console.WriteLine($"La IP introducida es {ip}");
        }
    }
```

Pero... **¿Es posible controlar que los bytes estén entre 0 y 255 en la propia expresión?**

En este caso tendremos que indicar algún tipo de alternancia de posibilidades para incluir rángos excluyentes en la expresión...

1. Podemos asegurar que cualquier byte de 1 o 2 dígitos es válido **0 a 99**, pero de 3 dígitos no es cualquiera válidos ya que los números mayores a 255 no lo son. Luego `\d{1,2}` sería válida para byte pero nos dejamos fuera del rango de 100 a 255.
2. Si empieza por 1 el byte, después cualesquier combinación de 2 dígitos son válidos, esto es: de **100 a 199**. Luego `1\d{2}` añadiría el rango de 100 a 199 dejando fuera de 200 a 255.
3. Si empieza por 2 el byte el siguiente dígito va de 0 a 4 el tercer dígito podrá ir de 0 a 9 → `\d` , ya que si es 5 el tercer dígito solo podrá ir de 0 a 5. Luego `2[0-4]\d` añadiría el rango de **200 a 249** dejando fuera de 250 a 255.
4. Ya nos queda añadir el último rango de valores válidos de **250 a 255** a través de la expresión `25[0-5]`

Podemos concluir que la expresión final para un byte válido será `(\d{1,2}|1\d{2}|2[0-4]\d|25[0-5])` con lo cual la expresión final será...

`^(\d{1,2}|1\d{2}|2[0-4]\d|25[0-5])(\.(\d{1,2}|1\d{2}|2[0-4]\d|25[0-5])){3}$`

Nos queda una ER muy larga y ofuscada de leer, con partes repetidas y donde cometer un error es bastante sencillo. Una posible solución será definir los sub-grupos en variables separadas y así auto-documentar y simplificar quedando la función de arriba así.

```
static bool IPCorrecta(string entrada)
{
    string patronByteIP = @"(\d{1,2}|1\d{2}|2[0-4]\d|25[0-5])";
    string patronIP = "^"
                    + patronByteIP + @"(\."
                    + patronByteIP + "){3}$";
    return Regex.IsMatch(entrada, patronIP);
}
```

Pero... **¿Sería posible obtener los bytes de la IP?**

Supongamos que los vamos a devolver en un array de bytes subordinado a si es correcta o no. Por tanto nuestro interfaz quedaría así:

`static bool IPCorrecta(string entrada, out byte[] bytes)`

y si quisiéramos ver si la IP es correcta sin obtener los bytes, podríamos llamar a la función de siguiente manera:

`bool correcta = IPCorrecta(entrada, out _);`

Fíjate que el caracter `_` significa descarte como en las expresiones switch.

En este caso sería más simple hacer un Split de la cadena correcta por el caracter de separación de los bytes, que recurrir a definir grupos en la expresión. Una propuesta podría ser el código siguiente...

```
static bool IPCorrecta(string entrada, out byte[] bytes)
{
    string patronByteIP = @"(\d{1,2}|1\d{2}|2[0-4]\d|25[0-5])";
    string patronIP = "^"
                    + patronByteIP + @"(\."
                    + patronByteIP + "){3}$";
    bool ipCorrecta = Regex.IsMatch(entrada, patronIP);

    if (ipCorrecta)
    {
        bytes = new byte[4];
        int i = 0;
        foreach(string textoByte in entrada.Split('.'))
            bytes[i++] = byte.Parse(textoByte);
    }
    else
        bytes = null;

    return ipCorrecta;
}

static byte[] LeeIP(string mensaje)
{
    byte[] bytes;
    bool válida;
    do
    {
        Console.Write($"{mensaje}: ");
        string ip = Console.ReadLine();
        válida = IPCorrecta(ip, out bytes);
        if (!válida)
            Console.WriteLine($"IP {ip} no es válida.");
    } while (!válida);

    return bytes;
}

static void Main()
{
    byte[] ip = LeeIP("Introduce la IP");
    Console.WriteLine($"La IP introducida es {string.Join('.', ip)}");
}
```

Pero... **¿Sería posible hacerlo definido grupos en la expresión?**

Tendríamos que etiquetar cada grupo y con la expresión actual donde agrupamos los 3 últimos grupos no sería posible. Por lo tanto deberíamos desagrupar la ER y el código podría quedar ....

```
static bool IPCorrecta(string entrada, out byte[] bytes)
{
    string[] grupos = new string[]{"b1", "b2", "b3", "b4"};
    string patronByteIP = @"\d{1,2}|1\d{2}|2[0-4]\d|25[0-5]";
    string patronIP = "^"
                    + $"(?<{grupos[0]}>{patronByteIP})" + @"\."
                    + $"(?<{grupos[1]}>{patronByteIP})" + @"\."
                    + $"(?<{grupos[2]}>{patronByteIP})" + @"\."
                    + $"(?<{grupos[3]}>{patronByteIP})" + "$";
    Match m = Regex.Match(entrada, patronIP);

    if (m.Success)
    {
        bytes = new byte[4];
        for (int i = 0; i < grupos.Length; i++)
            bytes[i] = byte.Parse(m.Groups[grupos[i]].Value);
    }
    else
        bytes = null;

    return m.Success;
}
```

Pero... ya que vamos a sacar grupos, **la comprobación de que el byte esta entre 0 y 255 la podríamos hacer al extraer el byte, y así simplificar bastante la ER haciéndola más legible**.

El código quedaría más legible, pues no tendríamos que pensar tanto la ER. Una posible propuesta podría ser la siguiente:

```
static bool IPCorrecta(string entrada, out byte[] bytes)
{
    string[] grupos = new string[]{"b1", "b2", "b3", "b4"};
    string patronByteIP = @"\d{1,3}";
    string patronIP = "^"
                    + $"(?<{grupos[0]}>{patronByteIP})" + @"\."
                    + $"(?<{grupos[1]}>{patronByteIP})" + @"\."
                    + $"(?<{grupos[2]}>{patronByteIP})" + @"\."
                    + $"(?<{grupos[3]}>{patronByteIP})" + "$";
    Match m = Regex.Match(entrada, patronIP);
    bool ipCorrecta = m.Success;
    if (ipCorrecta)
    {
        bytes = new byte[4];
        for (int i = 0; i < grupos.Length; i++)
        {
            bytes[i] = byte.Parse(m.Groups[grupos[i]].Value);
            ipCorrecta = bytes[i] >= 0 && bytes[i] <= 255;
            if (!ipCorrecta)
            {
                bytes = null;
                break;
            }
        }
    }
    else
        bytes = null;

    return ipCorrecta;
}
```

¿Podríamos llevar este último planteamiento a la expresión anterior con el Split?

Por supuesto y el código también quedaría simple y legible.

```
static bool IPCorrecta(string entrada, out byte[] bytes)
{
    string patronIP = @"^\d{1,3}(\.(\d{1,3})){3}$";
    bool ipCorrecta = Regex.IsMatch(entrada, patronIP);
    if (ipCorrecta)
    {
        string[] textoBytes = entrada.Split('.');
        bytes = new byte[4];
        for (int i = 0; i < textoBytes.Length; i++)
        {
            bytes[i] = byte.Parse(textoBytes[i]);
            ipCorrecta = bytes[i] >= 0 && bytes[i] <= 255;
            if (!ipCorrecta)
            {
                bytes = null;
                break;
            }
        }
    }
    else
        bytes = null;

    return ipCorrecta;
}
```

A través, de este caso de estudio, hemos visto variar formas de abordar el problema. Siempre tratando de refactorizar y mantera el código lo más legible y simple posible. Con ello ganaremos mantenibilidad y evitaremos errores.

¿Qué versión crees que es la mejor?