EXPRESIONES REGULARES

Manual básico de referencia

Contenido

Introducción

El tratamiento de cadenas de caracteres y validar los datos que un usuario escribe con su teclado son problemas cotidianos, que todo programador enfrentará en un momento u otro. No importa si desarrollas aplicaciones de escritorio, para SmartPhones o la Web, tendrás que poder procesar cadenas de caracteres y verificar que cumplan ciertas condiciones o criterios.

Es por esto, y por otras cosas, que existen las ***expresiones regulares***, que no son más que descripciones de patrones que deben cumplir las cadenas de caracteres que son evaluadas con ellas. Por tanto, esta herramienta no es algo inherente a Pascal ni a ningún otro lenguaje en concreto, sino que cada lenguaje tiene su propio motor de expresiones regulares permitiendo a sus usuarios programar aplicaciones que puedan utilizarlas.

Expresiones regulares

Veámoslo lento para que se entienda de qué estamos hablando: supón que tienes un formulario donde un campo debe solicitar una ***fecha de nacimiento*** en el formato `DD/MM/AAAA`, es decir, dos cifras para el día, una barra, dos cifras para el mes, una barra y cuatro cifras para el año. Se hace necesario verificar que el usuario ha ingresado la fecha en dicho formato; bien podríamos hacerlo a mano implementando una función que verifique carácter a carácter que esto es así, sin embargo, existe una forma super fácil de lograrlo mediante ***expresiones regulares***.

En Pascal, y en cualquier otro lenguaje, una *expresión regular* será un `String`. Los `Strings` que representan *expresiones regulares* ya están predefinidos. Antes de construir el ejemplo de la fecha veamos un ejemplo donde pedimos al usuario una cadena y verificamos si esa cadena está compuesta únicamente por números del **0** al **9** inclusive. El código fuente sería este:

```
program ExpresionesRegulares;

uses RegExpr, SysUtils;

var cadena: string;
    expresionRegular: TRegExpr;

begin
    write('Ingresa un numero entero: ');
    readln(cadena);

    expresionRegular:= TRegExpr.Create('[0-9]+');

    if expresionRegular.Exec(cadena) and (CompareStr(expresionRegular.Match[0],cadena)=0) then
        writeln('ES UN ENTERO')
    else
        writeln('NO ES UN ENTERO');
end.
```

Tenemos una variable de tipo `TRegExpr` (disponible en la unidad `RegExpr`) llamada `expresionRegular`. Esta variable en realidad es capaz de recibir una expresión regular propiamente dicha y evaluar luego si otras cadenas de caracteres cumplen los patrones que la expresión describe. En concreto, esta línea es donde todo inicia:

```
expresionRegular:= TRegExpr.Create('[0-9]+');
```

Lo que se hace allí es inicializar la variable expresionRegular a través de una operación llamada Create dentro de la clase `TRegExpr`. Eso es **Programación Orientada a Objetos**, y de momento no nos interesa la mecánica detrás de ello, simplemente nos importa saber que así podemos inicializar cualquier variable de tipo `TRegExpr`, pasando como argumento un `String` que es, ni más ni menos, que la expresión regular que evaluará luego a las cadenas de caracteres que queramos verificar. En este caso se pasó como argumento el literal `'[0-9]+'`, pero bien podría haberse pasado una variable `String`.

Rangos de caracteres

La expresión **`[0-9]`** representa a un carácter que puede estar entre **0** y **9** inclusive. Si yo quito el signo de **+** que coloqué en el ejemplo, lo que hará `Exec` es verificar que la cadena sea uno de esos caracteres y nada más, o sea, uno solo. Si yo quiero ver si un patrón se repite **una o más veces** debo agregar luego el signo de más (**+**). En este caso, el patrón **`[0-9]`** indica *"un carácter entre 0 y 9"*, al poner el signo de más luego de él estamos diciendo *"un carácter en 0 y 9, una o más veces"*. Por tanto, lo que hará `Exec` es verificar si la cadena tiene uno o más caracteres entre 0 y 9. Como no hay ningún otro patrón, la cadena no podrá contener otros caracteres ya que en tal caso no respetarán el patrón indicado y obtendremos `FALSE`.

En *expresiones regulares*, los rangos de caracteres, es decir, cuando yo quiero establecer que una cadena puede tener caracteres entre **`X`** e **`Y`** siendo **`X`** e **`Y`** justamente caracteres cualesquiera, de forma que **`X<Y`**. Es básicamente como definir subrangos en Pascal. Ejemplos:

- **`[A-Z]`**: Caracteres entre **`A`** y **`Z`** inclusive (mayúsculas). Es como un subrango `'A'..'Z'`.
- **`[A-D]`**: Caracteres entre **`A`** y **`D`** inclusive (mayúsculas). Es como un subrango `'A'..'D'`.
- **`[a-z]`**: Caracteres entre **`a`** y **`z`** inclusive (minúsculas). Es como un subrango `'a'..'z'`.
- **`[a-zA-Z]`**: Caracteres entre **`a`** y **`z`** inclusive, y entre **`A`** y **`Z`** inclusive. No se podría definir un subrango que abarque ambos rangos.
- **`[a-jE-G5-8]`**: Caracteres entre **`a`** y **`j`**, entre **`E`** y **`G`**, y entre **`5`** y **`8`**.
- **`[5-0]`**: Está mal porque **`5`** es mayor que **`0`**.
- **`[j-a]`**: Está mal porque **`j`** es mayor que **`a`**.

Todos estos rangos son para cadenas de un carácter, es decir que si la cadena que queremos evaluar tiene más de un carácter ya no verificará el patrón. Si queremos admitir cadenas con uno o más caracteres debemos agregar el cuantificador **+** al final.

Cuantificadores

Los cuantificadores indican, justamente, cuántas veces puede repetirse el patrón inmediato anterior. En lo que hemos visto hasta ahora, el cuantificador **+** indica *"una o más veces"*. Si yo quisiera verificar que un patrón se repite **cero o más veces** uso el cuantificador asterisco (**`*`**). En los ejemplos anteriores, poner **`[0-9]*`** verificaría si hay **cero o más dígitos**.

Otro cuantificador que tenemos es el **`?`** que indica *"cero o una vez"*. Pero también podemos indicar nosotros mismos un número específico de veces, o un rango de posibilidades. Por ejemplo, si yo quiero saber si un patrón se repite un número concreto de veces puedo usar el cuantificador **`{n}`** donde **`n`** es un natural (número entero positivo) que indica la cantidad exacta de veces que el patrón

anterior debe repetirse. Por ejemplo, si yo quiero que se ingrese cualquier cadena de caracteres enteros pero que sean exactamente cuatro dígitos escribo: `[0-9]{4}`

En el ejemplo anterior, cadenas válidas serían `4567`, `8795`, `5555` o cualquier otra que solo contenga dígitos pero que además sean exactamente cuatro. Listaré entonces todos los cuantificadores que tenemos disponibles. En estos ejemplos **X** representa al patrón completo:

Expresión	Significado
X*	X cero o más veces
X+	X una o más veces
X?	X cero o una vez
X{n}	X exactamente n veces.
X{n,}	X n o más veces.
X{m,n}	X entre m y n veces, donde m<n.

En los ejemplos anteriores usé el patrón genérico **[m-n]** que indica que se admite cualquier valor entre **m** y **n** donde justamente **m** y **n** determinan un intervalo. Sin embargo, cualquier carácter puede ser un patrón. Por ejemplo, si yo quiero verificar que una cadena comienza con una o más **a** minúsculas, luego tiene entre cero y tres **b** seguidas y termina con exactamente tres **l** minúscula, escribo esto:

```
a+b*l{3}
```

Esa hermosa expresión regular determina justamente cadenas que cumplen lo que describí anteriormente. Veámosla en detalle:

1. Comienza con **a+**, lo cual indica justamente **a** una o más veces.
2. Sigue con **b***, lo cual indica justamente **b** cero o más veces.
3. Termina con **l{3}**, lo cual indica **l** exactamente 3 veces.

Como puedes observar, cada cuantificador aplica únicamente al patrón inmediato anterior. Así el **+** aplica a la **a**, el ***** solo aplica a la **b** (no afecta a la **a** que está antes) y **{3}** solo aplica a **l**.

Ejercicio

Dada la expresión anterior: **a+b*l{3}** indica cuáles de las siguientes cadenas la verifican. Luego pruébalo con un programa en Pascal para corregir tus errores e incluso entra en regex101.com o cualquier otra Web que quiera utilizar para practicar expresiones regulares. Analiza aquellos en los que te equivocaste para entender qué es lo que estás entendiendo mal:

- alll
- aaaaaall
- aaaaaalll
- aaablll
- blll
- baaalll
- lllbaaa
- bblll
- bbllllllll
- al
- ablll
- abl
- abll
- lllba

Precedencia de cuantificadores

Mencioné que los cuantificadores aplican únicamente al patrón inmediato anterior. Todo lo que haya antes de eso no será afectado por el cuantificador. Así, en la expresión **a+b*1{3}**, el cuantificador **+** aplica solo a la **a**, el ***** solo a la **b** y el **{3}** solo a la **1**. Si yo quiero cambiar eso puedo usar paréntesis, por ejemplo:

```
(a+b)*1{3}
```

Ahora el ***** aplica a **(a+b)** . Si pensamos un poco, **a+** es *"a una o más veces"*, luego tenemos **b** una única vez. Todo eso puede ocurrir cero o más veces. Entonces, la cadena vacía verifica la expresión del paréntesis porque lo que está dentro de él puede ocurrir cero veces. ¿Qué cadenas verifican **(a+b)***?

- **ab**
- **aaaaab**
- **abab**
- **ababab**
- **abb**
- **aabb**
- **ab**
- **aaaaaaaabaa**
- **ababa**
- **aba**
- **abababaaaaaaaab**

Como puedes observar, con una expresión regular podemos representar infinidad de cadenas. Entonces, la expresión regular **(a+b)1{3}** ya se ha vuelto bastante compleja y sus resultados son muy variados aunque siempre siguen justamente le patrón definido por la expresión.

Ejercicio

Escribe cinco cadenas que verifiquen la siguiente expresión regular: **a{1}(2a+)*XZ{2,3}**

Luego implementa un pequeño programa en Pascal para poder verificar tus respuesta, así como también utiliza cualquier sitio Web para trabajar con expresiones regulares.

Operador de opción (|)

La barra vertical **|** funciona como operador de opción, decir que, si escribimos **X|Y** estamos diciendo que esperamos **X** o **Y**. Si yo escribo la expresión **(a|b)123** tenemos dos cadenas que la verifican:

- a123
- b123

Esto es porque el paréntesis **(a|b)** indica que se espera **a** o **b**, y luego tenemos **123** sin cuantificador. ¿Qué pasa si quito los paréntesis? Es decir ¿Qué cadenas verifican **a|b123**? Ahora indica cuáles de las siguientes cadenas verifican esta expresión: **(a*|b)123**, la gracia es que lo hagas a mano y luego lo verifiques con un programa Pascal o una Web.

- **123**
- **ab123**
- **123**
- **aa123**
- **aa**
- **b**
- **aaaaaaaaaa123**
- **123a**

Esto puede ser entonces muy complejo, es necario practicar y lograr comprenderlo ya que una vez que entiendes cómo funciona podrás lograr muchas cosas.

Intervalos de caracteres

Ya vimos una manera de marcar un intervalo de caracteres, por ejemplo `[0-9]` que indica *cualquier carácter entre 0 y 9 inclusive*. Esta notación puede usarse para mirar varios intervalos en uno; por ejemplo, si yo quisiera admitir cualquier carácter entre `0` y `9` o entre `a` y `z` escribo: `[0-9a-z]`. Si extendemos esto podemos marcar cualquier cantidad de intervalos, por ejemplo:

```
[a-zA-Z0-9]
```

Aquí estamos indicando que admitimos cualquier carácter entre `a` y `z`, entre `A` y `Z` o entre `0` y `9`. Entonces, la expresión `[a-zA-Z0-9]+` representa a cualquier cadena de caracteres que tenga solo letras y números.

Ejemplo práctico: Formato de dirección de correo electrónico

Así como vimos que las expresiones regulares pueden usarse para verificar formatos de fechas, suelen ser muy usadas para verificar formatos de correos electrónicos. Veamos entonces cómo lograrlo. Un correo electrónico tiene tres partes principales:

1. Lo que está antes del símbolo arroba @.
2. El propio símbolo arroba @.
3. Lo que está después del símbolo arroba @.

Lo que está antes refiere al identificador del correo, es decir, eso que el usuario mismo inventa. Esta cadena puede contener únicamente letras, guiones bajos, puntos y números teniendo en cuenta que:

- No puede comenzar con números.
- No puede comenzar ni terminar con guion bajo o punto.
- No puede dos guiones bajos seguidos; lo mismo para los puntos.
- No puede haber guiones bajos y puntos seguidos. Es decir, siempre antes de un guion bajo o un punto hay letras o números; y siempre después también.

Cadenas que verifican esto pueden ser:

- `Hola09`
- `Hola_09`
- `Hola.09_mi.cuenta`

Cadenas que no verifican esto pueden ser:

- `_hola`
- `Hola_`
- `Hola.`
- `Hola._09`
- `Hola_09.`
- `Hola__09`

Armemos una expresión regular que represente estas cadenas con sus reglas. En primer lugar, vimos que aceptan letras mayúsculas y minúsculas y números. Eso se representa fácilmente con:

```
[a-zA-Z0-9]
```

Sabemos que toda cadena debe comenzar con al menos uno de esos símbolos, así que le agregamos el cuantificador **+** (recuerden que este cuantificador indica una o más repeticiones). Así, vamos teniendo la expresión regular:

```
[a-zA-Z0-9]+
```

Esto representa entonces a una cadena de letras y números de cualquier largo.

Tenemos claro que un correo debe comenzar con una cadena de cualquier largo, es decir, no comienza con guiones bajos ni puntos. Ahora, sabemos que luego de una cadena de caracteres podría ir un punto o un guion bajo. Representemos primero eso, punto o guion bajo:

```
_|.
```

¿Estás de acuerdo? Ahora, sabemos que puede haber un punto o un guion bajo, o bien, puede no haberlo, entonces:

```
(_|.)?
```

Recuerda que el cuantificador **?** representa *cero o una vez*. Entonces, la expresión anterior verifica que haya un guion o un punto y que esto suceda cero o una vez. Si a la expresión inicial le agregamos esta quedaría:

```
[a-zA-Z0-9]+(_|.)?
```

Esa expresión representa entonces a cadenas de caracteres que tienen letras y/o números y que al final pueden tener un guion bajo o un punto e incluso podrían no tenerlo.

Sabemos que si hay un guion bajo o un punto luego debe ir algún otro carácter, por tanto agregamos a esa expresión lo mismo que teníamos al inicio: `[a-zA-Z0-9]+`. Así, tenemos:

```
[a-zA-Z0-9]+(_|.)?[a-zA-Z0-9]+
```

Esa expresión representa a cualquier cadena alfanumérica que puede contener en medio un punto o un guion. Es decir, cosas como:

- `Hola`
- `Hola702`
- `Hola_123`
- `Hola.123`

Ahora bien, si yo quiero que existan más guiones y puntos por ahí ¿cómo hago? No puedo seguir repitiendo esa expresión porque siempre estaré limitando el número de puntos o guiones bajos que se puedan contener, además de que el resultado es una expresión ilegible. Por tanto, lo único que hay que pensar es que con ese patrón enorme que acabamos de escribir exista al menos una vez alcanza, luego si se repite también sirve, por tanto, lo encerramos entre paréntesis y le ponemos el cuantificador **+**:

```
([a-zA-Z0-9]+(_|.)?[a-zA-Z0-9]+)+
```

Uff… ¿lindo no? Seguro que a golpe de vista una expresión como la anterior resulta horrible, pero créeme que he visto por la Web algunas más horribles que sirven para lo mismo. En fin, también hay otras mucho más simples. Continuemos entonces. ¿Qué va luego de eso? Pues el `@` ¿verdad? Así pues, lo agregamos:

```
([a-zA-Z0-9]+(_|.)?[a-zA-Z0-9]+)+@
```

Ahora analicemos lo que va luego del arroba: *cadenas de caracteres que solo contienen letras y pueden estar separadas por puntos*. Básicamente es el mismo patrón que antes solo que ya no admite números ni guiones bajos. En concreto es esto:

```
([a-zA-Z]+.?[a-zA-Z]+)+
```

Como verás quité el rango numérico de los intervalos y la opción del guion bajo. Si unimos todo tenemos esta hermosa expresión regular:

```
([a-zA-Z0-9]+(_|.)?[a-zA-Z0-9]+)+@([a-zA-Z]+.?[a-zA-Z]+)+
```

Ahí tenemos representadas todas las posibles direcciones de e-mail. Sin embargo, hay un error sutil en la parte que está luego del `@`, Y esto causa que por ejemplo la dirección **`mi_direccion@algo`** sea válida, pero realmente no debería serlo porque luego del símbolo arroba va el dominio y no existen dominios que no tengan un punto, o sea, debería ser algo así: **mi_direccion@dominio.algo**. ¿Cómo lo solucionas?

Caracteres especiales

La expresión regular anterior, si probamos distintos formatos de correos electrónicos, parecería funcionar bien, sin embargo si yo escribo la dirección de abajo también funcionará:

```
Mi#direccion@dominio.algo
```

¿Dónde indicamos que el carácter **#** es admitido? Pues lo que sucede es que, para las expresiones regulares, el punto (`.`) es un comodín, es decir, el punto representa a ***cualquier carácter***. Por tanto, donde está el punto en la expresión regular podemos poner cualquier carácter que se nos ocurra en la cadena y ésta funcionará bien. Así, si yo pongo el punto, una letra, un número, #, %, &, o cualquier otro carácter, la expresión indicará que es correcto. ¿Cómo hago entonces para indicar que el punto represente únicamente al punto?

Las expresiones regulares tienen el carácter de escape `\` (esto es común en Java, C++ y otros lenguajes, pero en Pascal no funciona así directamente en los Strings). Así, si el punto en una expresión regular es cualquier carácter, poner `\.` representa únicamente al punto. Entonces, lo que tendríamos que hacer ahora es ir a la expresión anterior y cambiar los puntos por **`\.`** (barra punto).

Corrige entonces la expresión anterior para que realmente chequee si se usa guion bajo o punto.

Existen más caracteres especiales que utilizan la barra de escape y que sirven para diversas tareas como verificar si una palabra empieza con cierta cadena, etc. Si has realizado los ejercicios que ha aparecido en este documento y si has analizado el ejemplo del correo electrónico, serás capaz de crear cualquier expresión regular. En la Web hay más información al respecto y más aplicaciones. No es el objetivo del curso profundizar en este tema sino mostrarte que existe esta herramienta y que veas cómo se usa para que luego tú mismo/a, si te hace falta, puedas profundizar en ella. En general, encontrarás expresiones regulares ya diseñadas por otras personas, y solo te limitarás a copiarlas y pegarlas en tu código fuente.

NOTA: Si escribimos una expresión regular errónea, o sea, que no esté representando a ninguna cadena porque sintácticamente está mal, Pascal nos arrojará una excepción y el programa se va a caer en tiempo de ejecución. Hay que tener mucho cuidado con esto.

Ejercicio

Crea un programa Pascal que lea un `documento`, un `nombre`, un `apellido` y una `edad`, y verifique si:

- El `documento` contiene únicamente números (al menos 3 caracteres).
- El `nombre` y el `apellido` contienen únicamente letras (al menos 2).
- La `edad` contiene únicamente números y que no exceda las dos cifras.

Esta es una forma perfecta y poco tediosa de verificar que los datos de entrada son correctos.

Hay mucho más para aprender sobre expresiones regulares, pero con este documento será más que suficiente para que entiendas las bases y para avanzar en el curso. Si luego en tu vida, trabajo o proyectos requieres mayor profundidad en este tema, deberás estudiarlo por tu cuenta, que es parte de lo que todo programador deberá siempre hacer.