Expresiones regulares

PID_00270627

Guillem Lluch Moll

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Guillem Lluch Moll

Licenciado en Matemáticas por la Universidad Autónoma de Barcelona (UAB), máster en Software libre por la Universitat Oberta de Catalunya (UOC) y máster en Lenguajes y sistemas informáticos (UNED). De larga trayectoria docente, en diferentes niveles y asignaturas, actualmente trabaja como coordinador TIC en un instituto de secundaria.

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Julià Minguillón Alfonso (2020)

Primera edición: febrero 2020

Avda. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

1. Introducción

Durante el tratamiento de los diversos datos surgen muchas situaciones en las que hay que encontrar, comprobar o sustituir fragmentos que tienen una cierta regularidad en su representación. Pensamos, por ejemplo, en situaciones como:

- **Nombre de ficheros similares**: compra1, compra2, ..., compra18, etc.
- **Identificadores**: como el DNI, formado por ocho dígitos y una letra. A veces la letra está separada por un guion y a veces no hay ninguna separación.
- **Números de teléfono**: siguen un patrón fácil de reconocer para las personas. Pero el software se tiene que enfrentar a las diferentes formas de expresarlos, como, por ejemplo, en grupos de tres dígitos separados por espacio o guion, o por un grupo inicial de tres y entonces de dos en dos, etc. Además, se puede incluir el prefijo del país.
- **Direcciones postales**: formadas por la calle, el número, el código postal, la ciudad y el país.
- **Fechas**: las diferencias pueden ser muy grandes. En Estados Unidos emplean un orden diferente al que se usa en castellano o en catalán. Además, hay detalles que complican su tratamiento automático, como, por ejemplo, la separación, típicamente con guiones o con barras, si se pone el año completo o solo las dos últimas cifras, etc.
- **Tratamiento de ficheros estructurados**: como csv, tsv o incluso json o xml.
- Etc.

La manipulación de estos tipos de fragmentos teóricamente se podría hacer empleando una programación de propósito general, pero de forma muy compleja (dependiendo de lo que busquemos). En general, lo que se utiliza para estos tipos de problemas son las **expresiones regulares** o **RegEx**.

Para entenderlas, volvamos al ejemplo del nombre de los ficheros: compra1, compra2, ..., compra18, etc. Informalmente, podríamos referirnos a estos como «compraX» o «compraN». La idea detrás de las RegEx es formalizar y me-

jorar este proceso de generalización. Fijémonos en la «X» o la «N» de la expresión «compraX» o «compraN». Estas letras pierden su significado habitual y pasan a representar, en este caso, un número natural.

Esto es el núcleo de las expresiones regulares: usar unos determinados símbolos para representar un **conjunto de caracteres** o para señalar la **posición** en la frase o para denotar el **número de apariciones** de uno o más caracteres.

Estos caracteres especiales son elegidos de forma que sean unos símbolos poco empleados y, consecuentemente, convierten las expresiones regulares en una secuencia de caracteres extraños, al menos al principio de su aprendizaje.

2. La sintaxis

Las expresiones regulares son una secuencia de símbolos que representan un conjunto de caracteres o realizan una función especial. Por ejemplo, «.» representa cualquier carácter, como, por ejemplo, un «9», una «n», etc. En cambio, el símbolo `$` se emplea para identificar el final de línea.

Desafortunadamente, no todos los programas utilizan el mismo juego de símbolos para las expresiones regulares, aunque sí que utilizan una sintaxis y una forma de construcción similares. Nosotros, en este módulo, emplearemos los estándares propios de **POSIX**, concretamente **BRE**[1] con Grep y Sed. POSIX también define la notación **ERE**,[2] muy similar y la empleada por Bash (a partir de la versión 3) y AWK.

(1)Acrónimo de *Basic Regular Expressions*.

(2)Acrónimo de *Extended Regular Expressions*.

Familias de RegEx

Hay que tener muy presente, especialmente cuando navegamos por internet, que hay otras familias de RegEx, como, por ejemplo, las de Javascript (muy útiles para validar formularios en la web) o las de Perl, que no siempre funcionan como RegEx POSIX.

2.1. Comandos Grep

Para probar las próximas expresiones, lo haremos con Grep.

El comando `grep` muestra solo aquellas líneas que cumplen el patrón RegEx suministrado y oculta el resto.

Iremos ilustrando las explicaciones con el fichero de texto plano «deudas.txt», con el contenido siguiente (vigilad los espacios):

```
cat deudas.txt

amigo, 25, euros
Carina, -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
Lourdes,-,dolares
```

Si ejecutáis `grep d deudas.txt` veréis que salen las líneas 3.ª, 4.ª y la última, pues todas tienen una «d». Si ponéis `grep j deudas.txt`, veréis que no sale ninguna línea, puesto que las «j» que hay son mayúsculas, y aquí se diferencian las minúsculas de las mayúsculas.

Diferencia minúsculas-mayúsculas

En inglés, este comportamiento se llama *case sentive*.

```
# Busca líneas con la letra d.
grep d deudas.txt

Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Lourdes,-,dolares
```

```
# Busca líneas con la letra j. Como que no hay ninguna, no devuelve nada.
grep j deudas.txt
```

Los comandos `grep` también se usan mucho para filtrar los resultados de otros comandos, empleando *pipes* (`|`). Por ejemplo, `ls -ltr` lista los ficheros de un directorio. Si la salida la redirigimos a `grep`, podremos conseguir ver solo los nombres de los ficheros que coinciden con el patrón suministrado:

```
# Filtra la salida de ls, mostrando solo los ficheros tipo csv.
ls -ltr | grep .csv$

-rw-rw-r-- 1 gandalf comunidad 1236531 de ma 26 14:10 IncomeLeft.csv
-rw-rw-r-- 1 gandalf comunidad 3922 de ju 21 19:35 comp.csv
```

En este tema, probaremos los patrones tanto sobre un fichero con Grep patrón fichero como redirigiendo la salida del comando `echo`.

```
# Cuando el texto del echo coincide con el patrón, se muestra la línea.
echo "Martín" | grep Mar

Martín
```

```
# Si el texto del echo no coincide, no se muestra nada.
echo "Martín" | grep " "
```

2.2. Literales y la secuencia de escape

Si queremos encontrar un determinado carácter exactamente o una secuencia de caracteres, en la mayoría de casos tenemos que escribir la secuencia directamente:

```
# Busca líneas con los caracteres ar (seguidos).
grep ar deudas.txt

Carina, -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Lourdes,-,dolares
```

```
# Busca líneas con al menos un espacio en blanco.
grep " " deudas.txt

amigo, 25, euros
Carina, -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
```

```
# Busca líneas con un nombre concreto.
grep "Juan López" deudas.txt

Juan López, 3, dolares
```

Dado que el símbolo «.» sirve para representar cualquier carácter, hay que usar algún tipo de truco cuando lo que queremos encontrar es el carácter «.» literalmente. Para cambiar el significado por defecto de un símbolo se emplea una **secuencia de escape**, compartida con muchos lenguajes de programación, que es la barra invertida (`\`). Así:

```
# Busca líneas con un punto.
grep "\." deudas.txt

Juan Pérz, 34.8, dolar
```

```
Amorós López, 110.000, euros
```

2.3. Clases de caracteres

Las **clases** representan un grupo de caracteres. Las clases POSIX se denotan poniendo un texto que las define entre dos puntos y entre dos corchetes. Por ejemplo `[[:digit:]]`, que representa cualquier dígito o `[[:upper:]]`, que representa cualquier mayúscula:

```
# Busca líneas con algún dígito.
grep [[:digit:]] deudas.txt

amigo, 25, euros
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
```

```
# Busca líneas con alguna mayúscula.
grep [[:upper:]] deudas.txt

Carina, -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
Lourdes,-,dolares
```

Obviamente, los literales y las clases se pueden (y se suelen) combinar para obtener patrones más precisos:

```
# Líneas con una mayúscula seguida de una o acentuada.
grep "[[:upper:]]ó" deudas.txt

Juan López, 3, dolares
Amorós López, 110.000, euros
```

```
# Líneas con un espacio en blanco y con un dígito.
grep " [[:digit:]]" deudas.txt

amigo, 25, euros
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
```

Las clases de caracteres pueden ser diferentes según el *locale* actual. Por tanto, es importante definirlo en consonancia con los datos y textos que queremos tratar.

Más información

Para más información, ved el manual de *locale* en Linux.

2.4. Cuantificadores

Solo con los literales y las clases estaríamos muy limitados respecto a las posibilidades de precisar la búsqueda. Imaginad que queremos que las líneas tengan dos o más «o» seguidas. En el caso de querer dos exactamente, lo que podríamos hacer es:

```
# Intento obtener las líneas con exactamente 2 o.
grep "oo" deudas.txt
```

```
Carina, -, eurooos
```

No obstante, el resultado incluye la cadena con 3 «o». Esto es a causa de que con POSIX **se devuelven los fragmentos más grandes** que cumplen el patrón. Siendo estrictos, donde hay tres «o», realmente también hay dos.

Comportamiento avaricioso

En inglés, este comportamiento se denomina *greedy*.

Si lo que queremos es poder hacer búsquedas que puedan contener valores con un número concreto de caracteres, dígitos o símbolos, resulta evidente que esta forma de proceder, repitiendo el símbolo las veces que haga falta, **no es escalable**. Por este motivo, las expresiones regulares permiten emplear **cuantificadores** que regulan el número de veces que puede aparecer un carácter o un conjunto de caracteres. Son muy comunes:

Cuantificadores	Descripción
`\?`	Para un elemento opcional.
`\+`	Para denotar que puede aparecer una o más veces.
`\*`	Para ninguna o muchas veces.

Si hace falta más precisión, tenemos `\{n}`, `\{n,m}`, `\{,m}` o `\{n,}`, donde *n* es el mínimo número de ocurrencia y *m* el máximo. Por ejemplo:

```
# Líneas que contienen una e, tres caracteres cualesquiera y una s.
grep "e.\{3\}s" deudas.txt

amigo, 25, euros
Amorós López, 110.000, euros
```

```
# Líneas que contienen una e y, más adelante, una s.
grep "e.\+s" deudas.txt

amigo, 25, euros
Carina, -, eurooos
Juan López, 3, dolares
Amorós López, 110.000, euros
Lourdes,-,dolares
```

No obstante, si intentamos volver a reescribir la obtención de exactamente dos «o», nos encontramos que seguimos sin conseguirlo.

```
# Intento obtener las líneas con exactamente 2 o.
grep "o\{2\}" deudas.txt

Carina, -, eurooos
```

Podría parecer imposible, pero se puede hacer con un poco de ingenio y con la negación, que veremos más adelante.

```
# Líneas que contienen un dígito, un punto y un dígito.
grep "[[:digit:]]\.[[:digit:]]" deudas.txt

Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
```

```
# Líneas que contienen al menos un dígito, un punto y uno o más dígitos.
grep "[[:digit:]]\+\.[[:digit:]]\+" deudas.txt
```

```
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
```

Estos dos últimos ejemplos son muy similares. Los fragmentos que coinciden con la primera expresión son «4.8» y «0.0» y con la segunda «34.8» y «110.000». En principio, puede parecer mejor la segunda expresión, pero esto depende de la intención y el uso que queramos darle.

```
# Líneas que contienen al menos dos dígitos seguidos.
grep "[[:digit:]]\{2\}" deudas.txt

amigo, 25, euros
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
```

2.5. Delimitadores (*anchors*)

Otro elemento muy utilizado en las expresiones regulares son los **delimitadores**, con los cuales podremos buscar patrones que estén en una **determinada posición**.

Tenemos el de final y el de principio de línea, `$` y `^`, respectivamente.

Por ejemplo:

```
# Líneas que empiezan por minúscula.
grep "^[[:lower:]]" deudas.txt

amigo, 25, euros
```

```
# Líneas que acaban en os.
grep "os$" deudas.txt

amigo, 25, euros
Carina, -, eurooos
```

```
# Líneas que acaban en os o con os y un espacio.
grep "os \?$" deudas.txt

amigo, 25, euros
Carina, -, eurooos
Amorós López, 110.000, euros
```

Hay que fijarse en la diferencia entre los dos últimos ejemplos. Inapreciablemente, la línea de Amorós López contiene un espacio en blanco al final y esto hace que no cumpla con el patrón `"os$"`.

Los espacios en blanco y, en general, los caracteres no visibles pueden suponer grandes problemas si no se consideran a tiempo o pasan desapercibidos.

2.6. Caracteres optativos, rangos y negación

Los **caracteres optativos** y los **rangos** ofrecen mucha flexibilidad a la hora de construir las expresiones regulares.

Ambos se denotan con corchetes (`[]`). Veamos unos cuantos ejemplos:

```
# Líneas que empiezan por J o L.
grep "^[JL]" deudas.txt

Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Lourdes,-,dolares
```

```
# Líneas que contienen una e, incluso acentuada.
grep "[eèé]" deudas.txt

amigo, 25, euros
Carina, -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
Lourdes,-,dolares
```

La otra posibilidad es usar **rangos**. Así:

Rangos	Descripción
`[a-z]`	Rango de todas las minúsculas.
`[c-n]`	Rango de todas las minúsculas desde la c hasta la n.
`[a-zA-Z]`	Podemos mezclar opciones y rangos. Encaja con minúsculas y mayúsculas.

```
# Líneas que empiezan por una mayúscula entre A y D.
grep "^[A-D]" deudas.txt

Carina, -, eurooos
Amorós López, 110.000, euros
```

El manual de Grep asegura que los rangos tienen en cuenta las particularidades del juego de caracteres empleados.

Es decir, si tenemos los *locale* de Linux configurados de acuerdo con los datos, se tendrán en cuenta las particularidades de la lengua en uso. No obstante, en caso de que los rangos no se comporten como queremos, están las clases, que ya hemos visto, o podemos emplear la «fuerza bruta» y escribir **todos los caracteres posibles uno a uno**, y así tener independencia del *locale* (pero no de la codificación).

Vocal minúscula en castellano

Por ejemplo, si queremos asegurar que detectamos cualquier vocal minúscula en castellano tenemos que emplear `[aáeéiíoóuú]`.

Otra casuística que nos puede ocurrir es **buscar la negación de un patrón**. La negación se denota con ^ como primer elemento dentro de los corchetes. No se debe confundir con ^ que denota el principio de línea y que va fuera de los corchetes.

```
# Líneas que no empiezan por una mayúscula entre A y D.
grep "^[^A-D]" deudas.txt
# No hay que confundirse con ^. El primero, fuera de los corchetes, es el de principio de línea.
# El ^ que va dentro es la negación.

amigo, 25, euros
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Lourdes,-,dolares
```

```
# Líneas que no contienen la J ni la C (incorrecto).
grep "[^JC]" deudas.txt
# Lo que tenemos que hacer realmente son líneas que tienen algún carácter que no sea ni J ni C.

amigo, 25, euros
Carina, -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
Lourdes,-,dolares
```

Una vez más, hay que recordar que Grep es *Greedy* y, por lo tanto, en todas las líneas anteriores hay caracteres que no son J ni C. Observad:

```
# Mostramos si hay algún carácter diferente de J y de C.
echo "J" | grep "[^JC]"

echo "V" | grep "[^JC]"

V

echo "J.C." | grep "[^JC]"

echo "JCV" | grep "[^JC]"

JCV

echo "JJJJJJCCCC" | grep "[^JC]"
```

Con la negación, podemos ajustar para buscar exactamente un número concreto de veces:

```
# Obtener las líneas con exactamente 2 o.
# Para hacerlo, pedimos que no haya una o ni antes ni después.
echo "koole" | grep "[^o]oo[^o]"

koole
```

Debemos pensar en el patrón anterior "`[^o]oo[^o]`". ¿Se puede asegurar que cubrirá todas las situaciones en las que puedan aparecer dos «o» seguidas? Antes de continuar leyendo, conviene reflexionar.

Fijaos en las situaciones siguientes que no encajan con el patrón anterior, analizadlas y haced una hipótesis de lo que está fallando.

```
echo "noom" | grep "[^o]oo[^o]"
```

```
noom

echo "oom" | grep "[^o]oo[^o]"

echo "noo" | grep "[^o]oo[^o]"

echo "oo" | grep "[^o]oo[^o]"
```

Realmente, el patrón encuentra dos «o» seguidas siempre que no estén a principio o final de línea.

2.7. Alternativas

Para poder acabar el patrón de las dos «o» consecutivas, podemos emplear las alternativas. Las alternativas sirven para encontrar las ocurrencias de dos o más expresiones regulares. Las diferentes expresiones regulares se separan por `|`.

> Cualquier línea que coincida con cualesquiera de los patrones será devuelta.

Lo veremos primero con un ejemplo simple y después con las dos «o».

```
# Líneas que empiezan por A o acaban por r y espacio.
grep "^A\|r $" deudas.txt

Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
```

```
# Líneas que tienen un espacio y guion o un espacio y 3.
grep "-\| 3" deudas.txt

Carina, -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
```

Volvemos así al patrón de las dos «o». Lo que tenemos que hacer es crear un patrón para cuando las oo estén al principio, al final o al principio y al final, y juntarlo con la expresión regular que ya tenemos `[^o]oo[^o]`.

```
# oo al principio.
echo "noom" | grep "^oo[^o]"
echo "oom" | grep "^oo[^o]"
echo "noo" | grep "^oo[^o]"
echo "oo" | grep "^oo[^o]"

oom
```

```
# oo al final.
echo "noom" | grep "[^o]oo$"
echo "oom" | grep "[^o]oo$"
echo "noo" | grep "[^o]oo$"
echo "oo" | grep "[^o]oo$"

noo
```

```
# oo al principio y al final.
echo "noom" | grep "^oo$"
echo "oom" | grep "^oo$"
echo "noo" | grep "^oo$"
echo "oo" | grep "^oo$"
```

```
oo
```

```
Ahora, para tener el patrón, tenemos que unirlas todas:
# Obtener las líneas con exactamente 2 «o».
# Para hacerlo, pedimos que no haya una o ni antes ni después.
echo "noom" | grep "^oo[^o]\|[^o]oo$\|^oo$\|[^o]oo[^o]"
echo "oom" | grep "^oo[^o]\|[^o]oo$\|^oo$\|[^o]oo[^o]"
echo "noo" | grep "^oo[^o]\|[^o]oo$\|^oo$\|[^o]oo[^o]"
echo "oo" | grep "^oo[^o]\|[^o]oo$\|^oo$\|[^o]oo[^o]"

noom
oom
noo
oo
```

2.8. Agrupamientos y referencia

Tal como hemos empleado los cuantificadores, solo podremos asignar repeticiones a un carácter, un grupo de opciones o un rango. Pero también se dan situaciones en las que hay que contar o hacer otras operaciones con un conjunto arbitrario de símbolos. En estos casos, para crear grupos, se emplean los paréntesis (con el símbolo de escape).

```
# Buscamos dos oh repetidos.
echo "ohohaahjijiooohf" | grep "\(oh\)\{2\}"

ohohaahjijiooohf

# Buscamos tres oh repetidos.
echo "ohohaahjijiooohf" | grep "\(oh\)\{3\}"
```

Las **agrupaciones** tienen un papel muy importante en las sustituciones. En el editor Sed podemos buscar un patrón y manipular todo o una parte de lo que se ha encontrado para crear una nueva expresión.

Ved también

Ved el apartado «El editor Sed.»

2.9. Tipos de expresiones regulares POSIX

Hay dos tipos de expresiones regulares POSIX: **BRE** (*basic regular expressions*) y ERE (*extended regular expressions*). En «sed, a stream editor», se puede ver que las diferencias se dan en los símbolos `?`, `+`, `{.}`, `|`, aunque también habría que incluir aquí `(,)`. Todos estos símbolos, con BRE, tienen un significado literal, excepto cuando van precedidos por el carácter de escape. Con ERE pasa al contrario, y los símbolos mencionados son especiales, salvo que vayan precedidos por el carácter de escape. Puede ser un poco confuso, puesto que los humanos tenemos una alta capacidad para llegar a acuerdos... y también para mantener discrepancias.

Aquí tenemos un listado de las **clases POSIX**:

Clase POSIX	Similar a	Significado
`[:upper:]`	`[A-Z]`	Mayúsculas

Clase POSIX	Similar a	Significado
`[:lower:]`	`[a-z]`	Minúsculas
`[:alpha:]`	`[A-Za-z]`	Mayúsculas y minúsculas
`[:digit:]`	`[0-9]`	Dígitos
`[:xdigit:]`	`[0-9A-Fa-f]`	Dígitos en hexadecimal
`[:alnum:]`	`[A-Za-z0-9]`	Dígitos, mayúsculas y minúsculas
`[:punct:]`		Puntuación (caracteres imprimibles, excepto números y letras)
`[:blank:]`	`[ \t]`	Espacios y tabuladores
`[:space:]`	`[ \t\n\r\f\v]`	Espacio tabulador, final de línea, etc.
`[:cntrl:]`		Caracteres de control
`[:graph:]`	`[^ [:cntrl:]]`	Todos los caracteres imprimibles
`[:print:]`	`[[:graph] ]`	Caracteres gráficos y espacios

Y a continuación, los caracteres especiales o metacaracteres :

Metacarácter	Descripción
.	Cualquier carácter individual. Eso sí, si está dentro de una expresión con corchetes, el punto coincide con el punto literal. Por ejemplo, `a.c` coincide con «aac», «abc», «acc», etc., mientras `[a.c]` coincide con cualquier conjunto de caracteres que incluya uno o más de los caracteres «a», «.» o «c».
`[ ]`	Una expresión entre corchetes coincide con un solo carácter que está dentro de los corchetes. Por ejemplo, `[abc]` coincide con «a», «b» o «c» y `[a-z]` especifica un intervalo que coincide con cualquier letra minúscula de «a» a «z». Estas formas se pueden mezclar: `[abcx-z]` coincide con «a», «b», «c», «x», «y» o «z», igual que `[a-cx-z]`. El carácter – se trata como carácter literal si es el último o el primer carácter entre los corchetes: `[abc-]`, `[-abc]`. El carácter `]` se puede incluir en una expresión entre corchetes si es el primer carácter: `[]abc]`. La expresión entre corchetes también puede contener clases.
`[^ ]`	Coincide con un carácter que no está dentro de los corchetes. Por ejemplo, `[^abc]` coincide con cualquier carácter que no sea ni «a», ni «b», ni «c» y `[^a-z]` coincide con cualquier carácter que no sea una letra minúscula de «a» a «z». Estas formas se pueden mezclar: `[^abcx-z]` coincide con cualquier carácter que no sea ni «a», ni «b», ni «c», ni «x», ni «y», ni «z». El carácter – se trata como carácter literal si es el último carácter o el primer carácter después de `^`: `[^abc-]`, `[^-abc]`. El carácter `]` se trata como carácter literal si es el primer carácter después de `^`: `[^]abc]`. La expresión también puede contener clases.
`^`	Coincide con la posición inicial dentro de la cadena si es el primer carácter de la expresión regular.
`$`	Coincide con la posición final de la cadena si es el último carácter de la expresión regular.
BRE: `\?` ERE: `?`	Convierte el elemento anterior en optativo. Por ejemplo, `a\?` coincide con «» o «a».
*	Coincide con el elemento anterior o con el vacío. Por ejemplo, `ab*c` coincide con «ac», «abc», «abbbc», etc. `[xyz]*` coincide con «», «x», «y», «z», «zx», «zyx», «xyzzy», y así sucesivamente.
BRE: `\{m\}` ERE: `{m}`	Coincide exactamente m veces con el ítem anterior. Por ejemplo, `a\{3\}` coincide con «aaa».
BRE: `\{m,\}` ERE: `{m,}`	Coincide con el elemento anterior al menos m veces. Por ejemplo, `a\{3,\}` coincide con «aaa», «aaaa», «aaaaa», «aaaaaa», «aaaaaaa», etc.

Metacarácter	Descripción
BRE: `\{m,n\}` ERE: `{m,n}`	Coincide con el elemento anterior como mínimo m veces y no más de n veces. Por ejemplo, `a\{3,5\}` coincide con «aaa», «aaaa» y «aaaaa».
BRE: `\( \)` ERE: `( )`	Define una **subexpresión**. Se trata como si fuera un solo elemento. Por ejemplo, `ab*` coincide con «a», «ab», «abb» y así sucesivamente, mientras que `(ab*)` coincide «», «ab», «abab», «abababab», etc. La cadena correspondiente dentro de los paréntesis se puede recuperar (con `\n`). Una subexpresión también se denomina *subexpresión marcada*, un *bloque* o un *grupo de captura*.
solo en BRE: `\n`	Recupera el fragmento entre paréntesis en lugar de n, donde n es un dígito de 1 a 9. No se ha adoptado en la sintaxis de POSIX ERE.
solo en BRE: `&`	Recupera toda la expresión marcada. No se ha adoptado en la sintaxis de POSIX ERE.

Ya hemos visto cómo probar las expresiones regulares BRE con Grep. Como con todos los comandos, conviene consultar el manual de Grep, en este caso, para explorar las posibilidades que ofrecen. Por ejemplo `grep -i` busca coincidencias, independientemente de mayúsculas/minúsculas, o `grep -E` que usa las expresiones ERE en lugar de BRE.

```
# Líneas que empiezan por C o J.
grep "^C\|^J" deudas.txt

Carina, -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar

# Líneas que empiezan por C o J. ERE.
grep -E "^C|^J" deudas.txt

Carina, -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar

# Líneas que empiezan por C, J, j o c. ERE.
grep -iE "^c|^j" deudas.txt

Carina, -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
```

2.10. Las expresiones regulares con Bash

Bash, desde la versión 3, permite hacer comparaciones, empleando la variante ERE, con el operador =~. La comparación devuelve 0 si hay coincidencia o 1 si ha fallado. Cada coincidencia se guarda en el *array* `BASH_REMATCH`. `$BASH_REMATCH` es toda la cadena, mientras que `BASH_REMATCH[1]` es la primera coincidencia, `BASH_REMATCH[2]` la segunda, etc.

Se pueden probar expresiones regulares empleando Bash:

Referencia

Este ejemplo está hecho a partir del *script* del artículo «Bash Regular Expressions» de Linuxjournal.com.

Si al fichero lo denominamos «bashre.sh» y se le dan permisos de ejecución, se pueden probar las expresiones regulares y varias cadenas por si coinciden:

```
cat bashre.sh

#!/bin/bash
```

```
if [[ $# -lt 2 ]]; then
 echo "Usage: $0 PATTERN STRINGS..."
 exit 1
fi
regex=$1
shift
echo "regex: $regex"
echo

while [[ $1 ]]
do
 if [[ $1 =~ $regex ]]; then
 echo "$1 -> coincide"
 i=1
 n=${#BASH_REMATCH[*]}
 while [[ $i -lt $n ]]
 do
 echo " capture[$i]: ${BASH_REMATCH[$i]}"
 let i++
 done
 else
 echo "$1 -> no coincide"
 fi
 shift
done

# Se prueba el patrón para cada uno de los argumentos (3, en este caso).
# ^A|d$ implica expresiones que empiecen por A o que acaben en d.
./bashre.sh '^A|d$' Asociación Cariño "Visita Madrid"

regex: ^A|d$

Asociación -> coincide
Cariño -> no coincide
Visita Madrid -> coincide
```

Otro ejemplo de uso de expresiones regulares con Bash podría ser para comprobar si un correo electrónico es válido:

```
cat checkmail.sh

#!/bin/bash

# Comprueba si hay suficientes argumentos
if [[ $# -lt 1 ]]; then
 echo "Usage: $0 Email"
 exit 1
fin
if [[ "$1" =~ ^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}$ ]]
then
 echo "$1 could be a valid email address"
else
 echo "$1 can't be a valid email"
fin

./checkmail.sh "hola"

hola can't be a valid email

./checkmail.sh "hola@algundominio.res"

hola@algundominio.res could be a valid email address
```

3. El editor Sed

Sed es un editor no interactivo orientado a trabajar línea a línea. Lo que lo hace realmente interesante y cómodo es poder realizar cambios en ficheros de texto de forma automática. Permite **buscar**, **sustituir**, **borrar**, **insertar**, etc. en uno o más ficheros a la vez.

Es la herramienta ideal para convertir grupos de ficheros de forma automática. El editor Sed se puede emplear directamente en el *prompt* o se puede crear un fichero con las acciones que se van a ejecutar y llamarlas con `-f`.

Para ilustrar las explicaciones, se empleará el fichero siguiente, denominado «deudas.txt»:

```
cat deudas.txt # Muestra el fichero de ejemplo.

amigo, 25, euros
Carina, -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
Lourdes,-,dolares
```

3.1. Sustitución

El comando más importante y más empleado de Sed es la **sustitución**. Consiste en «[dirección] s/patrón BRE/sustitución/[modificadores]».

Opcionales

La *dirección* y los *modificadores* son opcionales.

- La **dirección** es el número de línea o una expresión regular BRE. Se intentará llevar a cabo la sustitución solo en las líneas que cumplan la dirección. Si no está, el cambio se hará en todas las líneas.

- La **dirección** también puede ser un rango, donde el principio y el final se separan por coma. Tanto el inicio como el final pueden ser un número de línea o una expresión regular. La operación solo se llevará a cabo entre las líneas comprendidas entre las direcciones.

- **s** sirve para denotar que se tiene que hacer una sustitución, puesto que Sed tiene más comandos, como veremos más adelante.

- El **patrón BRE** es el patrón que se quiere sustituir.

- La parte que coincida con el patrón BRE será cambiada por lo que haya en **sustitución**. Este valor de sustitución puede ser un valor fijo o estar en función de toda la parte coincidente, empleando `&` o un grupo con `\n`.

- Los **modificadores** cambian la forma de actuar por defecto de Sed. Por ejemplo, `g` hace que se sustituyan todas las apariciones del patrón en lugar de solo la primera, que es el comportamiento por defecto.

3.1.1. Direccionamiento

Las **direcciones** pueden ser un número de línea, un rango de líneas o una expresión regular:

```
# Direccionamiento por rango.
# Aplicamos el patrón a las líneas 2 a 4.
# Cambiamos la coma por el símbolo del pipe.
sed '2,4 s/,/|/' deudas.txt

amigo, 25, euros
Carina| -, eurooos
Juan López| 3, dolares
Juan Pérz| 34.8, dolar
Amorós López, 110.000, euros
Lourdes,-,dolares


# La última línea se denota con $.
sed '3,$ s/,/|/' deudas.txt

amigo, 25, euros
Carina, -, eurooos
Juan López| 3, dolares
Juan Pérz| 34.8, dolar
Amorós López| 110.000, euros
Lourdes|-,dolares


# Direccionamiento por número.
# Aplicamos el patrón a la línea 2.
sed '2 s/,/|/' deudas.txt

amigo, 25, euros
Carina| -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
Lourdes,-,dolares


# Direccionamiento por RegEx.
# Cambia la primera coma por el tabulador solo de las líneas que tienen el símbolo -
sed '/-/s/,/\t/' deudas.txt

amigo, 25, euros
Carina     -, eurooos
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
Lourdes    -,dolares
```

```
# Direccionamiento por rango RegEx.
# Cambia la primera coma por el tabulador del rango de líneas desde la que empieza por C hasta
la que empieza por A.
sed '/^C/,/^A/s/,/\t/' deudas.txt

amigo, 25, euros
Carina     -, eurooos
Juan López     3, dolares
Juan Pérz     34.8, dolar
Amorós López     110.000, euros
Lourdes,-,dolares
```

3.1.2. Modificadores

Por defecto, Sed solo hace la sustitución en la primera coincidencia. Esto se puede cambiar con los modificadores `/g` y `/n`.

```
# Cambia todas las comas por tabuladores.
sed 's/,/\t/g' deudas.txt

amigo     25     euros
Carina     -     eurooos
Juan López     3     dolares
Juan Pérz     34.8     dolar
Amorós López     110.000     euros
Lourdes    -    dolares


# Cambia la segunda ocurrencia de la coma por un tabulador.
sed 's/,/\t/2' deudas.txt

amigo, 25     euros
Carina, -     eurooos
Juan López, 3     dolares
Juan Pérz, 34.8     dolar
Amorós López, 110.000     euros
Lourdes,-    dolares
```

Otro modificador de interés es `p`, que se suele emplear con la opción `-n` de Sed. Así hacemos que Sed actúe de forma similar a Grep y nos devuelva solo las líneas que coinciden con la dirección. Esto es a causa de que `sed -n` hace que no se imprima ninguna línea (por defecto Sed imprime todas las líneas que le llegan) y con `p` imprime la línea que ha coincidido con la dirección.

```
# Se imprime cada línea que contiene - (con la modificación hecha).
sed -n '/-/s/,/\t/p' deudas.txt

Carina     -, eurooos
Lourdes    -,dolares
```

3.1.3. Referencias

Los cambios pueden estar en función de lo que se cambia empleando las **referencias**. El símbolo `&` sirve para volver a mostrar el patrón encontrado:

```
# Sustituye línea (que tenga contenido) por "deuda, línea".
sed 's/./deuda, &/' deudas.txt

deuda, amigo, 25, euros
deuda, Carina, -, eurooos
deuda, Juan López, 3, dolares
deuda, Juan Pérz, 34.8, dolar
```

```
deuda, Amorós López, 110.000, euros
deuda, Lourdes,-,dolares
```

Aparte de la referencia del total, con los grupos podemos hacer referencia a una parte. Así, `\1` es el primer grupo, `\2` el segundo, etc.

```
# Cambia el último campo por euros. Mantiene las otras partes iguales.
sed 's/\(.*\),\(.*\),\(.*\)/\1,\2, euros/' deudas.txt

amigo, 25, euros
Carina, -, euros
Juan López, 3, euros
Juan Pérz, 34.8, euros
Amorós López, 110.000, euros
Lourdes,-, euros
```

3.1.4. Sed y ERE

```
# Con sed -r podemos emplear expresiones regulares extensas.
sed -r 's/(.*),(.*),(.*)/\1\2, pendiente/' deudas.txt

amigo 25, pendiente
Carina -, pendiente
Juan López 3, pendiente
Juan Pérz 34.8, pendiente
Amorós López 110.000, pendiente
Lourdes-, pendiente
```

3.2. Borrar, añadir, insertar, cambiar

Los comandos para hacer estas acciones son los siguientes:

Comando	Acción
`d`	Borrar.
`a`	Añadir. Añade una línea después de la dirección.
`i`	Insertar. Consiste en insertar una línea antes de la dirección.
`c`	Cambiar. Cambia toda la línea.

```
# Borrar las líneas con -
sed '/-/ d'  deudas.txt

amigo, 25, euros
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
```

```
# Borrar las 3 primeras líneas.
sed '1,3 d'  deudas.txt

Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
Lourdes,-,dolares
```

```
# Añade una línea después de la dirección.
sed '1,3 a Pendiente' deudas.txt

amigo, 25, euros
Pendiente
```

```
Carina, -, eurooos
Pendiente
Juan López, 3, dolares
Pendiente
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
Lourdes,-,dolares
```

```
# Añade una línea antes de la dirección.
sed '1,3 i Debt:' deudas.txt

Debt:
amigo, 25, euros
Debt:
Carina, -, eurooos
Debt:
Juan López, 3, dolares
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
Lourdes,-,dolares
```

```
# Cambia una línea.
sed '4 c Paid' deudas.txt

amigo, 25, euros
Carina, -, eurooos
Juan López, 3, dolares
Paid
Amorós López, 110.000, euros
Lourdes,-,dolares
```

```
# Cambia todas las líneas del rango por una sola línea.
sed '1,3 c Paid' deudas.txt

Paid
Juan Pérz, 34.8, dolar
Amorós López, 110.000, euros
Lourdes,-,dolares
```

A veces queremos que la acción se lleve a cabo en todas las líneas, excepto las que cumplen el patrón. El símbolo para hacerlo, la negación, es `!`.

```
# Borrar las líneas que no tengan -
sed '/-/ !d'  deudas.txt

Carina, -, eurooos
Lourdes,-,dolares
```

3.3. *Scripts* Sed

Hasta este momento, hemos visto cómo aplicar un comando línea por línea sobre un fichero. Pero es posible que nos interese aplicar no una única orden, sino unas cuántas, una detrás de la otra. Para hacerlo, tenemos dos opciones:

1) Agrupar los comandos Sed entre {}.
2) Crear un fichero con todas las órdenes y llamarlo para aplicarlas.

```
# Agrupación de comandos.
# Colapsa «o» y elimina la última coma.
sed '{
 s/o\+/o/g
 s/\(.*\),\(.*\),\(.*\)/\1,\2\3/
}' deudas.txt

amigo, 25 euros
```

```
Carina, - euros
Juan López, 3 dolares
Juan Pérz, 34.8 dolar
Amorós López, 110.000 euros
Lourdes,-dolares


# Comandos desde un fichero.
# Contenido del fichero.
cat demo.sed

#En los scripts Sed se pueden poner comentarios.
s/o\+/o/g
s/\(.*\),\(.*\),\(.*\)/\1,\2\3/

# Aplicación de los comandos del fichero demo.sed a deudas.txt
sed -f demo.sed deudas.txt

amigo, 25 euros
Carina, - euros
Juan López, 3 dolares
Juan Pérz, 34.8 dolar
Amorós López, 110.000 euros
Lourdes,-dolares
```

Para sacar provecho de los *scripts* Sed, conviene conocer cómo funcionan:

- **Sed actúa línea a línea**: coge una línea y le aplica todos los comandos del *script*. Entonces coge la línea siguiente y le aplica todos los comandos del *script* y continúa así hasta que acaba el fichero que se quiere tratar.

- **Las modificaciones no se hacen en el fichero original, sino en una copia temporal de cada línea** (por tanto, consume poca memoria). Si queremos guardar los cambios, podemos emplear la redirección de Bash, por ejemplo, pero nunca en el mismo fichero.

- **El orden de los comandos importa**. Después de los cambios hechos por un comando, los siguientes actúan sobre la línea modificada, no sobre el original.

- **Una vez tratada una línea del fichero, se muestra la modificación en la salida estándar** (excepto si se invoca Sed con la opción `-n`).

4. Construcción de expresiones regulares

Dominar las expresiones regulares tiene similitudes con aprender la notación matemática. Para entender las fórmulas, hay que conocer el significado de cada uno de los símbolos que aparecen.

> En expresiones regulares, la ventaja, si se compara con el álgebra, es que los símbolos mantienen el significado constantemente.

Pero aprender RegEx, igual que para la notación matemática, exige poner atención y práctica. Al principio, el esfuerzo necesario es alto y puede parecer que no vale la pena aprenderlo, pero a la larga sale a cuenta.

Es imprescindible practicar. Poder entender y saber construir RegEx no solamente consiste en saberse el listado de todos los símbolos, sino emplearlos y conjugarlos para obtener patrones útiles. Una forma de ir avanzando es analizando y tratando de mejorar las RegEx construidas por otros, identificando las ideas que han servido para crearlas. Acostumbrarse a entender las expresiones regulares sirve para estar preparado por si los datos en los que se aplican cambiaran.

Los patrones se construyen para un conjunto de datos, que aunque puede ser muy amplio, es finito.

> Por ejemplo, se pueden hacer expresiones regulares para encontrar el DNI, números de la seguridad social (que depende del país), números de tarjetas de crédito, fechas (con formato diverso), precios, etc. Por ejemplo, si pensamos en las fechas, 01-02-04 puede querer decir, según el formato, el 1 de febrero de 2004 o de 1904, o, incluso, día 4 de febrero de 2001 o de 1901. Es decir, el patrón que vamos a construir servirá para uno o algunos de este casos, pero probablemente no para los otros.

Tampoco podemos pretender crear el patrón perfecto a la primera. La tarea de ir encontrando el patrón adecuado es incremental y nos sirve para ir estudiando los datos. Cada patrón que vamos sacando es como una hipótesis que se tiene que validar con los datos y muchas veces también implica suposiciones sobre los propios datos.

Las herramientas que usan las expresiones regulares son muy potentes y, por tanto, muy complejas. Sin embargo, como dice Barnett (2001), esta complejidad no puede ser una excusa para no aprovechar la parte más simple. Y es que con las expresiones regulares podemos hacernos una idea rápida, en la línea de comandos, directamente e inmediatamente, del número de apariciones de

un determinado patrón, por ejemplo. Estas primeras pruebas pueden ser la semilla de la construcción de patrones más complejos y refinados que serán los que acabarán en producción.

Nuestro objetivo al emplear las expresiones regulares también marcará la estrategia que vamos a seguir. Si lo que queremos es saber si un patrón aparece en un conjunto de datos, es decir, queremos saber si aparece al menos una vez, nos bastará una expresión simple con la que veremos si el patrón se presenta, aunque deje muchas apariciones sin encontrar. Pero si lo que nos interesa es encontrar todas las ocurrencias, tendremos que construir una expresión mucho más precisa.

Finalmente, hay que destacar la importancia de elegir la herramienta adecuada en cada caso. Para hacer una exploración rápida *grep*, *cut*, *head*, *tail*, etc. tienen que ser la primera opción. Si queremos modificar (sustituir, insertar, borrar) determinadas partes de uno o muchos ficheros, Sed o AWK pueden ser adecuados. Si los datos no responden a expresiones regulares, no están bastante estructurados, se tendrá que recurrir a otras herramientas de tratamiento de datos, de procesamiento del lenguaje natural o de programación.

4.1. Ejemplos de RegEx

4.1.1. Filtrar, encontrar y validar

1) Con **Ubuntu** podemos mostrar todas las carpetas en «/usr/share/locale» que contengan «es»:

```
# ERE
# Empieza por d, puesto que es un directorio, después hay varios caracteres y "es".
ls -ltr /usr/share/locale | grep -E '^d.*es'

rwxr-xr-x 3 root root 4096 d'abr 16 2018 es
drwxr-xr-x 3 root root 4096 d'abr 26 2018 es_VE
drwxr-xr-x 3 root root 4096 d'abr 26 2018 es_MX
drwxr-xr-x 3 root root 4096 d'abr 26 2018 es_CO
drwxr-xr-x 3 root root 4096 d'abr 26 2018 es_CL
drwxr-xr-x 3 root root 4096 d'abr 26 2018 es_AR
```

```
# BRE
ls -ltr /usr/share/locale | grep '^d.*es'

drwxr-xr-x 3 root root 4096 d'abr 16 2018 es
drwxr-xr-x 3 root root 4096 d'abr 26 2018 es_VE
drwxr-xr-x 3 root root 4096 d'abr 26 2018 es_MX
drwxr-xr-x 3 root root 4096 d'abr 26 2018 es_CO
drwxr-xr-x 3 root root 4096 d'abr 26 2018 es_CL
drwxr-xr-x 3 root root 4096 d'abr 26 2018 es_AR
```

2) Comprobar que un nombre de usuario solo tiene los caracteres permitidos (letras minúsculas, números, guion y guion bajo) y que tiene una longitud de entre 3 y 16:

Referencia

J. Fox (2019). «Regex cookbook - Top 10 Most wanted regex» [en línea]. Factory Mind.

- ERE: `/^[a-z0-9_-]{3,16}$/`
- BRE: `/^[a-z0-9_-]{3,16}$/`

Esta *regex* obliga a empezar por una letra minúscula, un número, un guion o guion bajo y que haya entre 3 y 16 símbolos.

```
# BRE
echo "petito" | grep "^[a-z0-9_-]\{3,16\}$"
echo "petito y juan" | grep "^[a-z0-9_-]\{3,16\}$"
echo "p" | grep "^[a-z0-9_-]\{3,16\}$"

petito
```

```
# ERE
echo "petito" | grep -E "^[a-z0-9_-]{3,16}$"
echo "petito y juan" | grep -E "^[a-z0-9_-]{3,16}$"

petito
```

3) Comprobar que un número de DNI es válido. La letra puede estar unida al número, separada por un espacio o por un guion.

- ERE: `/[0-9]{8}( |-)?[a-zA-Z]/`
- BRE: `/[0-9]{8}( |-)\?[a-zA-Z]/`

Esta *regex* coincide con las cadenas con ocho dígitos `[0-9]{8}`, un espacio, un guion o ningún símbolo `( |-)?`, y una letra minúscula o mayúscula `[a-zA-Z]`.

```
# BRE
echo "el dni es 41000000e." | grep "[0-9]\{8\}\( |-\)\?[a-zA-Z]"
echo "el dni es 41000000--e." | grep "[0-9]\{8\}\( |-\)\?[a-zA-Z]"

el dni es 41000000e.
```

```
# ERE
echo "41000000e" | grep -E "[0-9]{8}( |-)?[a-zA-Z]"
echo "41000000--e" | grep -E "[0-9]{8}( |-)?[a-zA-Z]"

41000000e
```

4.1.2. Sustituir

En el fichero que hemos ido utilizando para hacer pruebas, podemos eliminar las líneas sin un valor numérico, unir la moneda con la cantidad y sustituir el nombre de la moneda por su símbolo.

```
sed '{
/-/ d # Borramos las líneas con guion.
s/, euro*s/€/ # Sustituimos euro por su símbolo.
s/, dolar.\?/\$/ # Sustituimos dólar por su símbolo.
}' deudas.txt

amigo, 25€
```

```
Juan López, 3$
Juan Pérz, 34.8$
Amorós López, 110.000€
```

4.1.3. Reordenar

Con Sed y las referencias, podemos reordenar o fusionar campos de ficheros estructurados.

```
# Reordena los campos poniendo la cantidad en primer lugar.
# Hay que fijarse en los espacios.
sed 's/\(.*\), \(.*\), \(.*\)/\2, \3, \1 /' deudas.txt

25, euros, amigo
-, eurooos, Carina
3, dolares, Juan López
34.8, dolar , Juan Pérz
110.000, euros , Amorós López
Lourdes,-,dolares
```

```
# Si hay que ordenar las líneas entre ellas, lo podemos hacer redirigiéndolas a sort.
sed 's/\(.*\), \(.*\), \(.*\)/\2, \3, \1 /' deudas.txt | sort -rn -k1

110.000, euros , Amorós López
34.8, dolar , Juan Pérz
25, euros, amigo
3, dolares, Juan López
Lourdes,-,dolares
-, eurooos, Carina
```

4.2. Construcción incremental de expresiones regulares

A la hora de construir una expresión regular compleja, es recomendable empezar por una más simple e irla refinando hasta que se ajuste a lo que se quiere conseguir.

Para ilustrarlo, trabajaremos con fechas. Concretamente, emplearemos el fichero «fechas.txt» siguiente:

```
cat fechas.txt

Día 1 marzo
El 8 Marzo es el día Internacional de la Mujer
Algún día de Marzo es el día de los amigos
Día Nacional del Trabajador de Construcción
Día Mundial del Consumidor
Marzo Día del Psicoorientador
El 22 de Marzo no es el día del optómetra
24-03-2020 Día del Locutor
27-3-2020 Día Internacional del Teatro
El 31-3-20 es el Día internacional contra el cáncer de Colon
1-4-2020 Abril Día del Controlador Técnico de Audio
8-05-20 Día de la Madre
01/06/20 Día Internacional del Niño
1/06/2020 Día del Campesino
05/6/20 Día del medio ambiente
14 de 06 del 2020 es el día mundial del donante de Sangre
17 del 6 de 2020 es el día del Higienista Dental
```

Queremos tratar las líneas que tengan el día del mes, el mes y el año y dejar el resto. Y queremos poner todas las fechas en un formato común, separado por un guion. Podemos empezar encontrando dónde hay números:

```
# Busca dígitos.
grep '[0-9]' fechas.txt

Día 1 marzo
El 8 Marzo es el día Internacional de la Mujer
El 22 de Marzo no es el día del optómetra
24-03-2020 Día del Locutor
27-3-2020 Día Internacional del Teatro
El 31-3-20 es el Día internacional contra el cáncer de Colon
1-4-2020 Abril Día del Controlador Técnico de Audio
8-05-20 Día de la Madre
01/06/20 Día Internacional del Niño
1/06/2020 Día del Campesino
05/6/20 Día del medio ambiente
14 de 06 del 2020 es el día mundial del donante de Sangre
17 del 6 de 2020 es el día del Higienista Dental
```

Ahora refinamos algo más, puesto que las tres primeras líneas no las queremos modificar. Observad que los datos que queremos están formados por uno o dos números, elementos en medio, uno o dos números, más elementos en medio y dos o cuatro números.

```
# Identificamos grupos de uno o dos dígitos.
grep '\([0-9]\{1,2\}\)' fechas.txt

Día 1 marzo
El 8 Marzo es el día Internacional de la Mujer
El 22 de Marzo no es el día del optómetra
24-03-2020 Día del Locutor
27-3-2020 Día Internacional del Teatro
El 31-3-20 es el Día internacional contra el cáncer de Colon
1-4-2020 Abril Día del Controlador Técnico de Audio
8-05-20 Día de la Madre
01/06/20 Día Internacional del Niño
1/06/2020 Día del Campesino
05/6/20 Día del medio ambiente
14 de 06 del 2020 es el día mundial del donante de Sangre
17 del 6 de 2020 es el día del Higienista Dental
```

```
# Identificamos grupos de uno o dos dígitos y que después tienen un espacio, un guion, de o del.
grep '\([0-9]\{1,2\}\)\( \|-\|/\| de \| del \)' fechas.txt

Día 1 marzo
El 8 Marzo es el día Internacional de la Mujer
El 22 de Marzo no es el día del optómetra
24-03-2020 Día del Locutor
27-3-2020 Día Internacional del Teatro
El 31-3-20 es el Día internacional contra el cáncer de Colon
1-4-2020 Abril Día del Controlador Técnico de Audio
8-05-20 Día de la Madre
01/06/20 Día Internacional del Niño
1/06/2020 Día del Campesino
05/6/20 Día del medio ambiente
14 de 06 del 2020 es el día mundial del donante de Sangre
17 del 6 de 2020 es el día del Higienista Dental
```

```
# Repetimos el patrón anterior dos veces.
grep '\([0-9]\{1,2\}\)\( \|-\|/\| de \| del \)\([0-9]\{1,2\}\)\( \|-\|/\| de \| del \)' fechas.txt

24-03-2020 Día del Locutor
27-3-2020 Día Internacional del Teatro
El 31-3-20 es el Día internacional contra el cáncer de Colon
1-4-2020 Abril Día del Controlador Técnico de Audio
```

```
8-05-20 Día de la Madre
01/06/20 Día Internacional del Niño
1/06/2020 Día del Campesino
05/6/20 Día del medio ambiente
14 de 06 del 2020 es el día mundial del donante de Sangre
17 del 6 de 2020 es el día del Higienista Dental
```

```
#  añadimos el año, que es \([0-9]\{2,4\}\), es decir, puede ser de dos o cuatro dígitos.
grep '\([0-9]\{1,2\}\)\( \|-\|/\| de \| del \)\([0-9]\{1,2\}\)\( \|-\|/\| de \| del \)\
([0-9]\{2,4\}\)' fechas.txt

24-03-2020 Día del Locutor
27-3-2020 Día Internacional del Teatro
El 31-3-20 es el Día internacional contra el cáncer de Colon
1-4-2020 Abril Día del Controlador Técnico de Audio
8-05-20 Día de la Madre
01/06/20 Día Internacional del Niño
1/06/2020 Día del Campesino
05/6/20 Día del medio ambiente
14 de 06 del 2020 es el día mundial del donante de Sangre
17 del 6 de 2020 es el día del Higienista Dental
```

Ahora tenemos cinco grupos. En el primero está el día (`[0-9]{1,2}`), en el segundo y cuarto el separador (`|-|/| de | del`), en el tercero el mes (`[0-9]{1,2}`), y en el quinto el año (`[0-9]{2,4}`). Apliquemos Sed.

Antes, sin embargo, hay un aspecto de Sed que nos conviene usar para este caso: en lugar de emplear la barra / como separador, podemos emplear el símbolo que más nos convenga.

Dado que ya estamos usando esta barra dentro de la expresión regular y, como no usamos la arroba, podemos emplearla como separador.

```
sed 's@\([0-9]\{1,2\}\)\( \|-\|/\| de \| del \)\([0-9]\{1,2\}\)\( \|-\|/\| de \| del \)\
([0-9]\{2,4\}\)@\1-\3-\5@' fechas.txt

Día 1 marzo
El 8 Marzo es el día Internacional de la Mujer
Algún día de Marzo es el día de los amigos
Día Nacional del Trabajador de Construcción
Día Mundial del Consumidor
Marzo Día del Psicoorientador
El 22 de Marzo no es el día del optómetra
24-03-2020 Día del Locutor
27-3-2020 Día Internacional del Teatro
El 31-3-20 es el Día internacional contra el cáncer de Colon
1-4-2020 Abril Día del Controlador Técnico de Audio
8-05-20 Día de la Madre
01-06-20 Día Internacional del Niño
1-06-2020 Día del Campesino
05-6-20 Día del medio ambiente
14-06-2020 es el día mundial del donante de Sangre
17-6-2020 es el día del Higienista Dental
```

4.3. Ventajas y limitaciones

Emplear las expresiones regulares POSIX hace que la primera **toma de contacto y exploración de los datos sea inmediata, directa y llevable**. No hay que instalar nada, ya que las herramientas que hemos visto vienen por defec-

to o son muy fáciles de instalar con el gestor propio de la distribución en la gran mayoría de las distribuciones Linux y Unix. Así, desde la misma línea de comandos, podemos aproximarnos a los datos muy rápidamente.

Las herramientas que hemos visto, tanto Grep como Sed, y también otras, hacen **un uso eficiente de la memoria**, así que son adecuadas para tratar ficheros grandes. Esto es debido a que, dado que los ordenadores de los años sesenta eran muy limitados, están diseñadas para trabajar con esta limitación de recursos.

El uso de expresiones regulares supone un ahorro de trabajo y de mantenimiento. La alternativa de hacer un programa en su lugar no comporta estos mismos beneficios. Un programa de estas características tendría que ir mirando carácter a carácter e ir guardando el estado (que es lo que se hace con las expresiones regulares). Sería un programa con muchos casos, muchas reglas, difícil de seguir y muy propenso a contener errores. Si hubiera que realizar un cambio, sería extremadamente difícil de hacer. No hay duda de que es mucho mejor emplear una o más expresiones regulares, comentadas y documentadas.

Las RegEx son ideales para tratar secuencias de caracteres que presentan un cierto grado de uniformidad. Pero, dependiendo del tipo de documento con el que hayamos de trabajar, las expresiones regulares no son las más adecuadas, aunque pueden ser útiles para tratar patrones simples. Lo que conviene en estos casos (json, xml, html, etc.) es emplear un *parser* o una herramienta específica, como, por ejemplo, *xpath*, para extraer información de xml.

Bibliografía

Bibliografía básica

Barnett, B. (2019). «The Grymoire's tutorial on SED». Disponible en: www.grymoire.com/unix/sed.html.

Dougherty, D.; Robbins, A. (1997). *Sed & Awk: UNIX Power Tools*. Massachusetts: O'Reilly Media.

Shotts, W. (2009). «20 – Regular Expressions». *The Linux Command Line* (1.ª ed., págs. 261-281). Linuxcommand.org.

Referencias

Gnu.org (2018). «Sed, a stream editor». Disponible en: www.gnu.org/software/sed/manual/html_node/index.html#SEC_Contents.

Fox, J. (2019). «Regex cookbook — Top 10 Most wanted regex». Medium. Disponible en: medium.com/factory-mind/regex-cookbook-most-wanted-regex-aa721558c3c1.

Frazier, M. (2008). «Bash Regular Expressions». Linuxjournal.com. Disponible en: www.linuxjournal.com/content/bash-regular-expressions.