# Expresiones Regulares II: sed y awk

Alejandro Viloria Lanero (aviloria@infor.uva.es)
Teoría de Autómatas y Lenguajes Formales
Universidad de Valladolid

Una vez presentada la sintaxis de las expresiones regulares, nos disponemos a conocer un poco más a fondo qué herramientas del entorno UNIX / LinuX nos permiten un procesamiento de la información más eficiente utilizando las expresiones regulares, tanto para la localización, como para el reemplazo, así como otras operaciones de edición.

## 1 Introducción a la herramienta 'sed'

La herramienta 'sed' es un editor de textos no interactivo que contiene al editor de líneas 'ed'. Procesa el fichero desde el comienzo hasta el final y no permite la selección de órdenes de edición una vez invocado. No se guarda el fichero a editar en un buffer así que puede manejar ficheros de cualquier tamaño.

### 1.1 Ejecución de sed

| sed [-f\|e\|n] 'lista de comandos de ed' file/s | |
|---|---|
| -f | Lee los comandos ed desde un fichero. El nombre del fichero a continuación de la opción es el que leerá. |
| -e | Coloca varias órdenes de edición en la propia línea (un -e por cada orden). |
| -n | Suprime la salida normal de todas las líneas del fichero y sólo aparecen en la salida las líneas especialmente solicitadas con una operación de imprimir p o l (como p pero muestra los caracteres especiales). Sin la opción -n una salida p (ya la veremos posteriormente) duplica las líneas seleccionadas. |

Aquí aplica los comandos de la lista, por orden, a cada renglón y escribe el resultado en la salida estándar. Los ficheros no se modifican y el resultado del comando se da a la salida estándar. Todas estas opciones pueden combinarse unas con otras.

Un comando 'ed' tiene, en general, la forma

```
[linea[,linea]]operacion[parametro]
```

donde la operación puede ser buscar, cambiar, añadir, imprimir, etc. Por comodidad incluye un carácter especial \n para representar una nueva línea.

NOTA: A lo largo de todos estos ejercicios, veremos que los parámetros de sed van entrecomillados con el carácter " y no el '. La razón de esto es muy simple. El comando sed admite ambos formatos, pero el shell puede evaluar las variables en las comillas dobles. Por ello, si usamos "$VAR", sed utiliza el valor de la variable VAR pero '$VAR' lo tomaría literalmente como $VAR.

### 1.2 Búsqueda de cadenas

```
sed [opciones] "/patron/accion" file
```

Busca un patrón y realiza una acción sobre la línea encontrada. Hemos utilizado el carácter / para separar el patrón de búsqueda, pero podemos utilizar cualquiera que no aparezca en el patrón (como - ó ,). Se suele utilizar / por coherencia con 'ed'.

### 1.3 Sustitución de patrones

```
sed [opciones] "s/patron1/patron2/X" file
```

Busca un patrón y lo sustituye por otro. Aquí, el carácter X puede ser g (que realiza la sustitución en todas las apariciones de patron1 en la línea) o ser un valor numérico n (que realiza la sustitución de la n-aparición del patron1 en la línea).

### 1.4 Borrado de líneas

```
sed [opciones] "comando_ed d" file
```

Borra la línea a la que se refiere el comando 'ed'.

### 1.5 Escritura sobre otros ficheros

```
sed [opciones] "w file" file
```

Guarda la salida sobre file.

### 1.6 Lectura de un fichero

```
sed [opciones] "r file1" file2
```

Lee la entrada desde file1 y file2. Si no existiera file1 no aparecería error y sed continuaría con el file2 de entrada normal.

### 1.7 Salida del procedimiento

```
sed [opciones] "comando_ed q" file
```

Finaliza en cuanto acaba el comando 'ed'.

### 1.8 Trabajo sobre varios ficheros a la vez

Se pueden especificar varios ficheros como entrada de un comando sed de forma que la salida será la suma de todos. Si no se especifica ningún fichero, se toma como entrada la entrada estándar.

### 1.9 No correspondencia de patrón

Hasta ahora sed efectúa su trabajo sobre todas las líneas del fichero. Esto puede invertirse precediendo la operación por el carácter de admiración (!) que hace que el comando se realice sobre todas las líneas que no correspondan a las indicadas. Así, para saber todos los directorios que no son del grupo 'gestion':

```
$ sed -n /gestion/!p ll.lis > sed.out
```

Y para saber todos los directorios que no son del grupo 'gestion' ni 'users':

```
$ sed -n /gestion/!p ll.lis | sed -n /users/!p
drwxr-xr-x 3 root root 1024 Abr 1 1993 delia
```

Notar que esto no podría hacerse así

```
$ sed -n -e /gestion/!p -e /users/!p ll.lis > sed.out
```

o de forma más comprimida

```
$ sed -n -e "/(gestion|users)/!p" ll.lis > sed.out
```

pues en ambos casos el fichero sed.out contendría los mismos datos que ll.lis.

### 1.10 Añadir líneas al fichero

Pueden añadirse líneas a un fichero con los comandos a, i ó c. Dichos comandos (al contrario que con ed) deben ir seguidos por \ antes de añadir líneas, cada una de las cuales deberá también ir seguidas por \. Así

```
$ sed '$a\
> nueva linea1\
> nueva linea2\
> nueva linea' file_origen > file_destino
```

añade tres líneas a la salida del fichero origen.
Con i se inserta en una posición dada. Por ejemplo, para insertar una nueva línea en la segunda del fichero origen

```
$ sed '2i\
> nueva lineai' file_origen > file_destino
```

Por último con c se cambia la línea o líneas dadas por las siguientes (puede ser varias por una o una por varias).

### 1.11 Ámbito del fichero

Hasta ahora hemos hecho cambios sobre todo el fichero. Sin embargo es posible referirse a líneas de él. Estas pueden referirse de forma absoluta (así 2,24 son las líneas que van desde la 2 hasta la 24) o en forma de contexto (/AMAL/,/SANCHO/ líneas que van desde la primera aparición de la palabra AMAL hasta la primera aparición de SANCHO a partir de la anterior). Podemos combinar valores absolutos con expresiones obtenidas de contexto.

### 1.12 Guiones

Podemos preparar guiones de comandos ‘ed’ que puedan ser parámetros del programa sed. En ellos pueden incluirse varios comandos seguidos que podrán ser ejecutados. El programa sed realiza sobre los comandos un preproceso de forma que evalúa en qué orden deben ejecutarse. Así, antes de insertar o realizar una búsqueda, borrará las líneas que se pidan, etc.

### 1.13 Pero, ¿en qué línea estamos?

Para saber en qué línea nos encontramos utilizamos el significado especial del carácter =. Al contrario que en ‘ed’, aquí no es posible hacer una referencia relativa a una línea actual. Así, no podríamos decirle que cambiara la línea anterior a una dada por otra. Los datos que contiene la línea se representan simbólicamente por el meta-carácter &.
El problema del significado especial del carácter = es que sólo tiene valor especial en el caso de ser una acción a ejecutar. Por ello el comando

```
$ sed -n "s/.*/= &/p" ll.lis > out.lis
```

no pone el número de línea en cada línea. La salida sería algo como

```
= drwxr-xr-x 2 abad gestion 2048 May 6 1993 abad
= drwxrwx--- 2 acceso gestion 1024 Mar 22 11:47 acceso
= drwxr-xr-x 2 administ gestion 1024 Feb 16 15:42 administ
(etc)
```

Aunque hay una forma (bastante complicada) de conseguir con sed cosas como

```
1 drwxr-xr-x 2 abad gestion 2048 May 6 1993 abad
2 drwxrwx--- 2 acceso gestion 1024 Mar 22 11:47 acceso
3 drwxr-xr-x 2 administ gestion 1024 Feb 16 15:42 administ
(etc)
```

veremos que esto lo hacen otros programas de forma mucho más eficiente.

## 2 Introducción a la herramienta ‘awk’

Algunas limitaciones del ‘sed’ se remedian con el ‘awk’. Este es igual al ‘sed’ pero más potente y utiliza comandos de C en vez del ‘ed’.

### 2.1 Ejecución de sed

```
awk 'programa' archivos ...
```

donde ‘programa’ puede ser:

```
/patrón/{acción} /patrón/{acción} ...
```

‘awk’ lee la entrada de archivos un renglón a la vez. Cada renglón se compara con cada patrón en orden; para cada patrón que concuerde con el renglón se efectúa la acción correspondiente.

### 2.2 Búsqueda de cadenas

```
awk '/expresión/{ print }' archivos
```

Imprime cada línea que encaja con la expresión regular especificada.

```
awk '{ print }' archivos
```

Puede obviarse la expresión. Muestra todas las líneas del fichero.

### 2.3 Extracción de campos

El ‘awk’ divide automáticamente la entrada del renglón en campos, es decir cadena de caracteres que no sean blancos separados por blancos o tabuladores, por ejemplo el ‘who’ tiene 5 campos

```
user1    tty2   Sep 29  11:53
user2    tty3   Sep 29  14:54
```

El ‘awk’ llama a estos campos $1 $2 ... $NF donde NF es una variable igual al número de campos. En este caso NF=5. Por ejemplo:

```
du -a | awk '{print $2}'
```

sólo imprime los nombre de los archivos.

```
who | awk '{print $5, $1}' | sort
```

imprime los usuarios ordenados y su hora de conexión (primero este campo).

También pueden usarse expresiones aritméticas:

```
awk '{print $1*$1, $2 - 5.0}' arch
```

Lo cual imprime el cuadrado de la primera columna, etc.

### 2.4 Formatos de impresión

También puede configurarse el formato de la línea al estilo que se hace en C:

```
awk '{printf "%4d %s\n", NR, $0}'
```

imprime un número decimal reservando espacio para 4 dígitos (NR = número de línea actual), un string ($0 = la línea completa), y salto de línea.

```
awk '{print NR, $0}
```

## 2.5 Condicionando las acciones

La expresión puede escribirse de forma que la acción se ejecute si se cumple una determinada condición.

```
awk ' $3 > 0 { print $1, $2*$3} ' datos.emp

awk ' $1~/a.*/ { print $1, $2*$3} ' datos.emp
```

Los operadores especiales más utilizados son:

| | |
|---|---|
| ~ | indica coincidencia con una expresión |
| !~ | significa lo contrario (no coincidencia) |
| ! | significa negación |
| $2 == "" | si el segundo campo es vacío |
| $2 ~ /^$/ | si el segundo campo coincide con la cadena vacía |
| $2 !~ /./ | si el segundo campo no concuerda con ningún carácter |
| length($2) == 0 | si la longitud del segundo campo es 0. length es una función integrada en 'awk' |
| NF % 2 != 0 | muestra el renglón solo si hay un número par de campos. |
| substr($2,1,20) == $3 | Si la subcadena de $2, desde el carácter 1 al 20, coincide con el campo $3. substr e suna función integrada en 'awk' |

## 2.6 Patrones especiales

Existen dos patrones especiales BEGIN y END. Las acciones BEGIN se realizan antes que el primer renglón se haya leído; puede usarse para inicializar las variables, imprimir encabezados o posicionar separadores de un campo asignándoselos a la variable:

```
awk 'BEGIN {FS = ":"} $2 == "" ' /etc/passwd
```

inicializa la variable global FS (especifica el carácter de separación de los campos), antes de buscar a los usuarios en el fichero passwd que no han especificado su clave.

```
awk 'END { print NR } '
```

imprime el número de líneas procesadas al final de la lectura del último renglón.

## 2.7 Variables locales

```
awk '{ s = s + $1} END { print s, s/NR}'
```

Tal y como ya hemos comentado, 'awk' permite operaciones numéricas. Por ejemplo para sumar todos los números de la primera columna y mostrar su media.

Las variables se inicializan a cero al declarase y se declaran al utilizarse por lo cual es muy simple. Los operadores son los mismo que en C.

También 'awk' maneja arreglos, por ejemplo para hacer un 'tail' de fichero.c:

```
awk '{ line[NR] = $0} END { for (i=NR; i 0; i--) print line[i]}
' fichero.c
```

### 2.7 Variables predefinidas

| | |
|---|---|
| FILENAME | nombre del archivo actual |
| FS | carácter delimitador del campo |
| NF | número de campos del registro de entrada |
| NR | número del registro de la entrada |
| OFMT | formato de salida para números (%g por defecto) |
| OFS | cadena separadora de campo en la salida (blanco por defecto) |
| ORS | cadena separadora de registro de salida (new line por defecto) |
| RS | cadena separador de registro de entrada (new line por defecto) |

### 2.8 Funciones predefinidas

cos(), exp(), getline(), index(), int(), length(), log(), sin(), split(), sprintf(), substr()

### 2.9 Sentencias de control

Además soporta dentro de acción sentencias de control con la misma sintaxis que C:

```
if (condición)

proposición1

else

proposición2
```

```
for (exp1;condición;exp2)

expr1
```

```
while (condición) {

proposición

expr2

}
```

Otras instrucciones que nos ayudan al control del flujo son:
**continue**: sigue, evalúa la condición nuevamente
**break**: quiebra la condición
**next**: lee la siguiente línea de entrada
**exit**: salta a END

## 2.10 Ejemplos de awk

Este ejemplo obtiene la hora y la transforma en letras

%cat > vdate.a

```
#
BEGIN {
units = "zero one two three four five six seven eight nine"
double = \
"ten eleven twelve thirteen fourteen fifteen sixteen seventeen eighteen nineteenn
tens = "twenty thirty fourty fifty"
split(units, spku); split(double, spkd); split(tens, spkt)
# obtain the time
{split($4, time, ":")
# obtain hour of the day (zero .. twenty-three)
if (time[1] < 20) {
if (time[1] < 10) hour = spku[time[1] + 1]
else hour = spkd[time[1] -10) +11}
else {
if ((time[1] % 10) < 1) hour = spkt[int(time[1] / 10)  -1]
else hour = spkt[int(time[1] / 10) -1]1 " " spku[(time[1] % 10) + 1]}
# obtain minutes of the hour (zero fifty-nine)
if (time[2] < 20) {
if (timet2] < 10) minute = spku[time[2] +1]
else minute = spkd[(time[2] -10) +1]}
else {
if ((time[2] % 10) < 1) minute = spkt[int(time[2] / 10)  -1]
else minute = spkt[int(time[2] / 10) -1] " " spku[(time[2] % 10) + 1]}
# obtain seconds of the minute (zero fifty-nine)
if (time[3] < 20) {
if (time[3] < 10) second = spku[time[3] + 1]
else second = spkd[(time[3] -10) + 1 ]}
else {
if ((time[3] % 10) < 1) second = spkt[int(time[3] / 10) -1]
else second = spkt[int(time[3] / 10) -1] " " spkut(time[3] % 10) + 1]}
}
END{
printf "La Hora es \n"
printf "%s horas %s minutos y %s segundos exactamente\n",hour,minute,second
}
```

%date ; date | awk -f vdate.a

Tue Feb 15 15 47 36 GMT 1994

La Hora es
fifteen hours fourty seven minutes y thirty six seconds exactamente