Automatizar y programar ediciones de texto con SED (versión GNU)

Joaquín Ataz López

1 de enero de 2011

Resumen

SED es un «editor programable» y no interactivo, ideal para automatizar la modificación de ficheros de texto. En la presente guía se explica su funcionamiento, incluyendo una explicación completa del conjunto de expresiones regulares usado por SED que es el mismo que usan numerosas aplicaciones GNU como «ed» o «grep».

Índice

1. Introducción

1.1. Qué es SED y para qué se puede usar

SED* se define a sí mismo como «editor de flujos de texto». Pero esa definición no es muy clara, que digamos. Yo prefiero, además, considerarlo como un «editor programable». Es decir: SED nos permite programar qué modificaciones queremos hacer a un fichero de texto, de tal modo que luego se pueden realizar de forma automática y sin necesidad de que el usuario tenga que intervenir. Esto convierte a SED en una herramienta ideal para:

- Aplicar el mismo cambio (o los mismos cambios) a una multitud de ficheros.
- Realizar con rapidez un cambio sencillo en un fichero de cierta extensión.

El nombre SED proviene de **s**tream **ed**itor, que podríamos traducir por «editor de flujos». Eso significa que SED lo que hace es filtrar un «flujo de texto»: recibe un texto, le aplica ciertas modificaciones previamente programadas y especificadas en un guión, y devuelve el texto resultante.

- El texto recibido puede provenir de cualquier entrada de datos admisible (lo que es relativamente corriente en las aplicaciones de UNIX). Habitualmente se usa un fichero, pero puede ser la salida de otro programa, o el teclado, si bien, esto último, sólo es útil para hacer pruebas mientras estamos aprendiendo a usar SED.
- El texto devuelto se envía a la salida estándar (normalmente la pantalla), a no ser que se utilice un redireccionamiento[1].
- Las instrucciones de manipulación del texto se pueden incluir en la propia línea de comandos, o guardarlas en un fichero para que SED las lea de él.

*__Nota del autor:__ Para diferenciar gráficamente los nombres de programas, o de sistemas informáticos, en el presente documento los escribo en mayúsculas, aunque, en realidad, los programas al menos, se escriben en minúsculas. Así, por ejemplo, escribo SED, ED, GREP, UNIX... Esto me ahorra tener que entrecomillar cada aparición de uno de esos nombres. Pero el lector debe tener en cuenta que eso es sólo un recurso tipográfico.

[1] Las nociones de entrada, salida, redireccionamiento, tubería... son nociones básicas usadas en los sistemas UNIX. No las explico porque asumo que nadie que las desconozca intentaría nunca utilizar un programa como SED. Pero si no se conocen, recomiendo leer alguna introducción básica a UNIX.

Las características de SED que lo convierten en una herramienta tan potente, son las siguientes:

1. Su capacidad para usar cualquier entrada de datos, que hace hace de él una herramienta ideal para, en combinación con el uso de tuberías de UNIX, filtrar y modificar la salida de cualquier otro programa.
2. La no necesidad de interacción con el usuario, que lo convierte en la herramienta idónea para aplicar cambios automáticos a un fichero de texto.
3. Su capacidad para aplicarse a varios ficheros simultáneamente, que hace de él una herramienta muy adecuada para aplicar, con una sola orden, cambios similares a un grupo grande de ficheros.
4. En fin: su manejo de las expresiones regulares, que permite programar cambios verdaderamente complejos, e incluso escribir «scripts» o «guiones» que constituyen verdaderos «programas». De hecho SED está considerado como un lenguaje de programación «turing-completo» y es el precursor de AWK y de PERL.

Podríamos decir que, aplicado a los usos en los que SED destaca, la diferencia entre usar SED o un editor interactivo ordinario es la misma que la que hay entre realizar trabajos de carpintería con una sierra eléctrica o con una sierra de mano. La sierra eléctrica mal manejada puede provocar accidentes, pero bien manejada permite realizar el trabajo mucho antes que con una sierra manual, y mucho mejor. Aunque hay trabajos que, por el nivel de precisión que requieren, es preferible hacerlos con sierra manual.

Ahora bien: SED es un programa peculiar, que no siempre está bien documentado, y del que apenas hay información en español. Su aprendizaje lleva cierto tiempo ya que, sobre todo, exige comprender que podemos hacer cosas que, si se ignora la existencia de SED, simplemente no se nos pasaría por la cabeza ni siquiera intentarlas. Por eso, al principio, manejar SED puede parecer que supone escoger "la forma difícil de acometer una tarea sencilla". Pero esto forma parte del aprendizaje, ya que, además de aprender a usar SED hay que aprender a reconocer las situaciones en las que SED ofrece unas utilidades de las que otras herramientas carecen.

Como ejemplo de situaciones en las que SED ofrece utilidades de las que otras herramientas carecen, puedo poner la página web que tengo dirigida a los alumnos de la Facultad de Derecho de la Universidad de Murcia. En ella (`http://webs.um.es/jal`) se alojan numerosas leyes que están escritas en formato XML: habitualmente cuando una ley menciona a otra ley, si la ley mencionada está también alojada en mi sitio web, incluyo un enlace al texto de la ley citada. Y eso significa que cada vez que incorporo una ley nueva al sitio web, tengo que revisar todas las leyes viejas, para comprobar si citan o no a la ley que acabo de incorporar, y, en caso afirmativo, añadir el enlace.

Pues bien: si decido incluir una nueva ley en la base de datos, y quiero insertar un enlace a ella en todas las leyes que la mencionen, puedo hacerlo de las siguientes maneras:

1. Editar manualmente todas las leyes incluidas en mi página web (más de 100), buscar en ellas si se menciona a la nueva ley, y, en caso afirmativo, añadir a mano el enlace.

2. Bastante más rápido: Hacer previamente una búsqueda con GREP que me indique qué ficheros son los que hay que editar, porque mencionan a la ley que acabo de incluir, y luego, editar manualmente esos ficheros. Nos ahorramos así la edición de varios de los ficheros.

3. Usar SED para hacer los cambios. El resultado será el mismo, pero mucho más rápido.

He mencionado, como paso intermedio entre la edición manual y la edición con SED, el uso de GREP, porque SED presupone GREP. GREP nos permite detectar, en un conjunto de ficheros de texto, cuáles de ellos contienen una determinada expresión. SED va un paso más allá: nos permite identificar esos ficheros, y modificarlos directamente.

Otra situación para la que SED es bueno es para convertir ficheros de texto de un formato a otro. Por ejemplo: Si queremos convertir un conjunto de ficheros HTML a LaTeX, o tenemos una biblioteca basada en LaTeX que queremos convertir al moderno formato EPUB para libros electrónicos... SED podría ser la herramienta que, previo análisis detallado de cómo están escritos nuestros documentos, nos permitiría realizar la conversión entre formatos con mucha más sencillez y rapidez que haciéndola manualmente.

1.2. ED y GREP, los antecesores de SED

El editor originalmente incluido en UNIX era ED. Se trataba de un editor de líneas, pensado para poder funcionar en sistemas con pocos recursos gráficos; a veces incluso en sistemas que carecían de pantalla propiamente dicha, como, por ejemplo, un teletipo. Por ello ED fue concebido como un "editor de líneas": leía y modificaba las líneas de un fichero de una en una y ello lo hacía, no de forma interactiva, como los actuales editores de texto, sino mediante comandos dirigidos a provocar alteraciones en la línea editada. Se preveían una serie de alteraciones *típicas* y, para cada una de ellas, se disponía de un comando específico que la realizaba.

De ED, SED ha heredado varias características. En particular:

1. El hecho de que la unidad básica de trabajo sea la "línea". Tanto ED como SED, dividen los ficheros de texto en líneas y, en principio, sus comandos actúan línea a línea.

   > SED incorpora un conjunto de comandos que permiten trabajar simultáneamente con varias líneas y que analizaremos en la sección 5.5. Pero aún así es innegable que SED está pensado para trabajar con líneas individuales, y el trabajo en modo multilínea resulta siempre más engorroso y difícil.

2. El nombre de varios de sus comandos, que también ha sido heredado por otros editores, como por ejemplo el popular VI y sus derivados, incluyendo VIM. Por lo tanto, si conocemos el manejo de VIM (cosa que recomiendo a todo escritor de ficheros de texto[2]) muchos de los comandos de SED nos resultarán familiares.

[2]Hace años escribí una guía básica para aprender a manejar VIM. Puede descargarse de internet en http://webs.um.es/jal/docs/Guia-Vim.pdf.

3. La capacidad de programar las ediciones. Porque aunque ED inicialmente estaba pensado como editor interactivo, era posible también escribir guiones autoejecutables que realizaran las modificaciones de modo no interactivo.

4. En fin: el uso de las expresiones regulares, si bien estas antes de pasar a SED fueron depuradas y mejoradas por otro de los sucesores de ED: GREP, cuyo nombre representa un comando de ED: «`g/re/p`» que podríamos traducir por "busca e imprime todas las apariciones en un fichero de cierta expresión regular" («`g`» era el comando "global" que significaba en ED que el siguiente comando no se refería sólo a la línea actual, sino a todo el fichero. El carácter «`/`» se usaba (y usa) para delimitar un texto a buscar; "re" son las iniciales en inglés de "*regular expression*", y «`p`» es el comando para imprimir ("print").

1.3. Versiones de SED

Como es lógico existen diferentes versiones de SED. La versión que pudiéramos llamar estándar de UNIX es la versión conocida como POSIX.

En los sistemas LINUX es corriente que encontremos instalada la implementación GNU de SED. En ella se añaden a SED numerosas habilidades, y se le quitan bastantes limitaciones. La versión POSIX, por ejemplo, no podía manejar líneas que superaran cierto número de caracteres.

Como GNU es software libre, y mejora bastante a la versión POSIX, hoy día es raro encontrar implementaciones que no incluyan las mejoras GNU. La documentación oficial de dicha versión, además, incluye una completa información sobre qué aspectos de SED pertenecen a la versión POSIX y cuáles son un añadido GNU. En el presente documento he eliminado esa información, porque entiendo que usar la versión POSIX, estando disponibles las extensiones GNU, es una pérdida de tiempo.

La versión de SED para la que se ha hecho este documento es la 4.2.1 y fue liberada en el año 2009.

1.4. Obtener ayuda de SED

En un sistema linux estándar, existen tres fuentes de información sobre SED que normalmente estarán instaladas. De más extensa o menos extensa son:

1. La página info de SED. Se invoca ejecutando en una consola «`info sed`» y contiene un manual del programa, en inglés, en el que se recogen con cierto detalle todas las características de SED, y se incluye un grupo muy numeroso de ejemplos que van desde algunos triviales hasta otros verdaderamente complejos. La mayor parte de este documento ha salido de dicha página info.

2. La página man de SED. Se invoca ejecutando desde una consola «`man sed`» y contiene un resumen de la anterior información: es útil para recordar nociones que ya se saben, pero aprender con ella a manejar SED y a aprovechar siquiera una mínima parte de su potencia, es bastante difícil.

3. La opción «`--help`» de SED. Se invoca ejecutando desde una consola «`sed --help`» o simplemente «`sed`» sin ningún otro argumento, y muestra en pantalla una explicación todavía más resumida del uso del programa. Esta información nos puede aclarar qué está mal en nuestra línea de comandos y por qué SED no funciona... pero desde luego no está pensada para enseñar a manejar el programa.

1.5. Dos comandos de SED para empezar

En las páginas que siguen vamos a explicar SED. Pero para poder poner ejemplos, lo ideal es que ya, desde el principio, conozcamos algún comando de SED que nos permita usar ejemplos reales. Y para esa finalidad he elegido el comando más usado de SED, y también el más potente: el comando «`s`» cuyo efecto es sustituir una cadena de texto por otra.

Más adelante (en el epígrafe 5.1.1) explicaré con detenimiento dicho comando. De momento sólo indicaré su uso general que se ajusta a la siguiente sintaxis:

```
s/Texto de búsqueda/Texto de reemplazo/g
```

«`s`» es el nombre del comando, a continuación una barra que indica que en ese punto se inicia el el texto que hay que buscar; una segunda barra indica que en ese punto termina el texto a buscar y empieza el texto con el que queremos reemplazar el anterior; y finalmente una tercera barra que indica que allí termina el texto de reemplazo. La «`g`» con la que acaba el comando indica a SED que queremos hacer una sustitución "global" del texto; si no la incluyéramos SED reemplazaría, en cada línea, exclusivamente la primera aparición del texto de búsqueda.

Por ejemplo:

```
s/unix/linux/g
```

cambiaría, en nuestro documento, todas las referencias a unix por referencias a linux.

Hay dos características importantes de este comando que, de momento, hay que retener:

1. «`s`» busca una secuencia de texto, y no una palabra; y si, por ejemplo, escribimos «`s/codo/muñeca/g`» y en nuestro texto aparecía la palabra "recodo", el comando la convertirá en "remuñeca".

2. «`s`», como regla, diferencia entre mayúsculas y minúsculas. Por tanto

   ```
   s/día/noche/g
   ```

   no cambiará las apariciones de "día" en las que aparezca alguna letra mayúscula como "Día", "DÍA", "díA", etc.

Un segundo comando, que usaré a veces en los ejemplos, es el comando «`d`» (de "delete") cuyo efecto es borrar la línea completa a la que se aplica.

2. Aspectos básicos de SED

2.1. Nociones fundamentales de SED

SED recibe un texto de entrada al que aplica una serie de comandos. El conjunto de comandos se aplica a cada línea del texto, en una especie de bucle o ciclo que empieza en la línea 1 del texto y acaba en la última.

En consecuencia las nociones fundamentales de SED son:

- **Texto de entrada:** Es el texto con el que SED trabaja. Puede provenir de uno o varios ficheros, de la salida de otro programa, o de *stdin*. Cuando proviene de varios ficheros, por defecto SED une el contenido de éstos que serán tratados como un sólo gran fichero, a no ser que se haya activado la opción «separate» (o la opción «in-place» que implica el uso de la opción «separate»). Sobre estas opciones véase la sección 2.3.2.

- **Comando:** La manipulación que SED hace en el texto consiste en aplicarle «comandos». El nombre de los comandos de SED consta siempre de una sola letra que, habitualmente, se corresponde con la inicial del nombre en inglés de la acción realizada por el comando, y así el comando para borrar es «d» (de *delete*) y el comando para imprimir es «p» (de *print*). De cara a los nombres de los comandos es importante tener en cuenta que SED también en ellos distingue entre mayúsculas y minúsculas y, por lo tanto, el comando «`d`», por ejemplo, no equivale al comando «`D`»; aunque sí es cierto que cuando dos comandos usan la misma letra, uno en mayúsculas y el otro en minúsculas, existe algún tipo de relación ente ambos comandos: hacen ambos algo parecido, aunque diferente.

  Se explica con más detalle el formato general de los comandos en la sección 4.1, y los concretos comandos se explican en la sección 5. En el apéndice se contiene un listado completo de los comandos, indicando cada uno de ellos en donde se explica.

- **Ámbito:** En SED los comandos tienen (o pueden tener) un ámbito que indica a qué líneas del texto de entrada se aplican. Los ámbitos y su forma de ser especificados se explican en la sección 4.2.

- **Guión o Script:** Un guión (en inglés *script*) es el conjunto de comandos de SED que hay que aplicar al texto de entrada. El guión, contiene, por lo tanto, la especificación detallada de todas las modificaciones que hay que hacer al texto, indicando qué comandos aplicar, en qué orden, y a qué líneas debe aplicarse cada cual.

  Sobre la forma de indicarle a SED el guión que debe ejecutar, véase la sección 2.4.

- **Expresión regular** Es un patrón de caracteres que tiene ciertas características. Se usa en SED para determinar a qué líneas aplicar cierto comando y en el comando «`s`», para indicar que un texto debe ser sustituido por otro.

  A la sintaxis de las expresiones regulares se dedica por completo la sección 3 de este documento.

- **Ciclo:** SED procesa el texto de entrada por líneas. Se llama ciclo al conjunto de acciones que empieza cuando SED carga una línea nueva en el área de trabajo y termina cuando SED termina de trabajar con dicha línea y pasa a la siguiente (el siguiente ciclo).

  El ciclo, por otra parte, es relativamente sencillo: Empieza cuando se carga una nueva línea en el área de trabajo. Tras cargarla, SED comprueba si alguno de los comandos del guión debe aplicarse a esa línea, en cuyo caso así lo hace. Tras ello se envía el contenido del área de trabajo a la salida estándar (salvo que se haya activado la opción «no-autoprint»), y se inicia un nuevo ciclo.

- **Área de trabajo:** Cuando SED carga una línea nueva, la almacena en una zona de memoria llamada «área de trabajo». Los comandos de SED normalmente actúan sobre el contenido del área de trabajo.

- **Área auxiliar:** Junto con el área de trabajo, SED gestiona otra zona de memoria denominada «área auxiliar». En este área se pueden guardar datos para usarlos más tarde. Un conjunto de comandos de SED permiten comunicar entre sí los contenidos del área de trabajo y del área auxiliar.

2.2. Invocación de SED

SED es un editor no interactivo. Para invocarlo desde la línea de comandos se usa el siguiente formato:

```
sed [OPCIONES] [GUIÓN] [FICH_1] [FICH_2]... [FICH_N]
```

donde:

- **Opciones:** Cualquiera de las admitidas por SED y que se explican en el próximo epígrafe. Las opciones empiezan siempre por el carácter «-».

- **Guión:** Se refiere al conjunto de comandos de SED que hay que aplicar sobre el texto de entrada. Puede consistir en una cadena de texto o en el nombre de un fichero que lo contenga. No hay que especificarlo si se ha usado algunas de las opciones que permiten indicar el guión (véase la sección 2.3.1).

  Como SED no puede trabajar sin guión, el programa es muy flexible respecto a cómo indicársele cual es el guión. En la sección 2.4 he resumido todas las posibilidades.

- **Ficheros:** Se puede indicar uno o varios ficheros. Para indicar varios ficheros se pueden separar sus nombres mediante un carácter en blanco, o también se puede invocar simultáneamente a varios ficheros usando alguno de los comodines de la *shell*.

  Cuando SED es invocado en el lado derecho de una tubería, no hay que indicar ficheros, pues SED tomará su entrada de la salida del programa que esté al otro lado

de la tubería. En los demás supuestos, si no se indica algún fichero, o, como nombre de fichero, se indica «-», SED tomará los datos de la entrada estándar (*stdin*), que habitualmente es el teclado. Esto es, en general, poco eficaz, pero nos puede servir para comprobar cómo funcionaría cierto comando en una frase. Por ejemplo, si en nuestro terminal escribimos:

```
$ sed -e "s/perro/gato/g"
```

Sed quedará esperando que introduzcamos una frase para aplicar en ella el comando que acabamos de escribir. Podemos, por lo tanto ir probando distintas frases: cada vez que pulsemos la tecla ENTER, sed nos devolverá la frase tras aplicarle el comando: Si, por ejemplo, a continuación escribimos "tengo tres perros ladradores" sed nos devolverá "tengo tres gatos ladradores".

```
$ sed -e "s/perro/gato/g"
tengo tres perros ladradores
tengo tres gatos ladradores
```

y seguirá en espera de una nueva frase, hasta que pulsemos CTRL-C, para interrumpir la ejecución del programa.

2.3. Opciones de SED en la línea de comandos

SED admite varias opciones en la línea de comando. Podemos agruparlas atendiendo a su finalidad, y así tendremos:

2.3.1. Opciones para indicar el guión a ejecutar

Se trata de dos opciones, aunque cada una de ellas admite dos formas de escritura, por lo que, en total, son cuatro las opciones que podemos usar:

1. Para indicar el guión como una cadena de texto podemos usar las opciones «`-e "texto"`» o «`--expression="texto"`».
2. Para indicar un nombre de fichero cuyo contenido es el guión a ejecutar, disponemos de otras dos opciones: «`-f NombFich`» y «`--file=NombFich`», donde `NombFich` es el nombre del fichero que contiene el guión.

Una explicación más detallada del uso de estas opciones se encuentra en la sección 2.4 de este documento.

2.3.2. Funcionamiento por defecto de SED y opciones que permiten alterarlo

Un grupo de opciones de SED afecta a su funcionamiento por defecto. Para entenderlas bien, como es obvio, hay que empezar en cada una de ellas explicando como es el funcionamiento "normal" de SED.

- **Lo que SED envía a la pantalla**: Por defecto SED, tras procesar una línea, la envía a la salida estándar (normalmente la pantalla). Este funcionamiento, al que la documentación de SED suele referirse con el nombre de *autoprint*, se puede alterar mediante cualquiera de las siguientes tres opciones (que son absolutamente intercambiables): «`-n`», «`--quiet`» y «`--silence`».

  Estas opciones activan el modo «no autoprint» en virtud del cual SED sólo enviará datos a la salida estándar cuando expresamente se le indique mediante los comandos «`p`», «`P`», «`l`», «`L`» o «`=`».

  El mismo efecto se obtiene si los dos primeros caracteres del guión a ejecutar son "#n" (pero no "#N" porque SED distingue entre mayúsculas y minúsculas).

- **Manejo de múltiples ficheros:** Cuando SED trabaja sobre varios ficheros, los yuxtapone de tal modo que todos son tratados como un único flujo de datos en el que todas las líneas se cuentan secuencialmente; de modo que si, por ejemplo, el primer fichero tenía 86 líneas, la primera línea del segundo fichero será considerada la línea 87.

  Este funcionamiento cambia si se usan las opciones «`-s`» o «`--separate`». El efecto de cualquiera de ellas es que cada fichero sea tratado de forma separada a los demás ficheros. Esto, a su vez, tiene varias consecuencias, sobre todo en la manera de indicar el ámbito de los comandos de SED (véase la sección 4.2) ya que:

  — Ningún comando podrá tener un ámbito que implique líneas de más de un fichero.

  — La numeración de líneas de cada fichero se hará con respecto a su propio inicio.

  — El carácter "$", que en SED representa la última línea del fichero, pasará a representar la última línea *de cada fichero*.

  — Los ficheros invocados con el comando «`R`» se releerán cada vez que se inicie el procesado de un nuevo fichero.

- **Modificación del fichero original:** SED no modifica el fichero original. El resultado de sus manipulaciones lo envía a la pantalla (o al lugar hacia donde se redirija la salida).

  Las opciones «`-i`» o «`--in-place`» alteran este comportamiento y provocan que, tras la manipulación del texto, se guarde, en lugar del fichero original, un fichero con el resultado de la ejecución de SED. Para hacer esto, si se están manejando varios ficheros, SED tiene que tratarlos de forma separada, y por ello el uso de la opción «`-i`» implica que se activará también la opción «`-s`».

Tanto «`-i`» como «`--in-place`» aceptan un argumento opcional consistente en un *sufijo* que, si es introducido, provoca que se guarde una copia de respaldo del fichero original, antes de sustituirlo, cuyo nombre será el del fichero original, seguido del texto del sufijo. Por ejemplo, si el fichero original se llamaba "MiFich.txt", el comando

```
sed --in-place=.bak -e '14d' MiFich.txt
```

guardará el contenido actual de MiFich.txt en MiFich.txt.bak, y guardará en MiFich.txt la versión resultante de aplicar a su contenido el comando «`14d`» el cual lo que hace es borrar la línea 14.

La forma de indicar este argumento varía según se use la forma «`-i`» o la forma «`--in-place`». En el primer caso el texto del sufijo se debe escribir inmediatamente después de la opción (sin usar ningún espacio de separación). En el segundo caso, entre el nombre de la opción y el texto del sufijo hay que introducir el carácter «=». Así, las siguientes dos órdenes son equivalentes:

```
sed --in-place=.bak -e '4d' MiFich.txt
sed -i.bak -e '4d' MiFich.txt
```

Por otra parte el carácter «`*`» usado en el nombre del sufijo provoca que el texto ya no se considere un *sufijo* sino que el nombre del fichero de respaldo se construirá a partir del texto especificado, pero sustituyendo el asterisco por el nombre completo del fichero original. Esto nos da mucha flexibilidad para determinar el nombre de la copia de respaldo y así, por ejemplo, el comando

```
sed -iRES* -e '4d' MiFich.txt
```

provocará que en lugar de un sufijo, para generar la copia de respaldo se use el prefijo RES. Mientras que el comando:

```
sed -iRespaldos/* -e '4d' MiFich.txt
```

hace que la copia de respaldo se guarde, con el mismo nombre del fichero original, en el directorio "Respaldos" (que debe existir previamente).

- **Conjunto de expresiones regulares a utilizar:** Por defecto SED utiliza el conjunto "normal" de expresiones regulares de GNU, que es el mismo que usan ED y GREP. Pero con las opciones «`-r`» o «`--regexp-extended`» se activa el uso del conjunto extendido de expresiones regulares usado por EGREP. Estas *expresiones regulares extendidas* se explican en la sección 3.6.

- **Compatibilidad de SED:** Existen diferentes implementaciones de SED. En sistemas Linux se incluye habitualmente la versión de GNU que incluye una serie de mejoras y extensiones sobre el SED estándar original. Pero si queremos desactivar todas esas funciones, para asegurarnos de que nuestro guión es totalmente compatible con cualquier versión de SED, podemos usar la opción «`--posix`» que inhabilita todas las extensiones GNU.

- **Otros aspectos de SED controlables mediantes opciones:**

  Las siguientes opciones afectan al funcionamiento de SED en aspectos de importancia secundaria o, al menos, más coyuntural:

  — «`-l`» ó «`--length`» Estas opciones indican el ancho de línea por defecto para el comando «`l`» (véase la sección 5.2). En la versión «`-l`» el ancho se introduce inmediatamente después del nombre de la opción (sin dejar espacios en blanco) y en la versión «`--length`» el número se separa del nombre de la opción con el signo «=».

  — «`-b`» o «`--binary`»: Estas opciones tienen sentido sólo en sistemas como MS-DOS o MS-Windows, que abren de modo distinto los ficheros de texto y los binarios. Mediante ellas se fuerza a SED a abrir un fichero de texto como si fuera binario. No estoy muy seguro de eso para qué se hace, aunque supongo que tiene que ver con el hecho de que en esos sistemas, para indicar el salto de línea no se usa un solo carácter (LF) sino dos caracteres (LF seguido de CR)[3].

  — «`--follow-symlinks`»: En sistemas que admiten ficheros que apuntan a otros ficheros (enlaces simbólicos), como UNIX, esta opción fuerza a SED a que si uno de los ficheros con los que está trabajando es un enlace simbólico, en lugar de ese fichero abra el fichero de destino de dicho enlace.

  — «`-u`», «`--unbuffered`»: Por defecto SED (y la mayoría de aplicaciones estándar GNU), cuando va a leer un fragmento del fichero, carga en memoria más datos de los que necesita, porque, según la experiencia, es probable que más adelante se necesiten esos datos, y teniéndolos ya cargados, se reducen los accesos a disco y se acelera la ejecución del programa. Ahora bien: si esos datos extra no se iban a usar, haberlos cargado en memoria habrá sido una pérdida de tiempo.

  Si abrimos un fichero grande para hacer en él una edición muy reducida, puede ser buena idea especificar las opciones «`-u`» o «`--unbuffered`», cuyo efecto es que se reduzca, tanto como sea posible, el tamaño de los buffers de memoria. Esto, a su vez, se traduce en que el programa se ejecutará más rápido (si se necesitaban pocos datos del fichero) o más lento (si se necesitaban muchos datos del fichero y al reducir los buffers aumentamos la cantidad de veces que hay que acceder a él).

2.3.3. Opciones para obtener información sobre SED

Las dos opciones de SED que quedan por explicar sirven para informar sobre el programa propiamente dicho. Se trata de:

- «`--version`»: Esta opción imprime en pantalla el número de versión de SED que está instalada en el sistema, así como un mensaje de copyright, y termina.

[3] LF son las iniciales en inglés de *line feed*, se corresponde con el código ASCII 10 y se suele representar como \n. CR son las iniciales de *carriage return*, se corresponde con el código ASCII 13 y se suele representar con la secuencia de escape \r.

- «`--help`»: Esta opción muestra por pantalla un mensaje con un brevísimo resumen sobre el manejo de SED. Ese mismo mensaje se muestra también si se intenta ejecutar SED sin ninguna opción ni parámetro.

2.4. Resumen de los procedimientos para especificar el guión

De lo visto hasta ahora resulta que el guión, o conjunto de comandos que SED ejecutará, lo podemos indicar:

- Como cadena de texto.
- Con el nombre de un fichero que contenga el guión.

Y cada una de estas dos posibilidades, por su parte, se puede incluir mediante dos opciones de SED, o sin usar ninguna opción, resulta que el guión lo podemos indicar de seis maneras distintas:

1. Como cadena de texto con la opción «`-e`»
2. Como cadena de texto con la opción «`--expression`»
3. Como cadena de texto entrecomillada, incluida en la línea de comandos antes del nombre de los ficheros a procesar, sin necesidad de usar ninguna opción.
4. En un fichero invocado mediante la opción «`-f`».
5. En un fichero invocado mediante la opción «`--file`».
6. En un fichero mencionado antes de los ficheros a procesar, sin necesidad de usar ninguna opción.

2.4.1. Indicar el guión como cadena de texto

Cuando se indica el guión como cadena de texto, se puede hacer saber a SED expresamente que lo que va a continuación es un comando a ejecutar, usando las opciones «`-e`» o «`--expression`». Pero no es imprescindible. Si ninguna opción ha indicado qué guión ejecutar, SED interpretará el primer parámetro que no sea una opción, como guión.

No es imprescindible que el comando a ejecutar se encierre entre comillas, pero es aconsejable, para evitar malas interpretaciones de la shell. Las comillas pueden ser simples o dobles. En la mayor parte de los casos da igual usar unas u otras, siempre que se sea consistente; no podemos, por tanto, especificar un texto que se inicie con una comilla simple y termine con comillas dobles. Si nuestro comando incluye alguno de los dos tipos de comillas, habrá que usar, para encerrar el comando, el otro tipo. Por ejemplo: si queremos cambiar en nuestro fichero las comillas dobles por el carácter « o » según convenga, podríamos escribir algo así como:

```
sed -e 's/\([\n ¡¿({]\)/\1«/g;s/"\([ ?)!}\n,;:\.]\)/»\1/g'
```

Lo anterior incluye dos expresiones regulares que aún no estamos en condiciones de entender, por lo tanto, de momento no lo explicaré, simplemente hay ahora que observar que como en nuestra expresión regular se usaba el carácter de las comillas dobles, la hemos encerrado entre comillas simples.

Por lo demás, en el ejemplo se incluían dos comandos (en realidad un comando –«`s`»- – ejecutado dos veces). Para indicar más de un comando a ejecutar, hay varias maneras, las principales son:

1ª Separar los comandos mediante un punto y coma. Así lo hemos hecho en el ejemplo anterior.

2ª Indicar cada comando mediante una opción «`-e`» distinta.

3ª Usar la capacidad multilínea de la las *shell* tipo *bourne*. Por ejemplo: si trabajamos en *bash* podemos, tras la comilla simple inicial del comando a ejecutar, pulsar ENTER. Bash entra entonces en la modalidad multilínea hasta que cerremos dichas comillas. Así el ejemplo anterior lo podríamos haber escrito de la siguiente manera:

```
$ sed -e '
> s/\([ ¡¿({\n]\)"/\1«/g
> s/"\([?)!} ,:\.]\)/»\1/g' prueba.txt
```

En fin: cuando hay que ejecutar más de un comando, debe tenerse en cuenta lo que en el próximo epígrafe se explica sobre el orden de ejecución de los scripts, pues, en caso contrario, se obtendrán resultados inesperados.

2.4.2. Usar el contenido de un fichero como guión: Los scripts de SED

Si el conjunto de tareas a realizar sobre el texto es complejo, o pensamos aplicarlo varias veces, en lugar de teclearlo en la línea de comandos, es preferible grabarlo en un fichero y luego indicarle a SED el nombre de dicho fichero para que extraiga de él los comandos a ejecutar.

El nombre del fichero lo podemos indicar explícitamente mediante las opciones «`-f`» o «`--file`» seguidas del nombre del fichero. Pero también podemos introducir directamente el nombre del fichero: Si SED, tras leer todas las opciones, no ha encontrado ningún comando a ejecutar, analizará el primer parámetro recibido que no sea una opción, asumiendo que en él se encuentra, bien un texto con los comandos a ejecutar, bien el nombre del fichero que los contiene[4].

[4]Lo que digo en el texto recoge al pie de la letra lo que dice la página info de SED. Mi experiencia, sin embargo, es que un mismo script funciona si es invocado usando la opción «`-f`», pero a veces genera un mensaje de error si es invocado "a pelo"; y no he conseguido determinar el por qué del fallo. Es posible que ello se deba al funcionamiento de BASH más que al funcionamiento de SED.

En cuanto al nombre del fichero, si en él hay espacios en blanco, debe entrecomillarse, en otro caso las comillas son opcionales.

En sentido estricto un *script* de SED es el conjunto de comandos a ejecutar. Pero, en la práctica, muchas veces se reserva el término exclusivamente para el caso de que esos comandos hayan sido previamente almacenados en un fichero.

Al procesar el script SED ignora:

— Las líneas en blanco.

— Los espacios en blanco y tabuladores al principio de cada línea.

— Todo lo que haya en una línea a partir del carácter "#", el cual se usa, por lo tanto, para insertar comentarios dentro del script que ayuden a entenderlo, o para inhabilitar temporalmente la ejecución de algún comando[5].

  Respecto de los comentarios hay que tener en cuenta que si los dos primeros caracteres de un script son "#n" se deshabilita la opción *autoprint* de SED. Sería como si se hubiera llamado a SED con la opción «`-n`». Por lo tanto si en la primera línea del script queremos escribir un comentario que empiece con la letra n, debemos asegurarnos de usar una «`N`» mayúscula, o de que entre la marca del comentario y su inicio hay al menos un espacio en blanco.

El resto del script se considera que son comandos de SED a ejecutar. Estos se pueden separar por el signo ";" o por saltos de línea, aunque es costumbre usar saltos, pues éstos, acompañados de los correspondientes sangrados de línea, aumentan la legibilidad del script y hacen más sencillo seguir su lógica.

Para comprender bien como funciona un script de SED hay tres principios básicos que conviene recordar:

1. Los comandos del script se ejecutan en el orden en que se encuentran.

2. Cada comando se aplica a las líneas de texto que se hayan especificado en su ámbito. Si para un comando no se ha especificado ámbito, como regla general se aplicará a todas las líneas.

3. El fichero original no se ve modificado por la ejecución del script, a no ser que ejecutemos SED con la opción -i.

Las dos primeras reglas afectan a la manera en que el script se ejecuta, y hay que tenerlas presentes para poder imaginar cómo afectan unos comandos del script a otros. Los usuarios novatos tienden a pensar que el script es como un programa clásico: SED va

[5] En algunas implementaciones de SED el carácter "#" sólo es considerado como marca de comentario si es el primer carácter de la línea; y en otras, aún mas antiguas, sólo podía haber un comentario en el script, que tenía además que estar en su primera línea. Pero en la mayor parte de las implementaciones, el carácter "#" funciona como se acaba de describir: si se encuentra en medio de una línea, se procesará lo que haya a su izquierda y se dejará sin procesar lo que haya a su derecha.

leyendo el script línea a línea, y aplicándolo al texto. Pero en realidad es al revés: SED va leyendo el texto línea a línea, y a cada línea le aplica la totalidad del script.

El orden de los comandos es muy importante, porque cada comando recibe el texto de la línea, tal y como lo dejó el comando anterior. Así, por ejemplo, si queremos, en un documento que habla de animales, cambiar las referencias a las vacas por referencias a los toros, y las referencias a los toros por referencias a los bueyes podríamos pensar en dos scripts distintos:

```
s/vaca/toro/g ; s/toro/buey/g
s/toro/buey/g ; s/vaca/toro/g
```

El primero no funcionaría correctamente y el segundo sí lo haría. En el primer script la primera orden convierte todas las vacas en toros, y luego, la segunda orden convertiría todos los toros (incluyendo los que inicialmente eran vacas) en bueyes, por lo que al final sólo habría bueyes.

Es pues muy importante valorar el orden de los comandos de un script.

A propósito de los scripts para SED conviene distinguir las siguientes posibilidades:

1. Scripts que se limitan a contener los diferentes comandos que SED ejecutará. Estos scripts deben ser incluidos en la línea de comandos de SED mediante las opciones «`-f`» o «`--file`» y en el fondo equivalen al resultado que se obtendría mediante múltiples opciones «`-e`» en la llamada a SED.

   Es costumbre en estos scripts que la primera línea sea

   ```
   #!/usr/bin/sed -f
   ```

   para informar de que se trata de un script para ser usado con SED mediante la opción «`-f`».

   Estos scripts no pueden recibir argumentos; y aunque en un sistema como UNIX, cualquier fichero puede convertirse en ejecutable, si lo hiciéramos, SED no sabría a qué ficheros aplicarse, por lo que habría que aprovechar el redireccionamiento de UNIX para poder ejecutarlo, y así, por ejemplo, si el script ejecutable se llama “MiScript.sed” y queremos ejecutarlo en el fichero “MiFich.txt” deberíamos invocarlo de la siguiente manera:

   ```
   ./MiScript.sed >MiFich.txt
   ```

2. Scripts de la shell que utilizan SED. En estos casos, además de usar SED podremos usar cualquier otro comando, podremos llamar al script con argumentos y, sobre todo, podremos realizar, en el mismo script, más de una llamada a SED.

   Esto último es muy útil pues, dado que un uso eficiente de SED exige que el documento esté escrito con una estructura consistente, podremos hacer una primera llamada a SED para que transforme el documento, regularizando en él ciertas pautas, y una segunda pasada, para realizar en él ciertas transformaciones.

3. Expresiones regulares

SED trabaja con expresiones regulares, y, antes de seguir avanzando, hay ya que introducirse en este terreno.

3.1. Concepto

Todos los editores de texto permiten buscar texto y, en su caso, reemplazarlo por otro. Pero sólo los editores verdaderamente avanzados incluyen la posibilidad de buscar texto utilizando expresiones regulares. En estos casos la búsqueda puede afinarse hasta límites insospechados.

Un concepto simple de expresión regular diría que estas consisten en una especie de *patrón de búsqueda* que permite no sólo localizar un texto exacto (aunque eso también puede hacerse) sino, sobre todo, localizar un texto que tenga ciertas características que lo ajusten a ciertas pautas o patrones. Por ejemplo: podríamos intentar localizar, mediante una expresión regular, todas las palabras de un texto que empiecen por la letra "c", tengan cuatro letras y acaben en vocal. Ese patrón se indicaría con la siguiente cadena de texto:

```
c..[aeiouáéíóú]
```

Un buen ejemplo de uso de expresiones regulares serían los llamados caracteres comodín de MS-DOS, o de UNIX. Los que hemos usado alguna vez MS-DOS (o, para personas más jóvenes, lo que hoy día en Windows se llama "ventana de DOS") sabemos que la orden «`dir`» devuelve un listado con los ficheros que hay en un directorio (equivale a la orden «`ls`» de bash), y que si queremos filtrar el resultado podemos usar algún carácter comodín. Por ejemplo «`dir *.doc`» (o «`ls *.doc`») nos devolvería un listado con los ficheros cuya extensión fuera ".doc". Esto significa que la frase «`*.doc`» funciona como un patrón de búsqueda que se podría interpretar más o menos como "cualquier secuencia de caracteres que termine con «.doc»". Pues bien: eso sería una expresión regular.

Pero las expresiones regulares son mucho más que eso y, una vez que las conocemos, no comprendemos como durante algún tiempo pudimos trabajar sin usarlas. Yo ahora recuerdo los tiempos en los que manejaba MS-WORD (que no conoce las expresiones regulares) y me escandalizo de la cantidad de tiempo que perdí realizando numerosas veces tareas repetitivas que habría podido resolver mediante una simple expresión regular.

Ahora bien: la potencia de las expresiones regulares tiene un precio: su complejidad. El lenguaje de expresiones regulares es tan compacto que para el que no lo entienda resulta críptico e ininteligible. Y además, para seguir complicando la cosa, existen diferencias entre el lenguaje de expresiones regulares que utilizan las distintas aplicaciones que hacen uso de las mismas. Las diferencias no son muy importantes, pero sí las suficientes como para nunca estar completamente seguro de que una expresión regular escrita para una aplicación funcionará correctamente en otra.

En este documento explicaré el estándar GNU usado en ED, GREP, EGREP y SED.

La complejidad de las expresiones regulares, por otra parte, se debe a que en ellas es corriente que aparezcan numerosos caracteres precedidos de la barra invertida «\» que obliga a tener que leerlas con mucho detenimiento, y, sobre todo, a que como la tecnología de las expresiones regulares usa la técnica conocida como *backtracking*, o vuelta atrás[6], en muchas ocasiones (pero no siempre) entender la lógica de una expresión regular exige leerlas de derecha a izquierda.

3.2. Operadores en las expresiones regulares

Una expresión regular es una cadena de texto y, como tal, se compone de caracteres. Estos caracteres pueden ser *normales* o *especiales*. Los primeros se interpretan tal cual, y a estos efectos es importante tener en cuenta que SED distingue entre mayúsculas y minúsculas. Los caracteres especiales, por su parte, se asume que, a diferencia de los normales, significan algo distinto de lo que literalmente dicen. Es decir: los caracteres especiales deben ser "interpretados".

En las líneas que siguen para referirme a los caracteres especiales usaré el nombre de *operadores*. Porque eso es lo que en su mayor parte son: caracteres que indican qué hay que hacer con los caracteres que están a su derecha o a su izquierda.

Los operadores usados en las expresiones regulares son, en realidad, muy pocos:

```
$ ^ . * [ ] \
```

Pero como el efecto del último de ellos es que el carácter que le sigue debe interpretarse de cierta manera, en realidad podemos considerar operadores también a todos los caracteres que, precedidos del símbolo «\» funcionen como caracteres especiales.

Estos últimos son los siguientes:

```
\+ \? \{ \} \( \) \| \n \t \c \d \o \x \w \W \b \B \` \'
```

Además el carácter "\" seguido de cualquiera de los otros operadores hace que este se interprete literalmente, y no como un operador.

Para explicar el uso de estos operadores, creo que lo mejor es agruparlos por tareas o utilidad.

3.2.1. Operadores de reemplazo

Este grupo de operadores funciona como una especie de comodín que puede asumir distintos valores o, lo que es lo mismo, que puede ser reemplazado por distintos caracteres. Cuando están presentes provocan que la expresión regular pueda coincidir con varias cadenas de texto diferentes.

Los operadores básicos de reemplazo son el punto y los corchetes:

[6] Esta es la técnica que utilizan algunos lenguajes de programación, como PROLOG, de los que se dice que andan cerca de la inteligencia artificial; y el uso de que las expresiones regulares usen la misma técnica nos da una idea de su potencia, pero también de su complejidad.

El operador punto (.) Es el más general de los operadores de reemplazo. En una expresión regular un punto coincidirá con cualquier carácter incluido el de nueva línea. Así, por ejemplo, «`a.b`» devolverá cualquier secuencia de tres caracteres que empiece por "a" y termine por "c".

Los corchetes Mediante los corchetes podemos indicar un conjunto de caracteres alternativos, considerándose que hay una coincidencia cuando se encuentre cualquiera de ellos. Así, por ejemplo, «`ca[srm]a`» coincide con "casa", "cara" y con "cama", pero no con "cana" ni con "capa".

Algunos caracteres, al introducirse entre corchetes asumen un valor especial, o deben colocarse en algún punto concreto para poder ser interpretados correctamente:

] El carácter de cierre de corchetes, puede incluirse dentro de unos corchetes, con la condición de que sea el primer carácter de los corchetes, pues en otro caso SED lo interpretaría no como un carácter *en* los corchetes, sino como el cierre de los mismos. Así, por ejemplo, «`b[]a`» es una secuencia correcta que encontrará coincidencias con "b]" y con "ba".

- El guión sólo se interpreta como un guión si es el primero o el último de los caracteres en los corchetes, en cualquier otro caso, si el guión se encuentra entre otros dos caracteres, se asume que está funcionando como un *operador de rango* y significa que hay que considerar incluidos en el corchete todos los caracteres que se encuentren entre el carácter a la izquierda del guión (rango inferior) y el carácter a la derecha del mismo (rango superior).

Así, por ejemplo, la expresión «`[1-8]`» significa que buscamos cualquier carácter en el rango del 1 al 8, o sea: 1, 2, 3, 4, 5, 6, 7 u 8, mientras que «`[a-z]`» significa cualquier carácter entre la a y la z, o, lo que es lo mismo, cualquier letra minúscula.

> Al trabajar con rangos de caracteres hay que tener en cuenta que éste depende de la concreta codificación que se use. En la codificación ASCII por ejemplo, las vocales acentuadas, o la ñ, al no existir en el alfabeto anglosajón, no están en el rango "a-z", mientras que si se trabaja con la codificación UTF8, ocurrirá lo contrario. En consecuencia el mismo script de SED arrojará resultados distintos en dos sistemas que utilicen una codificación diferente. Los ficheros de texto, por otra parte, tampoco informan de la codificación que en ellos se usa, y aunque hay editores que son bastante buenos a la hora de detectar la codificación, no sé cómo lo hacen, ni he encontrado información sobre en qué medida SED maneja bien o mal esta detección. Como, por otra parte, no hay en SED ninguna opción que permita indicar la codificación del fichero, supongo (aunque es sólo una suposición) que SED usará la codificación por defecto del sistema.
> Por todo ello, en este punto no me atrevo a generalizar. Sólo diré que en mi sistema, que usa por defecto UTF8, y con ficheros que usan en su totalidad la misma codificación SED (versión 4.2.1) funciona correctamente a la hora de determinar los rangos de las letras aunque se usen caracteres españoles que no existen en inglés.

^ El acento circunflejo, cuando ocupa la primera posición en los corchetes se interpreta como *operador de negación* y significa que hay que buscar cualquier carácter *que no esté en los corchetes*. Y así, por lo tanto, «`[a-z]`» buscará cualquier letra minúscula, y «`[^a-z]`» buscará cualquier carácter que no sea una letra minúscula.

Cuando ^ actúa como operador de negación no se considera, propiamente, un carácter, y por lo tanto el siguiente carácter de los corchetes será considerado *el primer carácter*.

Los demás caracteres, cuando están entre corchetes, se consideran siempre caracteres normales, aunque, fuera de los corchetes, tengan la consideración de caracteres especiales. Por lo tanto se pueden emplear en los corchetes sin necesidad de precederlos de una barra invertida.

Además del punto y los corchetes, determinadas secuencias de escape se interpretan en una expresión regular como operadores de reemplazo. Se trata de:

\w Encuentra caracteres que puedan formar parte de una palabra: letras, números y signos de subrayado (_).

\W Encuentra caracteres que no pueden formar parte de una palabra, por no ser letras, números o signos de subrayado.

Por último, los operadores «`\b`» y «`\B`», que se explican entre los operadores posicionales (sección 3.2.3), también son operadores de reemplazo, aunque su uso principal es el posicional.

3.2.2. Operadores de repetición

Este grupo de operadores indica que el carácter o la expresión que se encuentra a su izquierda debe o puede repetirse cierto número de veces. Así:

***** Significa que el carácter anterior puede repetirse cero o más veces. Así, por ejemplo, la expresión «`mo*`» coincidirá con “m”, “mo”, “moo”, “mooo”, etc.

\+ Similar a *, salvo que el carácter anterior debe aparecer al menos una vez. Así, por ejemplo, «`mo\+`» coincidirá con “mo”, “moo”, “mooo”... pero no con “m”.

\? Indica que el carácter anterior puede encontrarse cero veces o una vez, pero no más de una. Así, por ejemplo «`flo\?r`» coincidirá con “flr” y con “flor”, pero no con “floor”.

\{N\} Indica que el carácter anterior debe repetirse exactamente N veces, siendo N un número entero. La ayuda oficial de SED recomienda que, a efectos de compatibilidad, N no sea un número superior a 255. Por ejemplo «`x\{3\}`» coincidirá exclusivamente con “xxx”, pero no con “xx” ni con “xxxx”.

\{N,M\} Indica que el carácter anterior debe coincidir entre N y M veces: no menos de N ni más de M veces. Tanto N como M deben ser números enteros.

\{N,\} Indica que el carácter anterior debe repetirse al menos N veces, siendo N un número entero.

Estos operadores se aplican al carácter anterior. Pero si el carácter anterior fuera, a su vez, un operador, hay que distinguir:

- Si el carácter anterior era el cierre de un grupo (véase la sección 3.3), se asume que es el grupo entero el que debe repetirse para considerar que hay una coincidencia.
- Si el carácter anterior era un operador, pero no un operador de repetición, lo que se repetirá es el resultado de dicho operador. Por ejemplo: «`[aeiou]\+`» buscará dos o más vocales seguidas.
- Si el carácter anterior era también un operador de repetición... Hay que hacer un gran esfuerzo mental para predecir el resultado de nuestra búsqueda, aunque como regla debe asumirse que los operadores de repetición (salvo «`\?`» tienen lo que podríamos llamar *naturaleza ávida* o, si se prefiere, *carácter voraz*, y por lo tanto siempre devolverán la coincidencia más larga posible.

3.2.3. Operadores que indican posición del texto buscado

Este grupo de operadores permite indicar en qué lugar debe encontrarse el texto al que se aplican. Son los siguientes:

^ En general este carácter se usa, fuera de los corchetes, para indicar que el carácter buscado debe estar al principio de una línea. Pero como en SED las líneas, para ser procesadas, se copian al área de trabajo, aquí se dice que ^ identifica caracteres que estén al principio del área de trabajo.

^, por otra parte, actúa sólo como operador cuando es colocado al principio de la expresión regular (o de una subexpresión que constituya un grupo). Fuera de ese lugar podemos usar tranquilamente el carácter ^ sin necesidad de precederlo de una barra invertida.

$ Indica que el carácter inmediatamente anterior es el último de la línea (o del área de trabajo). Y al igual que ocurre con ^, $ sólo funciona como carácter especial cuando se encuentra al final de una expresión regular. En cualquier otro lugar podemos usarlo sin necesidad de precederlo de una barra.

\b Identifica caracteres que estén al principio o al final de una palabra, por tanto la coincidencia se producirá si el carácter a la izquierda coincide con \w y el carácter a la derecha con \W, o al revés.

\B Identifica caracteres que no estén en el extremo de una palabra, o, lo que es lo mismo: la coincidencia se producirá cuando, simultáneamente, los caracteres a izquierda y derecha coincidan con \w, o cuando ambos coincidan con \W.

3.3. Creación de grupos en una expresión regular

La pareja de operadores \(y \) funcionan como operadores de agrupación: todo lo que haya entre ellos se interpreta como un grupo.

En una expresión regular los grupos se pueden emplear para dos finalidades distintas:

1. Para conseguir que varios caracteres sean tratados como uno solo a los efectos de aplicarles algún operador de repetición. Así, por ejemplo, «`Ba\(na\)\+`» localizará “Bana”, “Banana”, “Bananana”, etc.

2. Identificar la secuencia encerrada dentro del grupo de tal modo que podamos referirnos a ella, más adelante.

La primera utilidad es muy sencilla y no requiere mayor explicación. La segunda por el contrario, ofrece más complejidad pero, al mismo tiempo, constituye uno de los aspectos que dotan a las expresiones regulares de más potencia.

Cuando SED analiza una expresión regular, si en ella encuentra un grupo, le asigna un número. El primer grupo encontrado será el número 1, el segundo el nº 2... así hasta el número 9. Estos números sirven para identificar el grupo de tal manera que podamos referirnos a él más adelante, bien en la misma expresión regular, bien en otra relacionada con ella.

Para referirnos a un grupo concreto se usa el operador \N, donde N representa el número asignado al grupo. Por tanto \1 se refiere al primer grupo, \2 se refiere al segundo grupo..., y así hasta \9. No se pueden usar más de 9 grupos con esta finalidad. Incluir en una expresión regular \N se interpreta como “insertar aquí el mismo texto que devolvió el grupo nº N”.

Los grupos usados en este sentido son fundamentales para las operaciones de reemplazo basadas en expresiones regulares. Véase, más adelante, el epígrafe 3.3. Un ejemplo de este uso de los grupos se contiene en la sección 5.6.

3.4. Combinar expresiones regulares

Podemos combinar dos o más expresiones regulares creando una expresión nueva, la cual puede buscar, bien cadenas de texto que coincidan simultáneamente con las dos expresiones constituyentes, bien cadenas de texto que coincidan con cualquiera de ellas.

Para la primera posibilidad no hay ninguna sintaxis especial. Basta con escribir una expresión regular detrás de la otra. Pero para la segunda posibilidad se usa el operador \| cuyo resultado es el del llamado “o lógico”: la expresión resultante será verdad cuando cualquiera de las expresiones que la componen sea verdad.

Así, por ejemplo, la expresión «`UNIX\|LINUX`» localizará, indistintamente, todas las menciones a UNIX o a LINUX.

3.5. Uso de secuencias de escape para referirse a caracteres no imprimibles

Ya hemos visto que, salvo los caracteres especiales, en una expresión regular cualquier carácter se interpreta literalmente. Ahora bien: hay algunos caracteres no imprimibles, como el tabulador, o el salto de línea, que, al ser invisibles, es difícil incluirlos en una expresión regular.

Para referirse a estos caracteres SED puede usar secuencias de escape.

> La noción de *secuencia de escape* apareció cuando se implementó el Código ASCII. Los 33 primeros caracteres de este código no tienen representación gráfica y, por lo tanto, para referirse a ellos se adoptó la convención de hacerlo mediante una barra invertida seguida de un carácter que simbolizara a dicho código. A partir de ahí se llama "secuencia de escape" a una combinación de caracteres que empieza con la barra invertida y que simboliza un carácter no directamente imprimible.

SED acepta las secuencias de escape *ordinarias*. Entre ellas las más importantes, en la práctica, son las siguientes:

\n Localiza (o produce[7]) un salto de línea (ASCII 10).

\r Localiza (o produce) un salto de carro (ASCII 13)[8].

\t Localiza (o produce) un salto de tabulador.

\cX Localiza (o produce) un Control-X, donde X es cualquier carácter.

\dXXX Localiza (o produce) el carácter cuyo valor decimal sea XXX.

\oXXX Localiza (o produce) el carácter cuyo valor octal sea XXX.

\xXXX Localiza (o produce) el carácter cuyo valor hexadecimal sea XXX.

\` Coincide con el principio del área de trabajo.

\' Coincide con el final del área de trabajo.

3.6. Expresiones regulares extendidas

El efecto de llamar a SED con las opciones «`-e`» o «`--regexp-extended`» es el de que en lugar de usar las expresiones regulares tipo GREP, se usa el conjunto extendido de expresiones reguladas de EGREP. La diferencia entre ambos conjuntos se encuentra exclusivamente en los caracteres «`?`», «`+`», «`{`», «`}`», «`(`» y «`)`»:

- En GREP (y en SED por defecto) estos caracteres se consideran "normales", salvo si van precedidos de la barra invertida, en cuyo caso funcionan como operadores.
- En EGREP estos caracteres se consideran operadores, salvo si van precedidos de la barra invertida, en cuyo caso son tratados como caracteres ordinarios.

[7] Cuando digo que estos caracteres localizan o producen algo, me refiero a que, si se usan como texto de búsqueda de un comando «`s`» o para determinar el ámbito de aplicación de un comando, entonces lo que hacen es localizar el carácter en cuestión. Pero cuando se usan como texto de reemplazo en un comando «`s`», lo que hacen es producir dicho carácter.

[8] En MS-DOS y en Windows, como regla el salto de línea en un fichero de texto se hace con dos caracteres: «`\n`» seguido de «`\r`». Creo, no obstante (aunque no lo he comprobado), que SED comprueba en que sistema está, y si está en uno de esos sistemas, directamente interpreta «`\n`» como la combinación de «`\n`» y «`\r`», para evitar que los scripts pensados para un sistema provoquen resultados inesperados en otro sistema.

De donde resulta que el nombre *conjunto extendido de expresiones regulares* es, cuando menos, equívoco; porque no se trata de que en EGREP puedan escribirse expresiones regulares que no puedan representarse en GREP, sino de que las mismas expresiones regulares se escriben de forma distinta: en EGREP son muchos menos los operadores que requieren el uso previo de una «\», lo que, en teoría, hace que sus expresiones regulares sean más sencillas de leer y de entender. Pero eso es, claro está, si el texto a buscar no incluye demasiados paréntesis, llaves, interrogaciones... porque en tal caso todas las apariciones de estos caracteres que no fueran a funcionar como operadores, deberían ir precedidas de la correspondiente barra invertida.

Personalmente tiendo a usar el conjunto extendido, porque yo aprendí a manejar las expresiones regulares en EMACS, y el conjunto de operadores usado por ese programa se parece más al de EGREP que al de GREP[9]. No obstante en este documento haré uso siempre de la sintaxis normal (la de GREP).

3.7. Consejos para trabajar con expresiones regulares

Aquí acaba la explicación de las expresiones regulares. Como se verá su teoría no es tan difícil. Sin embargo estoy seguro de que quien no conociera de antes el uso de estas expresiones, aunque ya haya aprendido todo lo que acabo de explicar, sigue sin tener ni idea de todo lo que se puede conseguir con ellas. Y es que el diseño de expresiones regulares es un arte cuyo único límite es la imaginación del artista.

Pero, como en todo arte, la experiencia hace mucho. Por lo tanto, para trabajar con expresiones regulares mi principal recomendación es que se empiecen a usar; para ello, claro está, lo mejor es acostumbrarse a manejar un editor de texto que las maneje: VIM o EMACS son, en mi opinión, las mejores opciones. Empezar a manejar expresiones regulares nos irá acostumbrando, poco a poco, a aprender a *pensar en términos de patrones o pautas*; a analizar los textos fijándonos en a qué pauta responden, porque son esas pautas las que nos permitirán diseñar las expresiones regulares que las capturen.

Aunque, sin duda, lo mejor es aprender de los maestros. En internet hay numerosos sitios donde se recogen scripts construidos con expresiones regulares. En la última sección de este documento se menciona alguno: lo mejor es visitarlo y estudiar las expresiones regulares que allí, o en otros lugares, se vayan encontrando.

Para estudiar expresiones regulares hay que tener en cuenta que los operadores de repetición funcionan como "*absorbedores*": absorben todos los caracteres que se ajustan al patrón dado. Por lo tanto, muchas veces, entender la expresión regular exige buscar en ella algún elemento que con seguridad deba aparecer y, a partir de él, leer el resto de la expresión regular.

Conforme nos vayamos acostumbrando a trabajar con expresiones regulares, iremos teniendo la tendencia a construir nuestros propios ficheros de tal manera que sea más

[9] Aunque tampoco es idéntico. Al respecto recuérdese lo que se dijo al principio de esta sección: uno de los inconvenientes de las expresiones regulares es que no están lo suficientemente estandarizadas, y cada programa que hace uso de ellas, trabaja de forma ligeramente diferente.

sencilla la modificación masiva de los mismos mediante expresiones regulares, con lo que cada vez las iremos usando más e iremos perfeccionando nuestra técnica.

4. Aspectos generales de los comandos de SED

4.1. Descripción general de la sintaxis de los comandos

Para indicarle a SED cómo debe manipularse el texto de entrada se usan comandos. La sintaxis general de un comando de SED es la siguiente:

```
[Ámbito] Comando[Argumentos]
```

Donde

- **Ámbito:** indica la línea o líneas a que se aplicará el comando. Si no se indica ámbito, como regla el comando se aplicará a todas las líneas del texto de entrada.
- **Comando:** El nombre del comando. Consta siempre de un sólo carácter que suele ser la inicial de la palabra en inglés que describe lo que dicho comando hace.

  Entre el ámbito y el comando puede insertarse, si se desea, un espacio en blanco.
- **Argumentos:** Si el comando indicado requiere o admite argumentos, hay que incluirlos inmediatamente después. Algunos comandos admiten un espacio en blanco entre el nombre del comando y sus argumentos, y los hay incluso que exigen dicho espacio; pero otros requieren que los argumentos se escriban inmediatamente detrás del comando, sin separación ninguna.

4.2. El ámbito de los comandos

El ámbito de los comandos informa a SED de las líneas que se verán afectadas por el comando. En este sentido podemos especificar, para cada comando:

- Ningún ámbito. En tal caso la regla es que el comando se ejecutará en todas las líneas, si bien eso depende de lo que haga el comando, pues en algunos no tiene sentido ejecutarlos más de una vez. Por ejemplo: el comando «`q`» termina la ejecución del script, y una vez que éste ha terminado... ¡ya no puede volverse a ejecutar ningún comando de él!
- Algún ámbito: el comando se ejecutará en todas las líneas que coincidan con el ámbito. Dependiendo del modo en que se exprese el ámbito se distingue entre:
  - *Ámbitos simples*: se indican mediante una sola expresión.
  - *Ámbitos dobles*: se indican mediante dos expresiones separadas entre sí por una coma. El ámbito así expresado se interpreta como “todas las líneas comprendidas entre la indicada en primer lugar y la indicada en segundo lugar”.

¡Ojo! No cometamos el error de pensar que el ámbito simple sólo se refiere a una línea y el ámbito doble se refiere a un grupo de líneas. La segunda parte es cierta, pero la primera no, ya que si, por ejemplo, como selector de ámbito se usa una expresión regular, el ámbito será simple, pero el comando se aplicará a todas las líneas que coincidan con dicha expresión regular. Por lo tanto la diferencia tiene que ver con *cómo se indica el ámbito*: si mediante una sola expresión, o mediante dos expresiones separadas por una coma, y así, por ejemplo:

```
14-3 d
14,+3 d
```

en el primer caso el ámbito es simple y en el segundo caso el ámbito es doble.

Insisto en donde está la diferencia, porque hay comandos que sólo admiten ámbitos simples.

Los ámbitos, por otra parte, se pueden indicar numéricamente, mediante expresiones regulares, o por una combinación de ambos procedimientos:

4.2.1. Indicación numérica de los ámbitos

Mediante la especificación numérica de los ámbitos podemos indicar que determinado comando:

- Se aplique exclusivamente a una línea concreta. Para ello basta con escribir el número de la línea antes del comando. Por ejemplo «`12d`» aplicará el comando «`d`» a la línea 12.
- Se aplique desde cierta línea hasta otra. Para ello hay que indicar, antes del comando, la línea inicial y la línea final. Por ejemplo «`1,12d`» aplicará el comando «`d`» a las líneas 1 a 12. Aunque también se admiten las siguientes sintaxis para esta modalidad:

  N,+M donde N es la línea inicial y M es la cantidad de líneas adicionales a contar desde la inicial a las que debe aplicarse el comando.

  N,~M donde N es la línea inicial, a partir de la cual se aplicará el comando hasta encontrar una línea cuyo número sea múltiplo de M.

- Se aplique desde cierta línea hasta el final. En este caso, si no sabemos cuántas líneas tiene el fichero, podemos representar simbólicamente la última línea del fichero con el carácter "$", y así por ejemplo «`12,$d`» borrará desde la línea 12 hasta el final del fichero. Este uso del carácter $, no debe confundirse con el uso que del mismo carácter se hace en las expresiones regulares.
- Se aplique a una línea concreta y, a partir de ella, cada cierto número de líneas. Para ello se usa la sintaxis «`N~M`» donde N representa a la línea inicial, y M representa el número de líneas que hay que saltar en cada repetición del comando. Así, por

ejemplo «1~2d» borrará la primera línea y, a partir de ella, cada dos líneas. El resultado será, por tanto, que se borrarán todas las líneas impares del texto.

En orden a la numeración de las líneas hay que tener en cuenta que, cuando se están manejando simultáneamente varios ficheros, SED los yuxtapone uno tras otro, de tal manera que, por ejemplo, la primera línea del segundo fichero recibirá como número el siguiente al que recibió la última línea del primer fichero. Así habrá sólo una línea nº 1 y una línea nº $ con independencia del número de ficheros que se esté manejando.

Este funcionamiento por defecto puede ser inhibido mediante el uso de las opciones «-s», «--separate», «-i» o «--in-place» en la línea de comandos (explicadas en la sección 2.3.2).

4.2.2. Uso de expresiones regulares para indicar los ámbitos

Esta característica es responsable, en gran medida, de la potencia de SED: podemos usar una expresión regular para aplicar un comando a todas las líneas en donde dicha expresión regular se cumpla. Por ejemplo: si queremos sustituir la palabra "California" por "San Francisco", pero sólo en los líneas en donde aparezca también la palabra "Massachusett", escribiríamos lo siguiente:

```
/Massachusett/s/California/San Francisco/g
```

Conviene analizar despacio la anterior expresión: Empieza con una expresión regular delimitada entre dos barras. Esa expresión regular no va precedida de ningún comando, porque su finalidad es, simplemente, indicar el ámbito. Tras ella se introduce el comando que queremos ejecutar en todas las líneas en donde se encuentre la expresión regular.

Para indicar que un ámbito se introduce mediante una expresión regular, ésta hay que introducirla entre delimitadores. Por defecto SED espera que como delimitador se use el carácter «/». Para usar otro carácter, hay que precederlo de una barra invertida la primera vez que aparece. Así, en los siguientes ejemplos

```
\cREGEXPc
\-REGEXP-
\?REGEXP?
```

usaríamos como delimitadores, respectivamente, el carácter "c", el guión y el cierre de interrogación.

En general, como delimitador, debemos usar un carácter que no forme parte de la expresión regular que vamos a escribir, puesto que su aparición indica a SED que ahí termina la expresión regular. No obstante, si dentro de la expresión regular nos vemos obligados a usar el delimitador, pero queremos que sea interpretado literalmente, y no como delimitador, deberemos precederlo de una barra invertida.

La expresión regular usada para indicar el ámbito distinguirá, como regla, entre mayúsculas y minúsculas. Para evitarlo hay que añadir, tras el delimitador de cierre, el modificador «I», y así, por ejemplo:

```
/c.c./d
```

no afectará a las líneas donde aparezca “CACA” o “Coco”. Por el contrario

```
/c.c./Id
```

borrará todas las líneas en que aparezca la palabra “caca”, “cuca”, “coco”, etc., Pero también aquellas donde estén “CUCO” o “Caco”.

> Obsérvese que el uso de expresiones regulares para indicar ámbitos significa que el comando que sigue se ejecutará si y solo si se cumple en la línea la expresión regular indicada como ámbito, lo que, desde la perspectiva de un programa de ordenador, equivale a un comando tipo IF. Por eso decíamos al principio que, SED es, en realidad, un lenguaje de programación ya que admite la ejecución condicional de comandos, y los saltos entre distintas secciones del script (véase, sobre esta última posibilidad, la sección 5.4.2).

Al igual que ocurre con la indicación numérica de los ámbitos, también podemos usar expresiones regulares para referirnos a la primera y la última línea del rango al que aplicar el comando, separadas por comas. Por ejemplo:

```
/coco/,/fruta/d
```

buscará las líneas en que aparezca el texto “coco” y, cuando encuentre una, la borrará, y seguirá borrando hasta que encuentre una línea en donde aparezca el texto “fruta”, que también se borrará, pero será la última borrada hasta encontrar una nueva línea con el texto “coco”.

4.2.3. Ámbitos dobles indicados numéricamente y mediante expresiones regulares

También podemos combinar la indicación numérica de ámbitos con la indicación por expresiones regulares y así, por ejemplo, serían admisibles ámbitos como los siguientes:

```
4,/fruta/
/coco/,56
/c.c./,$
```

En el primer caso, el rango irá desde la línea 4 hasta la primera línea posterior a ella en que aparezca la palabra “fruta”, en el segundo caso el rango empieza en la primera línea donde aparezca el texto “coco” y terminará en la línea 56. En el tercer ejemplo se buscará el patrón “c.c.” y el rango irá hasta la última línea.

En estos casos hay que tener en cuenta una peculiaridad del modo de trabajo de SED cuando usa expresiones regulares para delimitar ámbitos, y es que, una vez encontrada la primera línea, no se empieza a buscar la expresión regular que cierra el rango hasta que se ha ejecutado el comando en la línea en cuestión y se ha cargado la siguiente línea. Eso significa que si, por ejemplo, hemos usado como ámbito el del ejemplo anterior:

```
/coco/,/fruta/
```

y en una línea del texto aparecen las dos palabras, “coco” y “fruta”, SED no aplicará el comando exclusivamente a dicha línea sino que cuando encuentre la primera aparición de “coco”, ejecutará el comando que sea y luego empezará a buscar la palabra “fruta” *a partir de la línea siguiente*.

Precisamente por esa característica de SED se admite que, como ámbito, se indique

```
0,/REGEXP/
```

0 como número de línea no tiene normalmente sentido, pues la primera línea del texto es siempre la número 1. Pero en este contexto sí lo tendría: significa que hay que buscar la expresión regular de cierre, desde la línea 1.

4.2.4. Uso del operador de negación (!) en la especificación de ámbitos

Normalmente la especificación de un ámbito significa que sólo hay que aplicar el comando a la línea o líneas incluidas en dicho ámbito. Pero SED también admite que el ámbito se interprete en el sentido de que el comando debe aplicarse a todas las líneas salvo las incluidas en el ámbito.

Para ello se usa el operador de negación « ! » inmediatamente después del ámbito, y así:

```
14d
14!d
/fruta/,$d
/fruta/,$!d
```

El primer ejemplo borra la línea 14. El segundo ejemplo borra todas las líneas salvo la 14. El tercer ejemplo borra desde la primera línea en que aparezca el texto “fruta” hasta el final. El último ejemplo hace lo contrario borra todas las líneas hasta encontrar una en que aparezca el texto “fruta”.

4.3. Agrupación de comandos

Dentro de un script podemos agrupar un bloque de comandos mediante los caracteres “{” y “}”.

En estos casos el ámbito se indica antes de la llave inicial, y todos los comandos encerrados entre las llaves se aplicarán al mismo ámbito, en el orden en el que aparezcan.

El formato es, por lo tanto, el siguiente:

```
Ámbito {
comando1
```

```
comando2
comando3
...
}
```

5. Los concretos comandos de SED

A continuación expondré los distintos comandos reconocidos por SED. Para ello los agruparé en varias categorías atendiendo a su finalidad principal. Los comandos que encajan en más de una categoría se incluyen en la que pienso que mejor los refleja. Téngase en cuenta, por otra parte, que esta explicación se basa en la versión GNU de SED y algunos de los comandos que se incluyen no existen en la versión POSIX.

5.1. Los comandos básicos de manipulación del texto

Llamo comando de manipulación del texto al que produce modificaciones en él cambiando, quitando o añadiendo algo. Son los siguientes

5.1.1. El comando «s»: la navaja suiza de SED

El comando «s» es el más usado de sed, y posiblemente el más versátil y potente. Como ya sabemos su función es buscar un texto y reemplazarlo con otro. Su potencia está en que a la hora de buscar un texto usa expresiones regulares.

El formato general de este comando es:

```
s/REGEXP/REEMPLAZO/Indicadores
```

Donde:

s Es el nombre del comando.

/ Es el carácter que se usa para delimitar la expresión regular a buscar y el texto de reemplazo. Por tradición se usa como delimitador el carácter «/», pero puede usarse cualquier otro carácter (salvo el de salto de línea). SED interpretará que el carácter escrito a continuación del nombre del comando «s» va a funcionar como delimitador. Por ello este comando *nunca* debe separarse de sus argumentos por un espacio en blanco a no ser que queramos que los espacios en blanco funcionen como delimitadores.

Si el texto a buscar, o el texto de reemplazo contienen el carácter usado como delimitador, hay que escribirlo precediéndolo del carácter "\".

REGEXP Es la expresión regular a buscar. Téngase en cuenta que esta expresión regular será buscada en el área de trabajo, o lo que es lo mismo: en la línea actual. Por lo tanto si el texto a buscar está a lo largo de dos líneas, SED no lo localizará. Por otra parte,

si se usó para localizar el ámbito una expresión regular y no se indica aquí ninguna expresión regular, se usará la expresión regular que determinó el ámbito.

REEMPLAZO Es el texto con el que hay que sustituir cada aparición de REGEXP. Sobre él véase más adelante, en esta misma sección.

INDICADORES Se trata de una serie de modificadores que afectan al modo de trabajo del comando. Puede ser cualquiera de los siguientes:

- **g** Realiza la sustitución en todas las apariciones de REGEXP. Si no se incluye este indicador sólo se realizará la primera sustitución.
- **N** Sustituye sólo la aparición número N de REGEXP.
- **p** Envía a la salida estándar el texto tras realizar la sustitución. Si no se hace ninguna sustitución no envía nada.
- **e** Si se realiza una sustitución el texto resultante en el área de trabajo se interpreta como un comando, es ejecutado, y el contenido del área de trabajo es sustituido por el resultado de dicha ejecución.
- **I, i** Hace que la búsqueda de la expresión regular no distinga entre mayúsculas y minúsculas.
- **M, m** En los casos en que el área de trabajo consta de más de una línea (véase 5.5), estos indicadores afectan a cómo funcionarán los operadores ^ y $ en la expresión regular. Por defecto ya sabemos que el primero identifica el principio del área de trabajo (que es también el principio de la primera línea) y el segundo identifica el final del área de trabajo (que es también el final de la última línea); pero usando el modificador «M» se indica que queremos que los anteriores operadores identifiquen, respectivamente, el principio y el final de alguna de las líneas alojadas en el área de trabajo. En tal caso para identificar el principio del área de trabajo se usaría el operador \‘, y para identificar el final del área de trabajo se usaría \’.
- **w Nombre-FICH** Envía el resultado de la sustitución, si esta tiene lugar, al fichero llamado Nombre-FICH. Se admiten como nombres de fichero “/dev/stderr”, que enviaría el resultado de la sustitución a la salida estándar de errores, y “/dev/stdout” que lo haría a la salida estándar (la pantalla).

Estos indicadores, por otra parte, pueden combinarse entre sí, pero si aparece el indicador «w», éste debe ser el último porque tras él necesariamente hay que incluir un espacio en blanco para, a continuación, indicar el nombre del fichero al que hay que escribir.

El texto de reemplazo no es una expresión regular, sin embargo es posible rescatar en él el texto concreto que la expresión regular localizó, o alguna de sus partes. Para ello se usan las siguientes convenciones:

- **&** Este carácter incluido en el texto de reemplazo devuelve siempre el concreto texto que la expresión regular encontró. Por lo tanto si queremos usar, como parte del

texto de reemplazo, el carácter "&", habrá que precederlo de una barra invertida, para conseguir que SED lo interprete en su sentido literal.

\N Si en la expresión regular se usaron grupos delimitados por los caracteres \(y \) (véase sección 3.3), \N reproduce en la cadena de reemplazo el resultado devuelto por el grupo nº N de la expresión regular. Es decir: \1 devuelve el resultado del primer grupo, \2 el del segundo grupo, etc.

Por lo demás, si usamos las anteriores posibilidades, dado que al escribir el texto de reemplazo es posible que no sepamos exactamente lo que éste dirá, las siguientes secuencias nos permiten controlar si el texto de reemplazo será en mayúsculas o en minúsculas. Para ello podemos usar los siguientes modificadores:

\L Activa el modo minúsculas: todo el texto de reemplazo irá en minúsculas hasta que se encuentre un \U o un \E.

\l Pone en minúsculas el próximo carácter.

\U Activa el modo de mayúsculas: todo el texto de reemplazo irá en mayúsculas hasta que se encuentre un \L o un \E.

\u Pone en mayúsculas el próximo carácter.

\E Desactiva los modos mayúsculas y minúsculas.

En fin: en el texto de reemplazo también podemos añadir, por supuesto, secuencias de escape, a las que hicimos referencia en la sección 3.5.

5.1.2. El comando «d»: Borrado de líneas

El comando que borra una línea, y que también conocemos desde el principio (véase la sección 1.5), es el comando «d».

«d» borra totalmente la línea a la que se aplique, y cuando no se le indica ámbito se aplica a todas las líneas, por lo tanto:

```
sed -e "d" MiFich.txt
```

no producirá ninguna salida, pues todas las líneas del fichero, conforme van siendo procesadas, van siendo borradas.

Tras ejecutar «d» ya no hay línea. No es que la línea esté vacía, es que ya no está, y, por lo tanto, no se puede hacer nada con ella. Esta es la razón de un importante efecto colateral del uso de este comando: tras su ejecución se inicia un nuevo ciclo. Los comandos que quedaban por ejecutarse en el ciclo actual son ignorados, y SED carga la próxima línea y vuelve al principio del script. Por ello, si queremos borrar una línea con la idea de, más tarde, volverla a llenar, el comando que hay que usar no es «d» sino «z» (véase la sección 5.7).

Cuando el ámbito se delimita mediante una expresión regular, los usuarios novatos de SED tienden a pensar que «d» borrará sólo la parte de la línea que coincidía con la expresión regular. Pero eso no es así: «d» siempre borra la línea por entero. Para borrar una parte de la línea hay que usar el comando «s» sin especificar texto de reemplazo, y para borrar el texto que se usó para delimitar el ámbito, hay que usar el comando «s» sin texto de reemplazo y sin texto de búsqueda, porque, como ya sabemos, cuando «s» no incluye texto de búsqueda, se usa el mismo que se usó para delimitar el ámbito. Así por ejemplo:

```
/palabra/ s///g
```

borrará todas las apariciones de la palabra "palabra", dejando intacta el resto de la línea.

5.1.3. Los comandos «a», «i» y «c»: Añadir, insertar y cambiar texto

Para añadir texto al área de trabajo, o para sustituir totalmente el contenido del área de trabajo, se usan los siguientes tres comandos:

a (de *append*) Deja el área de trabajo como estaba y añade el texto recibido como parámetro detrás.

i (de *insert*) Escribe el texto recibido como parámetro delante del texto contenido en el área de trabajo.

c (de *change*) Sustituye enteramente el texto del área de trabajo por el que recibe como parámetro.

Los tres se ajustan a la misma sintáxis, la cual es, además, algo peculiar para lo que es normal en SED:

```
Comando \
texto
```

donde Comando es cualquiera de los tres indicados («a», «i» o «c»). Tras el comando hay que incluir una barra invertida y un salto de línea. A continuación va el texto que queremos añadir. Si el texto a añadir consta de más de una línea, cada salto de línea debe ir precedido del carácter "\".

Los comandos «a» e «i», por otra parte, sólo admiten ámbitos simples. No se pueden usar con ámbitos del tipo «Inicio,Fin». El comando «c» si admite ámbitos dobles, pero reemplazará todas las líneas especificadas en el ámbito con una sola aparición del texto indicado como parámetro.

También difieren estos comandos en cómo afectan al área de trabajo, ya que mientras el comando «c» produce el mismo efecto colateral que el comando «d», en el sentido de que tras su ejecución se inicia un nuevo ciclo, sin aplicar los subsiguientes comandos

del script a la línea actual, los comandos «a» e «i» se considera que no afectan al área de trabajo; y, en consecuencia, el texto insertado no podrá ser tomado en consideración por ningún otro comando del mismo script, no afecta, tampoco, al contador de líneas de SED, y será enviado a la salida estándar incluso aunque se haya inhabilitado la opción *autoprint* y, por lo tanto, no se envíe el área de trabajo propiamente dicha.

Otros dos comandos, que permiten añadir texto, pero que examinaremos en la sección 5.3 son «r» y «R».

5.1.4. El comando «y»: Transliteración de texto

El comando «y» produce una transliteración de texto. Para ello recibe dos cadenas de texto y procede a buscar en el área de trabajo cada aparición de alguno de los caracteres de la primera cadena y, si los encuentra, los sustituye por el carácter correspondiente de la segunda cadena.

El formato del comando es:

```
y/origen/destino/
```

Las cadenas "origen" y "destino" deben tener la misma extensión, y al igual que ocurre en el comando «s», como delimitador de estas cadenas se puede usar cualquier carácter: SED asume que el primer carácter tras el comando «y» será el delimitador.

El uso más evidente de este comando es el de convertir de mayúsculas a minúsculas y viceversa. Así, en el siguiente ejemplo, todas las vocales, acentuadas o no, de una línea se convertirían en mayúsculas sin acentuar.

```
y/aáàâäeéèêëiíìîïoóòôöuúùûü/AAAAAEEEEEIIIIIOOOOOUUUUU/
```

Es importante tener en cuenta que este comando afecta a la totalidad de la línea y no sólo a una o varias palabras concretas. Para transformar una palabra concreta deberíamos usar el área auxiliar (ver sección 5.6).

5.2. Comandos que envían datos a la salida estándar

Como ya sabemos, por defecto SED, tras procesar una línea, la envía a la salida estándar, a no ser que se haya activado la opción *no autoprint* llamando a SED con la opción «-n». En este caso las salidas por pantalla las tiene que controlar el propio SED, y para ello se dispone de los siguientes comandos:

p (de *print*) Es el comando básico de este grupo: envía el contenido del área de trabajo a la salida estándar. Su uso sólo tiene sentido cuando está activada la opción *no autoprint*, pues en caso contrario, todas las líneas son enviadas a la salida estándar sin necesidad de que se pida expresamente. De hecho usar «p» con la opción *autoprint* activada provoca que la línea en cuestión se imprima dos veces.

= Envía a la salida estándar el número actual de línea. Tiene la peculiaridad de que no puede trabajar con ámbitos dobles. Es útil sobre todo de cara a la depuración de scripts, aunque, en ciertos contextos, puede tener su utilidad. Por ejemplo: si queremos emitir un listado de un programa en el que las líneas aparezca numeradas, bastará con la siguiente orden:

```
sed -e "=" MiPrograma.txt
```

Pueden verse, no obstante, en la última sección, algunos ejemplos que mejorarían bastante el aspecto de este listado.

l (de *list*) Imprime el área de trabajo aplicando una serie de convenciones que hacen que la salida sea "*no ambigua*". Esas convenciones son:

- Los caracteres no imprimibles y el carácter "\" se imprimen en el estilo de C.
- Las líneas excesivamente largas se parten con un carácter "\" al final. Por defecto se considera que una línea es excesivamente larga si tiene más de 70 caracteres[10]. Pero esto puede cambiarse indicando una nueva longitud tras el comando. Si se indica una longitud de 0 caracteres, las líneas largas no se dividirán.
- El final de cada línea se marca con "$".

Al igual que el anterior, este comando es útil de cara a la depuración de un script, ya que permite detectar caracteres habitualmente invisibles.

5.3. Lectura y escritura de ficheros

Un conjunto de comandos de SED permiten leer el contenido de un fichero, e insertarlo en nuestro texto, o escribir en un fichero alguna línea de nuestro texto. Los comandos básicos para ello son:

r Fichero Envía el contenido del fichero llamado Fichero a la salida estándar. Si estamos capturando la salida estándar, esto es tanto como insertar el contenido completo del fichero en el punto del texto en el que nos encontremos.

R Fichero Similar al anterior. La diferencia esta en que mientras «`r`» inserta el contenido completo del fichero, «`R`» inserta una línea del fichero y cada vez que es ejecutado en el mismo script y con el mismo fichero, insertará la línea siguiente. Pero debe tenerse en cuenta que si SED se está usando con las opciones «`-s`», «`--separate`», «`-i`» o «`--in-place`» (véase la sección 2.3.2), el contador de líneas del Fichero usado para insertar líneas, se reiniciará cada vez que SED empiece a procesar un fichero distinto[11].

[10] O, si se usaron las opciones «`-l`» o «`--length`» de SED (véase la sección 2.3.2), si tiene más caracteres que los indicados como parámetro de dicha opción.

[11] En la frase anterior se habla de dos tipos de ficheros distintos y por ello el resultado es algo ambiguo: el Fichero mencionado por el comando «`R`» y los ficheros que SED está procesando porque fueron llamados al invocar SED. Pues bien: cada vez que SED empieza a trabajar con uno de estos últimos ficheros, reiniciará el contador de líneas del fichero llamado por el comando «`R`».

w Fichero Envía el contenido actual del área de trabajo al fichero llamado Fichero. Si el fichero no existía lo crea, si ya existía, será sobreescrito. En sucesivas llamadas de «w» se irán añadiendo líneas al fichero en cuestión.

Como nombre de fichero se puede especificar "/dev/stdout" y "/dev/stderror", lo que provocará que el área de trabajo se envíe a la salida estándar o a la salida de errores.

W Fichero Similar al anterior pero si en el área de trabajo hay, cuando es invocado, más de una línea, sólo se envía la primera línea.

Entre el comando y el nombre del fichero hay que dejar un espacio en blanco. A partir del primer espacio en blanco todo lo que haya, hasta el carácter de nueva línea, será considerado nombre del fichero, incluyendo los posibles espacios en blanco adicionales.

Una diferencia importante entre «r» y «w» es que el primero no acepta ámbitos dobles, mientras que el segundo sí.

Algunas versiones de SED tienen limitado el número máximo de ficheros que, además de los que están siendo procesados, se pueden mantener abiertos en un mismo script.

5.4. Comandos que alteran el flujo normal de trabajo de SED

Como ya dije al principio SED trabaja por ciclos consistentes en leer una línea de texto, aplicarle uno por uno los comandos del script (comprobando previamente si, según el ámbito del comando, hay que aplicarlo) y enviar la línea transformada a la salida estándar, salvo que se haya activado la opción *no autoprint*.

Algunos comandos, sin embargo, alteran este flujo normal, y por lo tanto son especialmente importantes para escribir scripts de cierta complejidad. Los comandos que alteran el flujo normal son varios: «d», por ejemplo, lo hace pues ya hemos visto que tras haberlo ejecutado SED siempre inicia un nuevo ciclo. También lo hacen los comandos que permiten trabajar en áreas multilínea, y que más adelante veremos. Pero los más importantes comandos que hacen eso son los que a continuación vamos a examinar.

5.4.1. El comando *next*

El comando «n» carga en el área una nueva línea, *y continúa el ciclo* en el punto en el que estaba, de tal modo que los comandos posteriores a «n» no se aplicarán a la línea original, sino a la línea posterior y los comandos anteriores a «n» que se aplicaron a la línea original, no llegan nunca a aplicarse a la línea posterior.

En suma: «n» es uno de los comandos capaces de alterar el flujo de trabajo de SED, y de ahí su importancia de cara a la escritura de scripts complejos.

Antes de cargar la nueva línea, la línea original será enviada a la salida estándar si la opción *autoprint* estaba activada. En caso contrario no se enviará, salvo que antes de «n» se haya aplicado algún comando «p».

5.4.2. Etiquetas y saltos dentro de un script

En un script de SED podemos denominar a una sección del mismo con una etiqueta y generar saltos a dicha sección. Esto convierte a SED en un verdadero sistema de programación: cierto que primitivo y tosco, pero lenguaje a fin de cuentas.

Para crear una etiqueta en un script se usa, al principio de una línea, el signo de los dos puntos seguido (sin ningún espacio en blanco por en medio) del nombre de la etiqueta[12], el cual se considera que sigue hasta llegar al próximo salto de línea. Por tanto los espacios en blanco a la derecha del nombre de la etiqueta se considerará que forman parte de él.

Para saltar a una etiqueta se dispone de los siguientes comandos:

b [Etiqueta] (de *branch*) Si tras el comando se incluye el nombre de una etiqueta, se salta a ella, si no se incluye ninguna etiqueta, se inicia un nuevo ciclo. En terminología normal de lenguajes de programación diríamos que en el primer caso el comando b equivale a un GOTO, y en el segundo a un BREAK.

t [Etiqueta] de (*test*) Provoca un salto a Etiqueta, o el inicio de un nuevo ciclo (si no se incluye ningún nombre válido de etiqueta) si y solo si desde que se inició el ciclo ha habido alguna operación de sustitución (comando «`s`») con éxito. Este comando genera, por tanto, saltos condicionales.

T [Etiqueta] Similar a «`T`», pero provoca el salto en el caso contrario: si desde que se inició el ciclo ninguna operación de sustitución ha tenido éxito.

En los tres casos entre el comando y el nombre de la etiqueta puede haber un espacio en blanco, pero tras el nombre de la etiqueta no debe haber espacios en blanco, a no ser que éstos formen parte de él.

El comando «`b`», en combinación con la especificación de ámbitos mediante expresiones regulares, se puede usar para generar subprocedimientos, o para escribir bucles tipo “do while”. Por ejemplo las siguientes líneas:

```
:Inicio
Comando1
Comando2
/condicion/b Inicio
Comando3
```

ejecutarán los comandos 1 y 2, comprobarán si se da la condición. En caso afirmativo se salta de nuevo a Inicio, para volver a ejecutar los comandos 1 y 2. Sólo cuando la condición deje de cumplirse se permitirá la ejecución del comando 3.

[12] En algunas implementaciones de SED la longitud de la etiqueta está limitada a 7 caracteres. En la versión GNU se admiten etiquetas de cualquier longitud.

5.5. Comandos para trabajar con varias líneas simultáneamente

Aunque SED trabaja habitualmente con una línea de texto en cada ciclo, dispone de varios comandos que permiten trabajar en modo multilínea: Los más importantes son «N», «D» y «P»[13]. Si nos fijamos los tres comandos tienen el nombre en mayúscula, y en los tres casos hay un comando que empieza con la misma letra pero en minúsculas. Esto es porque «N», «D» y «P» son la versión multilínea de, respectivamente, «n», «d» y «p». Veamóslos por separado:

5.5.1. Añadir una línea al área de trabajo: El comando «N»

Mediante el comando «N» provocamos que SED lea la próxima línea, y la añada al área de trabajo, sin descartar la línea actual, con lo que pasaremos a tener, en el área de trabajo, dos líneas. Si de nuevo ejecutamos el comando pasaremos a tener tres líneas... y así sucesivamente.

Como se verá el comando «N» difiere del comando «n» en que si bien los dos cargan una nueva línea, el segundo termina de procesar la línea previa, y sustituye el área de trabajo con la nueva línea, mientras que «N» añade la nueva línea, sin eliminar la línea anterior.

La existencia de un área de trabajo con varias líneas afecta a la escritura de expresiones regulares que vayan a funcionar en ellas. Así:

- Las líneas se encuentran separadas por un carácter de nueva línea. Para referirnos a él en una expresión regular debemos usar la secuencia de escape \n.
- El operador ^ Identificará el principio de la primera línea, y el operador $ identificará el final de la última línea.

5.5.2. Borrar e imprimir líneas en un espacio multilínea

Si usamos los comandos «d» o «p» en un área de trabajo multilínea, borraremos o enviaremos a la salida todo el contenido del área de trabajo. Pero si lo que queremos es borrar exclusivamente alguna de las líneas que componen el área de trabajo, debemos usar los comandos «D» y «P» respectivamente:

D Borra en el área de trabajo hasta que encuentra el primer carácter de nueva línea; o, dicho con otras palabras, borra la primera línea en un área de trabajo multilínea. Además, a diferencia del comando «d» no inicia un nuevo ciclo, sino que reinicia el ciclo actual, sobre el resultante tras la operación de borrado.

P Envía a la salida estándar la primera línea de un área de trabajo multilínea.

[13] Además, también trabajan en modo multilínea los comandos «G», «H», «L» y «W», que, si nos fijamos, son todos los comandos cuyo nombre va en mayúsculas, salvo «Q», «R» y «T».

5.6. Comandos para usar el área auxiliar

Junto con el área de trabajo, SED mantiene otro buffer al que se llama *área auxiliar* (en inglés *hold space*), que está pensada para almacenar temporalmente en ella el contenido del área de trabajo antes de su manipulación. SED no proporciona ningún comando que permita manipular el texto directamente en el área auxiliar, sino exclusivamente comandos para comunicar entre sí los dos buffers: el área de trabajo y el área auxiliar.

Los comandos básicos son:

h (de *hold space*) Envía al área auxiliar el contenido actual del área de trabajo. Cualquier contenido previo del área auxiliar es sustituido por el nuevo contenido.

H Añade al área auxiliar, sin borrar su contenido previo, el contenido actual del área de trabajo. Entre el contenido anterior y el que ahora se añade se inserta un carácter de nueva línea.

g (de *get hold space*) Sustituye el contenido actual del área de trabajo por el del área auxiliar.

G Añade al área de trabajo, sin borrar su contenido previo, el contenido del área auxiliar. Entre el contenido anterior y el que ahora se añade se inserta un carácter de nueva línea.

x Intercambia los contenidos de las áreas de trabajo y auxiliar.

Obsérvese que las versiones en mayúsculas de ambos comandos, generan áreas multilínea.

Estos comandos son útiles para tareas bastante refinadas. Por ejemplo, si quisiéramos usar el comando «`y`» (véase la sección 5.1.4) para convertir a mayúsculas sólo una palabra de una línea, deberíamos construir un script que hiciera lo siguiente:

1. Enviara una copia de la línea que contiene la palabra al área auxiliar.
2. En el área de trabajo sustituyera el contenido completo de la línea por la palabra en la que queremos trabajar. Esto lo conseguiríamos mediante un simple «`s/.*/palabra/`», que reemplaza toda la línea por el texto “palabra”.
3. Aplicara el comando «`y`» al contenido del área de trabajo, donde ya sólo queda la palabra con la que queremos trabajar.
4. Añadiera al área de trabajo el contenido del área auxiliar.
5. Eliminara del área de trabajo el salto de nueva línea y restituyera la palabra transformada a su ubicación original. Esto lo conseguiríamos mediante:

```
s/\(PALABRA\)\n\(.*\)palabra\(.*\)/\2\1\3/
```

Merece la pena analizar la anterior expresión regular, pues, si la entendemos, significa que seguimos avanzando en nuestro aprendizaje. En ella se hace uso de la posibilidad de agrupar parte de una expresión regular para luego referirnos a ella en el texto de reemplazo. Veámosla poco a poco:

- Como punto de partida sabemos que el área de trabajo empieza por la palabra que hemos transformado, seguida de un salto de línea, por lo tanto empezamos generando un grupo que aisle dicha palabra, para luego volverla a usar.
- Tras el salto de línea generamos un segundo grupo que encierre todos los caracteres de la línea que pueda haber antes de la palabra que buscamos.
- Tras la aparición de la palabra buscada en la segunda línea, generamos un tercer grupo que contenga todo lo que haya en la línea a la derecha de nuestra palabra.
- Una vez que tenemos organizada toda la información en tres grupos, nos basta con, en el texto de reemplazo, reordenarlos. Primero irá el grupo \2 que contiene lo que había a la izquierda de la palabra; luego el grupo \1 que contiene la palabra propiamente dicha y, finalmente, el grupo \3, con todo lo que hubiera en la línea detrás de la palabra.

En el ejemplo anterior asumo que sabemos con qué palabra habíamos trabajado. Si no lo supiéramos deberíamos reescribir el comando de la siguiente manera:

```
s/\([^\n]\+\)\n\(.*\)\1\(.*\)/\3\1\2/I
```

Obsérvese que nos hemos limitado a encerrar en el primer grupo todo lo que haya hasta el salto de línea: esa es nuestra palabra. Y para referirnos a ella en la segunda línea, simplemente hacemos referencia a \1. Pero como en la primera línea la palabra está en mayúsculas y en la segunda línea la palabra está en minúsculas, tenemos que indicar al comando «`s`» que queremos que no diferencie entre mayúsculas y minúsculas, para así asegurarnos de que la primera referencia a \1 funcione correctamente.

5.7. Otros comandos

Además de los comandos vistos, SED dispone de los siguientes comandos, difíciles de clasificar:

q[Código] (de *quit*) Termina la ejecución del script. No admite ámbitos dobles. Puede ser útil para, por ejemplo, emular el funcionamiento del comando «`head`» de UNIX y mostrar por pantalla las primeras n líneas de un fichero. Así:

```
sed -e "25q" MiFich.txt
```

Imprimirá las primeras 25 líneas de MiFich.txt.

«`q`» puede recibir un parámetro opcional que se enviará al sistema operativo como código de salida del script.

Q[Código] Similar a «`q`» pero evita termina la ejecución de SED sin enviar el contenido del área de trabajo a la salida estándar. Por lo tanto usando «`Q`» no se mostrará la línea de texto en la que se produjo la finalización del script.

L[N] (de *List*) Este comando está considerado un experimento fallido, apenas tiene uso y no se excluye que en versiones futuras de SED desaparezca, por lo que no se recomienda su uso. Fue concebido como una extensión del comando «`l`». «`L`» llena y junta líneas en el área de trabajo hasta lograr líneas de, al menos, «`N`» caracteres. Si no se indica un valor para «`N`» se usará el ancho de línea estándar.

v[Versión] (de *version*) Permite asegurarnos de cuál es la versión instalada de SED, por si nuestro script incluye comandos que sólo están implementados a partir de cierta versión. Si no se incluye como argumento un número de versión se comprobará sólo si la versión instalada soporta las extensiones GNU de SED, introducidas a partir de la versión XXX

z (de *zap*) Vacía completamente el área de trabajo. Equivale al uso de «`s/.*//`» pero, según la ayuda oficial de SED, el comando «`z`» es más eficiente.

Es, por otra parte, importante comprender la diferencia entre «`z`» y «`d`». El primero vacía el área de trabajo, pero continúa con la ejecución del ciclo: todavía podemos usar algún comando «`a`», «`i`», «`c`», o traer el contenido del área auxiliar, e incluso si no hacemos nada, al terminar de procesar la línea, SED la enviará a la salida estándar, generando así una línea vacía. Por el contrario «`d`»: borra la línea, no envía nada a la salida estándar, y termina la ejecución del ciclo.

e (de *execute*) Este comando permite que el contenido del texto de entrada interactúe con el script pues hace que SED interprete el contenido actual del área de trabajo como un comando con sus especificaciones: «`e`» ejecutará ese comando, y sustituirá el contenido del área de trabajo por el resultado de la ejecución del comando.

«`e`» puede, por otra parte, recibir un parámetro opcional que se interpreta como comando a ejecutar. En tal caso el efecto de «`e`» será ejecutar dicho comando y cambiar el contenido del área de trabajo por el resultado de la ejecución.

6. Algunos ejemplos concretos

Hasta aquí la explicación teórica. Para asegurarnos de que la hemos entendido creo que lo mejor es incluir una serie de ejemplos reales, explicados. Eso nos ayudará a digerir mejor el uso de los distintos comandos de SED y a aprender a identificar las situaciones en las que SED puede ser útil.

En Internet hay decenas de páginas con ejemplos de scripts de SED. En

`http://sed.sourceforge.net`

es posible encontrar los más importantes enlaces. Tras estudiar los ejemplos que a continuación se incluyen, recomiendo que se visiten esas páginas y se estudien los scripts,

no tanto para aprender cómo hacer ciertas cosas, sino, sobre todo, para comprender cómo las hace el script en cuestión, y aumentar así nuestro conocimiento de SED y de las expresiones regulares.

6.1. Scripts de una sola línea

Los siguientes ejemplos los he extraído, en su mayoría, de

http://sed.sourceforge.net/grabbag/tutorials/sed1line.txt.

Están todos basados en una sola línea de texto, y sirven para demostrar cómo, con una sola línea, SED puede conseguir resultados asombrosos. Todos los ejemplos se aplicarán a un hipotético fichero llamado "prueba.txt" y, tras el ejemplo comentaré por qué funciona, y en qué sección de este documento se explican los comandos usados en el ejemplo:

6.1.1. Líneas en blanco

Los siguientes ejemplos muestran como podemos con facilidad añadir o eliminar líneas en blanco encima o debajo de todas las líneas, o sólo de las que cumplan cierta condición.

1. Duplicar los saltos de línea de un fichero:

   ```
   sed "G" prueba.txt
   ```

   El comando G (sección 5.6) añade al área de trabajo el un salto de línea y el contenido del área auxiliar. Al empezar el script se añade sólo el salto de línea, pues el área auxiliar está vacía. Como hemos indicado el comando sin ámbito, se aplica a todas las líneas. El resultado es, por lo tanto, que tras cada línea, se añade una línea adicional vacía.

   Por lo tanto «`sed "G;G" prueba.txt`» triplicaría los saltos de línea, y «`sed "G;G;G" prueba.txt`» los cuadruplicaría.

2. Deshacer el duplicado de líneas generado con el comando anterior:

   ```
   sed "n;d" prueba.txt
   ```

   Se asume que, tras aplicar el duplicado de líneas, en el fichero las líneas pares están siempre vacías. Por lo tanto el script carga la primera línea. El comando «`n`» (sección 5.4.1) la envía a la salida estándar y carga la segunda línea (que está vacía), tras lo que el comando «`d`» (sección 5.1.2) borra esta nueva línea.

3. Insertar una línea en blanco encima de todas las líneas en las que aparezca cierta expresión regular:

   ```
   sed "/regexp/ {x;p;x}" prueba.txt
   ```

/regexp/ (sección 4.2.2) determina que el script sólo se aplique a las líneas en las que aparezca dicha expresión regular. El comando «x» (sección 5.6) intercambia las áreas de trabajo y la auxiliar. Como el área auxiliar está vacía, el área de trabajo pasará a consistir en una línea en blanco. El comando «p» (sección 5.2) imprime dicha línea en blanco, y un nuevo uso de «x» devuelve al área de trabajo su contenido habitual que será enviado a la salida estándar al terminar el ciclo.

4. Insertar una línea en blanco debajo de todas las líneas en las que aparezca cierta expresión regular:

```
sed "/regexp/ G" prueba.txt
```

   Este comando es una combinación de los vistos en los ejemplos 1 y 3.

5. Insertar una línea en blanco encima y debajo de las líneas que contengan cierta expresión:

```
sed "/regexp/ {x;p;x;G}" prueba.txt
```

   Este script es una combinación de los ejemplos examinados en los números 3 y 4.

6. Eliminar las dobles líneas en blanco:

```
sed "/^$/ {n; /^$/ d}" prueba.txt
```

   La expresión regular inicial selecciona las líneas vacías, es decir: aquellas en las que tras el principio de la línea (operador ^) se encuentra el final de la línea (operador $). El script entonces carga la siguiente línea mediante el comando «n» (sección 5.4.1) y, si también cumple la condición de ser una línea vacía, ejecuta el comando «d» (sección 5.1.2) y borra esta segunda línea. Obsérvese como, en este ejemplo, los ámbitos funcionan como condición para ejecutar el comando. En un lenguaje de programación entandar este ejemplo se representaría así:

```
IF linea-vacía
  cargar próxima línea
  IF linea-vacía
    borrar línea
  ENDIF
ENDIF
```

   Si añadiéramos una etiqueta y usáramos los comandos para generar saltos (sección 5.4.2) podríamos escribir un bucle que detectara todas las líneas vacías consecutivas y las borrara. Pero ese ejemplo ya no podría escribirse en una sola línea, por lo que no lo incluyo aquí. Puede ser un buen ejercicio para el lector intentar escribir ese script.

6.1.2. Numeración de líneas

1. Numerar las líneas del fichero separando por un tabulador el número de línea de su contenido:

   ```
   sed "=" prueba.txt | sed "N;s/\n/\t/"
   ```

   Este ejemplo es interesante, porque aprovecha la capacidad de SED para tomar su entrada de la salida de otro programa que en este caso es también SED. Es decir: ejecutamos dos veces SED.

   - La primera ejecución se limita a aplicar a cada línea el comando «=» (sección 5.2) que, al ejecutarse sin ámbito, añade a cada línea una línea previa con el número asignado a la misma.
   - La salida del anterior comando, es capturada de nuevo por SED que ahora aplica a cada línea el script «`N;s/\n/\t/`». Este script empieza con el comando «`N`» (sección 5.5.1) que añade a la línea actual la próxima línea, creando así un área de trabajo multilínea. A continuación se realiza una sustitución (sección 5.1.1), y el salto de línea que separa a las dos líneas (secuencia de escape «`\n`») es sustituido por un tabulador (secuencia de escape «`\t`»).

2. Numera las líneas de un fichero de modo similar al ejemplo anterior, pero sólo si la línea no está en blanco.

   ```
   sed "/./=" prueba.txt | sed "/./N;s/\n/\t/"
   ```

   La mayor parte es idéntica al ejemplo anterior. La única diferencia está en que en la primera ejecución de SED se limita el ámbito del comando «=» y sólo se aplica a las líneas que no están vacías, y en la segunda ejecución, se pone la misma restricción, ya que sólo las líneas no vacías irán precedidas de su número.

3. Numera las líneas de un fichero, alineando decimalmente los números:

   ```
   sed "=" prueba.txt | sed "N;s/^/     /;s/ *\(.\{6,\}\)\n/\1 /"
   ```

   El interés de este ejemplo está en el uso que se hace de los grupos de sustitución y en la gran sutileza de los dos comandos de sustitución que se emplean. La primera parte es igual que en el ejemplo anterior: Se ejecuta SED una primera vez para añadir los números de línea, y se captura su propia salida, haciendo que, tras cada línea (que contiene el número), se añada la siguiente línea (que contiene el texto). A partir de aquí empiezan los cambios:

   - En el comando posterior a «`N`» (véase la sección 5.5.1), añadimos cinco espacios en blanco al principio de la línea, mediante un comando «`s`» (sección 5.1.1); eso es porque, para este ejemplo, hemos asumido que el fichero no tendrá más de 99999 líneas, pero podríamos cambiar el número de espacios

en blanco. Para que el script funcione bien debe haber, al menos, tantos espacios en blanco como cifras tenga el número de la última línea, pues lo que pretendemos es, más adelante, sustituir esos espacios en blanco por cifras.

El resultado del primer comando será que el área de trabajo empezará por cinco espacios en blanco, luego irá el número, luego irá el salto de línea («`\n`») y, por último, el contenido original de la línea.

- El siguiente comando es muy sutil y se basa en la distribución del área de trabajo que el comando anterior realizó. Antes del carácter «`\n`» habrá un numero de caracteres igual al número de cifras de que conste el número de línea + 5. Es decir: en las líneas 1 a 9, habrá seis caracteres antes del salto de línea. En las líneas 10 a 99 habrá 7 caracteres, etc.

  Pues bien: aquí viene la sutileza: el último comando de nuestro script encierra en un grupo todos los caracteres que preceden al salto de línea, asegurándose de que el grupo conste de al menos seis caracteres y descartando los caracteres en blanco a la izquierda del grupo. Esto es lo que hace la expresión regular «`␣*\(.\{6,\}\)\n`». Si la analizamos por separado, veremos que consta de los siguientes elementos:

  — Empieza por «`␣*`» lo que significa, un número indeterminado de espacios en blanco, que incluso puede no tener ninguno. Esta parte de la expresión regular absorberá los caracteres que no quepan en la siguiente parte.

  — A continuación tenemos un grupo (encerrado entre «`\(`» y «`\)`») que puede contener cualquier carácter («`.`») y debe tener una extensión de, al menos, 6 caracteres («`\{6,\}`»)

  — Tras el grupo se encuentra el salto de línea («`\n`»). En realidad, como las dos partes anteriores de la expresión regular pueden tener una longitud variable, ambas se calculan a partir del salto de línea, lo que significa que esta expresión regular, para ser comprendida, debe ser leída de derecha a izquierda.

  El resultado será que, para la línea primera, el grupo contendrá cinco espacios en blanco y el número de línea. Pero para la línea 10, el grupo contendrá *sólo* cuatro espacios en blanco delante del número de línea; y para la línea número 100 los espacios en blanco serán sólo 3. El efecto es que, como cada número va precedido por un número de espacios en blanco inversamente proporcional al número de cifras de que consta, los números se muestran alineados a la derecha. Por lo tanto ya sólo queda sustituir esa cadena de búsqueda, por el resultado del grupo seguido de un espacio en blanco, que es lo que hace la última parte del comando.

4. Cuenta las líneas de un fichero. Emula, en este aspecto, al comando «`wc`» de UNIX.

```
sed -n "$=" prueba.txt
```

Llamamos a SED con la opción «`-n`» para evitar que las líneas vayan siendo enviadas a la pantalla. El comando que ejecutamos, «`=`» (ver sección 5.2) restringe

su ámbito exclusivamente a la última línea del fichero, representada por «$». El resultado es que sólo se muestra por pantalla el número de línea de la última línea.

6.2. Un script, en varias líneas, que elimina los espacios en blanco y saltos de línea superfluos

6.2.1. Por qué eliminar estos caracteres de un fichero de texto

La separación de palabras es siempre un problema en las expresiones regulares. Si buscamos más de una palabra, entre ellas habría siempre que incluir no un simple espacio en blanco, sino la secuencia «[\t]\+», para asegurarnos de que cubrimos cualquier posibilidad. Esto complica sobremanera la escritura de las expresiones regulares. Pero además no resuelve totalmente el problema, pues es posible que la frase que buscamos esté repartida entre dos líneas, en cuyo caso, el hecho de que SED analice el texto de entrada línea a línea, hace que, en principio, sea imposible encontrar frases repartidas en más de una línea.

Aunque esta imposibilidad no es tal, sino más bien, dificultad: El comando «N» nos permitiría buscar la primera palabra de nuestra expresión regular y, si está a final de línea, cargar la línea siguiente. Pero... ¿Y si la expresión buscada consta de tres palabras? El salto de línea podría estar entre la primera y la segunda o entre la segunda y la tercera. ¿Y si buscamos una frase de cuatro palabras?...

Como se ve, se podría hacer, pero sería un script complicadísimo.

Las cosas se simplificarían mucho si pudiéramos eliminar los saltos de línea superfluos, es decir: los que no indican separación de párrafo. Y ya puestos, podríamos también eliminar los espacios en blanco y tabuladores superfluos.

Un script que hiciera eso simplificaría el tratamiento con SED del texto en cuestión; y por ello ese es el ejemplo que a continuación propongo. El siguiente script, por lo tanto, es ideal para aplicarlo a ficheros escritos en formatos en los que los espacios en blanco extra son ignorados, y donde la separación entre párrafos se indica mediante una o más líneas vacías. Por ejemplo: un fichero escrito en LaTeX o en XML.

6.2.2. La lógica básica del script

El script que hiciera eso, podría ser, más o menos, así:

```
#!/usr/bin/sed -f
# Elimina los caracteres en blanco, tabuladores
# y saltos de línea superfluos en un fichero de
# texto. Debe ser llamado con la opción -n.

$! H

$ {
  H
  g
```

```
s/^\n//
s/\n[ \t]*\(.*\)[ \t]*\n/\n\1\n/g
s/[ \t]\{2,\}/ /g
s/\n\n\+/@@##@@??@@/g
s/\n/ /g
s/[ \t]\?@@##@@??@@[ \t]\?/\n\n/g
p
}
```

Analicemos ahora el script. Básicamente lo que se hace en él es cargar todo el fichero en un área multilínea. Para ello se usa el área auxiliar.

En el script se distingue entre la última línea (cuyo ámbito es «`$)`» y las demás (es decir, las que no son la última, o sea «`$!`» (véase la sección 4.2.4).

Con las líneas que no son la última, nos limitamos a enviarlas al área auxiliar mediante el comando «`H`» (sección 5.6). Pero al llegar a la última línea, cuando tras enviarla también al área auxiliar ya estamos seguros de tener allí almacenado todo el fichero, entonces hay que empezar el trabajo de ajuste.

- Lo primero que hacemos es traer al área principal el contenido del área auxiliar. Esto se puede hacer con dos comandos: «`g`» o «`x`» (véase sección 5.6). Como en este caso no nos importa perder el contenido del área de trabajo tras cargar el área auxiliar, hemos usado «`g`». En caso contrario habríamos debido usar «`x`». Si aquí hubiéramos usado «`x`», el resultado sería el mismo.

- Una vez que tenemos en el área de trabajo todo el fichero, ya podemos empezar a trabajar. Básicamente aquí se usa el comando «`s`» (5.1.1), pues lo que hacemos son sustituciones:

  1. En primer lugar quitamos el primer salto de línea del fichero. Este salto de línea es consecuencia de que la primera línea fue enviada al área auxiliar mediante «`H`», y no mediante «`h`», por lo que la primera línea se colocó en el área auxiliar como segunda línea.
  2. La segunda sustitución elimina los espacios en blanco o tabuladores que pueda haber al principio o al final de cada línea. Para ello usa una expresión regular que podemos analizar por partes:
     - «`\n[ \t]*`» es decir, cualquier espacio en blanco o tabulador al principio de una línea. Para indicar el principio de línea usamos la secuencia de escape «`\n`» porque, en un entorno multilínea, el operador «`^`» se refiere sólo al principio de la primera línea.
     - «`\(.*\)`», se lee como: “lo anterior seguido de cualquier secuencia de caracteres”. Esa secuencia de caracteres constituye el contenido real de la línea, y por ello la encerramos en un grupo, para poderla rescatar en el texto de reemplazo del comando «`s`».
     - «`[ \t]*\n`» Esta tercera parte de la expresión, absorbe todos los caracteres en blanco o tabuladores que se encuentren al final de la línea, indicado mediante la secuencia de escape «`\n`».

El texto de reemplazo es «\n\1\n», es decir: el contenido real de la línea, encerrado entre dos marcas de línea.

3. La tercera sustitución detecta los espacios en blanco y marcas de tabulación repetidos y que, al no estar al principio o al final de las líneas, no fueron suprimidos por la anterior operación. Obsérvese que la expresión regular usa el operador de repetición «\{2,\}» que significa que la expresión anterior se debe repetir 2 o más veces.
4. Tras las anteriores sustituciones podemos estar seguros de que los saltos de línea repetidos marcan saltos de párrafo, y los no repetidos, pueden sustituirse por un espacio en blanco. Por lo tanto lo que hacemos es, antes de sustituir los saltos de línea, *marcar* los lugares donde hay auténticos saltos de párrafo. *Marcarlos* significa, poner en ellos una secuencia de caracteres que nos permita más tarde localizarlos. Debe ser por lo tanto una secuencia que sea imposible que, de modo natural, aparezca en el fichero. He elegido: «@@##@@??@@». Como, por otra parte, es posible que existan saltos de línea superfluos, como expresión regular no se usa solamente «\n\n», sino que se añade el operador «\+» para, de paso, eliminar estos saltos de línea superfluos.
5. El penúltimo paso consiste en sustituir los saltos de línea que quedan (que son los superfluos) por un espacio en blanco.
6. Y, finalmente, insertamos dos saltos de línea en los lugares que antes marcamos para indicar que ahí iba un salto de párrafo.

- Por último, enviamos a la salida estándar el contenido del área de trabajo, con todas nuestras modificaciones.

6.2.3. Refinando nuestro script

El siguiente script hará su trabajo. Pero la verdad es que, cuando trabajamos con ficheros de texto escritos para cierto formato (LaTeX, HTML, XHTML, etc), habrá líneas en las que no nos interese suprimir el salto de línea, aunque no sea doble. En LaTeX, por ejemplo, el carácter «%» indica que lo que vaya tras él *hasta el final de la línea* es un comentario, y por lo tanto sería un error suprimir el salto de línea posterior...

Para trabajar bien con SED es fundamental conocer los detalles de la construcción del fichero de texto al que aplicamos SED, pues sólo así podremos refinar lo bastante los scripts, y conseguiremos un auténtico ahorro de tiempo. Una parte importante en la escritura de todo programa es el análisis, y SED, a fin de cuentas, lo que hace es programar ediciones de texto. El análisis es fundamental.

Por lo tanto podríamos refinar el anterior script para aplicarlo a algún formato concreto. Básicamente el refinado exigiría, en primer lugar, decidir en qué líneas no hay que suprimir el salto de línea final, y detectar esas líneas y marcarlas, tal y como hicimos en el paso 4 del script básico, para poder restaurar en ellas el salto de línea simple. La marca debería, por otra parte, ser distinta a la usada para marcar los saltos de párrafo, y debería

quedar claro que una marca de salto de párrafo absorbe a la marca de salto de línea que pueda estar antes o después de ella.

En fin: esta labor la dejo como “deberes” para los lectores del presente documento. Si son capaces de hacerla, aplicándolo a su formato de texto favorito... ¡Enhorabuena! este documento habrá cumplido con la función para la que fue diseñado.

Apéndice: El alfabeto de SED

Los comandos de SED constan de un solo carácter que, salvo en un caso, es siempre una letra. Por suerte hay letras suficientes en el alfabeto, sobre todo porque, al distinguir SED entre mayúsculas y minúsculas, se duplica el tamaño del alfabeto. En la siguiente lista se recogen, por orden alfabético, los distintos comandos de SED, sintetizando, en una sola frase, para qué sirven, e indicando la sección de este documento donde se explica el comando en cuestión.

Al indicar la función de cada comando, utilizo la expresión *área de trabajo* para referirme a la línea en la que se está trabajando en cada momento. Recuérdese que, como regla, SED trabaja línea a línea. Asimismo, para los comandos que generan o trabajan en entorno multilínea, añado, tras su función, y entre paréntesis, las siglas (ML) que deben interpretarse como "este comando trabaja o genera un área multilínea).

Comando	Función	Explicado en
=	Envía a la salida estándar el número de la línea actual	Sección 5.2
a	Añade texto tras el área de trabajo*	Sección 5.1.3
b	Provoca un salto incondicional	Sección 5.4.2
c	Sustituye todo el contenido del área de trabajo	Sección 5.1.3
d	Borra la línea e inicia un nuevo ciclo	Sección 5.1.2
D	Borra la primera línea del área de trabajo (ML)	Sección 5.5.2
e	Ejecuta un comando	Sección 5.7
g	Sustituye el contenido del área de trabajo por el del área auxiliar	Sección 5.6
G	Añade el contenido del área auxiliar tras el área de trabajo (ML)	Sección 5.6
h	Sustituye el contenido del área auxiliar por el del área de trabajo	Sección 5.6
H	Añade al área auxiliar el contenido del área de trabajo (ML)	Sección 5.6
i	Inserta texto delante del contenido del área de trabajo*	Sección 5.1.3
l	Imprime el área de trabajo, aplicando ciertas convenciones	Sección 5.2
L	Extensión del comando «`l`». No se recomienda su uso (ML)	Sección 5.7
n	Sustituye el área de trabajo por la siguiente línea	Sección 5.4.1
N	Añade al área de trabajo la siguiente línea (ML)	Sección 5.5.1
p	Envía a la salida estándar el contenido del área de trabajo	Sección 5.2
P	Envía a la salida estándar la primera línea del área de trabajo (ML)	Sección 5.5.2
q	Termina la ejecución del script	Sección 5.7
Q	Similar a «`Q`», aunque con diferencias de matiz	Sección 5.7
r	Añade, tras el área de trabajo, el contenido de un fichero	Sección 5.3
R	Añade, tras el área de trabajo, una línea de un fichero	Sección 5.3
s	Realiza sustituciones de texto dentro del contenido del área de trabajo	Sección 5.1.1
t	Provoca un salto condicional	Sección 5.4.2
T	Provoca un salto condicional	Sección 5.4.2
v	Se asegura de que se esté manejando cierta versión de SED	Sección 5.7
w	Envía a un fichero el contenido del área de trabajo	Sección 5.3
W	Envía a un fichero la primera línea del área de trabajo (ML)	Sección 5.3
x	Intercambia los contenidos del área auxiliar y el área de trabajo	Sección 5.6
y	Aplica una transliteración al área de trabajo	Sección 5.1.4
z	Vacía el contenido del área de trabajo	Sección 5.7

*No he considerado multilínea a los comandos «`a`» e «`i`» porque, aunque de facto provocan que el área de trabajo pase a tener más de una línea, se trata de líneas adicionales que no se pueden manipular mediante ningún otro comando.