Introducción a Sistemas Operativos: Usando el shell

Francisco J Ballesteros

1. Comandos como herramientas

En UNIX la idea es combinar programas ya existentes para conseguir hacer el trabajo. Actualmente, por desgracia, hay muchos usuarios de UNIX que lo han olvidado y corren a implementar programas en C u otros lenguajes para tareas que ya se pueden resolver con los programas existentes.

Por ejemplo, quizá conozcas un programa llamado *head(1)* que imprime las primeras líneas de un fichero. Si recuerdas que *seq(1)* escribe líneas con enteros hasta el valor indicado, verás que en

```
unix$ seq 10 | head -2
1
2
unix$
```

`head` ha impreso las primeras dos líneas aunque `seq` escriba 10 en este caso. Pues resulta que no era preciso programar `head`, dado que con un editor de *streams* (texto según fluye por un pipe) ya era posible imprimir las primeras *n* líneas. Baste un ejemplo:

```
unix$ seq 10 | sed 2q
1
2
unix$
```

Se le pide educadamente a *sed(1)* que termine tras imprimir las primeras dos líneas sin hacer ninguna edición en ellas y ya lo tenemos.

Saber utilizar y combinar los comandos disponibles en UNIX te resultará realmente útil y ahorrará gran cantidad de tiempo. Además, dado que estos programas ya están probados y depurados evitarás bugs innecesarios. Simplemente utiliza el manual (¡Y aprende a buscar en él!) para localizar comandos que hagan o todo o parte del trabajo que deseas hacer. Combina varios de ellos cuando no sea posible efectuar el trabajo on uno sólo de ellos, y recurre a programar una nueva herramienta sólo como último recurso. Los comandos de unix son una caja herramientas, ¡úsala!.

La utilidad del shell es, además de permitirnos escribir comandos simples, permitirnos combinar programas ya existentes para hacer otros nuevos. Ya lo hemos estado haciendo anteriormente, pero ahora vamos a dedicarle algo de tiempo a ver cómo programar utilizando el shell. Recuerda que *sh(1)* está disponible en todos los sistemas UNIX, por lo que aprender a utilizarlo te permitirá poder combinar comandos y programar scripts en todos ellos.

2. Convenios

Para facilitarte el trabajo a la hora de utilizar el shell, deberías ser sistemático y seguir tus propios convenios. Por ejemplo, si las funciones en C las escribimos siempre como en

```
int
myfunc(int arg)
{
    ...
}
```

entonces sabemos que todas las líneas con un identificador a principio de linea seguido de un "`(`" está definiendo una función.

El comando *egrep(1)* imprime líneas que encajan con una expresión (como veremos más adelante). Si hemos sido sistemáticos con nuestro convenio para programar funciones, podríamos pedirle que escriba todas las líneas que declaran funciones en un directorio dado. No te preocupes por cómo se usa `egrep` por el momento, presta atención a cómo un simple convenio nos permite trabajar de forma efectiva.

Podemos utilizar el comando *egrep(1)* para ver qué funciones definimos y en qué ficheros y líneas:

```
unix$ egrep -n '^[a-z0-9A-Z_]+\(' *.c
alrm.c:11:tout(int no)
alrm.c:17:main(int argc, char* argv[])
broke.c:8:main(int argc, char* argv[])
fifo.c:14:main(int argc, char* argv[])
...
```

O para contar cuántas funciones definimos...

```
unix$ egrep -n '^[a-z0-9A-Z_]+\(' *.c | wc -l
      24
unix$
```

Programando en un entorno integrado de desarrollo (IDE o *integrated development environment*) puedes hacer este tipo de cosas pulsando botones con el ratón. ¡Pero ay de ti si quieres hacer algo que no esté disponible en tu IDE! Con frecuencia notarás que estás haciendo trabajo repetitivo y mecánico, ¡incluso si utilizas un IDE en lugar de un editor sencillo! Cuando te suceda eso, recuerda que seguramente puedas programar un script que haga el trabajo por ti.

El shell es muy bueno manipulando ficheros, bytes que proceden de la salida de un comando y texto en general. Igual sucede con los comandos que manipulan ficheros de texto en UNIX. Para programar utilizando el shell hay que pensar que no estamos utilizando C e intentar escribir comandos que operen sobre todo un fichero a la vez o sobre todo un flujo de bytes a la vez. La idea es ir adaptando los datos que tenemos a los que queremos tener. Otra estrategia es adaptar los datos que tenemos para convertirlos en comandos que, una vez ejecutados, produzcan el efecto que deseamos. Iremos viendo esto poco a poco con los ejemplos que siguen.

En muchos casos es posible que estés intentando resolver problemas que has creado tu mismo. Lo mejor en la mayoría de los casos suele ser intentar no crear los problemas en lugar de resolverlos. Baste un ejemplo.

Supón que estás programando una aplicación que requiere de un fichero de configuración. Tradicionalmente estos ficheros suelen guardarse en el directorio `$HOME` con nombres que comienzan por "`.`", pero eso nos da igual en este punto. Ya sea por seguir la moda o por no saber cómo leer líneas y cómo partirlas en palabras, quizá programemos la aplicación utilizando XML como formato para el fichero de configuración. ¡Qué gran error! Al menos, en los casos en que nuestro fichero de configuración necesite información tan simple como

```
shell /bin/sh
libdir /usr/lib
```

para indicar qué shell queremos que ejecute y qué directorio queremos que utilice para almacenar sus

librerías (por ejemplo). El lugar de un fichero tan sencillo como el que hemos mostrado, quizá termines con algo como

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE MyConfig SYSTEM "Crap_Config.dtd">
<Crap_Config>
    <Parameters>
        <Param>
            <Name>shell</Name>
            <Value>/bin/sh</Value>
        </Param
        <Param>
            <Name>libdir</Name>
            <Value>/usr/lib</Value>
        </Param
    </Parameters>
</Crap_Config>
```

Es cierto que tienes librerías ya programadas para la mayoría de lenguajes que saben entender ficheros en este formato, leerlos y escribirlos. Pero eso no es una excusa si el fichero en cuestión guarda información que puedas tabular (líneas de texto con campos en cada línea). En realidad, ni siquiera es una excusa si necesitas guardar una estructura de tipo árbol que sea sencilla, como por ejemplo este fichero:

```
/
    home
        nemo
    bin
    tmp
```

que podría indicar con qué partes de un árbol de ficheros queremos trabajar.

Pero volvamos a nuestro fichero de configuración. Supón que deseas comprobar que para todos los usuarios que utilizan tu aplicación en el sistema, los shells que han indicado existen y son ejecutables y los directorios existen. ¿Cómo lo harás? Con el formato sencillo para el fichero bastaría hacer algo como

```
unix$ shells=‘egrep ’^shell’ /home/*/.myapp | cut -f2‘
```

para tener en `$shells` la lista de shells y luego podemos utilizar esta variable y un bucle `for` del shell para comprobar que existen. Si has utilizado XML también podrías programar un script para hacerlo, pero te resultará mucho más difícil. Te has buscado un problema que no tenías.

Pero vamos a ver cómo usar el shell para hacer este tipo de cosas...

3. Usando expresiones regulares

Vamos a resolver el problema que teníamos. Supongamos que tenemos nuestra aplicación `myapp` con su fichero de configuración en `$HOME/.myapp` para cada usuario, y que su aspecto es como habíamos visto:

```
shell /bin/sh
libdir /usr/lib
bindir /usr/bin
logdir /log
```

Vamos a hacer un script llamado `chkappconf` que compruebe la configuración para todos los usuarios del sistema. Por el momento vamos a usar un fichero llamado `config` que tiene justo el contenido que mostramos arriba como ejemplo de fichero de configuración. Luego es fácil trabajar con los ficheros de verdad. La idea es procesar los ficheros *enteros*, poco a poco, e ir generando nuevos "ficheros" como salida de

comandos que procesan los datos que tenemos. En realidad, generamos bytes que fluyen por un pipe.

En este caso, vamos a necesitar utilizar *grep(1)*, o, concretamente *egrep(1)*. Este comando lee su entrada, o ficheros indicados en la línea de comandos, línea a línea y escribe aquellas que encajan con la expresión que le damos. Por ejemplo,

```
unix$ egrep dir config
libdir /usr/lib
bindir /usr/bin
logdir /log
```

El primer parámetro es la expresión que buscamos y el segundo es el fichero en que estamos buscando.

Otro ejemplo:

```
unix$ seq 15 | egrep 1
1
10
11
12
13
14
15
unix$
```

En este caso `egrep` lee de la entrada estándar, dado que no hemos indicado fichero alguno.

Las expresiones de `egrep` son muy potentes. Son de hecho un lenguaje denominado **expresiones regulares** que se utiliza en diversos comandos de UNIX, por lo que resulta muy útil aprenderlo. Mirando la página *egrep(1)* vemos hacia el final...

```
unix$ man egrep
...
SEE ALSO
     ed(1), ex(1), gzip(1), sed(1), re_format(7)
...
```

Y precisamente *re_format(7)* en este sistema documenta las expresiones regulares. Esto lo hemos hecho en un sistema OS X. Si usamos un sistema Linux podemos hacer la misma jugada

```
unix$ man egrep
...
SEE ALSO
awk(1),  cmp(1), diff(1), find(1), sed(1), sort(1),
       glob(7), regex(7).
...
```

y vemos que *regex(7)* documenta sus expresiones regulares. Las que usamos en este curso funcionan en ambos.

Una expresión regular es un string que describe otros strings. Decimos que un string encaja con la expresión si *contiene* uno de los strings que describe la expresión. Podemos definir las expresiones recursivamente como sigue:

- Cualquier carácter normal que no forma parte de la sintaxis de expresiones regulares encaja con el mismo. Por ejemplo, `a` describe el string `a`.
- La expresión "`.`" describe con cualquier carácter, pero sólo uno. Por ejemplo, `.` puede ser tanto `a` como `b`, pero no el string vacío ni tampoco on `ab`.

- Una expresión regular *r0* seguida de otra *r1* describe los strings que tienen un prefijo descrito por *r0* seguido de otro descrito por *r1*. Por ejemplo `ab` describe el string `ab` puesto que `a` puede ser `a` y `b` puede ser `b`. Igualmente, `.b` describe `ab` `bb` pero no `ba`.
- Si *r0* y *r1* son dos expresiones regulares, La expresión *r0*`|`*r1* describe los strings que describe alguna de las expresiones *r0* y *r1* (o que describen las dos).
- Si tenemos una expresión *r*, entonces *r*`*` describe los strings "" (la cadena vacía), los que describe *r*, los de *rr*, los de *rrr*, etc. (Repetimos la expresión cualquier número de veces, posiblemente ninguna).
- Si tenemos una expresión *r*, entonces *r*`+` es lo mismo que *r*`(`*r*`)*`. Esto es, *r* una o más veces.
- Si tenemos una expresión *r*, entonces *r*`?` es lo mismo que `(`*r*`)|()`. Esto es, *r* una o ninguna vez.
- `^` representa el principio del string en que buscamos encajes de la expresión regular. Por ejemplo, `^a` encaja con `ab` pero no con `ba`.
- `$` representa el final del string en que buscamos encajes de la expresión regular. Por ejemplo, `a$` encaja con `ba` pero no con `ab`.
- `\`*c* quita el significado especial a *c*, de tal modo que podemos utilizar caracteres que forman parte de la sintaxis de expresiones regulares como caracteres normales. Por ejemplo, `\\` encaja con `\` y `\.c` encaja con `.c` pero no con `ac`. En cambio `.c` encaja también con `ac`.
- `(`*r*`)` permite agrupar una expresión y describe los strings descritos por *r*. Por ejemplo, `(a|b)(x|y)` describe `ax` pero no `ab`.
- `[...]` describe cualquiera de los caracteres entre los corchetes. Y es posible escribir rangos como en `[a-c]` (de la `a` a la `c`). Por ejemplo, `[a-zA-Z0-9_]` es cualquier letra minúscula o mayúscula o dígito o bien "_". ¡Pero cuidado aquí con caracteres como "ñ"!
- `[^...]` describe cualquiera de los caracteres no descritos por `[...]`. Por ejemplo, `[0-9]` es cualquier carácter que no sea un dígito.

Veamos algunos ejemplos utilizando `seq` para usar `egrep` en su salida. Primero, buscamos un `1` seguido de un carácter:

```
unix$ seq 15 | egrep 1.
10
11
12
13
14
15
unix$
```

Ahora un carácter seguido de un `1`:

```
unix$ seq 15 | egrep .1
11
unix$
```

O bien `11` o bien `12`:

```
unix$ seq 15 | egrep '11|12'
11
12
unix$
```

Y tambien podemos combinar expresiones más complejas del mismo modo:

```
unix$ seq 15 | egrep '.2|1.'
10
11
12
13
14
15
unix$
```

Un `1` y cualquier cosa:

```
unix$ seq 15 | egrep '1.*'
1
10
11
12
13
14
15
unix$
```

¡Ojo al `.*`! Si usáramos `1*` entonces veríamos todas las líneas dado que todas contienen el string vacío y que `1*` encaja con el string vacío. Pero podríamos pedir las líneas que son cualquier número de unos:

```
unix$ seq 15 | egrep '^1*$'
1
11
unix$
```

Un `1` o el principio del texto y luego un `2` o un `3`:

```
unix$ seq 15 | egrep '(1|^)(2|3)'
2
3
12
13
unix$
```

Aunque tal vez sería mejor

```
unix$ seq 15 | egrep '(1|^)[23]'
2
3
12
13
unix$
```

para conseguir el mismo efecto.

Ahora cualquier línea que use sólo cualquier carácter menos los del `2` al `8`:

```
unix$ seq 15 | egrep '^[^2-8]*$'
1
9
10
11
```

Líneas que tengan `3` una o más veces:

```
unix$ seq 15 | egrep '3+'
3
13
unix$
```

Quizá un `1` y un `3`:

```
unix$ seq 15 | egrep '1?3'
3
13
unix$
```

Bueno, ya conocemos las expresiones regulares y podemos utilizar `egrep` para obtener las líneas de nuestro fichero de configuración que se refieren al shell:

```
unix$ egrep '^shell ' config
shell /bin/sh
unix$
```

Y también las que se refieren a directorios:

```
unix$
unix$ egrep '^[a-z]+dir ' config
libdir /usr/lib
bindir /usr/bin
logdir /log
unix$
```

Nótese que utilizar aquí "`dir`" como expresión habría sido seguramente un error. Líneas que contengan algo como "/opt/bindir/ksh" habrían salido y no tienen por qué ser las que buscamos en este caso.

4. Líneas y campos

Tenemos toda una plétora de comandos que saben trabajar con ficheros suponiendo que tienen líneas y que cada una tiene una serie de campos. Muchos de los comandos que trabajan con líneas no requieren siquiera que tengan campos o cualquier otra estructura predeterminada. Vamos a continuar nuestro script de ejemplo centrándonos ahora en este tipo de tarea: procesar líneas y campos.

Teníamos ya, para nuestro fichero

```
shell /bin/sh
libdir /usr/lib
bindir /usr/bin
logdir /log
```

un comando capaz de escribir las líneas que corresponden a directorios:

```
unix$
unix$ egrep '^[a-z]+dir ' config
libdir /usr/lib
bindir /usr/bin
logdir /log
unix$
```

Ahora nos gustaría tener la lista de directorios mencionados en dichas líneas. Lo primero que tenemos que hacer es pensar que la entrada que tenemos es un fichero con líneas que tienen dos campos separados por

blanco. En tal caso basta con utilizar un programa capaz de escribir el segundo campo. Hay muchas formas de hacer esto. Una es utilizar *cut(1)*. El flag `-f` de `cut` admite como argumento el número de campo que se desea, comenzando a contar desde `1`.¡Pero atención!

```
unix$ egrep '^[a-z]+dir ' config | cut -f2
libdir /usr/lib
bindir /usr/bin
logdir /log
unix$
```

Por omisión `cut` utiliza el tabulador como carácter separador de campos y, en nuestro caso, teníamos un espacio en blanco y no un tabulador separando los campos. En realidad queremos

```
unix$ egrep '^[a-z]+dir ' config | cut '-d ' -f2
/usr/lib
/usr/bin
/log
unix$
```

que utiliza la opción `-d` de `cut` para indicarle que el carácter escrito tras la opción es el que deseamos utilizar como separador. ¡Ya tenemos una lista de directorios!

No obstante, `cut` no es la mejor opción en la mayoría de los casos. Pensemos en obtener la lista de dueños de los ficheros del directorio actual... Sabiendo que podemos usar un listado largo en `ls` como en

```
unix$ ls -l config
-rw-r--r--  1 nemo  staff  58 Aug 28 11:27 config
```

podríamos intentar usar algo como

```
unix$ ls -l config | cut -f3
-rw-r--r--  1 nemo  staff  58 Aug 28 11:27 config
```

¡Pero `cut` escribiría todos los campos! El problema de nuevo es el separador entre un campo y otro.

El programa que menos problemas suele dar para seleccionar campos es *awk(1)*. Es un lenguaje de programación para manipular ficheros con aspecto de tabla, pero tiene muchos programas de una sóla línea que resultan útiles. En este caso,

```
unix$ ls -l config | awk '{print $3}'
nemo
```

consigue el efecto que buscamos. El programa significa "*en todas las líneas, imprime el tercer campo*".

Lo bueno de `awk` es que utiliza una expresión regular como separador de campos. Y puedes cambiarla escribiendo la que quieras como argumento de la opción `-F`:

```
awk '-F[: ]+' '{print $3}'
```

imprime el tercer campo suponiendo que un campo está separado de otro por uno o mas caracteres que sean bien un espacio en blanco o bien dos puntos.

Volviendo a nuestro problema, una vez tenemos los directorios mencionados en nuestro fichero de configuración, estaría bien tenerlos ordenados y sin tener duplicados en la lista. Para conseguirlo podemos utilizar el comando *sort(1)* que sabe ordenar líneas utilizando campos como clave (o la línea entera). Otro comando relacionado es `uniq(1)`, que elimina duplicados de una entrada ya ordenada. Así pues,

```
unix$ egrep '^[a-z]+dir ' config | awk '{print $1}' | sort | uniq
bindir
libdir
logdir
unix$
```

escribe los directorios ordenados y sin duplicados.

Esto resulta útil también al recolectar nombres o valores que aparecen en cualquier otro sitio. Por ejemplo, para obtener la lista de dueños de ficheros en el directorio actual:

```
unix$ ls -l * | awk '{print $3}' | sort -u
nemo
unix$
```

El flag `-u` de `sort` hace lo mismo que `uniq`. ¡Ya tenemos algo que podríamos incluso guardar en una variable de entorno!

```
unix$ owners=‘ls -l * | awk '{print $3}' | sort -u‘
unix$ echo $owners
nemo
unix$
```

De hecho, vamos a hacer justo con nuestros directorios, y ya podemos ver si existen o no...

```
unix$ dirs=‘egrep '^[a-z]+dir ' config | awk '{print $1}' | sort | uniq‘
unix$ for d in $dirs ; do
>    if test ! -d $d ; then
>        echo no dir $d
>    fi
>    done
unix$
```

¡Bien, todos los directorios existen!, ¡Problema resuelto!

Recuerda que es el shell el que escribe los "`>`" para indicarnos que es preciso escribir más líneas para completar el comando. Hemos usado el bucle `for` que vimos anteriormente y que ejecuta los comandos entre `do` y `done` para cada palabra que sigue a `in`. En cada iteración, la variable cuyo nombre precede a `in` toma como valor cada una de las palabras que siguen a `in`.

También hemos utilizado el comando `if` que ejecuta en este caso

```
test ! -d $d
```

como condición y, de ser cierta, ejecuta los comandos en el `then`. Aunque no lo hemos necesitado, es posible escribir

```
if ....
then
    ...
else
    ...
fi
```

como comando.

El comando `sort` merece que le dediquemos algo más de tiempo. Sabe ordenar la entrada asumiendo que son cadenas de texto o bien ordenarla segun su valor numérico (y de algunas otras formas). Por ejemplo,

```
unix$ seq 10 | sort
1
10
2
3
4
5
6
7
8
9
```

no produce el efecto que podrías esperar. Ahora bien...

```
unix$ seq 10 | sort -n
1
2
3
4
5
6
7
8
9
10
```

con la opción `-n`, `sort` ordena numéricamente.

Sea numérica o alfabéticamente, podemos pedir una ordenación en orden inverso:

```
unix$ seq 5 | sort -nr
5
4
3
2
1
unix$
```

Aquí utilizamos ambos flags para indicar que se desea una ordenación numérica y además en orden inverso.

Además podemos pedir a `sort` que utilice como clave para ordenar sólo alguno de los campos, por ejemplo

```
sort -k3,5
```

ordena utilizando los campos del 3 al 5 (contando desde uno). Consulta su página de manual en lugar de recordar todo esto.

¿Recuerdas el comando *du(1)*? !Podemos ver dónde estamos gastando el espacio en disco!

```
unix$ du -s * | sort -nr | sed 5q
23824    zx-spe
6088    wr_pic1.eps
6088    inkdump.eps
6088    clive_pic3.eps
5768    zx
```

Primero, hacemos que `du` liste el total (opción `-s`, *summary*) para todos los ficheros y directorios. Ordenamos entonces su salida numéricamente en orden inverso y nos quedamos con las 5 primeras líneas. Son los

5 ficheros o directorios en que estamos usando más disco. Ahora podemos pensar qué hacemos con ellos si necesitamos espacio...

¿Y si queremos los que menos espacio usan? Podríamos quedarnos con las últimas 5 líneas, por ejemplo:

```
unix$ du -s * | sort -nr | tail -5
```

utiliza *tail(1)* que escribe las últimas *n* líneas de un fichero. Aunque habría sido mejor no invertir la ordenación en sort:

```
unix$ du -s * | sort -n | sed 5q
```

El comando `tail` tiene otro uso para imprimir las últimas líneas pero empezando a contar desde el principio. Esto es útil, por ejemplo, para eliminar las primeras líneas de un fichero:

```
unix$ seq 5 | tail +3
3
4
5
unix$
```

También es posible seleccionar determinados campos y/o reordenarlos. Por ejemplo, con la opción `-l`, `ps` produce un listado largo...

```
unix$ man ps
...
     -l      Display information associated with the following keywords:
             uid, pid, ppid, flags, cpu, pri, nice, vsz=SZ, rss, wchan,
             state=S, paddr=ADDR, tty, time, and command=CMD.
...
```

Para cada item, `ps` produce una columna separada de las demás por espacio en blanco. Así pues podemos utilizar `awk` para imprimir el pid, tamaño de la memoria física (*resident set size*, o rss) y nombre de los procesos utilizando

```
unix$ ps -l | awk '{printf("%s\t%s\t%s\n", $2, $15, $9);}'
PID    CMD    RSS
448    -bash    372
519    acme    28
526    (fontsrv)    0
527    acme    104
528    acme    5576
534    9pserve    24
...
```

Como puedes ver, `awk` dispone de `printf`. Dicha función se utiliza prácticamente como la de C. En nuestro caso optamos por imprimir las columnas 2, 15, y 9 en ese orden. Utilizar `cut` sería más complicado dado que hay múltiples blancos entre un campo y el siguiente.

¿Qué proceso está utilizando más memoria? Bastar ordenar numéricamente por el tercer campo, de mayor a menor y quedarse con la primera línea.

```
unix$ ps -l | awk '{printf("%s\t%s\t%s\n", $2, $15, $9);}' | \
>    sort -nr -k3 | sed 1q
91136    acme    52920
unix$
```

Parece que `acme`, con el pid 91136.

5. Funciones y otras estructuras de control

Aún tenemos pendiente escribir nuestro script para procesar los ficheros de configuración de nuestra aplicación y ver si todos son correctos.

Lo primero que deberíamos hacer es pensar en qué opciones y argumentos queremos admitir en nuestro script. Por ejemplo, un posible uso podría ser

```
chkappconf [-sd] [fich...]
```

Hemos utilizado la sintáxis que vemos en la synopsis de las páginas de manual:

- Los flags `-d` y `-s` pueden utilizarse opcionalmente y permiten comprobar sólo los directorios utilizados en los ficheros de configuración o sólo los shells mencionados. Si no indicamos ninguno de estos flags queremos que se comprueben ambas cosas.
- Si se indican nombres de fichero (uno o más) como argumento entonces haremos que se comprueben sólo dichos ficheros, en caso contrario comprobaremos todos los ficheros en los directorios `$HOME` de todos los usuarios.

Sabemos que `$*` corresponde al vector de argumentos en un script, excluyendo el nombre del script (`argv[0]`, que sería `$0`). Una posibilidad es procesar dicha lista de argumentos mientras veamos que comienzan por "`-`", y luego veremos cómo procesar las opciones que indican.

Podemos empezar con algo como esto, que guardaremos en nuestra primera versión de `chkappconf`:

```
#!/bin/sh

while test $# -gt 0
do
    echo $1
    shift
done
```

Aquí utilizamos la estructura de control `while` del shell, que ejecuta el comando indicado como condición y, mientras dicho comando termine con éxito, ejecuta las sentencias entr `do` y `done`. Como condición,

```
test $# -gt 0
```

compara el número de argumentos con `0` usando *greater than* como comparación. Por último, dentro del cuerpo del `while` escribimos el primer argumento y posteriormente lo tiramos a la basura. La primitiva `shift` tira el primer argumento y conserva el resto.

Vamos a probarlo:

```
unix$ chkappconf -a b c
-a
b
c
unix$
```

Ahora estaría bien detener el while si el primer argumento no compienza por un "`-`". Por el momento podríamos usar `egrep` para eso. Hagamos un par de pruebas:

```
unix$ echo -a | egrep '^-'
-a
unix$ echo a | egrep '^-'
unix$
```

¡Estupendo! Pero sólo nos interesa ver si `egrep` dice que la entrada tiene ese aspecto o no. En lugar de enviar su salida a `/dev/null`, usaremos la opción `-q` (*quiet*) de `egrep` que hace que no imprima nada y tan sólo llame a *exit(2)* con el estatus adecuado.

```
unix$ echo -a | egrep -q '^-'
unix$ echo $?
0
unix$ echo a | egrep -q '^-'
unix$ echo $?
1
```

Ya casi estamos. Ahora necesitamos poder escribir como condición que tanto `test` compruebe que hay algún argumento como el comando anterior vea que comienza por un "`-`". Pero esto es fácil. El shell dispone de "`&&`" y "`||`" como operadores. Significan lo mismo que al evaluar condiciones en C, pero naturalmente esto es shell y utilizamos líneas de comandos. Nuestro script queda así por el momento:

```
#!/bin/sh

while test $# -gt 0 && echo $1 | egrep -q '^-'
do
    echo option arg: $1
    shift
done
echo argv is $*
```

Y si lo ejecutamos, vemos que funciona:

```
unix$ chkappconf -a -ab c d
option arg: -a
option arg: -ab
argv is c d
unix$
```

El siguiente paso es procesar cada uno de los argumentos correspondientes a opciones. Aunque podríamos recurrir a `egrep` de nuevo, vamos a utilizar la estructura de control `case` del shell, que sabe como comprobar si un string encaja con una o más expresiones. Veamos antes un ejemplo:

```
unix$ x=ab
unix$ case $x in
> a*)
> echo a;;
> b*)
> echo b;;
> esac
a
unix$
```

Esta estructura compara el string entre `case` e `in` con cada una de las expresiones de cada rama del `case`. Estas expresiones son similares a las utilizadas para globbing y terminan en un "`)`". Tras cada expresión se incluyen las líneas de comandos que hay que ejecutar en dicho caso, terminando con un "`;;`". Por último,

se cierra el `case` con "`esac`".

Podemos escribir expresiones como

```
case ... in
[ab]c|d)
    ...
    ;;
esac
```

Esto es, es posible escribir conjuntos de caracteres "`[...]`" y también indicar expresiones alternativas separadas por un "`|`". Dado que `case` intentar encajar las expresiones en el orden en que se dan, y que "`*`" encaja con cualquier cosa, no existe `else` o `default` para `case`. Basta usar

```
case ... in
...
*)
    ... este es el default ...
    ;;
esac
```

Volvamos a nuestro script. Vamos a definir un par de variables para los flags y a comprobar si aparecen como opciones.

```
#!/bin/sh

dflag=y
sflag=y
while test $# -gt 0 && echo $1 | egrep -q '^-'
do
    case $1 in
    *d*)
        dflag=y
        sflag=n
        ;;
    esac
    case $1 in
    *s*)
        sflag=y
        dflag=n
        ;;
    esac
    shift
done
echo dflag $dflag
echo sflag $sflag
echo argv is $*
```

Habitualmente habríamos usando "`n`" como valor inicial para los flags y los habríamos puesto a "`y`" si aparecen. Pero en este caso hay que procesar tanto directorios como shells si ninguno de los flags aparece. ¡Probémoslo!

```
unix$ chkappconf -d c d
dflag y
sflag n
argv is c d
unix$
```

Ya disponemos de `$dflag` para ver si hay que procesar sólo directorios y de `$sflag` para ver si hay que procesar sólo shells. Además, hemos dejado `$*` con el resto de argumentos.

Pero... ¿Y si hay una opción inválida? Bueno, siempre podemos recurrir a `echo` y `egrep` para ver si las opciones tienen buen aspecto:

```
...
while test $# -gt 0 && echo $1 | egrep -q '^-'
do
    if echo $1 | egrep -q '[^-sd]' ; then
        echo usage: $0 '[-sd] [file...]' 1>&2
        exit 1
    fi
    case $1 in
    ...
done
...
```

Ahora podemos ponernos a hacer el trabajo según las opciones y los argumentos. Lo primero que haremos es dejar en una variable la lista de ficheros que hay que procesar.

```
files=$*
if test $# -eq 0 ; then
    files="/home/*/.myapp"
fi
```

Suponiendo que los ficheros de configuración se llaman `.myapp` y que queremos procesar dichos ficheros para todos los usuarios si no se indica ningún fichero.

Ahora podríamos procesar un fichero tras otro...

```
for f in $files ; do
    echo checking $f...
    if test $dflag = y ; then
        checkdirs $f
    fi
    if test $sflag = y ; then
        checkshells $f
    fi
done
```

Aquí vamos a utilizar `checkdirs` y `checkshells` para comprobar cada fichero. Aunque podríamos hacer otros scripts, parece que lo más adecuado es definir funciones. Y sí, ¡el shell permite definir funciones!.

Para definir una función escribimos algo como

```
myfun() {
    ...
}
```

En este caso, `myfun` es el nombre de la función. Tras el nombre han de ir los paréntesis (¡Sin nada entre ellos!) y después los comandos que constituyen el cuerpo entre llaves. Recuerda que estás usando el shell, esto no es C.

Dentro de la función, los argumentos se procesan como en un script. Así pues, en nuestro caso "`$1`" es el nombre del fichero que hay que comprobar y las funciones podrían ser algo como

```
checkdirs()
{
    echo checking dirs in $1
}
checkshells()
{
    echo checking shells in $1
}
```

Hemos terminado. Mostramos ahora el script entero para evitar confusiones.

```
#!/bin/sh

checkdirs()
{
    file=$1
    dirs=‘egrep ’^[a-z]+dir ’ config | \
        awk ’{print $1}’ | sort | uniq‘
    for d in $dirs ; do
        if test ! -d $d ; then
            echo no dir $d in $file
        fi
    done
}


checkshells()
{
    file=$1
    shells=‘egrep ’^shell ’ config | \
        awk ’{print $1}’ | sort | uniq‘
    for s in $shells ; do
        if test ! -x $x ; then
            echo no shell $s in $file
        fi
    done
}
```

```
dflag=y
sflag=y
while test $# -gt 0 && echo $1 | egrep -q '^-'
do
    if echo $1 | egrep -q '[^-sd]' ; then
        echo usage: $0 '[-sd] [file...]' 1>&2
        exit 1
    fi
    case $1 in
    *d*)
        dflag=y
        sflag=n
        ;;
    esac
    case $1 in
    *s*)
        sflag=y
        dflag=n
        ;;
    esac
    shift
done


files=$*
if test $# -eq 0 ; then
    files="a b c"
fi
for f in $files ; do
    if test $dflag = y ; then
        checkdirs $f
    fi
    if test $sflag = y ; then
        checkshells $f
    fi
done
```

Hay formas mejores de hacer lo que hemos hecho. Pero lo que importa es que hemos podido hacerlo con las herramientas que tenemos. Por ejemplo, la práctica totalidad de los shells modernos incluyen *getopts(1)* para ayudar a procesar opciones y, además, tienes *getopt(1)* en cualquier caso para la misma tarea. Sólo con eso, ya podemos simplificar en gran medida nuestro script.

6. Editando streams

Ya hemos utilizado *sed(1)* para escribir las primeras líneas de un fichero. Pero puede hacer mucho más. Se trata de un editor similar a *ed(1)* (que fue el editor estándar en UNIX durante mucho mucho, pero hace ya mucho). Sus comandos están también disponibles (en general) en *vi(1)*, un editor "*visual*" que puede utilizarse en terminales de texto que no sean gráficos. La diferencia radica en que `sed` está hecho para procesar su entrada estándar, principalmente. De ahí el nombre *stream ed*.

`Sed` trabaja procesando su entrada línea a línea. En general, para cada línea, aplica los comandos de edición que se indican y después se escribe la línea en la salida.

Un comando en `sed` es una o dos *direcciones* seguidas de una función y, quizá, argumentos para la

función. Las direcciones determinan en qué líneas son aplicables los comandos.

Por ejemplo,

```
sed 5q
```

Significa "*en la línea 5*" ejecuta "*quit*". Dado que no se han ejecutado comandos que alteren las líneas, `sed` las escribe tal cual hasta llegar a la línea 5, momento en que termina. Por ello este comando escribe las 5 primeras líneas. La primera línea es la `1` para `sed`.

Sabiendo esto, podemos utilizar `sed` para escribir un rango de líneas. Por ejemplo

```
unix$ seq 15 | sed -n 5,7p
5
6
7
unix$
```

Aquí el flag `-n` hace que `sed` no imprima las líneas tras editarlas y utilizamos el comando `p` para imprimir las líneas entre la `5` y la `7`. Como puedes ver, dos direcciones separadas por una "`,`" delimitan un rango de líneas.

Veamos otro ejemplo, sabemos que `ps` escribe una primera línea con la cabecera que indica qué es cada campo. Por ejemplo:

```
unix$ ps
  PID TTY           TIME CMD
  448 ttys000    0:00.01 -bash
  519 ttys000    0:00.01 acme
...
```

Podemos utilizar

```
unix$ ps | tail +1
  448 ttys000    0:00.01 -bash
  519 ttys000    0:00.01 acme
...
```

como ya vimos o bien

```
unix$ ps | sed -n '2,$p'
  448 ttys000    0:00.01 -bash
  519 ttys000    0:00.01 acme
...
```

Como puedes ver, "`$`" equivale al número de la última línea cuando se utiliza como dirección en `sed`.

De un modo similar podemos eliminar un rango de lineas con el comando `d`:

```
unix$ seq 15 | sed 2,12d
1
13
14
15
unix$
```

¡O varios rangos! El flag `-e` permite utilizar como argumento un comando de edición, y podemos repetirlo el número de veces que haga falta. Nos permite aplicar más de un comando en el mismo `sed`.

```
unix$ seq 15 | sed -e 1,3d -e 5,12d
4
13
14
15
unix$
```

Pero recuerda... ¡`sed` trabaja línea a línea sobre un stream!

```
unix$ seq 15 | sed -e 2,12d -e 1,3d
unix$
```

¡Aquí la hemos liado parda! Este comando no tiene mucho sentido dado que hay varios comandos que se aplican a líneas del mismo rango.

Las direcciones pueden ser expresiones regulares escritas entre "`/`". Esto nos permite seleccionar líneas entre ciertas líneas de interés. Por ejemplo, suponiendo que tenemos un fichero fuente en C con la función `run`, este comando escribe la cabecera y el cuerpo de dicha función:

```
unix$ sed -n  '/^run/,/^}/p <run2.c
run(char *cmd, char *argv[])
{
    ...
}
unix$
```

Para entender por qué no se escriben todas las líneas hasta la última que contenga un "`}`" al principio, piensa cómo evalúa `sed` las direcciones:

- Primero esperará hasta tener una línea que encaja con "`^run`"
- Una vez encontrada, seguirá hasta una línea que encaja con "`^}`".

En cada una de esas líneas ejecutará "`p`" (que imprime la línea) y una vez alcanza la línea de la segunda dirección, el comando termina por lo que no se vuelve a ejecutar para las líneas que siguen.

Otro comando de los más utilizados es el de sustituir una expresión por otra. Por ejemplo, este comando aumenta la tabulación del texto a que se aplica:

```
sed 's/^/ /'
```

Lo que hace es reemplazar el principio de línea por un tabulador (que hemos escrito como " ".

Supongamos que queremos renombrar todos los ficheros "`.c`" a ficheros "`.C`" por alguna extraña razón. Podemos utilizar `sed` para ello:

```
unix$ echo foo.c | sed 's/\.c$/.C/'
foo.C
```

Así pues

```
unix$ for f in *.c ; do
>    nf=‘echo $f | sed 's/\.c$/.C/’‘
>    mv $f $nf
>    done
unix$
```

hace el trabajo. Cuidado aquí. El comando de `sed` está entre comillas simples, pero todo lo que hay tras el signo "`=`" está entre comillas invertidas. ¡No son iguales!

Por ejemplo, podemos tener ficheros llamados `ch1.w`, `ch2.w`, etc. y llegar al `ch10.w`. En ese momento

quizá queramos renombrar `ch1.w` para que sea `ch01.w`, y así hasta el `9`. ¿Por qué? Bueno, de ese modo `ls` los lista en el orden adecuado y otros programas los verán en el orden en que deberían estar. De otro modo `ch11.w` estará listado antes que `ch2.w`.

Una vez más `sed` nos puede ayudar. Para probarlo, vamos a crear esos ficheros, y vamos a utilizar `seq` para crearlos.

```
unix$ for n in ‘seq 12‘; do touch ch$n.w ; done
unix$ ls
ch1.w    ch11.w    ch2.w    ch4.w    ch6.w    ch8.w
ch10.w    ch12.w    ch3.w    ch5.w    ch7.w    ch9.w
unix$
```

Y ahora podemos probar...

```
unix$ for ch in ch[0-9].w ; do
>    echo mv $ch ‘echo $ch | sed ’s/\([0-9]\)/0\1/’ ‘
>    done
mv ch1.w ch01.w
mv ch2.w ch02.w
mv ch3.w ch03.w
mv ch4.w ch04.w
mv ch5.w ch05.w
mv ch6.w ch06.w
mv ch7.w ch07.w
mv ch8.w ch08.w
mv ch9.w ch09.w
unix$
```

En lugar de ejecutar `mv` directamente, lo hemos precedido de `echo` para ver lo que va a ejecutar. Si nos convence el resultado podemos añadir "`|sh`" para ejecutarlo o quitar el `echo`.

Pero la parte interesante es "`s/\([0-9]\)/0\1/`". Esto quiere decir que hay que sustituir el texto que encaja con "`[0-9]`" en cada línea por "`0\1`". El "`\1`" significa *el texto que encaja en la primera expresión entre* "`\(...\)`", que en este caso es "`[0-9]`". Así pues, "`\1`" es capaz de recuperar parte del texto que ha encajado en la expresión para utilizarlo al reemplazar. Vuelve a leer la sesión anterior y notarás el efecto de "`\1`".

Observa que esta facilidad te permite reordenar el texto que hay en la línea. Si contamos de 10 a 15 de 1 en 1 usando `seq`

```
unix$ seq 10 1 15
10
11
12
13
14
15
```

podemos ver el efecto de estas expresiones en otro ejemplo:

```
unix$ seq 10 1 15 | sed 's/\([0-9]\)\([0-9]\)/\2\1 and not \1\2/'
01 and not 10
11 and not 11
21 and not 12
31 and not 13
41 and not 14
51 and not 15
unix$
```

Ahora tenemos dos subexpresiones entre "`\(...\)`", y reemplazamos toda la expresión por texto consistente en los dos caracteres con el orden cambiado, luego "` and not `" y luego los dos caracteres en el orden original. Ni que decir tiene que puedes utilizar cualquier subexpresión y no una que opere sobre un sólo carácter.

Para borrar una expresión, podemos reemplazarla por *nada*. Por ejemplo, este comando elimina un nivel de tabulación de su entrada:

```
sed 's/^ //'
```

Debes tener cuidado con las sustituciones en `sed`. Se aplican al primer trozo de la línea que encaja con la expresión, pero no se repiten más adelante en la misma línea. Esto es, puedes reemplazar texto una sóla vez. Si deseas reemplazar todas las veces que sea posible puedes añadir un flag "`g`" al comando. Por ejemplo, dados los ficheros `chXX.w`", este comando nos imprime los números para esos ficheros:

```
unix$ ls
ch1.w    ch11.w    ch2.w    ch4.w    ch6.w    ch8.w
ch10.w    ch12.w    ch3.w    ch5.w    ch7.w    ch9.w
unix$ ls | sed 's/[.a-z]//g'
    1
    10
    11
    12
    2
    ...
    9
unix$
```

Compara con

```
unix$ ls | sed 's/[.a-z]//'
h1.w
h10.w
...
unix$
```

7. Trabajando con tablas

En gran cantidad de casos tendrás datos tabulados como en una hoja de cálculo. Gran número de ficheros de configuración en UNIX presentan este aspecto, y además la salida de muchos comandos está tabulada.

La mayoría de las veces puedes operar filtrando filas y columnas y trabajar con los comandos que hemos visto durante el curso. Para cálculos numéricos puedes utilizar cualquier de las calculadores de línea de comandos, aunque tal vez sea más simple utilizar *expr(1)* para evaluar expresiones numéricas simples, como en:

```
unix$ n=4
unix$ n=‘expr $n ’*’ 2‘
unix$ echo $n
8
unix$
```

No necesitas mucho más.

No obstante, dispones de *awk(1)* que es un lenguaje pensado para operar sobre datos tabulados. Ya lo hemos utilizado para seleccionar campos, pero ahora vamos a ver algunos otros usos típicos y como son los programas de `awk` en general.

La idea es similar a *sed(1)*. Para `awk` los ficheros constan de registros (líneas) que tienen campos (separados por blancos, aunque ya sabes como cambiar esto). Lo que hace `awk` es leer la entrada (o el fichero indicado como argumento) línea a línea y aplicar el programa sobre el mismo.

Un programa en `awk` se suele escribir directamente en el primer argumento en la llamada a `awk`, pero ya sabes que puedes utilizar "`#!`" en scripts para lo que gustes. El programa consta de **reglas**, que tienen una expresión (opcional) que selecciona a qué lineas se aplica la regla y una acción entre llaves (opcional) que indica qué debe hacer `awk` en esas líneas. Si no hay expresión se considera que la acción se aplica a todas las líneas. Si no hay acción se considera que la acción es imprimir.

En las expresiones y en las acciones tienes predefinidas las variables "`$1`" (el primer campo), "`$2`" (el segundo), etc. El registro entero, la línea, es "`$0`". Y además tienes definidas las variables `NR` *number of record* (número de línea) y `NF` *number of field* (número de campos).

Para jugar, vamos a utilizar el fichero `/etc/passwd` que define las cuentas de usuario. Tiene este aspecto:

```
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
...
```

El nombre de usuario, seguido de su uid, gid, nombre real, home y por último el shell que utiliza. Todos los campos separados por "`:`".

Vamos a imprimir tan sólo el nombre de usuario y el shell que utilizan:

```
unix$ awk -F: ’{print $1, $NF}’ /etc/passwd
nobody /usr/bin/false
root /bin/sh
daemon /usr/bin/false
...
```

En lugar de print, podemos utilizar `printf`. Las acciones de `awk` utilizan sintáxis casi como la de C y las variables no es preciso declararlas. Sabiendo esto puedes ser optimista y hacer intentos, la página de manual te sacará de dudas en cualquier caso.

```
unix$ awk -F: ’{printf("user %s shell %s\n", $1, $NF);}’ /etc/passwd
user nobody shell /usr/bin/false
user root shell /bin/sh
user daemon shell /usr/bin/false
...
```

¡Ahora sólo para *root*!

```
unix$ awk -F: '
>    $1 == "root" {
>        printf("user %s shell %s\n", $1, $NF);
>    }' /etc/passwd
user root shell /bin/sh
unix$
```

En este caso la expresión `$1 == "root"` hace que la acción ejecute sólo si el primer campo es el string `root`.

Pero compara con este otro comando:

```
unix$ awk -F: '
>    $1 ~ /root/ {
>        printf("user %s shell %s\n", $1, $NF);
>    }' /etc/passwd
user root shell /bin/sh
user _cvmsroot shell /usr/bin/false
unix$
```

El operador "`~`" permite ver si cierto texto encaja con una expresión regular (escrita entre "`/`"). Por eso aparecen dos cuentas, porque el nombre de usuario contiene `root` en ambas.

Igual que en el caso de `sed`, podemos utilizar como expresión un par de expresiones regulares separadas por una coma, lo que hace que la expresión sea cierta para las líneas comprendidas entre una que encaja con la primera y otra que encaja con la segunda.

```
unix$ seq 15 | awk '/3/,/5/ {print}'
3
4
5
13
14
15
unix$
```

Verás que aparecen las líneas entre la 3 y la 5 (que encajan con las expresiones) y las líneas entre la 13 y la 15 (que también lo hacen). Las expresiones pueden ser mas elaboradas y puedes utilizar los operadores "`&&`" y "`||`" casi como en C.

La acción `next` salta el registro en curso. Considerando que `awk` procesa las reglas en el orden en que las escribimos, podemos hacer que imprima un rango de líneas pero salte algunas. Por ejemplo,

```
unix$ seq 15 | awk '
>    $0 == "4" {next}
>    /3/,/5/ {print} '
3
5
13
14
15
unix$
```

Muy útil para utilizar

```
$0 ~ /^#/ {next}
```

al principio del programa de `awk` e ignorar las líneas que comienzan por un "#" (que suele ser el carácter

de comentario en el shell y en muchos ficheros).

Las expresiones `BEGIN` y `END` hacen que su acción ejecute antes de procesar la entrada y después de haberla procesado. Podemos utilizar esto para sumar los números de la entrada, por ejemplo:

```
unix$ seq 15 | awk ’
>    {tot += $0}
>    END {print tot}’
120
unix$
```

O para obtener la media:

```
unix$ seq 15 | awk ’
>    {tot += $0}
>    END {print tot/NR}’
8
unix$
```

Este otro obtiene la media de los números pares, y además los imprime:

```
unix$ seq 15 | awk ’
>    $0 % 2 == 0 {
>        print;
>        tot += $0;
>    }
>    END {print tot/NR}’
2
4
6
8
10
12
14
3.73333
```

Pero podríamos haberlo hecho así:

```
unix$ seq 15 | awk ’
>    { if ($0 %2 == 0) {
>        print;
>        tot += $0;
>      }
>    }’
2
4
...
```

Simplemente recuerda que las acciones disponen de sintaxis similar a la de C y se optimista. Consulta *awk(1)* para ver qué funciones tienes disponibles y que expresiones y estructuras de control. Si has entendido lo que hemos estado haciendo, seguro que te sobra con la página de manual.

¿Puedes escribir un comando que numere las líneas de un fichero? Para imprimirlo con números de línea, por ejemplo.