Prácticas de Fundamentos del Software

Módulo I. Órdenes UNIX y Shell Bash

Sesión Nº5: Expresiones con variables y expresiones regulares

31-Oct-2011

1 Objetivos principales

- Distinguir entre operadores aritméticos y relacionales para definir expresiones con variables.
- Conocer operadores de consulta de archivos y algunas órdenes para utilizarlos.
- Conocer el concepto de expresión regular y operadores para expresiones regulares.
- Saber utilizar distintos tipos de operadores con las órdenes **`find`** y **`grep`**.

Además, en esta sesión se verán las siguientes órdenes:

Órdenes Shell Bash					
`$(( … ))`	**`$[ … ]`**	**`bc`**	**`let`**	**`test`**	**`if/else`**

Tabla 1. Órdenes de la sesión.

2 Expresiones con variables

Como se vio en la sesión anterior, las variables son muy útiles tanto para adaptar el entorno de trabajo al usuario como en la construcción de guiones (o scripts). Pero en muchas ocasiones esas variables no se tratan de manera independiente, sino que se relacionan unas con otras, como puede ser mediante la orden **`expr`** que también se vio en esa sesión, o mediante expresiones aritméticas.

El shell bash ofrece dos posibles sintaxis para manejar expresiones aritméticas haciendo uso de lo que se denomina *expansión aritmética*, o *sustitución aritmética*, que evalúa una expresión aritmética y sustituye el resultado de la expresión en el lugar donde se utiliza. Ambas posibilidades son:

`$(( … ))`

`$[ … ]`

En estos casos, lo que se ponga en lugar de los puntos suspensivos se interpretará como una expresión aritmética, no siendo necesario dejar huecos en blanco entre los paréntesis más internos y la expresión contenida en ellos, ni entre los corchetes y la expresión que contengan. Además, hay que tener en cuenta que las variables que se usen en una expresión aritmética no necesitan ir precedidas del símbolo **`$`** para ser sustituidas por su valor, aunque si lo llevan no será causa de error, y que cualquier expresión aritmética puede contener otras expresiones aritméticas, es decir, las expresiones aritméticas se pueden anidar.

Por ejemplo, la orden **`date`**, que permite consultar o establecer la fecha y la hora del sistema, y que admite como argumento **`+%j`** para conocer el número del día actual del año en curso, puede utilizarse para saber cuántas semanas faltan para el fin de año:

```
$ echo "Faltan $(( (365 - $(date +%j)) / 7 )) semanas hasta el fin de año"
```

Ejercicio 1: Utilizando una variable que contenga el valor entero 365 y otra que guarde el número del día actual del año en curso, realice la misma operación del ejemplo anterior usando cada una de las diversas formas de cálculo comentadas hasta el momento, es decir, utilizando **`expr`**, **`$(( … ))`** y **`$[ … ]`**.

2.1 Operadores aritméticos

El shell bash considera, entre otros, los operadores aritméticos que se dan en Tabla 2.

Operador	Descripción
+ -	Suma y resta, o más unario y menos unario.
*** / %**	Multiplicación, división (truncando decimales), y resto de la división.
******	Potencia.
++	Incremento en una unidad. Puede ir como prefijo o como sufijo de una variable: si se usa como prefijo (`++`*`variable`*), primero se incrementa la variable y luego se hace lo que se desee con ella; si se utiliza como sufijo (*`variable`*`++`), primero se hace lo que se desee con la variable y después se incrementa su valor.
--	Decremento en una unidad. Actúa de forma análoga al caso anterior, pudiendo usarse como prefijo o como sufijo de una variable (`--`*`variable`* o *`variable`*`--`).
( )	Agrupación para evaluar conjuntamente; permite indicar el orden en el que se evaluarán las subexpresiones o partes de una expresión.
,	Separador entre expresiones con evaluación secuencial.
= **+= -=** ***= /=** **%=**	`x=expresión`, asigna a `x` el resultado de evaluar la `expresión` (no puede haber huecos en blanco a los lados del símbolo "`=`"); `x+=y` equivale a `x=x+y`; `x-=y` equivale a `x=x-y`; `x*=y` equivale a `x=x*y`; `x/=y` equivale a `x=x/y`; `x%=y` equivale a `x=x%y`.

Tabla 2. Operadores aritméticos.

Ejercicio 2: Realice las siguientes operaciones para conocer el funcionamiento del operador de incremento como sufijo y como prefijo. Razone el resultado obtenido en cada una de ellas:

```
$ v=1
$ echo $v
$ echo $((v++))
$ echo $v
$ echo $((++v))
$ echo $v
```

Ejercicio 3: Utilizando el operador de división, ponga un caso concreto donde se aprecie que la asignación abreviada es equivalente a la asignación completa, es decir, que `x/=y` equivale a `x=x/y`.

En el resultado del cálculo de expresiones aritméticas, bash solamente trabaja con números enteros, por lo que si se necesitase calcular un resultado con decimales, habría que utilizar una forma alternativa, como puede ser la ofrecida por la orden **`bc`**, cuya opción **`-l`**, letra "ele", permite hacer algunos cálculos matemáticos (admite otras posibilidades que pueden verse mediante **`man`**).

El ejemplo siguiente ilustra el uso de la orden **bc** para realizar una división con decimales:

```
$ echo 6/5|bc -l
```

Ejercicio 4: Compruebe qué ocurre si en el ejemplo anterior utiliza comillas dobles o simples para acotar todo lo que sigue a la orden **echo**. ¿Qué sucede si se acota entre comillas dobles solamente la expresión aritmética que se quiere calcular?, ¿y si se usan comillas simples?

Ejercicio 5: Calcule con decimales el resultado de la expresión aritmética `(3-2)/5`. Escriba todas las expresiones que haya probado hasta dar con una respuesta válida. Utilizando una solución válida, compruebe qué sucede cuando la expresión aritmética se acota entre comillas dobles; ¿qué ocurre si se usan comillas simples?, ¿y si se ponen apóstrofos inversos?

2.2 Asignación y variables aritméticas

Otra forma de asignar valor a una variable entera es utilizar la orden **let** de la shell bash. Aunque esta orden se usa para evaluar expresiones aritméticas, en su forma más habitual su sintaxis es:

let variableEntera**=**expresión

Como en otras asignaciones, a ambos lados del signo igual (=) no debe haber espacios en blanco y `expresión` debe ser una expresión aritmética.

Ejemplo: Compruebe el resultado de cada una de las asignaciones siguientes:

```
$ let w=3+2
$ let w='3 + 2'
$ let w='(4+5)*6'
$ let "w=4+5*6"
$ let w=4+5*6
$ y=7
$ let w=y%5                  # esta orden es equivalente a: let w=$y%5
```

Como habrá observado en el ejemplo anterior, las dos primeras asignaciones producen el mismo resultado, a pesar de que en la segunda hay espacios en blanco. Por el contrario, las asignaciones tercera y cuarta no dan el mismo resultado debido al uso o no de paréntesis. Las asignaciones cuarta y quinta son equivalentes, y las dos últimas ponen de manifiesto que en la expresión pueden intervenir otras variables.

Ejercicio 6: Consulte la sintaxis completa de la orden **let** utilizando la orden de ayuda para las órdenes empotradas (**help let**) y tome nota de su sintaxis general.

Ejercicio 7: Al realizar el ejercicio anterior habrá observado que la orden **let** admite asignaciones múltiples y operadores que nosotros no hemos mencionado anteriormente. Ponga un ejemplo de asignación múltiple y, por otra parte, copie en un archivo el orden en el que se evalúan los operadores que admite.

2.3 Operadores relacionales

A veces es necesario poder relacionar dos expresiones aritméticas, `A` y `B`, o negar una expresión aritmética, de forma que se pueda evaluar si se da o no cierta relación. La evaluación de una relación entre expresiones tomará finalmente un valor numérico, de manera que el 1 representa una evaluación "verdadera" (*true*), mientras que el 0 indica que la evaluación ha sido "falsa" (*false*). Observe que esta forma de evaluar resulta un poco discordante respecto a lo que sucede cuando se evalúa la variable `$?` que se mencionaba en la sesión anterior.

A continuación, en Tabla 3, se pueden ver diferentes operadores relacionales admitidos en el shell bash.

Operador	Descripción: el resultado se evalúa como "verdadero" -*true*- si ... (en otro caso sería "falso" -*false*-)
`A = B` | `A == B` | `A -eq B`	`A` es igual a `B`.
`A != B` | `A -ne B`	`A` es distinta de `B`.
`A < B` | `A -lt B`	`A` es menor que `B`.
`A > B` | `A -gt B`	`A` es mayor que `B`.
`A <= B` | `A -le B`	`A` es menor o igual que `B`.
`A >= B` | `A -ge B`	`A` es mayor o igual que `B`.
`! A`	`A` es falsa; representa al operador NOT (negación lógica).
`A && B`	`A` es verdadera y `B` es verdadera; es el operador AND (conjunción lógica).
`A || B`	`A` es verdadera o `B` es verdadera; es el operador OR (disyunción lógica).

Tabla 3. Operadores relacionales.

Por ejemplo, se puede comprobar que la expresión `(8<3)&&(9<5)` es falsa, ya que la primera parte de ella es verdadera, pero la segunda es falsa:

```
$ echo $[$[8>3]&&$[9<5]]
0
$ echo $[8>3] y $[9<5]
1 y 0
```

Ejercicio 8: Haciendo uso de las órdenes conocidas hasta el momento, construya un guion que admita dos parámetros, que compare por separado si el primer parámetro que se le pasa es igual al segundo, o es menor, o es mayor, y que informe tanto del valor de cada uno de los parámetros como del resultado de cada una de las evaluaciones mostrando un 0 o un 1 según corresponda.

En bash, los operadores `==` y `!=`, también pueden utilizarse para comparar si dos cadenas de caracteres `A` y `B` coinciden o no, respectivamente; además, en la versión 2.0 y las posteriores, los operadores `<` y `>` permiten comparar si una cadena de caracteres `A` se clasifica antes o después de otra cadena `B`, respectivamente, siguiendo el orden lexicográfico.

Por otra parte, cabe observar que existen otros operadores aparte de los mencionados aquí.

2.4 Operadores de consulta de archivos

A veces es necesario conocer características específicas de los archivos o directorios para saber cómo tratarlos. En Tabla 4 se pueden ver algunos operadores utilizados para la comprobación de características de archivos y directorios.

Para aplicar los operadores de consulta de archivos haremos uso de dos órdenes nuevas, **`test`** e **`if`**, aunque la segunda de estas órdenes la trataremos en un apartado posterior.

La sintaxis de la orden **`test`** es:

`test` *`expresión`*

Esta orden evalúa una expresión condicional y da como salida el estado 0, en caso de que *`expresión`* se haya evaluado como verdadera (*true*), o el estado 1, si la evaluación ha resultado falsa (*false*) o se le dio algún argumento no válido.

Operador	Descripción: el resultado se evalúa como "verdadero" -*true*- si ... (en otro caso sería "falso" -*false*-)
-a *archivo*	*archivo* existe.
-b *archivo*	*archivo* existe y es un dispositivo de bloques.
-c *archivo*	*archivo* existe y es un dispositivo de caracteres.
-d *archivo*	*archivo* existe y es un directorio.
-e *archivo*	*archivo* existe. Es igual que **-a**.
-f *archivo*	*archivo* existe y es un archivo plano o regular.
-G *archivo*	*archivo* existe y es propiedad del mismo grupo del usuario.
-h *archivo*	*archivo* existe y es un enlace simbólico.
-L *archivo*	*archivo* existe y es un enlace simbólico. Es igual que **-h**.
-O *archivo*	*archivo* existe y es propiedad del usuario.
-r *archivo*	*archivo* existe y el usuario tiene permiso de lectura sobre él.
-s *archivo*	*archivo* existe y es no vacío.
-w *archivo*	*archivo* existe y el usuario tiene permiso de escritura sobre él.
-x *archivo*	*archivo* existe y el usuario tiene permiso de ejecución sobre él, o es un directorio y el usuario tiene permiso de búsqueda en él.
archivo1 **-nt** *archivo2*	*archivo1* es más reciente que *archivo2*, según la fecha de modificación, o si *archivo1* existe y *archivo2* no.
archivo1 **-ot** *archivo2*	*archivo1* es más antiguo que *archivo2*, según la fecha de modificación, o si *archivo2* existe y *archivo1* no.
archivo1 **-ef** *archivo2*	*archivo1* es un enlace duro al *archivo2*, es decir, si ambos se refieren a los mismos números de dispositivo e *inode*.

Tabla 4. Operadores de consulta de archivos.

La orden **test** *expresión* es equivalente a la orden **[** *expresión* **]**, donde los huecos en blanco entre los corchetes y *expresión* son necesarios.

Ejemplo: A continuación se muestran algunos casos de evaluación de expresiones sobre archivos utilizando la orden `test` y corchetes:

```
$ test -d /bin      # comprueba si /bin es un directorio
$ echo $?           # nos muestra el estado de la última orden ejecutada, aunque
0                   # usado después de test o [ ] da 0 si la evaluación era verdadera

$ [ -w /bin ]       # comprueba si tenemos permiso de escritura en /bin
$ echo $?           # usado después de test o [ ] da 1 si la evaluación era falsa
1

$ test -f /bin/cat      # comprueba si el archivo /bin/cat existe y es plano
$ echo $?
```

```
0
$ [ /bin/cat -nt /bin/zz ]   # comprueba si /bin/cat es más reciente que /bin/zz
$ echo $?
1                            # la evaluación es falsa porque /bin/zz no existe
```

Ejemplo: Vemos algunas características de archivos existentes en el directorio `/bin` y asignamos el resultado a una variable:

```
$ cd /bin
$ ls -l cat
-rwxr-xrx 1 root root 38524 2010-06-11 09:10 cat

$ xacceso=`test -x cat && echo "true" || echo "false"`    # se pueden omitir las "
$ echo $xacceso
true                  # indica que sí tenemos permiso de ejecución sobre cat

$ wacceso=`test -w cat && echo "true" || echo "false"`    # se pueden omitir las "
$ echo $wacceso
false                 # indica que no tenemos permiso de escritura en cat
```

Ejercicio 9: Usando **`test`**, construya un guion que admita como parámetro un nombre de archivo y realice las siguientes acciones: asignar a una variable el resultado de comprobar si el archivo dado como parámetro es plano y tiene permiso de ejecución sobre él; asignar a otra variable el resultado de comprobar si el archivo es un enlace simbólico; mostrar el valor de las dos variables anteriores con un mensaje que aclare su significado. Pruebe el guion ejecutándolo con `/bin/cat` y también con `/bin/rnano`.

Ejercicio 10: Ejecute **`help test`** y anote qué otros operadores se pueden utilizar con la orden **`test`** y para qué sirven. Ponga un ejemplo de uso de la orden **`test`** comparando dos expresiones aritméticas y otro comparando dos cadenas de caracteres.

2.5 Orden if / else

La orden **`if/else`** ejecuta una lista de declaraciones dependiendo de si se cumple o no cierta condición, y se podrá utilizar tanto en la programación de guiones, como en expresiones más simples.

La sintaxis de la orden condicional **`if`** es:

```
if condición;
then
      declaraciones;
[elif condición;
      then declaraciones; ]...
[else
      declaraciones; ]
fi
```

La principal diferencia de este condicional respecto a otros lenguajes es que cada *`condición`* representa realmente una lista de declaraciones, con órdenes, y no una simple expresión booleana. De esta forma, como las órdenes terminan con un estado de finalización, *`condición`* se considera *true* si su estado de finalización (`status`) es 0, y *false* en caso contrario (estado de finalización igual a 1). Al igual que en otros lenguajes, en cualquiera de las *`declaraciones`* puede haber otra orden **`if`**, lo que daría lugar a un anidamiento.

El funcionamiento de la orden **`if`** es el siguiente: se comienza haciendo la ejecución de la lista de órdenes contenidas en la primera *`condición`*; si su estado de salida es 0, entonces se ejecuta la lista de *`declaraciones`* que sigue a la palabra **`then`** y se termina la ejecución del **`if`**; si el estado de salida fuese 1, se comprueba si hay

un bloque que comience por **elif**. En caso de haber varios bloques **elif**, se evalúa la *condición* del primero de ellos de forma que si su estado de salida es 0, se hace la parte **then** correspondiente y termina el **if**, pero si su estado de salida es 1, se continúa comprobando de manera análoga el siguiente bloque **elif**, si es que existe. Si el estado de salida de todas las condiciones existentes es 1, se comprueba si hay un bloque **else**, en cuyo caso se ejecutarían las *declaraciones* asociadas a él, y termina el **if**.

En el ejemplo siguiente se utiliza la orden **if** para tener una estructura similar a la que se había planteado anteriormente con **test** usando "**&&**" y "**||**":

```
$ cd /bin
$ ls -l cat
-rwxr-xrx 1 root root 38524 2010-06-11 09:10 cat

$ xacceso=`if test -x cat; then echo "true"; else echo "false"; fi`
$ echo $xacceso
true                    # indica que sí tenemos permiso de ejecución sobre cat

$ wacceso=`if test -w cat; then echo "true"; else echo "false"; fi`
$ echo $wacceso
false                   # indica que no tenemos permiso de escritura en cat
```

Como se puede apreciar, la condición de la orden **if** puede expresarse utilizando la orden **test** para hacer una comprobación. De forma análoga se puede utilizar **[...];** si se usa "`if [ expresión ];`", *expresión* puede ser una expresión booleana y puede contener órdenes, siendo necesarios los huecos en blanco entre los corchetes y *expresión*.

Ejercicio 11: Responda a los siguientes apartados:

a) Razone qué hace la siguiente orden:
   `if test -f ./sesion5.pdf ; then printf "El archivo ./sesion5.pdf existe\n"; fi`
b) Añada los cambios necesarios en la orden anterior para que también muestre un mensaje de aviso en caso de no existir el archivo. (Recuerde que, para escribir de forma legible una orden que ocupe más de una línea, puede utilizar el carácter "****" como final de cada línea que no sea la última.)
c) Sobre la solución anterior, añada un bloque **elif** para que, cuando no exista el archivo `./sesion5.pdf`, compruebe si el archivo `/bin` es un directorio. Ponga los mensajes adecuados para conocer el resultado en cada caso posible.
d) Usando como base la solución del apartado anterior, construya un guion que sea capaz de hacer lo mismo pero admitiendo como parámetros la ruta relativa del primer archivo a buscar y la ruta absoluta del segundo. Pruébelo con los dos archivos del apartado anterior.

Ejercicio 12: Construya un guion que admita como argumento el nombre de un usuario del sistema y que permita saber si ese usuario es el propietario del archivo `/bin/ls` y si tiene permiso de lectura sobre él.

El ejemplo que viene a continuación plantea algunos casos de comparaciones aritméticas utilizando corchetes:

```
$ valor=3
$ if [ $valor == "3" ]; then echo sí; else echo no; fi  # los huecos en blanco a los
sí                                                      # lados de los operadores
$ if [ $valor -eq "3" ]; then echo sí; else echo no; fi # relacionales son
sí                                                      # necesarios
$ if [ $valor == "4" ]; then echo sí; else echo no; fi
no
$ if [ $valor = 3 ]; then echo sí; else echo no; fi
```

```
sí
$ if [ $valor = 4 ]; then echo sí; else echo no; fi
no
```

En los ejemplos siguientes la `condición` del **`if`** es una orden:

```
$ valor=6
$ if [ valor=3 ]; then echo sí; else echo no; fi
sí                    # se hace internamente la orden que hay entre corchetes y no
$ echo $valor         # da error, pero la supuesta asignación en la condición del
6                     # if no tiene efecto sobre la variable que se estaba usando

$ if ls > salida; then echo sí; else echo no; fi
sí                     # además, el if hace la orden ls sobre el archivo salida
$ cat salida           # vemos que la orden ls anterior ha volcado su resultado en
                       # este archivo, poniendo cada nombre en una línea distinta
                       # e incluyendo también el nombre "salida"
$ valor=5
$ if valor=3 && ls ; then echo sí; else echo no; fi
                # muestra el resultado de ls y otra línea con sí;
                # además, hace la asignación correctamente, tal como podemos ver
$ echo $valor
3               # el if ha cambiado el contenido de la variable valor
$ if rm salida; then echo sí; else echo no; fi 2> sal
                # en caso de que el archivo salida exista antes del if, se borra y
                # escribe una línea en pantalla poniendo sí;
                # en caso de que ese archivo no exista, escribe en pantalla una
                # línea con no y pone un mensaje de error en el archivo sal
```

Ejercicio 13: Escriba un guion que calcule si el número de días que faltan hasta fin de año es múltiplo de cinco o no, y que comunique el resultado de la evaluación. Modifique el guion anterior para que admita la opción `-h` de manera que, al ejecutarlo con esa opción, muestre información de para qué sirve el guion y cómo debe ejecutarse.

El siguiente guion de ejemplo se puede utilizar para borrar el archivo `temporal` que se le dé como argumento. Si **`rm`** devuelve 0, se muestra el mensaje de confirmación del borrado; en caso contrario, se muestra el código de error. Como se puede apreciar, hemos utilizado la variable `$LINENO` que indica la línea actualmente en ejecución dentro del guion.

```
#!/bin/bash
declare -rx SCRIPT=${0##*/}  # donde SCRIPT contiene sólo el nombre del guión
                             # ${var##Patron} actúa eliminando de $var aquella parte
                             # que cumpla de $Patron desde el principio de $var
                             # En este caso: elimina todo lo que precede al
                             # último slash "/".

if rm ${1} ; then
      printf "%s\n" "$SCRIPT: archivo temporal borrado"
else
      STATUS=177
      printf "%s – código de finalizacion %d\n" \
      "$SCRIPT:$LINENO no es posible borrar archivo" $STATUS
fi 2> /dev/null
```

En la siguiente referencia se puede encontrar información adicional sobre la sustitución de parámetros y algunos ejemplos: `http://tldp.org/LDP/abs/html/parameter-substitution.html`

Ejercicio 14: ¿Qué pasa en el ejemplo anterior si eliminamos la redirección de la orden **`if`**?

3 Expresiones regulares

Una *expresión regular* es un patrón que describe un conjunto de cadenas y que se puede utilizar para búsquedas dentro de una cadena o un archivo. Un usuario avanzado o un administrador del sistema que desee obtener la máxima potencia de ciertas órdenes, debe conocer y manejar expresiones regulares. Las expresiones regulares se construyen de forma similar a las expresiones aritméticas, utilizando diversos operadores para combinar expresiones más sencillas. Las piezas fundamentales para la construcción de expresiones regulares son las que representan a un carácter simple. La mayoría de los caracteres, incluyendo las letras y los dígitos, se consideran expresiones regulares que se representan a sí mismos. Para comparar con el carácter asterisco (*****), éste debe ir entre comillas simples (**'*'**).

En las expresiones regulares se puede utilizar una barra inclinada invertida (****), denominada a veces como *barra de escape*, para modificar la forma en la que se interpretará el carácter que le siga. Cuando los metacaracteres "**?**", "**+**", "**{**", "**}**", "**|**", "**(**" y "**)**" aparecen en una expresión regular no tienen un significado especial, salvo que vayan precedidos de una barra de escape; si se utilizan anteponiendo esa barra, es decir "**\?**", "**\+**", "**\{**", "**\}**", "**\|**", "**\(**" o "**\)**", su significado será el que corresponda según la Tabla 5.

Para practicar con expresiones regulares puede ser útil la página web http://rubular.com , que dispone de una interfaz en la que se puede escribir un texto y probar diferentes expresiones regulares aplicables sobre el texto y, de manera interactiva, el editor mostrará qué se va seleccionando en función de la expresión regular escrita.

3.1 Patrones en expresiones regulares

La Tabla 5 muestra patrones simples que pueden utilizarse en expresiones regulares. Una expresión regular estará formada por uno o varios de estos patrones.

Dos expresiones regulares también pueden concatenarse; la expresión que se obtiene representa a cualquier cadena formada por la concatenación de las dos subcadenas dadas por las subexpresiones concatenadas. Esto no debe confundirse con el uso del operador OR mencionado en la Tabla 5.

Por defecto, las reglas de precedencia indican que primero se trata la repetición, luego la concatenación y después la alternación, aunque el uso de paréntesis permite considerar subexpresiones y cambiar esas reglas.

3.2 Expresiones regulares con órdenes de búsqueda

Las órdenes de búsqueda **`find`**, **`grep`** y **`egrep`** dadas en la sesión anterior, adquieren mayor relevancia al hacer uso de expresiones regulares en ellas, cosa que se verá con algunos casos. Es conveniente recordar las múltiples y útiles opciones de estas órdenes, además de una diferencia fundamental entre ellas: **`find`** sirve para buscar a nivel de características generales de los archivos, como, por ejemplo, nombre o condiciones de acceso, pero sin entrar en su contenido; por el contrario, **`grep`** y **`egrep`** examinan la cadena que se le dé mediante la entrada estándar o el contenido de los archivos que se le pongan como argumento.

Dado que la shell bash intenta interpretar algunos caracteres antes de pasárselos a las órdenes de búsqueda, es especialmente importante tener en cuenta lo mencionado anteriormente sobre la barra de escape en el uso de expresiones regulares.

A continuación proponemos algunos ejemplos y ejercicios que debe probar para aprender a realizar búsquedas con patrones más o menos complejos.

Ejemplo: Buscar en el directorio `/bin/usr` los archivos cuyo nombre comience con las letras `a` o `z` y acabe con la letra `m`:

```
$ find /usr/bin -name "[az]*m"
```

Patrón	Representa …
****	la barra de escape; si en un patrón se quiere hacer referencia a este mismo carácter, debe ir precedido por él mismo y ambos entre comillas simples.
.	cualquier carácter en la posición en la que se encuentre el punto cuando se usa en un patrón con otras cosas; si se usa solo, representa a cualquier cadena; si se quiere buscar un punto como parte de un patrón, debe utilizarse `\.` entre comillas simples o dobles.
( )	un grupo; los caracteres que se pongan entre los paréntesis serán considerados conjuntamente como si fuesen un único carácter. (Hay que usar `\`)
?	que el carácter o grupo al que sigue puede aparecer una vez o no aparecer ninguna vez. (Hay que usar `\`)
*****	que el carácter o grupo al que sigue puede no aparecer o aparecer varias veces seguidas. (No hay que usar `\`)
+	que el carácter o grupo previo debe aparecer una o más veces seguidas. (Hay que usar `\`)
{n}	que el carácter o grupo previo debe aparecer exactamente *n* veces. (Hay que usar `\`)
{n,}	que el carácter o grupo previo debe aparecer *n* veces o más seguidas. (Hay que usar `\`)
{n,m}	que el carácter o grupo previo debe aparecer de *n* a *m* veces seguidas; al menos *n* veces, pero no más de *m* veces. (Hay que usar `\`)
[ ]	una lista de caracteres que se tratan uno a uno como caracteres simples; si el primer carácter de la lista es "^", entonces representa a cualquier carácter que no esté en esa lista.
-	un rango de caracteres cuando el guión no es el primero o el último en una lista; si el guión aparece el primero o el último de la lista, entonces se trata como él mismo, no como rango; en los rangos de caracteres, el orden es el alfabético, pero intercalando minúsculas y mayúsculas – es decir: aAbB...-; en los rangos de dígitos el orden es 012...
^	la cadena vacía al comienzo de una línea, es decir, representa el inicio de línea; como ya se ha dicho, cuando se usa al comienzo de una lista entre corchetes, representa a los caracteres que no están en esa lista.
$	el final de la línea.
\b	el final de una palabra. (Debe utilizarse entre comillas simples o dobles)
\B	que no está al final de una palabra. (Debe utilizarse entre comillas simples o dobles)
\<	el comienzo de una palabra. (Debe utilizarse entre comillas simples o dobles)
\>	el final de una palabra. (Debe utilizarse entre comillas simples o dobles)
|	el operador OR para unir dos expresiones regulares, de forma que la expresión regular resultante representa a cualquier cadena que coincida con al menos una de las dos subexpresiones. (La expresión global debe ir entre comillas simples o dobles; además, cuando se usa con `grep`, esta orden debe ir acompañada de la opción `-E`)

Tabla 5. Operadores para expresiones regulares.

Ejemplo: Buscar en el directorio `/etc` los archivos de configuración que contengan la palabra "`dev`":

```
$ find /etc -name "*dev*.conf"
```

Seguidamente se utiliza la orden **find** con la opción **-regex** para especificar que se buscan nombres de archivos que satisfagan la expresión regular que se pone a continuación de esta opción (v.g., "regex" significa "regular expression"). El ejemplo siguiente se ha resuelto de dos formas equivalentes y sirve para localizar en el directorio `/usr/bin` los nombres de archivos que contengan las letras `cd` o `zip`:

```
$ find  /usr/bin  -regex  '.*\(cd\|zip\).*'

$ find  /usr/bin  -regex  '.*cd.*' -o -regex  '.*zip.*'
```

Ejemplo: Buscar en el archivo `/etc/group` las líneas que contengan un carácter `k`, o `y` o `z`:

```
$ grep [kyz] /etc/group
```

El ejemplo anterior también se puede resolver de otro modo que parece más complicado, pero que aumenta la capacidad de programación para construir guiones. Haciendo uso del mecanismo de cauces que nos ofrece Linux, el archivo cuyo contenido se examina con la orden **grep** se puede sustituir por la salida de una orden **cat** sobre ese mismo archivo:

```
$ cat  /etc/group  |  grep [kyz]
```

De manera análoga, utilizando el mecanismo de cauces se puede conseguir que la salida de otras órdenes se trate como si fuese un archivo cuyo contenido se desea examinar.

Ejercicio 15: Haciendo uso del mecanismo de cauces y de la orden **echo**, construya un guion que admita un argumento y que informe si el argumento dado es una única letra, en mayúsculas o en minúsculas, o es algo distinto de una única letra.

El siguiente ejemplo permite añadir a la variable `$PATH` un directorio que se pase como primer argumento del guion; si se da la opción `after` como segundo argumento del guion, el directorio se añadirá al final de la variable `$PATH`, en cualquier otro caso el directorio se añadirá al principio de `$PATH`:

```
#!/bin/bash
# Uso: pathmas directorio [after]
if ! echo $PATH | /bin/egrep -q "(^|:)$1($|:)" ; then
          if [ "$2" = "after" ] ; then
              PATH=$PATH:$1
           else
              PATH=$1:$PATH
          fi
else
    echo "$1 ya está en el path"
fi
```

Ejercicio 16: Haciendo uso de expresiones regulares, escriba una orden que permita buscar en el árbol de directorios los nombres de los archivos que contengan al menos un dígito y la letra `e`. ¿Cómo sería la orden si quisiéramos obtener los nombres de los archivos que tengan la letra `e` y no contengan ni el `0` ni el `1`?

Ejercicio 17: Utilizando la orden `grep`, exprese una forma alternativa de realizar lo mismo que con `wc -l`.

Para finalizar esta sesión y afianzarse en el uso de expresiones regulares utilice el archivo `s5_Complementos.txt` que hay en la plataforma tutor. En él encontrará instrucciones para crear un archivo de prueba y poder ejecutar algunos ejemplos, así como algunos ejercicios más.