Programación en shell Bash

Empezando a programar en shell Bash

Tabla de Contenidos

Autor: Lina García Cabrera & Francisco Martínez del Río

1 Introducción

El objetivo de este guión de prácticas es que el lector aprenda a **realizar guiones en bash**, esto le permitirá crear órdenes sencillas de una forma rápida y sin necesidad de utilizar un lenguaje de alto nivel.

2 Creación de Órdenes Nuevas

Dada una serie de órdenes cuya ejecución va a ser repetida varias veces, sería conveniente convertirlas en una nueva orden con un nombre propio, de manera que pueda usarse como una orden normal. Por ejemplo, supongamos que uno intenta contar usuarios frecuentemente mediante una interconexión:

```
$ who | wc -l
```

y desea escribir un nuevo programa **nu** que lo haga. El primer paso es crear un archivo ordinario que contenga '`who | wc -l`'. El lector puede emplear su editor favorito, o bien, si es creativo, teclear:

```
$ cat > nu
  #!/bin/bash
  who | wc –l
  ctrl-d
```

> Con **cat > nu** se redirige la salida a un fichero. Lo que escribimos desde el teclado se guarda en el fichero **nu**

Como se mencionó anteriormente, los shells son programas como un editor, **who** o **wc**; el *shell* que nosotros utilizamos tiene por nombre **bash**. Como un *shell* es un programa, se puede ejecutar, y su entrada puede ser redireccionada. Así, podemos ejecutar **bash** con la entrada proviniendo del fichero *nu*, en lugar del terminal:

```
$ who
  fmartin    pts/1 obre 16 13:20
  lina        pts/5 obre 15 17:40
  lperez    pts/6 obre 15 19:49
  $ bash <nu
      3
```

> El resultado de tu ejecución puede ser distinto, depende del número de usuarios que estén conectados. Ejecuta antes **who** para ver cuantos usuarios hay.

La salida es la misma que la que se produciría si hubiera tecleado '**who | wc –l**' desde el terminal. Al igual que muchos otros programas, los *shells* toman su entrada de un fichero si se le da alguno como argumento; se podría haber escrito:

```
$ bash nu
```

para obtener el mismo resultado. Pero es molesto tener que teclear bash en ambos casos: es más tedioso, y crea un distinción entre programas escritos en, por ejemplo C, y otros compuestos de órdenes de *shell*. Para evitar teclear **csh** lo único que hay que hacer es asignar a nu el permiso de ejecución:

```
$ chmod +x nu
```

y después se le puede invocar con:

```
$ nu ó $ ./nu
```

Los usuarios de **nu** no podrán saber, simplemente ejecutándolo, que el lector lo ha creado de esta forma tan sencilla. La manera en que **bash** ejecuta realmente **nu** es creando un nuevo proceso ***bash***, exactamente igual que si hubiera tecleado

```
$ bash nu
```

Este *shell* hijo se llama *subshell* (un proceso *shell* invocado por el *shell* actual del usuario). '**bash nu**' no es lo mismo que '**bash < nu**', ya que su entrada estándar está conectada aún al terminal.

Un fichero de texto que contiene órdenes se llama **fichero de shell o guión (*script*)**.

La primera línea de **nu** (`!#/bin/bash`) sirve para indicarle a UNIX que el guión debe ser interpretado por el *shell*
bash. Esto es importante porque los distintos *shells* pueden tener órdenes internas o metacaracteres distintos
para realizar las mismas tareas. Por ejemplo, la sintaxis para crear variables de shell varía en los shells bash y csh:

$ **set x = 3** *Se crea la variable de shell x con valor 3 en csh*
$ **x=5** *Se crea la variable de shell x con valor 5 en sh*

$ echo 'set x = 3; echo $x' >gcsh *Crea el guión gcsh con contenido set x = 3; echo $x*
$ echo 'x=5; echo $x' >gsh *Crea el guión gsh con contenido x=5; echo $x*
$ **csh gcsh** *Interpretamos gcsh con csh*
3 *Funciona*
$ **sh gcsh** *Interpretamos gcsh con sh*
$ *No funciona*
$ **sh gsh** Interpretamos gsh con sh
5 Funciona
$ **csh gsh** *Interpretamos gsh con csh*
x=5: Command not found.
x: Undefined variable.

Supongamos ahora que queremos ejecutar un guión al que hemos dado permiso de ejecución escribiendo únicamente su nombre (como hicimos con **nu**). La pregunta es: ¿qué *shell* interpretará el guión? Pues bien, cuando un shell detecta una orden que no es interna, es decir, que el *shell* no ejecuta directamente, llama al sistema operativo para que la ejecute. La orden puede ser un programa compilado o un guión de *shell*. En el segundo caso, el sistema operativo tiene que seleccionar un *shell* para ejecutar el guión.

La forma en que el sistema operativo realiza su selección depende del sistema. Por omisión se elige una versión del *shell Bourne* o, en algunas ocasiones, el *shell Korn* (que es casi completamente compatible con aquel). Sin embargo, la mayoría de los sistemas se adhieren al convencionalismo de BSD UNIX que especifica que la primera línea de un guión de *shell* tiene la forma

`#! shell`

donde *shell* es la trayectoria absoluta del *shell* que se usa para interpretar el guión. El espacio en blanco después de '`#!`' es opcional. Por ejemplo, si se escribe un guión usando el conjunto de órdenes del *shell* C, éste deberá comenzar con la línea: **`#! /bin/csh`**. Si quiere que sea interpreta con el shell de Bourne, debe comenzar con la línea: **`#! /bin/bash`**. De hecho, usted puede hacer que un guión sea interpretado por cualquier programa. Por ejemplo, suponga que emite una orden que nombra a un archivo ejecutable, cuya primera línea es

`#! /home/sibila/leer.entrañas`

Entonces, el programa **leer.entrañas** del directorio base de *sibila* asume el control, usando como entrada el
archivo ejecutable. Por supuesto, **leer.entrañas** debe tener la inteligencia suficiente para reconocer la primera línea como comentario e ignorarla.

Algunos sistemas que no respetan el convencionalismo '`#!`' ofrecen otra manera de etiquetar los guiones de shell: si el primer carácter es '`:`', se considera como un guión del shell Bourne; si el primer carácter es '`#`', se considera como un guión del shell C.

Compruebe si nu puede ser interpretado por csh y por sh.

- Funciona con bash pero no con csh
- No funciona ni con bash ni con csh

- Funciona con csh pero no con bash
- Funciona tanto con bash como con csh

Enviar respuestas

Asigne permisos de ejecución a gsh y gcsh. Modifíquelos para que al ser invocados de la forma $ gsh y $ gcsh sean ejecutados por los shells sh y csh respectivamente.

- No hay que hacer nada
- Primera fila de gsh #! /bin/csh
- Primera fila de gsh #! /bin/bash
- Primera fila de gcsh y de gsh #! /bin/csh

Enviar respuestas

3 Argumentos en los Guiones

Aunque **nu** es adecuado tal como se mostró, la mayoría de los guiones precisan trabajar con argumentos. Suponga que queremos hacer un programa, llamado **cx**, que cambia el modo de un fichero a ejecutable, de manera que

```
$ cx nu
```

sea la abreviatura de

```
$ chmod +x nu
```

Ya conocemos casi todo lo que se necesita para hacer esto. Necesitamos un fichero llamado **cx** cuyo contenido sea:

```
#!/bin/bash
chmod +x fichero
```

Lo único nuevo que necesita saber es cómo indicarle a **cx** el nombre del fichero, puesto que éste será diferente cada vez que se ejecute **cx**.

Cuando **bash** ejecuta un guión, cada ocurrencia de **${1}** se reemplaza por el primer argumento, cada ocurrencia de **${2}** se reemplaza por el segundo argumento, y así sucesivamente. Las llaves no son precisas si no existe ambigüedad para delimitar el número. En consecuencia, si el fichero **cx** contiene '**chmod +x $1**', cuando se ejecute la orden

```
$ cx nu
```

el *subshell* que interpreta las órdenes de **cx** reemplaza **$1** por su primer argumento: **nu**. Veamos la secuencia completa de operaciones:

```
$ cat > cx Cree cx originalmente
#!/bin/bash
chmod +x $1
ctrl-d
$ chmod +x cx Haga que cx sea ejecutable (También hubiera valido $ bash cx cx)
$ cat > hola Haga un guión de prueba
#!/bin/bash
echo ¡qué tal!
ctrl-d
$ hola Intente ejecutarlo
hola: Permission denied
$ cx hola Hágalo ejecutable con cx
$ hola Inténtelo otra vez
¡qué tal!
```

¿Y si se deseara gestionar más de un argumento? Por ejemplo, si cx gestionara varios archivos al tiempo. Un primer intento es poner varios argumentos en el guión, como en: '**chmod +x $1 $2 $3 $4 $5 $6 $7 $8 $9**'. Si el usuario de este guión proporciona menos de nueve argumentos, los restantes serán cadenas nulas; el efecto es que sólo los argumentos que fueron realmente tecleados son pasados a **chmod** por el *subshell*. De este modo, esto funciona, pero falla si se dan más de nueve argumentos. Previendo este problema, **bash** proporciona una abreviatura, **$***, que se sustituye por todos los argumentos. La manera correcta de definir **cx**, por lo tanto, es: **'chmod +x $*'**, que funciona sin importar cuántos argumentos se den.

Ya con **$*** en su repertorio el lector puede hacer algunos guiones, como **lc**:

```
$ cat lc
```

```
#!/bin/bash
wc -l $*
```

lc puede usarse sensatamente sin argumentos. Si no hay argumentos, **$*** será nulo, y no se pasará ningún argumento a **wc**. Con, o sin argumentos, la orden se invoca correctamente:

lc /etc/passwd /etc/group *Utilizamos lc para contar el número de líneas de dos ficheros*
```
390 /etc/passwd
46 /etc/group
436 total
```

> **Bash** utiliza principalmente variables de tipo cadena de caracteres. Los **parámetros posicionales** son variables que reciben los arguementos de entrada de un script. Sus nombres son 1, 2, 3, 4, etc. Para saber su contenido se preceden de $ ($1, $2, $3, $4...). El parámetro posicional 0 almacena el nombre del script que se ejecuta.

3.1 Variables $*, $@ y $#

La variable **$#** almacena el número de argumentos recibidos por el script y es de tipo cadena de caracteres.

Tanto $* como $@ nos devuelven los argumentos recibidos por el script. Cuando no se entrecomillan se comportan igual pero al encerrarlos entre comillas débiles "$*" crea un único token con todos los argumentos recibidos mientras que "$@" crea un token por cada argumento recibido.

4 Manipulación de vectores y cadenas

Manipulación de vectores

Un modo conveniente de **manipular los parámetros de un *script*** es como si éstos se trataran de un ***vector de cadenas***, de tal modo que luego podamos referir cualquier parámetro a través de un índice. El modo en que se crea en **bash** un índice es muy sencillo, basta con colocar entre parentesis los elementos que conforman el vector, separados por un espacio:

> dofer@robin:~/ssoo$ **animales_domesticos=(gato perro loro)**

Ahora, para acceder a un elemento del vector:

> dofer@robin:~/ssoo$ **echo ${animales_domesticos[1]}**
> **perro**

Nótese que el primer elemento está en el índice 0.
Si deseamos manipular los argumentos de un script como si de un vector se tratara todo lo que tenemos que hacer es poner los argumentos entre parentesis.

El siguiente ejemplo muestra los argumentos pasados a script utilizando esta técnica:

```
#!/bin/bash
i=0
para=($*)
until (( $i > $# ))
do
     echo ${para[$i]}
     let i=$i+1
done
```

> Nótese que en la expresión *${para[$i]}* es necesario indicar con llaves el ámbito al que afecta el primer $.

Manipulación de cadenas

La shell bash permite manipular una *cadena* mediante el operador ":".

Véase el siguiente ejemplo:

> *dofer@robin:~/ssoo$* **cadena=abcdefgh**
> *dofer@robin:~/ssoo$* **echo ${cadena:1:1}**
> **b**

En este ejemplo, indicamos que queremos seleccionar a partir del segundo carácter (se empieza a contar desde el 0), una cadena de longitud 1.

Es posible conocer la longitud de una cadena precediéndola de #

> *dofer@robin:~/ssoo$* ***cadena=abcdefgh***
> *dofer@robin:~/ssoo$* ***echo ${#cadena}***
> ***8***

5 Estructuras iterativas en bash: for, while y until

Para poder hacer guiones interesante necesitamos utilizar construcciones condicionales e iterativas. En este apartado estudiamos las órdenes internas de **bash** que permiten crear ciclos. La primera orden que analizamos es **for**, cuya sintaxis es:

```
1 for VARIABLE in [palabra ....]
2 do
3         orden1
4         orden2
5         .....
6         ordenN
7 done
```

El resultado es que la lista de órdenes (*orden1*, *orden2*, ...) se ejecuta una vez por cada *palabra* de la lista [*palabra* ...]. En cada ejecución de la lista de órdenes a la variable *nombre* se le asignará una palabra de la lista. En la primera ejecución la primera palabra de la lista, en la segunda ejecución la segunda palabra de la lista, y así sucesivamente. Por ejemplo:

```
1 #!/bin/bash
2 for i in 1 2 3 4 5
3 do
4    echo "Bienvenido $i vez"
5 done
```

```
1 #!/bin/bash
2 for i in {1..5}
3 do
4    echo "Bienvenido $i vez o veces"
5 done
```

```
1 #!/bin/bash
2 echo "Bash version ${BASH_VERSION}..."
3 for i in {0..10..2}
4   do
5      echo "Welcome $i times"
6  done
```

```
$ for x in a b c
> do
> echo $x
> done
a
b
c
```

Hace que la orden `'echo $x'` se ejecute tres veces, una por cada una de las palabras de la lista (a b c). En la primera ejecución la variable x toma el valor a, en la segunda b, y en la tercera c.

Observe de nuevo la sintaxis de **for**. Es muy importante que aprenda que los caracteres de nueva línea forman parte de su sintaxis. **for** espera que la primera línea termine con la lista de palabras, que le siga un **do** y que la última línea sólo contenga la palabra **done**. Por ejemplo

$ for x in a b c do echo $x done *No funciona, no verifica la sintaxis de for*

Aunque en el ejemplo previo se utilizó **for** en la línea de órdenes, lo usual es usarla en guiones, como el siguiente, que tiene por objeto mostrar cada uno de sus argumentos en una línea distinta:

```
$ cat mieco
#!/bin/bash
for arg in $*
do
echo $arg
done
$ mieco a b c
a
b
c
```

Observe cómo se utiliza la sustitución **$*** para producir todos los argumentos del guión. Existe una sustitución similar, **$@**. La diferencia entre ambos es bastante sutil, **$@** representa cada parámetro como una cadena entrecomillada, de tal modo que cada parámetro es palabra separada de las demás. En la práctica, en muchos casos ambos comodines son intercambiables. Sin embargo, con independencia de la sustutución utilizada, el guión no es correcto, como se deduce de la siguiente ejecución:

```
$ mieco "a b" c
a
b
c
```

En esta ejecución se ha utilizado el metacarácter comillas dobles para que **bash** no interprete el espacio en blanco entre la *a* y la *b* como un separador de palabras. Esto hace que el guión tenga dos argumentos: "a b" y c. Sin embargo, mieco muestra 3 parámetros, ¿por qué?. La respuesta es la siguiente: la sustitución **$*** produce el listado de los 2 argumentos del guión separados por espacios en blanco, es decir: "a b c". Como el primer argumento contiene un espacio en blanco, éste se va a convertir en dos palabras al ser empleado por **for**.

Estudiemos la siguiente orden de **bash** que permite realizar contrucciones iterativas , y construyamos una

orden alternativa a **mieco** . Su nombre es **while**, y al igual que en **for** y en el resto de órdenes iterativas o condicionales, los caracteres de nueva línea forman parte de su sintaxis, la cual es:

while expresion
do
orden1
orden2
.....
ordenN
done

El efecto de esta orden es el siguiente: mientras la expresión expr sea distinta de cero (verdadera) se ejecutarán la lista de órdenes orden1, orden2, ... (en la siguiente sección se describen las expresiones en bash). El siguiente programa realiza un **eco** empleando la orden **while** y la variable predefinida de argumentos de entrada.

> **$ cat mieco2**
> #!/bin/bash
> i=0
> para=($@)
> while [$i -le $#]
> do
> echo ${para[$i]}
> let i=$i+1
> done
> **$ mieco2 a b c**
> a
> b
> c

El siguiente programa es equivalente a **mieco2**, pero un poco más breve, e ilustra el funcionamiento de la orden **shift**.

> **$ cat mieco3**

```
1 #!/bin/bash
2 while (( $# > 0 ))
3 do
4   echo $1
5   shift
6 done
```

$ mieco3 "a b" c
a b
c

Como se indicó en el tema anterior
$ shift [var]

Desplaza las palabras de `var` una a la izquierda, eliminando la primera. Si se omite `var`, se considera que la variable sobre la que se actúa es `$@`.

En las contrucciones **for** y **while** puede utilizar la orden interna **break**, que produce que se ejecute la siguiente instrucción de la construcción **for** o **while** más cercana que la abarque.

En **for** y **while** también puede utilizar la orden interna **continue**, que produce que se pase al principio de la siguiente iteración de la construcción **while** o **for** más cercana que la abarque.

Para terminar con las construcciones iterativas de **bash** presentamos la orden **until** cuya sintaxis es:

```
1 until (expresion)
2 do
3         orden1
4         orden2
5         .....
6         ordenN
7 done
```

Y cuyo resultado es ejecutar las ordenes hasta que la expresión sea verdad.

```
1 #!/bin/bash
2 i=0
3 para=($@)
4 until (( $i > $# ))
5 do
6   echo ${para[$i]}
7   let i=$i+1
8 done
```

$ mieco4 a b c
a
b
c

6 Expresiones

Varias órdenes internas de **bash**, como **while** o **if**, aceptan expresiones. Los operandos de estas expresiones son números decimales, números octales, cadenas o pruebas. Una secuencia de dígitos que comienza por cero se considera como un número octal. Las pruebas se describen en "Pruebas". Los operandos se pueden obtener como resultado de sustituciones de variables.

Las expresiones se pueden clasificar en:

- Expresiones aritméticas: las que dan como resultado un número entero o binario.
- Expresiones condicionales: utilizadas por órdenes internas de BASH cuya evaluación indica si ésta es cierta o falsa.
- Expresiones de cadenas: aquellas que tratan cadenas de caracteres.

Las expresiones complejas cuentan con varios parámetros y operadores, se evalúan de izquierda a derecha. Sin embargo, si una operación está encerrada entre paréntesis seconsidera de mayor prioridad y se ejecuta antes.A modo de resumen, la siguiente tabla presenta los operadores utilizados en los distintos tipos de expresiones BASH

Operadores aritméticos:	+ - * / % ++ --
Operadores de comparación:	== != < <= > >= -eq –nt –lt -le –gt –ge
Operadores lógicos:	! && ||
Operadores binarios:	& | ^ << >>
Operadores de asignación:	= *= /= %= += -= <<= >>= &= ^= |=
Operadores de tipos de ficheros:	-e –b –c –d –f -h -L -p -S –t
Operadores de permisos:	-r -w -x -g -u -k -O -G –N
Operadores de fechas:	-nt –ot –et
Operadores de cadenas:	-z -n

$# Es remplazado por la cantidad de parámetros que el script recibe.

$* Que se expande a todos los parámetros que el script haya recibido, un parámetro se separa de otro con el valor de la variable IFS que normalmente es un espacio.

$@** Similar a **$*, pero cada párametro es entrecomillado.

$? Todo programa al terminar debe retornar un número al sistema operativo, por convención 0 significa operacióne exitosa y números diferente representan errores. $? se expande al número retornado por el último programa ejecutado en primer plano. Un script puede retornar un 3 en lugar de 0 con exit 3

$- Opciones que se pasaron al script durante su ejecución.

$$ Identificación del proceso del intérprete de comandos.

$! Identificación del proceso del último comando que se ejecutó en segundo plano.

$0 Nombre del script o del shell.

Pruebas o **test** son evaluados y según su valor de salida se asimila a verdadero o falso. En UNIX se asimila 0 a éxito, por lo que estos test devuelven 0 como equivalente a verdadero. Existen varios constructores de pruebas:

- Los corchetes simples **[]**, es la forma estandard de usar test. Nótese que si deseamos usarlos con los operadores aritméticos < o > es necesario prefijarlos con el carácter \, de modo que no se interpreten como redirección. Algunos ejemplos:

```
[dofer@pinguino ~]$ [ "abc" != "def" ];echo $?
0
[ian@pinguino ~]$ test "abc" = "def" ;echo $?
1
[ian@pinguino ~]$ [ "abc" != "def" ];echo $?
0
[ian@pinguino ~]$ [ "abc" \< "def" ];echo $?
```

- Los dobles paréntesis ((, evaluan una expresión aritmética y fijan como como valor de salida 1 si la

expresión aritmética vale 0, y 0 en otro caso. Algunos ejemplos:

```
(( 0 ))
echo "Exit status: \"(( 0 ))\" is $?." # 1

(( 1 ))
echo "Exit status: \"(( 1 ))\" is $?." # 0

(( 5 > 4 )) # true
echo "Exit status: \"(( 5 > 4 ))\" is $?." # 0

(( 5 > 9 )) # false
echo "Exit status: \"(( 5 > 9 ))\" is $?." # 1

(( 5 == 5 )) # true
echo "Exit status: \"(( 5 == 5 ))\" is $?." # 0


(( 5 - 5 )) # 0
echo "Exit status: \"(( 5 - 5 ))\" is $?." # 1
```

- **let** permite asignar a una variable el resultado de evaluar una expresión aritmético-lógica

```
let a=11
let a=a+5
let "a=5>4" #ponemos la expresión entrecomillada para evitar que se interprete el < como
redirección. Equivale a let a=5\>4
```

- Los dobles corchetes [[, son similares a [, pero con una sintaxis más natural. Además permiten evaluar expresiones regulares.

Las pruebas que siguen proporcionan información sobre el estado de un fichero.

- **-r fichero** determina si fichero tiene permiso de lectura.
- **-w fichero** determina si fichero tiene permiso de escritura.
- **-x fichero** determina si fichero tiene permiso de ejecución.
- **-e fichero** determina si fichero existe.
- **-o fichero** determina si fichero pertenece al usuario que llamó a bash.
- **-z fichero** determina si fichero tiene tamaño cero.
- **-f fichero** determina si fichero es un fichero ordinario.
- **-d fichero** determina si fichero es un directorio.

7 Estructuras condicionales en bash: if y case

En este apartado estudiamos las órdenes internas de bash que permiten crear construcciones condicionales. La primera que analizamos es **if**. Ésta orden admite dos formatos, el primero es el siguiente:

if [expresion]
then orden
fi

que ejecuta orden si expr es distinto de cero (verdadero), por ejemplo:

$ echo hola >f *Creamos f con contenido hola*
$ if [-e f] && [-r f]
then
cat f
fi *Si existe f, y tenemos permiso para leer f, mostramos su contenido*
hola
$ cat f
hola
$ chmod 000 f *Quitamos todos los permisos a f*
$ cat f *Intentamos mostrar el contenido de f*
cat: f: Permission denied

$ if [-e f] && [-r f]
then
cat f
fi *No llega a ejecutar cat f*

Pasemos a otros formatos de **if**, más potentes, cuya sintaxis es:

If – Then – Else If – Then – Else if – Else

if [expresión]	**if [expresión1]**
then	**then**
órdenes	**órdenes**
else	**elif [expresión2]**
órdenes	**then**
fi	**órdenes**
	else
	órdenes
	fi

Que prueba cada *expresión* hasta encontrar una con valor distinto de cero (verdadero), para luego ejecutar su lista de órdenes correspondiente, *órdenes*. Si no hay ninguna expresión distinta de cero y existe una parte 'else', se ejecuta su lista de órdenes.

El siguiente guión comprueba si los ficheros que se le pasan como parámetros existen, y si el usuario tiene permiso de lectura sobre ellos, antes de intentar leerlos con **cat**. El guión da información en el caso de que un fichero no exista o no se tenga acceso de lectura.

```
 1 #! /bin/bash
 2 while (( $# > 0 ))
 3 do
 4    if [ ! -e $1 ]
 5    then
 6         echo $1: no existe
 7    elif [ ! -r $1 ]
 8       then
 9            echo $1: no es posible leerlo
10       else
11           cat $1
12    fi
13    shift
14 done
```

Para terminar con las estructuras condicionales de **bash** tenemos la orden interna **case**, cuya sintaxis es:

```
 1 case cadena in
 2 cadena1)
 3     rdenes
 4 ;;
 5 cadena2)
 6    rdenes
 7 ;;
 8 *)
 9    rdenes
10 ;;
11 esac
```

case permite seleccionar las órdenes que se ejecutarán de acuerdo con la coincidencia con una cadena. **case** puede contener cualquier número de casos. *cadena* se expande con todas las sustituciones aplicables. Después se examina cada etiqueta (cadena1, cadena2, ...) de caso. La *cadena* se trata como un patrón y se compara con cadena. Si es posible obtener cadena a partir de *cadena* con la expansión de nombres de fichero, es decir, expandiendo los comodines, se selecciona el caso, y se ejecuta su lista de órdenes y las demás listas de órdenes relacionadas con otros casos, hasta llegar a **'esac'** o detectar el fin de uno de los case **;;**. Lo usual es terminar cada caso con **;;** para evitar que el control pase al siguiente caso.

Si ninguno de los casos coincide, y existe un enunciado por defecto ***)**, entonces se ejecuta su lista de órdenes. Si ninguno de los casos coincide y no hay enunciado por defecto, concluye la ejecución de todo el enunciado.

Como ejemplo de uso de **case** se presenta el guión **direc**, que permite realizar varias acciones sobre un directorio, su sintaxis es:

$ direc acción directorio

Donde *directorio* hace referencia al directorio sobre el que se aplica la acción, *acción* puede valer:

- **-c** Implica la creación del directorio *directorio*.
- **-r** Implica la eliminación del árbol del sistema de ficheros que empieza en *directorio*.
- **-l** Implica el listado del contenido de *directorio*.

El guión **direc** es el siguiente:

```
1 #! /bin/bash
2 case $1 in
3 -c)
4     mkdir $2
5 ;;
6 -r)
7     rm -r $2
8 ;;
```

```
 9 -l)
10     ls $2
11 ;;
12 *)
13    echo $1 no es una opcin vlida
14 ;;
15 esac
```

8 Sustitución de órdenes

En el módulo anterior ya se estudió el concepto de sustitución de órdenes que, recordemos, sirve para utilizar la salida de una orden como parte de otra orden. El empleo de este concepto en guiones permite incrementar en gran medida su la potencia de éstos. Estudiémoslo mediante ejemplos.

$ x=`grep lina /etc/passwd`; echo $x
`lina:x:1000:1000:Lina García Cabrera,,,:/home/lina:/bin/bash`

> La ` es la **comilla inversa o de ejecución**, está situada normalmente en la tecla situada a la derecha del teclado, a lado de la *p* y en la misma tecla en la que aparece los símbolos **^]**

La orden '**grep lina /etc/passwd**' extrae las líneas del fichero /etc/passwd en que aparece la cadena de caracteres *lina*. Al utilizar la orden en una sustitución se asignan las líneas (en este caso sólo hay una) a la variable *x*. Observe que una línea de `/etc/passwd` contiene información sobre un usuario, utilizándose los dos puntos para separar los distintos campos de información. El siguiente guión toma como argumento el login de un usuario y muestra información sobre él extraída de `/etc/passwd`. En el guión se utiliza la orden **cut**, que sirve para extraer zonas específicas de líneas de texto. Esta orden resulta muy útil para la realización de guiones, por lo que se describe al final del tema en el apéndice A.

$ cat datos
```
#! /bin/bash
fila=`grep $1 /etc/passwd`
echo Nombre: `echo $fila | cut -f5 -d:`
echo Directorio base: `echo $fila | cut -f6 -d:`
echo Shell de inicio: `echo $fila | cut -f7 -d:`
echo Identificador de usuario: `echo $fila | cut -f3 -d:`
echo Identificador de grupo: `echo $fila | cut -f4 -d:`
```

$ datos lina
```
Nombre: Lina García Cabrera
Directorio base: /home/lina
Shell de inicio: /bin/bash
Identificador de usuario: 1000
Identificador de grupo: 1000
```

En **cut** la combinación de las opciones **–f*n* -d*c*** extrae el campo *n* de la línea de texto, utilizándose el carácter *c* para delimitar los campos.

En la sustitución anterior sólo se obtiene una línea de texto, estudiemos un aspecto interesante en sustituciones de órdenes que producen más de una línea de texto. Por ejemplo, suponga que existen dos usuarios conectados al sistema:

$ who
```
fmartin pts/0 Oct 31 09:32
ajrueda pts/1 Oct 31 09:40
```

$ v=`who`
$ echo $#v ${v[*]}
`10 fmartin pts/0 Oct 31 09:32 ajrueda pts/1 Oct 31 09:40`
$ w[0]=`who | head -1`
$ w[1]=`who | tail -1`
$ echo $#w ${w[*]}
`2 fmartin pts/0 Oct 31 09:32 ajrueda pts/1 Oct 31 09:40`

Analicemos la primera sustitución de órdenes: en ella tanto los caracteres de nueva línea como los espacios en blanco y los tabuladores producidos por la sustitución separan las palabras de la salida. Para poder quedarnos con cada fila debemos extraer las líneas.

$ echo $v[2]
`pts/0`
$ echo $w[2]
`ajrueda pts/1 Oct 31 09:40`

9 Leer archivos de texto

Leer un archivo de texto palabra a palabra es muy sencillo en bash. Por ejemplo podemos usar un bucle "for". Supongamos el arhivo ejemplo.txt con el siguiente contenido:

```
Esta es la línea n° 1
Esta es la línea n° 2
Esta es la línea n° 3
Esta es la línea n° 4
Esta es la línea n° 5
```

Si quremos leerlo palabra por palabra podemos hacerlo con un bucle for:

```
dofer@robin:~/ssoo$ for line in $(cat file.txt); do echo "$line" ; done
Esta
Es
La
Línea
n°
1
Esta
Es
La
Línea
n°
2
Esta
...
```

Sin embargo, en ocasiones no queremos leer el archivo palabra por palabra, sino línea por línea. La solución consiste en utilizar un bucle "while" asociado al comando interno "read".

Sin embargo, también podemos obtener el mismo resultado con un bucle “for” con la condición de que cambiemos el valor de la variable "$IFS" (Internal Field Separator, separador de campo interno) antes de ejecutar el bucle. Es lo que veremos a continuación.

Bucle while

El bucle “while” sigue siendo el método más apropiado y simple para leer un archivo línea por línea.
Sintaxis

```
while read linea
do
comando
done < archivo
```

El archivo de inicio daría:

```
Esta es la línea n° 1
Esta es la línea n° 2
Esta es la línea n° 3
Esta es la línea n° 4
Esta es la línea n° 5
```

Las instrucciones en línea de comandos:

```
dofer@robin:~/ssoo$ while read line; do echo -e "$line\n"; done < file.txt
```

o en un script “bash”:

```
#!/bin/bash

while read line
do
echo "$line\n" #las comillas son necesarias para que se interprete correctamente la \n. Más
información en el tema 4.
done < $1
```

La salida en la pantalla será:

```
Esta es la línea n° 1
Esta es la línea n° 2
Esta es la línea n° 3
```

```
Esta es la línea n° 4
Esta es la línea n° 5
```

Trucos

También podemos a partir de un archivo estructurado (como una libretas de direcciones) obtener los valores de cada campo y asignarlos a varias variables con el comando “read”. Sin embargo hay que tener cuidado de asignar a la variable “IFS” el separador de campo adecuado (espacio por defecto).

Ejemplo:

```
#!/bin/bash

while read nombre edad telefono #se leen tres campos por línea separados por espacio, y se
les asigna a las variables nombre edad y telefono
do
echo -e "Nombre: $nombre :\n\
Edad: $edad\n\
Telefono:\t $telefono\n"
done < /etc/passwd
```

Bucle for para leer de línea en linea

Si bien es cierto que el bucle “while” es el método más simple, sin embargo este tiene un gran inconveniente, el de eliminar el formateado de las líneas y especialmente los espacios y tabulaciones. Felizmente el bucle “for” asociado a un cambio de IFS permite conservar la estructura del documento a la salida. Recuerda que IFS es el campo que usa bash para delimitar los elementos de una lista. Es por eso que si usamos "for" para leer una archivo, por defecto recupera una palabra en cada iteracción.

Sintaxis

```
oldIFS=$IFS # conserva el separador de campo
IFS=$'\n' # nuevo separador de campo, el caracter fin de línea
for línea in $(cat archivo)
do
comando
done
IFS=$old_IFS # restablece el separador de campo predeterminado
```

Bucle while

10 La orden bc

El orden **bc** se utiliza como calculador de la línea de órdenes. Es similar a una **calculadora básica**. Con **bc** podemos hacer cálculos matemáticos básicos.

¿Cómo hacer el cálculo si sólo tengo la línea de órdenes?. Se puede hacer un cálculo muy complicado pasándolo mediante una tubería a la orden **bc**.

Algunos ejemplos son:

```
echo "56.8 + 77.7" | bc
```

Para convertir de decimal a hexadecimal, podemos ejecutar

```
echo "obase=16; ibase=10; 56" | bc
```

ibase es la base de entrada y obase es la base de salida, de este modo se puede convertir un número de una base a otra.

Para hacer divisiones en coma flotante, es necesario especificar la escala. La escala es el número de decimales. Por defecto, la escala es 0, que significa división entera.

```
echo "scale=6;  60/7.02" | bc
```

Podemos guardar el resultado de una operación en una variable:

```
resultado=$(echo "scale=3;2/3" | bc)
```

11 Depuración de guiones

La depuración de guiones puede resultar bastante compleja. No existen programas que permitan analizar los errores sintácticos de un guión. No obstante, las siguientes opciones de **bash** son una gran ayuda a la hora de depurar un guión:

$ bash –x guión
$ bash –v guión

La opción –x hace que **bash** ejecute *guión* asignando previamente la variable *echo*, esto provoca que se muestren en la pantalla las órdenes que va ejecutando el guión.

La opción –v hace que **bash** ejecute guión asignando previamente la variable *verbose*, esto provoca que se muestren en la pantalla las órdenes que va ejecutando el guión después de las sustituciones históricas.

También es posible introducir comentarios en un guión utilizando el carácter #, todo lo que aparezca en la línea tras # no es ejecutado. Tenga presente que el carácter # no se interpreta si se utiliza en un *shell* interactivo:

$ echo a # b *Produce sólo a*

12 Ejercicios propuestos

La edición de la pregunta no ha terminado. Por favor, haga click y edite o elimine la pregunte.

Ejercicio Resuelto
invarg0.txt (149 B)
invarg1.txt (216 B)
invarg2.txt (229 B)

La edición de la pregunta no ha terminado. Por favor, haga click y edite o elimine la pregunte.

Ejercicio Resuelto
sumarg.txt (173 B)

La edición de la pregunta no ha terminado. Por favor, haga click y edite o elimine la pregunte.

Ejercicio Resuelto
maxfich.txt (347 B)

La edición de la pregunta no ha terminado. Por favor, haga click y edite o elimine la pregunte.

Ejercicio Resuelto
numUID.txt (443 B)

La edición de la pregunta no ha terminado. Por favor, haga click y edite o elimine la pregunte.

Ejercicio Resuelto
permisos.txt (348 B)

13 Apéndice A. El filtro cut

La orden **cut** extrae porciones específicas de cada línea de entrada. Las porciones se pueden definir especificando las posiciones que ocupan los caracteres, o los campos que ocupan y el carácter que delimita los campos. La sintaxis de **cut** es:
$ cut opciones [fichero] ...

donde la entrada consiste en la concatenación de los ficheros especificados en fichero ... . Si no se especifican ficheros **cut** leerá de la entrada estándar. La entrada estándar también se puede especificar con '**-**' como *fichero*. La opción **–c** especifica extracción de caracteres, la opción **–f** especifica extracción de campos. Sólo puede estar presente una de estas opciones.

En ambos casos, el argumento de la opción especifica las porciones que se extraerán mediante una lista (indicada en lo que sigue como lista) que consiste en una secuencia de números sencillos e intervalos. El primer carácter o campo se representa mediante el 1 y los números deben estar en orden ascendente[1] . Los intervalos se indican con un par de números separados por '-'. Se puede omitir el primer número de un intervalo (para indicar el primer elemento) o el último (para representar el último elemento). Por ejemplo,

```
$ cut –c2,5-8,14-
```

extrae los caracteres 2, 5 a 8, y 14 a `l` de cada línea introducida por el teclado, donde `l` representa la longitud de la línea.

Extracción de caracteres. La opción siguiente se aplica para la extracción de caracteres:

- **-c*lista*** Extrae los caracteres especificados en *lista*. No debe haber un espacio entre **–c** y *lista*.

Nota: Si la longitud de la línea de entrada es menor que el mayor número de carácter especificado, **cut** ignora los caracteres faltantes.

Extracción de campos delimitados. Las opciones siguientes se aplican a la extracción de campos separados por delimitadores. El delimitador por defecto es el tabulador.

- **-f*lista*** extrae los campos especificados en *lista*. No debe haber un espacio entre **–f** y *lista*. Los campos se concatenan y envían a la salida estándar.
- **-d*c*** usa el carácter *c* como delimitador de campos, en lugar del tabulador.
- **-s** Suprime las líneas que no contienen caracteres delimitadores. Estas líneas se incluyen en la salida por defecto. La opción **–s** puede servir para suprimir líneas de cabecera en ficheros tabulares

Una secuencia de delimitadores seguidos define una secuencia de campos; los campos vacíos están permitidos y se reconocen. Es probable que esta situación no sea la deseada si el delimitador es un espacio. **cut** inserta el carácter delimitador entre campos consecutivos de la salida, incluso si están vacíos. Por ejemplo,

```
$ cut –f1,3-4 –d:
```

especifica que deben extraerse los campos 1, 3 y 4 de cada línea introducida por el teclado, usando ':' como carácter delimitador. Al aplicar esta orden a la línea de entrada

```
yo:digo::lo:que:quiero
```

se genera la salida

```
yo::lo
```

Nota: si el número de campos de la línea de entrada es menor que el mayor número de campo especificado en la línea de órdenes, **cut** ignora los campos que faltan, y no produce delimitadores para ellos.

***[1] cut** no hará nada si los números no están en orden ascendente. Sin embargo, la salida siempre contiene las porciones de la línea de entrada en el orden en que aparecen en la línea, de izquierda a derecha. Si se especifican los elementos de la lista en orden no ascendente no se aumentará la potencia de **cut**; únicamente se creará confusión con respecto a lo que será la salida.*