Programación en Shell

¿Por qué la programación en shell?

Incluso existiendo varios interfaces gráficos disponibles para Linux el shell sigue siendo una herramienta muy eficiente. El shell no es sólo una colección de comandos sino un buen lenguaje de programación. Se pueden automatizar muchas tareas con él, el shell es muy útil para tareas de administración, puedes comprobar rápidamente si tus ideas funcionan lo que lo hace muy útil cuando estás creando un prototipo y es muy útil para pequeñas utilidades que realizan tareas relativamente simples donde la eficiencia importa menos que la facilidad de configuración, mantenimiento y portabilidad.
Por lo tanto veamos como funciona:

Creando un script

Existen muchos shells disponibles para Linux pero habitualmente se utiliza el bash (bourne again shell) para la programación en shell ya que está disponible libremente y es fácil de usar. Por lo tanto todos los scripts que escribamos en este artículo usarán el bash (aunque la mayoría de veces también funcionarán con su hermano mayor, el bourne shell).
Para escribir nuestros programas en shell usaremos cualquier clase de editor de texto, p.ej. nedit, kedit, emacs, vi... como con otros lenguajes de programación.
El programa debe empezar con la siguiente línea (debe ser la primera línea del fichero):

```
#!/bin/sh
```

Lo caracteres #! indican al sistema que el primer argumento que sigue en la linea es el programa a utilizar para ejecutar este fichero. En este caso usamos el shell /bin/sh.
Cuando hallas escrito y guardado tu script debes hacerlo ejecutable para poder usarlo.
Para hacer el script ejecutable escribe
chmod +x nombre-del-fichero
Después puedes ejecutar tu script escribiendo: `./nombre-del-fichero`

Comentarios

Los comentarios en la programación en shell comienzan con # se prolongan hasta el final de la línea. Te recomendamos que uses comentarios. Si tienes comentarios y no usas un cierto script durante algún tiempo inmediatamente sabrás qué hace y cómo funciona.

Variables

Como en otros lenguajes de programación no se puede vivir sin variables. En la programación en shell todas las variables son de tipo string y no es necesario declararlas. Para asignar un valor a una variable se escribe:

```
nombre_de_la_variable=valor
```

Para obtener el valor de la variable simplemente hay que poner el signo del dólar delante de la variable:

```
#!/bin/sh
# asignamos un valor:
a="hola mundo"
# ahora mostramos el contenido de  "a":
echo "A es:"
echo $a
```

Escribe estas líneas en tu editor de textos y guardalo p.ej. como primero. Después haz ejecutable el script escribiendo chmod +x primero en el shell y después ejecútalo escribiendo ./primero
El script mostrará lo siguiente:

```
A es:
hola mundo
```

Algunas veces es posible confundir nombres de variables con el resto del texto:

```
num=2
echo "este es el $num°"
```

Aquí no se mostrará "este es el 2º" si no "este es el " porque el shell busca una variable llamada numdo que no tiene valor. Para decirle al shell que queremos mostrar la variable num tenemos que usar llaves:

```
num=2
echo "este es el ${num}°"
```

Con esto imprimiremos lo que queremos: este es el 2º

Hay una serie de variables que se establecen automaticamente. Hablaremos sobre ellas más adelante cuando las usemos por primera vez.

Si necesitas manejar expresiones matemáticas entonces necesitas usar programas como expr (ver tabla abajo). Además de las variables de shell normales que son únicamente válidas dentro del programa en shell hay también variables de entorno. Una variable precedida de la palabra clave export es una variable de entorno. No vamos a comentar nada más sobre ellas ya que normalmente sólo se utilizan en scripts de entrada al sistema.

Comandos de shell y estructuras de control

Hay tres categorías de comandos que pueden ser usados en scripts de shell:

1)Comandos Unix:
Aunque un script de shell puede hacer uso de cualquier comando unix aquí tienes varios comandos utilizados con más frecuencia que el resto. Estos comandos se decriben como comandos para la manipulación de ficheros y texto.

Sintaxis del comando	Utilidad
echo "texto cualquiera"	escribir un texto cualquiera en tu pantalla
ls	listar ficheros
wc -l fichero wc -w fichero wc -c fichero	contar las líneas de un fichero o contar las palabras de un fichero o contar el número de carácteres
cp fichero_origen fichero_destino	copiar el fichero_origen al fichero_destino
mv nombre_antiguo nuevo_nombre	renombrar o mover un fichero
rm fichero	borrar un fichero
grep 'patrón' fichero	buscar cadenas en un fichero Ejemplo: grep 'cadena-a-buscar' fichero.txt
cut -b num_de_columna fichero	sacar datos de columnas de texto de ancho definido Ejemplo: sacar los caracteres de la posición 5 a la 9 cut -b5-9 fichero.txt No confundais este comando con "cat" que es algo totalmente diferente
cat fichero.txt	mostrar un fichero.txt por stdout (tu pantalla)

file un_fichero	describir qué tipo de fichero es un_fichero
read var	esperar a que el usuario introduzca algo y guardarlo en la variable (var)
sort fichero.txt	ordenar las líneas del fichero fichero.txt
uniq	borrar las líneas duplicadas, usado en combinación con sort ya que uniq sólo borra las líneas duplicadas consecutivas Ejemplo: sort fichero.txt | uniq
expr	hacer operaciones matemáticas en el shell Ejemplo: sumar 2 y 3 expr 2 "+" 3
find	buscar ficheros Ejemplo: buscar por nombre: find . -name nombre_del_fichero -print Este comando tiene muchas posibilidades y opciones diferentes. Desafortunadamente son demasiadas para explicarlas todas en este artículo.
tee	escribir datos por stdout (tu pantalla) y a un fichero Normalmente se usa así: un-comando | tee fichero-de-salida Escribe la salida de un-comando por la pantalla y a el fichero de salida
basename fichero	devolver el nombre del fichero dado un nombre y quitarle la ruta al directorio Ejemplo: basename /bin/tux devuelve sólo tux
dirname fichero	devolver el nombre del directorio dado un nombre y quitándole el nombre del fichero Ejemplo: dirname /bin/tux devuelve /bin
head fichero	mostrar algunas líneas del principio de un fichero
tail fichero	mostrar algunas líneas del final de un fichero
sed	sed es básicamente un programa para buscar y reemplazar. Lee el texto de la entrada estándar (p.ej. una tubería) y escribe el resultado por stdout (normalmente la pantalla). El patrón de búsqueda es una expresión regular (ver referencias). Este patrón de búsqueda de se debe confundir con la sintaxis de comodines (wildcard) del shell. Para reemplazar la cadena linuxfocus con LinuxFocus es un fichero de texto usa: cat texto.fichero | sed 's/linuxfocus/LinuxFocus/' > nuevotexto.fichero Esto reemplaza la primera aparición de la cadena linuxfocus en cada línea por LinuxFocus. Si hay líneas en que linuxfocus aparece varias veces y quieres reemplazarlas todas usa: cat texto.fichero | sed 's/linuxfocus/LinuxFocus/g' > nuevotexto.fichero

awk	La mayoría de las veces awk se utiliza para extraer campos de una línea de un texto. El campo separador por defecto es un espacio. Para especificar uno diferente se usa la opción -F.
	`cat fichero.txt | awk -F, '{print $1 "," $3 }'`
	
	Aquí hemos usado la coma (,) como separador de campos e imprimimos la columna primero y tercera ($1 $3). Si fichero.txt tiene líneas como:
	`Adam Bor, 34, India`
	`Kerry Miller, 22, USA`
	
	estonces mostraría lo siguiente:

	Adam Bor, India Kerry Miller, USA Se pueden hacer muchas más cosas con awk pero este es su uso más común.

2) Conceptos: Tuberías (pipes), redirecciones y comillas simples invertidas
Realmente no son comando pero son conceptos muy importantes.

tuberías (pipes) (|) envian la salida (stdout) de un programa a la entrada (stdin) de otro.

```
    grep "hola" fichero.txt | wc -l
```

encuentra las líneas con la cadena hola en fichero.txt y después cuenta el número de líneas.
La salida del comando grep se usa como entrada del comando wc. De esta manera se pueden contatenar tantos comandos como se quiera (dentro de unos límites razonables).

redirecciones: escribe la salida de un comando a un fichero o añade datos a un fichero
> escribe la salida a un fichero y sobreescribe el fichero antiguo en caso de existir
>> añade datos a un fichero (o crea uno nuevo si no existía previamente pero nunca sobreescribe nada).

Comillas simples invertidas (o tilde invertida)
La salida de un comando se puede usar como argumento de línea de comandos (no stdin como anteriormente, los argumentos de línea de comando son cadenas que se especifican destrás de un comando como con los nombres de ficheros y opciones) para otro comando. También se pueden usar para asignar la salida de un comando a una variable.
El comando

```
find . -mtime -1 -type f -print
```

encuetra todos los ficheros que han sido modificados en las últimas 24 horas (-mtime -2 serían 48 horas). Si quisieramos empaquetar estos fichero en un archivo tar (fichero.tar) la sintaxis sería:

```
tar xvf fichero.tar fichero1 fichero2 ...
```

En vez de escribirlo todo se pueden combinar los dos comandos (find y tar) usando comillas simples invertidas. Entonces tar empaquetará todos los ficheros que muestre find:

```
#!/bin/sh
# Las comillas son comillas simples invertidas (`)  no comillas normales ('):
tar -zcvf ultimodific.tar.gz `find . -mtime -1 -type f -print`
```

3) Estructuras de control
La **sentencia "if"** comprueba si una condición es verdadera (con salida de estado 0, correcto). Si es correcto entonces se ejecuta la parte "then":

```
if ....; then
   ....
elif ....; then
   ....
else
   ....
fi
```

La mayoría de las veces se utiliza un comando muy especial dentro de las sentencias if. Se puede usar para comparar cadena o comprobar si un fichero existe, tiene permiso de lectura etc...
El comando "test" se escribe como corchetes " [ ] ". Aquí los espacios son muy importantes: Asegúrate que siempre hay un espacio entre corchetes. Ejemplos:

```
[ -f "un-fichero" ]  : Comprueba si un-fichero es un fichero.
[ -x "/bin/ls" ]   : Comprueba si /bin/ls existe y es ejecutable.
[ -n "$var" ]      : Comprueba si la variable $var contiene algo
[ "$a" = "$b" ]    : Comprueba si las variables "$a" y  "$b" son iguales
```

Ejecuta el comando "man test" y obtendrás una larga lista con todo tipo de operadores test para comparaciones y ficheros.
Usarlo en scripts de shell es bastante directo:

```
#!/bin/sh
if [ "$SHELL" = "/bin/bash" ]; then
  echo "tu shell es el bash (bourne again shell)"
else
  echo "tu shell no es bash sino $SHELL"
fi
```

La variable $SHELL contiene el nombre de la shell donde estás y esto es lo que estamos comprobando aquí comparandola con la cadena "/bin/bash"

Operadores cortos

A aquellos que conozcan C les gustará la siguiente expresión:

```
[ -f "/etc/shadow" ] && echo "Este ordenador usa shadow passwors"
```

Los && se pueden usar como una pequeña sentencia if. Lo de la derecha se ejecuta si lo de la izquierda es verdadero. Podemos interpretarlo como un Y (AND). Por lo tanto el ejemplo quedaría: "El fichero /etc/shadow existe Y (AND) se ejecuta el comando echo". También está disponible el operador O (OR) (||). Un ejemplo:

```
#!/bin/sh
mailfolder=/var/spool/mail/james
[ -r "$mailfolder" ] || { echo "No se ha podido leer $mailfolder" ; exit 1; }
echo "$mailfolder tiene mensajes de:"
grep "^From " $mailfolder
```

El script comprueba primero si puede leer un buzón de correo. Si puede entonces imprime las líneas "From" del buzón. Si no puede leer el fichero $mailfolder entonces se activa el operador O. Sencillamente este código lo leeríamos como "Mailfolder legible o salir del programa". Aquí el problema es que debemos tener exactamente un comando detrás de O pero en este caso necesitamos dos:
-mostrar un mensaje de error
-salir del programa
Para manejar ambos como un solo comando podemos agruparlos en una función anónima usando llaves. Las funciones en general se explican con más profundidad más abajo.
Puedes hacerlo todo sin O e Ys usando simplemente sentencias if pero algunas veces los operadores O e Y son mucho más convenientes.

La sentencia **case** se puede usar para comparar (usando comodines del shell como * y ?) una cadena dada con varias posibilidades.

```
case ... in
...) hacer algo aquí;;
esac
```

Veamos un ejemplo. El comando file comprueba que tipo de fichero es el fichero que le pasamos:

```
file lf.gz

devuelve:

lf.gz: gzip compressed data, deflated, original filename,
last modified: Mon Aug 27 23:09:18 2001, os: Unix
```

Ahora vamos a usar esto para escribir un script llamado smartzip que puede descomprimir ficheros comprimidos con bzip2, gzip y zip automaticamente :

```
#!/bin/sh
tipofichero=`file "$1"`
case "$tipofichero" in
"$1: Zip archive"*)
    unzip "$1" ;;
"$1: gzip compressed"*)
    gunzip "$1" ;;
"$1: bzip2 compressed"*)
    bunzip2 "$1" ;;
```

```
*) error "El fichero $1 no se puede descomprimir con smartzip";;
esac
```

Te habrás fijado que hemos usado una nueva variable especial llamada $1. Esta variable contiene el primer argumento pasado a un programa. Digamos que ejecutamos
smartzip articles.zip
entonces $1 contendría la cadena articles.zip

La sentencia **select** es una extensión específica del bash y es muy útil para usos interactivos. El usuario puede escoger una opción de una lista de difirentes valores:

```
select var in ... ; do
  break
done
.... ahora podemos usar $var ....
```

Un ejemplo:

```
#!/bin/sh
echo "¿Cuál es tu sistema operativo favorito?"
select var in "Linux" "Gnu Hurd" "Free BSD" "Otros"; do
        break
done
echo "Has seleccionado $var"
```

Aquí tienes lo que haría el programa:

```
¿Cuál es tu sistema operativo favorito?
1) Linux
2) Gnu Hurd
3) Free BSD
4) Otros
#? 1
Has seleccionado Linux
```

En el shell tenemos están disponibles las siguientes **sentencias de bucle**:

```
while ...; do
 ....
done
```

El **bucle while** se ejecutará mientras la expresión que comprobamos sea verdadera. Se puede usar la palabra clave "break" para abandonar el bucle en cualquier punto de la ejecución. Con la palabra clave "continue" el bucle continua con la siguiente iteración y se salta el resto del cuerpo del bucle.

El **bucle for** toma una lista de cadenas (separadas por espacios) y las asigna a una variable:

```
for var in ....; do
  ....
done
```

El siguente ejemplo muestra por pantalla de la letra A a la C:

```
#!/bin/sh
for var in A B C ; do
  echo "var es $var"
done
```

Un script de ejemplo más útil es este, llamado showrpm, que muestra un resumen del contenido de una serie de paquetes RPM:

```
#!/bin/sh
# listar un resumen del contenido de una serie de paquetes RPM
# USO: showrpm ficherorpm1 ficherorpm2 ...
# EJEMPLO: showrpm /cdrom/RedHat/RPMS/*.rpm
for rpmpackage in $*; do
  if [ -r "$rpmpackage" ];then
    echo "=============== $rpmpackage ==============="
    rpm -qi -p $rpmpackage
  else
    echo "ERROR: no se pudo leer el fichero $rpmpackage"
  fi
```

```
done
```
Como se puede ver hemos usado la siguente variable especial, $* que contiene todos los argumentos de la línea de comandos. Si ejecutas
showrpm openssh.rpm w3m.rpm webgrep.rpm
entonces $* contiene las tres cadenas openssh.rpm, w3m.rpm y webgrep.rpm.

El bash GNU también entiende bucles until pero en general con los bucles while y for es suficiente.

Comillas
Antes de pasarle cualquier argumento a un programa el shell intenta expandir los comodines y variables. Expandir significa que el comodín (p.ej. *) se reemplaza por los nombres de fichero apropiados o que una variable se reemplaza por su valor. Para modificar este comportamiento se pueden usar las comillas: Pongamos que tenemos varios ficheros en el directorio actual. Dos de ellos son ficheros jpg, correo.jpg y tux.jpg.
```
#!/bin/sh
echo *.jpg
```
Imprimirá "correo.jpg tux.jpg".
Usar comillas (simples y dobles) evitará esta expansión de comodines:
```
#!/bin/sh
echo "*.jpg"
echo '*.jpg'
```
Esto mostrará "*.jpg" dos veces.
Las comillas simples son las más estrictas. Evitan incluso la expansión de variables. Las comillas dobles evitarán la exprasión de comodines pero no así la de variables:
```
#!/bin/sh
echo $SHELL
echo "$SHELL"
echo '$SHELL'
```
Esto mostrará:
```
/bin/bash
/bin/bash
$SHELL
```
Finalmente existe la posibilidad de eliminar el significado especial de un único caracter anteponiendole una barra invertida:
```
echo \*.jpg
echo \$SHELL
```
Esto mostrará:
```
*.jpg
$SHELL
```
Documentación aquí
Documentación aquí es una forma muy elegante de enviar varias líneas de texto a un comando. Es bastante útil para escribir un texto de ayuda en un script sin tener que poner echo delante de cada línea. Un "Documentación aquí" comienza por << seguido de una cadena que también debe aparecer al final del documentación aquí. Aquí tenemos un script de ejemplo, llamado ren, que renombra múltiples ficheros y usa documentación aquí para su texto de ayuda:
```
#!/bin/sh
# tenemos menos de 3 argumentos. Mostramos el texto de ayuda:
if [ $# -lt 3 ] ; then
cat <<AYUDA
ren -- renombra varios ficheros usando expresiones regulares de sed

USO: ren 'regexp' 'reemplazo' ficheros...

EJEMPLO: rename all *.HTM fles in *.html:
  ren 'HTM$' 'html' *.HTM

AYUDA
  exit 0
```

```
fi
VIEJO="$1"
NUEVO="$2"
# El comando shift elimina un argumento de la lista
# de argumentos de la linea de comandos.
shift
shift
# ahora $* contiene todos los ficheros:
for fichero in $*; do
    if [ -f "$fichero" ] ; then
      nuevofichero=`echo "$fichero" | sed "s/${VIEJO}/${NUEVO}/g"`
      if [ -f "$nuevofichero" ]; then
        echo "ERROR: $nuevofichero ya existe"
      else
        echo "renombrando $fichero como $nuevofichero ..."
        mv "$fichero" "$nuevofichero"
      fi
    fi
done
```

Hasta ahora este es el script más complejo que hemos visto. Hablemos sobre él un poco. La primera sentencia if comprueba si hemos introducido al menos 3 parámetros de línea de comandos. (La variable especial $# contiene el número de argumentos). Si no, se envia el texto de ayuda al comando cat que consecuentemente lo muestra por pantalla. Después de mostrar el texto de ayuda salimos del programa. Si hay 3 o más argumentos asignamos el primer argumento a la variable VIEJO y el segundo a la variable NUEVO. Lo siguiente es cambiar la posición de los parámetros de la lista de comandos dos veces para poner el tercero en la primera posición de $*. Con $* entramos en el bucle for. Cada uno de los argumentos de $* se asigna uno a uno a la variable $fichero. Aquí comprobamos primero que el fichero existe y después creamos un nuevo fichero usando buscar y reemplazar con sed. Las comillas simples invertidas se usan para asignar el resultado a la variable nuevofichero. Ya tenemos todo lo que necesitamos: El nombre de fichero de viejo y el nuevo. Ahora sólo tenemos que utilizarlos con el comando mv para renombrar los ficheros.

Funciones

Tan pronto como tengamos un programa más complejo nos encontraremos con que estamos usando el mismo código en varias partes y que sería de más ayuda el darle cierta estructura. Una función es similar a esto:

```
nombredelafuncion()
{
 # dentro del cuerpo $1 es el primer argumento dado a la función
 # $2 el segundo ...
 cuerpo
}
```

Se necesita "declarar" las funciones al principio del script antes de usarlas.

Aquí tenemos un script llamado xtitlebar (título de la ventana) que puedes usar para cambiar el nombre de una ventana de terminal. Si tienes varias abiertas es más fácil encontrar una. El script envia una secuencia de escape que es interpretada por el terminal y hace que este cambie su nombre el la barra de título. El script usa una función llamada ayuda. Como se puede ver la función se define una vez y después se utiliza dos veces:

```
#!/bin/sh
# vim: set sw=4 ts=4 et:

ayuda()
{
    cat <<AYUDA
xtitlebar -- cambiar el nombre de un xterm, gnome-terminal o konsole de kde

USO: xtitlebar [-h] "cadena_para_la_barra_de_titulo"

OPCIONES: -h texto de ayuda
```

```
EJEMPLO: xtitlebar "cvs"

AYUDA
    exit 0
}

# en caso de error o si le pasamos -h llamar a la función ayuda:
[ -z "$1" ] && ayuda
[ "$1" = "-h" ] && ayuda

# enviamos la secuencia de escape para cambiar el título de la ventana:
echo -e "\033]0;$1\007"
#
```

Se recomienda siempre documentar detalladamente el código de los scripts. Con esto posibilitamos que otras personas (y tú) puedan usar y entender el script.

Argumentos de línea de comandos

Hemos visto que $* y $1, $2 ... $9 contienen los argumentos que el usuario especificó en la línea de comandos (las cadenas escritas detrás del nombre del programa). Hasta ahora sólo hemos tratado con sintaxis de linea de comandos bastante sencilla y corta (un par de argumentos obligatorios y la opción -h para ayuda). Pero pronto descubrirás que necesitas algún tipo de analizador para programas más complejos donde defines tus propias opciones. Por convención se dice que los parámetros opcionales deben estar precedidos por un signo menos y deben ir antes de otros argumentos (como p.ej. nombres de ficheros).

Hay muchas posibilidades para implementar un analizador (parser). El bucle while siguiente combiando con una sentencia case es una solución muy buena para un analizador genérico:

```
#!/bin/sh
ayuda()
{
  cat <<AYUDA
Esta es la demostración de un analizador de línea de comandos genérico.
EJEMPLO DE USO: cmdparser -l hola -f -- -fichero1 fichero2
AYUDA
  exit 0
}
while [ -n "$1" ]; do
case $1 in
    -h) ayuda;shift 1;; # llamamos a la función ayuda
    -f) opt_f=1;shift 1;; # la variable opt_f existe
    -l) opt_l=$2;shift 2;; # -l toma un argumento -> cambiado 2 veces
    --) shift;break;; # end of options
    -*) echo "error: no existe la opción $1. -h para ayuda";exit 1;;
    *)  break;;
esac
done

echo "opt_f es $opt_f"
echo "opt_l es $opt_l"
echo "el primer argumento es $1"
echo "el 2° argumento es $2"
```

¡Pruébalo! Puedes ejecutarlo con p.ej.:

```
cmdparser -l hola -f -- -fichero1 fichero2
```

Produce

```
opt_f es 1
opt_l es hola
el primer argumento es -fichero1
el 2° argumento es fichero2
```

¿Cómo funciona? Basicamente se realiza un bucle a través de todos los argumentos y los compara con la sentencia case. Si encuentra alguna coincidencia establece una variable y cambia la línea de comandos en uno.

Por convención en unix las opciones (lo que tiene delante un signo menos) deben estar primero. Puedes indicar que han finalizado las opciones escribiendo dos signos menos (--). Lo necesitarás p.ej. con grep para buscar una cadena que comience con el signo menos:

```
Buscar la cadena -xx- en el fichero f.txt:
grep -- -xx- f.txt
```

Nuestro analizador de opciones también puede manejar los -- como se puede comprobar en el listado superior.

Ejemplos

Un esqueleto de propósito general

Hasta ahora hemos hablado de casi todos los componentes que se necesitan para escribir un script. Todos los buenos scripts deberían tener ayuda y también un buen analizador genérico de opciones incluso aunque el script tenga una sola opción. Por lo tanto es un buena idea tener un script básico, llamado framework.sh (framework = armazón), que se puede usar como esqueleto para otros scripts. Si se quiere escribir un nuevo script simplemente se tendría que hacer una copia:

```
cp framework.sh miscript
```

y después insertar las funcionalidades necesarias en "miscript".

Veamos ahora dos ejemplos más:

Un conversor numérico de binario a decimal

El script b2d convierte un número binario (e.g 1101) a su equivalente en decimal. Es un ejemplo que muestra cómo hacer sencillas operaciones matemáticas con expr:

```
#!/bin/sh
# vim: set sw=4 ts=4 et:
ayuda()
{
  cat <<AYUDA
b2d -- conversor binario a decimal

USO: b2d [-h] numeroenbinario

OPCIONES: -h texto de ayuda

EJEMPLO: b2d 111010
devolverá 58
AYUDA
  exit 0
}

error()
{
    # mostrar un error y salir
    echo "$1"
    exit 1
}

ultimocarac()
{
    # devolver el último caracter de una cadena en $rval
    if [ -z "$1" ]; then
        # cadena vacía
        rval=""
        return
    fi
    # wc deja espacio antes de su salida y por eso necesitamos usar sed:
```

```
    numdcarac=`echo -n "$1" | wc -c | sed 's/ //g' `
    # ahora cortamos el último caracter
    rval=`echo -n "$1" | cut -b $numdcarac`
}

chop()
{
    # borrar el último caracter de la cadena y retornarno como $rval
    if [ -z "$1" ]; then
        # cadena vacía
        rval=""
        return
    fi
    # wc deja espacio antes de su salida y por eso necesitamos usar sed:
    numdcarac=`echo -n "$1" | wc -c | sed 's/ //g' `
    if [ "$numdcarac" = "1" ]; then
        # sólo hay un carácter en la cadena
        rval=""
        return
    fi
    numdcaracmenos1=`expr $numdcarac "-" 1`
    # ahora cortamos todos excepto el último:
    rval=`echo -n "$1" | cut -b 0-${numdcaracmenos1}`
}


while [ -n "$1" ]; do
case $1 in
    -h) ayuda;shift 1;; # llamamos a la función ayuda
    --) shift;break;; # fin de las opciones
    -*) error "error: no existe la opción $1. -h para ayuda";;
    *)  break;;
esac
done

# El programa principal
suma=0
peso=1
# hay que pasar un argumento:
[ -z "$1" ] && ayuda
numbin="$1"
numbinorig="$1"

while [ -n "$numbin" ]; do
    ultimocarac "$numbin"
    if [ "$rval" = "1" ]; then
        suma=`expr "$peso" "+" "$suma"`
    fi
    # borrar la última posición en $numbin
    chop "$numbin"
    numbin="$rval"
    peso=`expr "$peso" "*" 2`
done

echo "el binario $numbinorig en decimal es $suma"
#
```

El algoritmo usado en este script toma el peso decimal (1,2,4,8,16,..) de cada dígito comenzando por el dígito más a la derecha y lo añade a la suma si el dígito es un 1. De este modo "10" es:

0 * 1 + 1 * 2 = 2

Para sacar lo dígitos de la cadena usamos la función ultimocarac. Esta usa wc -c para contar el número de caracteres en la cadena y después cut para cortar el último caracter. La función chop sigue la misma lógica pero borra el último caracter, es decir, corta todo desde el principio hasta el caracter anterior al último.

Un programa para rotar ficheros

Quizá eres uno de aquellos que guardan todo en correo enviado en un fichero. Después de un par de meses este fichero se ha hecho bastante grande y si lo cargas en tu programa de correo el acceso a él es muy lento. El siguiente script rotatefile puede ayudarte. Renombra el buzón de correo, llamémoslo correoenviado, a correoenviado.1 si ya habían un correoenviado.1 entonces se convierte en correoenviado.2 etc...

```
#!/bin/sh
# vim: set sw=4 ts=4 et:
ver="0.1"
ayuda()
{
    cat <<AYUDA
rotatefile -- rotar el nombre de un fichero

USO: rotatefile [-h]  nombre_del_fichero

OPCIONES: -h texto de ayuda

EJEMPLO: rotatefile algo
Esto p. ej. renombrará algo.2 a algo.3, algo.1 a algo.2, algo a algo.1
y creará un fichero vacío llamado algo.

El número máximo es 10

version $ver
AYUDA
    exit 0
}

error()
{
    echo "$1"
    exit 1
}
while [ -n "$1" ]; do
case $1 in
    -h) ayuda;shift 1;;
    --) break;;
    -*) echo "error: no existe la opción $1. -h para ayuda";exit 1;;
    *)  break;;
esac
done

# comprobamos la entrada:
if [ -z "$1" ] ; then
 error "ERROR: debe especificar un fichero, use -h para obtener ayuda"
fi
nomfich="$1"
# renombrar cualquier fichero .1 , .2 etc:
for n in  9 8 7 6 5 4 3 2 1; do
    if [ -f "$nomfich.$n" ]; then
        p=`expr $n + 1`
        echo "mv $nomfich.$n $nomfich.$p"
        mv $nomfich.$n $nomfich.$p
    fi
done
# renombrar el fichero original:
if [ -f "$nomfich" ]; then
    echo "mv $nomfich $nomfich.1"
    mv $nomfich $nomfich.1
fi
echo touch $nomfich
```

```
touch $nomfich
```

¿Cómo funciona el programa? Después de comprobar que el usuario ha proporcionado un nombre de fichero entramos en un bucle for desde 9 a 1. El fichero 9 se renombra como 10, el 8 como 9 así con todos. Después del bucle renombramos el fichero original como 1 y creamos un fichero vacío con el nombre del fichero original.

Depuración

La ayuda para depurar más sencilla es por supuesto el comando echo. Lo puedes usar para imprimir variables específicas en en lugar donde sospeches que está el fallo. Esto es probablemente lo que usan la mayoría de programadores de shell el 80% de las veces que intentan encontrar un error. La ventaja de la programación en shell es que no requiere ninguna recompilación e insertar una sentencia "echo" se hace rápidamente.

El shell también tiene un modo de depuración real. Si hay un error en tu script "scriptconerror" entonces puedes depurarlo con:

```
sh -x scriptconerror
```

Esto ejecutará el script y mostrará todas la sentencias que se ejecutan con las variables y comodines ya expandidos.

El shell también tiene un modo para comprobar errores de sintaxis sin ejecutar el programa. Se usa así:

```
sh -n tu_script
```

Si no retorna nada entonces tu programa no tiene errores de sintaxis.

Esperamos que ahora comiences a escribir tus propios scripts de shell. ¡Diviértete!