

2. Shell Scripts

La administración de red es una ardua, pero hermosa tarea; como administrador se debe conocer y tener a la mano un conjunto de herramientas o utilerías para prevenir, evitar y corregir problemas en la red. El administrador debe conocer, controlar y planificar los procesos del sistema, conocer los archivos donde se almacenan información de los usuarios, compartir archivos, controlar recursos, conocer e instalar herramientas de monitoreo, conocer e instalar herramientas para realizar encriptación, controlar el acceso al sistema, y muchas otras actividades extras; como se puede notar la labor de un administrador es una tarea titánica, y es mejor y más recomendado que estas tareas recaigan en un grupo de personas y no en una sola.

Para empezar este capítulo, hablaremos primero del shell del sistema, en que consiste y algunos comandos que consideramos importantes, en la siguiente sección, tocaremos el punto de su programación, y en la última sección hablaremos de la programación con el lenguaje AWK.

2.1 Introducción al shell

Cuando observamos una pantalla en la cual se esta ejecutando el sistema UNIX, lo que en verdad vemos es un programa en ejecución, el cual esta monitoreando y respondiendo a las ordenes que damos desde teclado. Este programa es llamado el *shell* del sistema.

Como el shell responde a un comando dado por el usuario (usuario ordinario o administrador) podemos editar (en vi, pico, emacs o algún otro editor de texto) un conjunto de comandos para automatizar alguna tarea, este programa fuente se le conoce como un *script*.

Muchos sistemas UNIX, incluyendo AT&T's System V (SV), XENIX, Berkeley Software Distribution (BSD), Ultrix, Linux, y SunOS, son distribuciones que cuentan por lo menos con 200 programas, comúnmente llamados *comandos* o *utilerías*. Para combinar dos o más utilerías los usuarios de UNIX pueden crear sus propios programas de comandos. El shell de UNIX proporciona un simple pero muy poderoso mecanismo para construir herramientas que los programados o no programadores pueden utilizar para automatizar tareas, la programación de script cuenta además con un conjunto de constructores como *loops*, condicionales y variables.

2.1.1 El ambiente del sistema UNIX

Antes de empezar a hablar de la programación del shell, veamos en que consiste el sistema operativo UNIX. El sistema operativo UNIX consiste de un *kernel*, un *shell* y cientos de comandos o utilerías. El kernel es la parte que interactúa directamente con el hardware, este administra la memoria, controla los dispositivos de entrada y salida, y proporciona una serie de servicios a través del shell. El shell interactúa con el kernel de modo transparente para el usuario. Así la combinación del kernel, el shell, los comandos y el hardware es lo que forma al sistema UNIX.

UNIX proporciona un *ambiente* de trabajo que consiste de dos grandes subsistemas:

- El sistema de archivos, el cual es una definición física, conceptual y organizacional de los archivos en el sistema, y

- El sistema de control de procesos, el cual se debe conocer para poder interactuar con el resto del ambiente, su tarea principal es relacionar un programa (especialmente el shell) y otros programas en ejecución.

2.1.2 El shell de UNIX y el usuario

El shell es uno de los programas que se distribuyen con el sistema UNIX, este se encarga de interpretar las órdenes que colocamos desde el teclado o desde un script.

Cuando nos conectamos a una computadora con UNIX, el proceso *login* invoca al programa de shell llámese Bourne shell, korn shell, C shell, etc, esto depende del sistema y del administrador, en la tabla 2-1 se muestran los shells típicos de UNIX. El shell de cada usuario esta determinado en una de las entradas del archivo */etc/passwd*, en la figura 2-1 se muestra parte de un archivo *passwd* y se coloca con negrita el shell de cada usuario. El shell se encarga de establecer y mantener el ambiente de trabajo, el cual esta definido en el archivo *.profile*. El *.profile* es un claro ejemplo de un script, en la figura 2-2 se muestra un archivo de este tipo.

Tabla 2-1. Los shells del sistema UNIX

Shell	Descripción
bash (Bourne-Again Shell)	Versión GNU del shell estándar con características añadidas del shell C. Este shell es el estándar de los sistemas LINUX.
Csh	El shell C de Berkeley.
Jsh	El shell de trabajo, una extensión del shell estándar.
Ksh	El shell Korn.
Sh	El tradicional shell de Bourne.
Tcsh	Versión mejorada de csh.
Zsh	Versión mejorada de ksh.

```

hsuarez:x:202:202:Hugo Suarez Onofre:/export/home/usuarios/hsuarez:/usr/bin/csh
ic981476:x:1476:211:Cruz Perez Javier:/export/home/alumnos/ic981476:/bin/csh
rglist:x:242:220:red-utm:/export/home/listas/rglist:/bin/csh
pglist:x:248:220:red-utm:/export/home/listas/pglist:/bin/csh
gcgero:x:373:201:Gabriel Geronimo Castillo:/export/home/usuarios/gcgero:/bin/ksh
bibliote:x:405:10:Proyecto Biblioteca:/opt/bibliote:/bin/csh
ohayna:x:1829:211:Otilia Hayna:/export/home/alumnos/ohayna:/bin/csh
ercomar:x:1287:211:Proyecto de Energia:/export/home/alumnos/ercomar:/bin/csh
id971355:x:1355:211:Montes Velasco Emma:/export/home/alumnos/id971355:/bin/csh
id941402:x:1402:211:Maur Cruz James:/export/home/alumnos/id941402:/bin/csh
ecg:x:396:207:Enrique Contreras Gonzalez:/export/home1/usuarios/ecg:/bin/sh
    
```

Figura 2-1. Parte de un archivo *passwd*.

```

# archivo .profile
# @(#)local.profile 1.4 93/09/15 SMI
#
#setenv DISPLAY 192.100.170.30:0.0
PS1="SUNOS@.mixteco>"
stty istrip
PATH=/usr/bin:/usr/ucb:/etc:.
export PATH
#setenv DISPLAY 192.100.170.30:0.0
#
# If possible, start the windows system
#
if [ `tty` = "/dev/console" ] ; then
    if [ "$TERM" = "sun" -o "$TERM" = "AT386" ] ; then

        if [ ${OPENWINHOME:-""} = "" ] ; then
            OPENWINHOME=/usr/openwin
            export OPENWINHOME
        fi
        echo ""
        echo "Starting OpenWindows in 5 seconds (type Control-C to interrupt)"
        sleep 5
        echo ""
        $OPENWINHOME/bin/openwin
        clear      # get rid of annoying cursor rectangle
        exit      # logout after leaving windows system
    fi
fi

```

Figura 2-2. Ejemplo de un archivo .profile

El shell cuenta además con un conjunto de variables de ambiente que se fijan automáticamente al inicio de una sesión o las puede fijar el usuario, esto con la finalidad de facilitar y personalizar el ambiente de trabajo, el comando *set* se puede utilizar para visualizar las variables actuales del shell y sus valores.

```

SUNOS@.mixteco>set
ERRNO=9
FCEDIT=/bin/ed
HOME=/export/home/usuarios/gcgero
HZ=100
IFS='
'
LC_COLLATE=es
LC_CTYPE=es
LC_MESSAGES=es
LC_MONETARY=es_MX
LC_NUMERIC=es
LC_TIME=es
LINENO=1
LOGNAME=gcgero
MAIL=/var/mail/gcgero

```

```
MAILCHECK=600
OPTIND=1
PATH=/usr/bin:/usr/ucb:/etc:.
PPID=14034
PS1='SUNOS@.mixteco>'
PS2='> '
PS3='#? '
PS4='+ '
PWD=/export/home/usuarios/gcgero
RANDOM=23554
SECONDS=7
SHELL=/bin/ksh
TERM=vt100
TMOUT=0
TZ=Mexico/General
_PATH
```

Las variables de shell comunes se muestran en la tabla 2-2. Algunas veces se necesita obtener el valor de una variable del shell, para esto debemos de colocar antes del nombre de la variable el signo \$, y para ver el valor de la variable utilizamos la orden *echo*.

```
SUNOS@.mixteco>echo $HOME
/export/home/usuarios/gcgero
```

Las variables de ambiente las podemos utilizar para no tener que teclear un camino completo, por ejemplo si está ubicados en el directorio /etc y queremos regresar a nuestro directorio de trabajo podemos utilizar la variable HOME:

```
SUNOS@.mixteco>pwd
/etc
SUNOS@.mixteco>cd $HOME
SUNOS@.mixteco>pwd
/export/home/usuarios/gcgero
SUNOS@.mixteco>
```

Tabla 2-2. Variable de ambientes habituales.

Variable de shell	Descripción	Ejemplo	
HOME	Nombre del camino de su directorio de trabajo	HOME= /export/home/usuarios/gcgero	Fijado por el usuario.
LOGNAME	Nombre de usuario	LOGNAME=gcgero	Fijado automáticamente en el inicio de la sesión.
MAIL	Nombre del camino que contiene su correo.	MAIL=/var/mail/gcgero	Utilizado por el shell para notificar el correo.
MAILCHECK=600	Checa cambios en los parámetros de los archivos especificados por MAILPATH o MAIL.	MAILCHECK=600	Frecuentemente checa cada 600 segundos (10 minutos).
PATH	Lista de directorios	PATH=/usr/bin:/usr/ucb:/etc:.	Fijada automáticamente

	donde el shell busca las órdenes.		al iniciar la sesión.
PS1	Indicador primario del shell	PS1='SUNOS@.mixteco>'	Fijado automáticamente o por el usuario.
PS2	Indicador secundario del shell	PS2='> '	Por omisión de >
SHELL	Ruta de su shell.	SHELL=/bin/ksh	Fijada automáticamente
TERM	Define su tipo de terminal para <i>vi</i> , <i>pine</i> y otras órdenes orientadas a pantalla.	TERM=vt100	Fijada por el usuario.
TZ	Información de la zona de horario.	TZ=Mexico/General	Fijada y utilizada por el sistema.
IFS	Utilizado para separar comandos, opciones y argumentos; usualmente es un espacio, un tab, o una nueva línea.	IFS=' '	Usualmente es cambiado por los usuarios.
PWD	Contiene el nombre del directorio de trabajo.	PWD=/export/home/usuarios/gcgero	El usuario lo puede fijar. Esta variable se encuentra en el shell korn.
OLDPWD	Contiene el nombre del directorio anterior.		Esta variable se encuentra en el shell korn.
RANDOM	Contiene un entero aleatorio.	RANDOM=23554	Toma una distribución uniforme entre 0 y 32767. Esta variable se encuentra en el shell korn.
SECONDS	Contiene el tiempo transcurrido desde el inicio de la sesión.	SECONDS=7	Esta variable se encuentra en el shell korn.

Existen también unas variables especiales que son muy útiles en el momento de programar el shell, estas variables son:

- # Utilizada para comprobar si existen argumentos de líneas de órdenes, y si es así cuantos existen.
- ? Contiene el valor devuelto por la última orden ejecutada; cuando se ejecuta una orden, esta devuelve un número al shell, 0 en caso de una orden con éxito, y un valor diferente de 0 en caso de que la orden falle.
- ! Contiene el ID (identificador) del último proceso subordinado (en background).

Las variables de shell son utilizadas fundamentalmente por los programas, incluyendo el propio shell, sin embargo, puede definir también nuevas variables de shell para su propio uso. Por ejemplo, si mueve con frecuencia archivos a un directorio particular, puede definir una variable con el nombre del directorio.

```
SUNOS@.mixteco> DEST = "\home\ usuario1\redes\script"
```

Ahora, ya puede mover archivos desde cualquier punto del sistema escribiendo

```
SUNOS@.mixteco> mv file $DEST
```

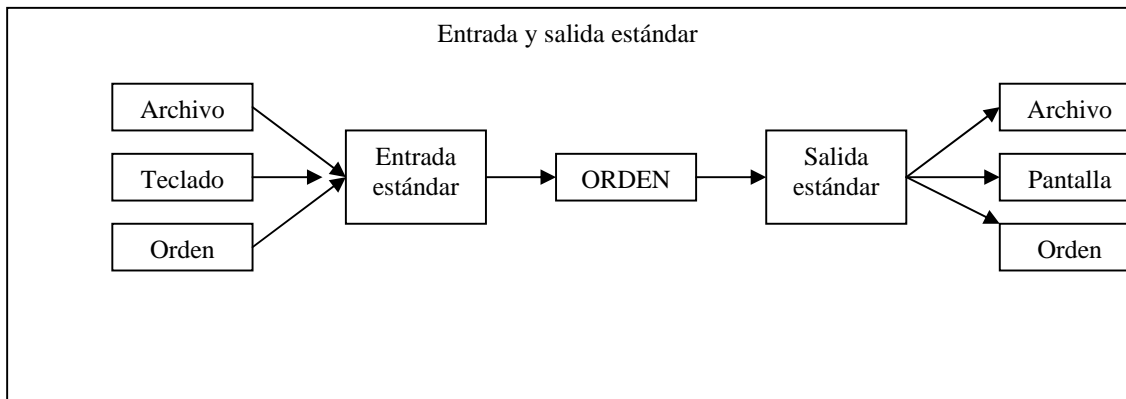
Si queremos referirnos a un simple archivo usualmente tenemos que colocar su nombre de entrada, pero si queremos referirnos a varios archivos en una sola línea, el shell nos proporciona caminos más cortos al poder utilizar caracteres especiales, llamados *metacaracteres* del shell. Estos metacaracteres dependen del tipo de shell de trabajo, en la tabla 2-3 se muestran algunos metacaracteres usados en la utilización y programación del shell.

Tabla 2-3. Metacaracteres

Metacaracter	Función	Ejemplo
<	Redirecciona la entrada de un archivo a xxx.	xxx < archivo
>	Redirecciona la salida de xxx a un archivo.	xxx > file
>>	Añade la salida de xxx a un archivo.	xxx >> archivo
2>	Envía el error estándar que se produzca a un archivo.	xxx 2> archivo
2>>	Añade el error estándar a un archivo.	xxx 2>> archivo
	Ejecuta un archivo y envía la salida a otro archivo.	archivo archivo
&	Ejecuta un archivo en modo subordinado o background.	archivo &
;	Ejecuta un archivo y después al siguiente que se encuentra delante de él.	archivo ; archivo
*	Comodín de nombre de archivo, sustituye a cualquier carácter en los nombres de archivo.	x*y
?	Comodín de nombre de archivo, sustituye un sólo carácter.	x?y
[...]	Identifica a cualquier carácter del conjunto encerrado entre [].	[Aa]rarchivo
=	Utilizado para asignar un valor a una variable de shell.	HOME= /export/usuarios/home/gcgero
\	Elimina el significado del siguiente carácter especial y la unión de línea.	
'...'	Se utiliza para que el shell no interprete el texto entre comillas simples.	

2.1.3 Entrada y salida del shell

El shell asocia a cada programa que ejecuta tres archivos abiertos: entrada estándar (por lo general el teclado), salida estándar (por lo general la pantalla) y error estándar (por lo general la pantalla). Estos tres archivos se asignan normalmente a la terminal, pero se pueden *redirigir* con facilidad utilizando los metacaracteres mostrados en la tabla 2-3.



El shell proporciona dos órdenes internas para la escritura de la salida y para la lectura de la entrada: **echo** y **read**.

La orden **echo** permite escribir salida desde la consola o desde un programa de shell, y puede trabajar con alguna o algunas de las siguientes secuencias de escape:

- \b Backspace.
- \c Imprime la línea sin nueva línea.
- \f Siguiete página.
- \n Nueva línea.
- \r Return.
- \t Tabulador.
- \v Tabulador vertical.
- \\ Barra invertida.

La orden **read** permite insertar la entrada del usuario en el *script*, **read** lee sólo una línea de entrada del usuario y la asigna a una o más variables del shell. Cuando programa puede asignarle un nombre a la variable en la cual se almacenará el dato leído o si no lo hace, se utiliza la variable **REPLY** por defecto. Cuando se utiliza **read** con varios nombres de variables, el primer campo tecleado por el usuario se asigna a la primera variable, el segundo campo a la segunda variable y así sucesivamente. Los campos que sobran a la izquierda son asignados a la última variable. Para que el shell reconozca donde empieza un campo, se auxilia de la variable de ambiente **IFS** (Internal Field Separator), la cual tiene el valor por defecto de un espacio en blanco; si desea utilizar un carácter diferente para separar campos simplemente se redefine la variable del shell **IFS**, por ejemplo **IFS = :** fijará el separador de campo al carácter dos puntos (:).

Ejemplo 1:

```
# Descompone la entrada del usuario en campos
IFS=:
echo "Escriba algunas palabras utilice la barra espaciadora como separación"
read palabra1 palabra2 palabra3 palabra4 palabra5
echo "\n Campo 1: " $palabra1
echo "\n Campo 2: " $palabra2
echo "\n Campo 3: " $palabra3
echo "\n Campo 4: " $palabra4
```

```
echo "\n Campos restantes: " $palabra5
```

2.1.4 Desplazamiento de los parámetros posicionales

Todos los argumentos de la línea de órdenes están asignados por el shell a parámetros posicionales; el valor del primer argumento de la línea de órdenes está contenido en \$1, el valor del segundo en \$2, el tercero en \$3 y así sucesivamente. Para reordenar los parámetros posicionales utilizamos el comando shift, el cual cambia \$2 a \$1, \$3 a \$2, \$4 a \$3 y así sucesivamente, y el valor original de \$1 se pierde.

2.2 Programación del shell

Para realizar un script, necesitamos:

1. Escribir los comandos del shell en un archivo de texto.
2. Hacer el archivo ejecutable (chmod +x archivo)
3. Escribir el nombre del archivo en la línea de comandos.

Cabe hacer algunas observaciones, si queremos utilizar otro tipo de shell para que interprete los comandos debemos de colocar como primera línea en el archivo de texto el nombre del shell; por ejemplo si nuestros comandos serán interpretados por el shell ksh, entonces colocamos como primera línea: #!/bin/ksh.

2.2.1 Ejecución condicional

Para proporcionar más potencia de programación, el shell incluye *constructores* para realizar decisiones en base a funciones de pruebas lógicas.

2.2.1.1 La orden if

La orden *if* proporciona control simple al programa a través de bifurcaciones simples, la forma general de la orden *if* es:

```
if orden  
  then ordenes  
fi
```

Las *ordenes* siguientes a *then* se ejecutan si se cumple con éxito la expresión *orden* de *if*. El *fi* marca el final de la estructura *if*.

En el sistema UNIX las ordenes a evaluar retorna 0 (verdadero) para indicar que se evaluaron normalmente, en caso contrario devuelve un número diferente de 0 (falso).

Ejemplo 2:

```
#!/bin/ksh  
# Script enviar
```

```
# Enviar un mensaje al usuario y eliminar el mensaje si el envío tiene éxito
if mail $LOGNAME < mensaje
then rm -r mensaje
fi
```

2.2.1.2 La orden test

Para utilizar las operaciones if-then en los programas de shell se necesita evaluar alguna expresión lógica y ejecutar alguna orden en base al resultado de esta evaluación. La orden *test* permite realizar comparaciones explícitas. *Test* retorna un valor igual a 0 si la expresión evaluada es verdadera y un número diferente de 0 si la expresión es falsa. *Test* permite evaluar cadenas, enteros y el estado de archivos del sistema UNIX, note además que *test* devuelve un estado diferente de 0 si no se le da una expresión.

Comprobación de enteros.

n1 -eq n2	verdadero si n1 = n2
n1 -ne n2	verdadero si n1 ≠ n2
n1 -gt n2	verdadero si n1 > n2
n1 -ge n2	verdadero si n1 >= n2
n1 -lt n2	verdadero si n1 < n2
n1 -le n2	verdadero si n1 <= n2

Comprobación de cadenas.

-z cadena	verdadero si la longitud de la cadena es 0.
-n cadena	verdadero si la longitud de la cadena es diferente de 0, es decir, si existe cadena.
cadena1 = cadena2	verdadero si cadena1 y cadena2 son idénticas.
cadena1 != cadena2	verdadero si cadena1 y cadena2 no son idénticas.
cadena1	verdadero si cadena 1 no es la cadena nula.

Comprobación de archivos.

-a archivo	verdadero si existe el archivo.
-r archivo	verdadero si existe el archivo y puede leerse.
-w archivo	verdadero si existe el archivo y puede escribirse.
-x archivo	verdadero si existe el archivo y es ejecutable.
-f archivo	verdadero si existe el archivo y es un archivo regular.
-d archivo	verdadero si existe el archivo y es un directorio.
-h archivo	verdadero si existe el archivo y es un enlace simbólico.
-c archivo	verdadero si existe el archivo y es un archivo de caracteres especiales.
-b archivo	verdadero si existe el archivo y es un archivo especial de bloque.
-p archivo	verdadero si existe el archivo y es un cauce con nombre.
-s archivo	verdadero si existe el archivo y tiene un tamaño mayor que cero.

Operadores de comprobación.

!	Operador negación.
-a	Operador binario and.

-o operador binario or.

Ejemplo 3:

```
#!/bin/ksh
if test $# -eq 0
  then echo "Use: posic archivo "
  exit 2
fi
mail $1 < mensaje
exit 0
```

Ejemplo 4:

```
#!/bin/ksh
if test -n "$1"
  then
  person= $1
fi
```

Ejemplo 5:

```
#!/bin/ksh
# Script catt
# verifica primero si existe el archivo para ver su contenido
# utilizando el programa cat
if test -r "$1"
  then
  if test -r "$1"
    then
    cat $1
  else
    echo " el archivo -" $1 "-no existe en esta ruta: " $PWD
    exit 1
  fi
  else
  echo "use : catt archivo"
  exit 2
fi
```

Existe una forma alternativa de especificar el operador de comprobación *test* por medio de [], así por ejemplo

```
test $# -eq 0    es equivalente a      [ $# -eq 0 ]
test -z $1      es equivalente a      [ -z $1 ]
test tam=$1    es equivalente a      [ tam = $1 ]
```

Note los espacios entre los [] y los operadores.

2.2.1.3 La orden *if-then-else*

La operación *if-then-else* permite bifurcación en doble camino en función del resultado de la orden *if*

```
if orden
  then ordenes
  else ordenes
fi
```

2.2.1.4 La orden *if-then-elif*

La orden *if-then-elif* permite crear un gran conjunto de comprobaciones *if-then* anidadas

```
if orden
  then ordenes
  elif orden
    then ordenes
  else ordenes
fi
```

2.2.1.5 La orden *case*

Una forma alternativa de *if-then-elif*, es la orden *case* que proporciona una forma más simple y legible para realizar las mismas comprobaciones.

```
case cadena in
  Lista-de-patrones )
    Ordenes
    ;;
  Lista-de-patrones )
    Ordenes
    ;;
.....
.....
.....
esac
```

case opera de la siguiente forma: El valor de ***cadena*** se compara con cada uno de los ***patrones***, si se encuentra una identificación, las órdenes que siguen al patrón se ejecutan hasta el doble punto y coma (;) donde finaliza la sentencia *case*.

El carácter * (asterisco) identifica a cualquier valor de ***cadena*** y proporciona una forma de especificar una acción por defecto, el uso de * es un ejemplo de comodín de nombre de archivo de shell en patrones *case*.

Ejemplo 6:

```
#!/bin/ksh
# script borrar
```

```

# Borra un archivo, sólo si el usuario esta seguro de hacerlo
# verifica que haya dado un argumento.
if [ -n "$1" ]
then
# verifica si el archivo es regular
if [ -f "$1" ]
then
echo "Borramos el archivo ?\c"
read ok
case $ok in
# verifiquemos si la respuesta es: y o Y o yes o YES o OK o ok
[Yy]* | OK | ok) echo "Borrando archivo ....."
sleep 1
rm $1
exit 0
;;
#verifica si la respuesta es: No o no
[Nn]*) echo "El archivo no fue borrado ...."
sleep 1
exit 1
;;
esac
else
# verifica si es un directorio
if [ -d "$1" ]
then
echo "No puedo borrar "$1 "es un directorio"
exit 1
else
echo "no puedo borrar el archivo"
exit 2
fi
fi
else
echo "use: borrar archivo"
exit 3
if

```

2.2.2 Loops

2.2.2.1 La orden for

El bucle *for* ejecuta una lista de órdenes una vez para cada miembro de una lista, el formato básico es:

```

for i [ in lista ]
do
Ordenes
done

```

La variable *i* en la sintaxis anterior puede ser cualquier nombre que se elija, si se omite la porción *in lista de la orden, las ordenes entre do y done* serán ejecutadas una vez por cada parámetro posicional que se coloque.

Ejemplo 7:

```
#!/bin/ksh
# script lista
# ejemplo que muestra como se utiliza una lista
for i in 1 2 3 4 5 6 7 8 9
do
  echo "Hola mundo"
done
```

Ejemplo 8:

```
#!/bin/ksh
#script buscar
# suponga que tiene un archivo (llamado agenda) en el cual coloco nombres, direcciones y teléfonos de
# sus amigos, y desea buscar si se encuentran ciertos nombres en el archivo
#
if [ -n "$1" ]
then
  for i
  do
    grep $i $PWD/agenda
  done
else
  echo "use: buscar agenda"
  exit 1
fi
```

al ejecutar colocamos: **buscar** hugo paco luis

2.2.2.2 Las ordenes *while* y *until*

Las ordenes *while* y *until* proporcionan una forma simple para implementar el control de comprobación-iteración-comprobación.

```
while ordenes1
do
  ordenes2
done
```

Esto es, si ordenes1 es verdadero entonces se ejecuta ordenes2, si no finaliza el *while*.

Para *until* se sigue la sintaxis:

```
until ordenes1
```

```
do
ordenes2
done
```

La única diferencia entre las ordenes *while* y *until* se encuentra en la naturaleza de comprobación lógica que se realiza en parte superior del bucle.

2.2.3 Las ordenes *break* y *continue*

Normalmente, cuando se establece un bucle utilizando las ordenes *for*, *while* y *until* (y la orden *select* en *ksh*), la ejecución de las órdenes englobadas en el bucle continúa hasta que se cumple la condición lógica del bucle. El shell proporciona dos formas para alternar el funcionamiento de las ordenes en un bucle, una de ellas es ***break***, el cual permite salir del bucle inmediato que lo engloba, y ***continue*** el cual es opuesto a *break*, controla la vuelta al principio del bucle más pequeño que lo engloba.

2.2.4 La orden *select*

La orden *select* se utiliza para visualizar un número de elementos sobre la salida estándar y espera entrada, si el usuario pulsa return sin realizar una selección, la lista de elementos se visualiza de nuevo hasta que se da una respuesta. *Select* es muy utilizado en programas que permiten a los nuevos usuarios selecciones de un menú en lugar de entradas por órdenes para operar un programa. La orden *select* sólo se encuentra en el shell *korn*.

2.2.5 Las ordenes *true* y *false*

Dependiendo de las comprobaciones y comparaciones los scripts retornan un valor verdadero (0) o falso ($\neq 0$), las ordenes *true* y *false* retornan dos valores específicos, *true* devuelve siempre 0 y *false* un número diferente de 0. El uso fundamental de estas ordenes es el establecimiento de bucles infinitos.

```
while true
do
ordenes
done
```

```
until false
do
ordenes
done
```

Note que dentro del bucle debe existir alguna acción que provoque un *break* para poder salir.

2.2.6 Operaciones aritméticas

Para evaluar expresiones aritméticas los shells cuentan con las siguientes instrucciones:

```
expr   en sh
let    en ksh
(( ))  en ksh, la cual es una abreviación de let.
```

Por ejemplo:

```
suma = expr 1 + 1 // los espacios son muy importantes
O
let suma =1+1
O
((suma=1+1))
```

Las instrucciones que trabajan mejor, sin ocasionar tantos problemas son *let* y *(())*. Cabe hacer notar que *let* automáticamente utiliza el valor de una variable, no debemos anteponerle el símbolo \$. La orden *let* se puede utilizar para todas las operaciones aritméticas básicas (+,-,*,/,%), también proporciona operaciones más especializadas, como conversión entre bases y operaciones bit a bit and (&) y or (|).

Ejemplo 9:

```
#!/bin/ksh
# script imprime
# script coloca en pantalla del 1 al 10
i = 1
while ((i<=10))
do
  echo $i
  ((i=i+1))
done
```

2.2.7 Depuración de programas de shell

Algunas veces nos damos cuenta que los script no funcionan de la manera que esperábamos, quizás escribimos mal una orden o no colocamos las comillas o caracteres especiales. Cuando cometemos un error en el mejor de los casos el script no se ejecutará, o en el peor de los casos un error en el nombre de una orden producirá que la orden realice algo indebido, por ejemplo, `rm *.tmp` es diferente que `rm * .tmp`, el espacio entre `*` y `.tmp` hará que todos los archivos del directorio actual se borren.

Para inspeccionar la ejecución de cada línea del script, se utiliza la orden:

```
sh -x script
```

Esta orden le indica al shell que imprima cada orden y sus argumentos conforme se van ejecutando.

2.3 El lenguaje awk

El nombre de awk viene de sus autores Aho, Weinberger, y Kernighan. La versión original de awk fue escrita en 1977 en AT&T Bell Laboratories.

Entre las cosas que podemos hacer con awk se encuentran: administrar pequeñas base de datos personales, generar reportes, validar datos, indexar productos, experimentar con algoritmos que pueden ser adaptados después a otros lenguajes de computo.

Awk tiene un amplio rango de comando y una variedad de elementos de programación como son: las variables, las decisiones, los ciclos de repetición (loops), arreglos, y mucho más.

Awk divide los datos de los archivos en registros y campos, cada registro esta separado por una nueva línea y cada campo se encuentra separado por tabs o espacios (esto es por default, pero se puede cambiar los caracteres de separación).

2.3.1 El formato de una script awk

Un script awk puede tener tres secciones:

- Una sección de *BEGIN*, la cual inicializa el script. Esta sección es ejecutada solamente al inicio del script, antes de que cualquier registro sea leído del archivo o archivos de entrada. La sintaxis es :

```
BEGIN {  
  declaración  
  declaración  
  declaración  
  ....  
}
```

o

```
BEGIN { declaración }
```

o

```
BEGIN { declaración; declaración; declaración }
```

- Una sección de *declaración de patrón o cuerpo principal* la cual tiene la siguiente sintaxis:

```
patrón { acciones }
```

En esta sección awk lee un registro del archivo de entrada, y si este cumple con el patrón realiza alguna o algunas acciones con el registro. Algunas formas que pueden aparecer en esta sección son:

```
/expresión regular / { declaración }
```

o

```

/expresión regular / {
    declaración
    declaración
    declaración
    .....
}

```

o

```

/expresión regular / { declaración; declaración; declaración }

```

- Una sección *END* que se ejecuta después de leer todas las entradas del archivo de datos. Usualmente en esta sección se realiza los cálculos finales, los cuales no pueden ocurrir hasta después de procesar la entrada de los datos. La sintaxis de esta sección es:

```

END {
    declaración
    declaración
    declaración
    .....
}

```

o

```

END { declaración }

```

o

```

END { declaración; declaración; declaración }

```

2.3.2 Como ejecutamos programas awk

Existen varias formas de ejecutar un programa awk, si el programa es corto, es fácil incluirlo en el comando que ejecuta awk:

```
awk 'PROGRAMA' Archivo-entrada1 Archivo-entrada2 ...
```

donde PROGRAMA consiste de una serie de patrones y acciones.

Cuando el programa es largo, es conveniente escribirlo en un archivo y ejecutarlo como sigue:

```
awk -f Archivo-del-PROGRAMA Archivo-entrada1 Archivo-entrada2 ...
```

Ejemplo 1:

```
$ cat uno.awk
```

```

BEGIN {
    print "Hola, mundo!"
}

```

```
$ awk -f uno.awk
```

```
Hola, mundo!
```

Este script sólo despliega el mensaje porque la acción se colocó en la sección de BEGIN.

Ejemplo 2:

```
$ cat dos.awk
```

```
{  
    print "Hola mundo!"  
}
```

```
$ awk -f dos.awk
```

Cuando ejecutamos este comando no hace nada hasta que presionemos enter, pero no retorna el prompt del sistema, debemos presionar ^C para retornar al prompt. La razón de esto es que awk necesita de un archivo para aplicar los comandos que se encuentran en el cuerpo principal del script, y como no existe archivo de entrada el espera leer un registro hasta que encuentre un enter. Si tenemos por ejemplo un archivo (llamémosle nombres) con 5 líneas y lo colocamos como archivo de entrada pasará lo siguiente:

```
$awk -f dos.awk nombres
```

```
Hola, mundo!  
Hola, mundo!  
Hola, mundo!  
Hola, mundo!  
Hola, mundo!
```

```
$
```

2.3.3 Registros y campos

Como mencionamos al inicio de la sección, cada línea del archivo de entrada es un registro, y cada registro está formado por campos que se encuentran delimitados por un carácter separador (por default un tab o espacio en blanco). Supongamos que tenemos el siguiente archivo (llamado autos.txt):

ES	Arturo	85	Honda	Prelude	190.00
BS	Brenda	90	Nissan	300ZX	130.00
AS	Sara	91	BMW	M-3	345.00
ES	Sandra	92	Honda	Civic	210.00
DS	Samuel	93	Honda	CRX-Si	230.00
ES	Daniel	86	VW	GTI	90.00
AS	Norma	89	Porsche	911	420.00
CS	Mario	90	VW	Golf	115.00

Donde cada columna representa la clase, el nombre del conductor, el año del modelo, el fabricante, el modelo y el precio. Este archivo contiene 8 registros, y cada registro contiene 6 campos. Con estos datos podemos usar awk para calcular el precio de todos los autos, el precio de los autos en cada clase, para todos VW, para todos los del año 90, etc..

Cada campo en el registro tiene asociado una variable interna de awk, la cual es un valor; para referirnos a este valor debemos utilizar:

\$0	Para el valor del registro de entrada
\$1	Para el valor del primer campo
\$2	Para el valor del segundo campo
\$n	En forma general, representa el valor del n-ésimo campo.

Ejemplo 3:

```
$cat tres.awk
```

```
#imprime el nombre del conductor y el fabricante del carro
#
    { print $2 $4 }
```

```
$awk -f tres.awk autos.txt
```

```
ArturoHonda
BrendaNissan
SaraBMW
SandraHonda
SamuelHonda
DanielVW
NormaPorsche
MarioVW
```

Si deseamos colocar un espacio separador entre los valores de los campos debemos colocar en el print una coma entre los campos, pero si queremos que aparezca impresa una coma debemos colocar la coma entre comillas “,”.

Ejemplo 4:

```
#imprime el nombre del conductor y el fabricante del carro, separados por ,
#
    { print $2 “,” $4 }
```

```
$awk -f tres.awk autos.txt
```

```
Arturo, Honda
Brenda, Nissan
Sara, BMW
Sandra, Honda
Samuel, Honda
Daniel, VW
```

Norma, Porsche
Mario, VW

Veamos un ejemplo donde utilizamos las tres secciones y utilizamos la variable NR, la cual contiene el número de registros que fueron leídos, para esto supongamos que tenemos un archivo llamado nombres.txt con diez registros y 5 campos: nombre, apellido paterno, apellido materno, grupo, semestre.

Ejemplo 5:

```
$cat grupo.awk
```

```
BEGIN {
    print
    print "Clase de redes"
    print
}

#cuerpo principal del script

    { print $2, $1 }

END {
    Print "Numero de estudiantes: " NR
}
```

```
$awk -f grupo.awk nombres.txt
```

```
Salazar Rose
Villalba Nancy
Cervantes Nashelli
Doquis Carlos
Benitez Blanca
Martínez Belén
Luna Lizbeth
López Mayra
Cruz Martha
Ruiz José
```

Numero de estudiantes: 10

2.3.4 Variables en awk

Awk contiene una serie de variables que proporcionan información acerca del tamaño y composición de los registros de entrada, y variables para el control de formato.

NR	Número de registros que fueron leídos.
NF	Número de campos en el registro actual.
FS	Separador de campo de entrada (por default, es un espacio).
RS	Separador de registros (por default, newline).

\$0	Valor de la entrada actual del registro.
\$n	Valor del n-ésimo campo del registro.
OF	Separador del campo de salida (por default, espacio).
OR	Separador de registro de salida (por default, newline).
FILENAME	Nombre del archivo de entrada.
ARGC	Número de argumentos en la línea de comandos.
ARGV	Arreglo de argumentos en la línea de comandos.
FNR	Número de registros en el archivo actual.

2.3.5 Patrones

Para buscar patrones en un archivo se utiliza el carácter de tilde (~). Por ejemplo el siguiente script usa el patrón \$1 ~/AS/ y si lo encuentra imprime el registro completo e incrementa un contador para que en la sección END imprima el número total de claves AS encontradas.

Ejemplo 6:

```
# Script que busca el patrón AS en el campo 1
#
$1 ~/AS/ {
    print $0
    num = num + 1
}
END {
    print "Fueron encontradas: " num "AS"
}
```

Awk usa el símbolo ! para negar una coincidencia o patrón, por ejemplo para mostrar todos los registros que no contengan AS.

Ejemplo 7:

```
# Script que encuentra todos los registros que no contengan AS
#
$1 !~/AS/ {
    print $0
    num++
}
END {
    print "Fueron encontrados : " num "registros que no contienen AS"
}
```

2.3.6 Operadores

Awk tiene un conjunto de operadores aritméticos y lógicos similares a los usados en C. En la tabla 2-4 se muestra los operadores aritméticos, y en la tabla 2-5 se muestran los operadores relacionales que son usados para comparar los valores de dos variables.

Tabla 2-4. Operadores aritméticos.

Símbolo	
+	Suma
-	Resta
*	Multiplicación
/	División
%	Operador modulo

Tabla 2-5. Operadores relacionales.

Símbolo	
==	Igualdad
!=	Desigualdad
>	Mayor que
>=	Mayor que o igual que
<	Menor que
<=	Menor que o igual que

Awk cuenta como ya hemos visto con el operador de asignación = para colocarle un valor a una variable, aparte de este operador, awk cuenta con operadores de asignación cortos tomado de C (tabla 2-6), con operadores de incremento (tabla 2-7) y los operadores lógicos && el cual equivale a un and y || el cual representa un or.

Tabla 2-6. Operadores de asignación compuestos

Símbolo	
+=	Suma y asigna
-=	Resta y asigna
*=	Multiplifica y asigna
/=	Divide y asigna
%=	Modulo y asigna

Tabla 2-7. Operadores de incremento.

Símbolo	
++	Incrementa en uno
--	Decrementa en uno

Veamos un ejemplo donde utilizamos el operador lógico &&.

Ejemplo 8:

```
# Script que muestra los id que se encuentran entre el intervalo [100,1300)
# tomados del archivo passwd, utilizando la variable FS para indicar que la separación
# de los campos es por medio de :
```

```
BEGIN { FS = ":" }
$3 >= 100 && $3 < 200 { print }
```

Existen dos formas de imprimir la salida en awk:

- La declaración **print**, es utilizado para imprimir salidas no formateadas.
- La declaración **printf** se utiliza para imprimir salidas formateadas, esta instrucción es similar al printf de C. La sintaxis del printf es:

```
printf "formato, expresion1, expresion2, ...
```

En la tabla 2-8 se muestran los formatos especificados en awk.

Tabla 2-8. Conversión de formatos.

Especificación	
%d	Imprime la salida como un entero decimal (en base 10).
%f	Imprime la salida como un valor de punto flotante.
%o	Imprime la salida como un valor octal.
%x	Imprime la salida como un valor hexadecimal.
%s	Imprime la salida como una cadena.
%e	Imprime la salida como un punto flotante con notación científica.

Ejemplo 9:

```
# Script para mostrar el uso del printf
{ printf "%s %s \t [ %.2f] \n", $1, $2, $6 }
```

Ejemplo 10:

```
# Script que utiliza printf y la salida
# la redirecciona a un archivo (result.txt)
{ printf (" %s %s \t [ %.2f] \n", $1, $2, $6) > "result.txt" }
```

2.3.7 Declaraciones condicionales

Awk soporta dos tipos de condicionales: implícitas y explícitas. Las condicionales *explícitas* son: if, if-else cuya sintaxis es:

```
if ( expresión )
    acción
```

Si *expresión* es verdadera entonces la *acción* se ejecuta

```
if ( expresión )
    acción-1
else
    acción-2
```

Si el valor de expresión es verdadero entonces *acción-1* es ejecutada, sino se ejecuta *acción-2*. Al igual que en C si la acción involucra más de una instrucción entonces se utilizan { }.

La sintaxis de una condición *implícita* es:

Patrón que contiene una expresión relacional { acción }

Ejemplo 11:

```
$3 > 1000 { print $3 }
```

2.3.8 Loops

Una de las declaraciones que controlan las repeticiones en awk es *while*, cuya sintaxis es:

```
while ( expresión )  
  acción
```

Ejemplo 12:

```
num = 1  
costo = 0  
while ( num <= 10 )  
{  
  costo += $6  
  num++  
}
```

La declaración *do-while* proporcionan una ligera diferencia con respecto al *while*, checa la expresión hasta el final, por lo que la acción por lo menos se ejecuta una vez.

```
do {  
  acción  
} while ( expresión )
```

Ejemplo 13:

```
num = 5  
costo = 0  
do {  
  costo += $3  
  num - -  
} while ( num > 0 )
```

La declaración *for* es otra forma de loop en awk. Este loop frecuentemente se usa cuando se conoce el número de iteraciones a ejecutar. La sintaxis del *for* es:

```
for ( expr1; expr2; expr3 )  
  acción
```

Ejemplo 14:

```
costo = 0  
for ( num =1; num <= 10; num++)
```

```
{
  costo += $6
}
```

2.3.9 Manipulación de cadenas

Awk es una herramienta excelente para manejar cadenas. Las funciones que utiliza se muestran en la tabla 2-9.

Tabla 2-9.Funciones de manipulación de cadenas.

Función	Uso	Ejemplo
length	Retorna la longitud de una cadena	print length (\$4)
substr	Retorna la subcadena de <i>n</i> caracteres, de la cadena <i>s</i> empezando en la posición <i>m</i> . La sintaxis es: substr (s, m, n). Si se omite <i>n</i> se retorna el resto de la cadena a partir de <i>m</i> .	print substr ("Domingo", 1,3) Despliega: Dom
index	Retorna la posición de la primera ocurrencia de la cadena <i>t</i> en la cadena <i>s</i> . La sintaxis es: index (s, t)	Supongamos que tenemos un archivo de números con punto flotante y queremos obtener sólo las centenas. { dp = index (\$1, ".") print substr (\$1, dp + 1) }
sprintf	Retorna una cadena completa a partir del formato usado por la declaración printf.	{ mensaje = sprintf ("Juan tiene %d años %d meses % días", año, meses, dia) print mensaje }
getline	Lee el siguiente registro de entrada sin saltar al inicio del programa.	

2.3.10 Funciones matemáticas

Awk cuenta con varias funciones matemáticas, las cuales se muestran en la tabla 2-10.

Tabla 2-10. Funciones matemáticas.

Función	Resultado
sqrt	Retorna la raíz cuadrada de un argumento. sqrt (x)
exp	Retorna el valor exponencial de un argumento. exp (x)
log	Retorna el logaritmo de base <i>e</i> (natural) de un argumento. log (x)
sin	Retorna el seno de un argumento en radianes. sin (x)
cos	Retorna el coseno de un argumento en radianes. cos (x)
int	Retorna el valor entero de un argumento. int (25.4567) retorna el valor de 25
Rand	Retorna un número aleatorio en el rango de (0, 1). rand ()
srand	Inicializa rand ()

2.3.11 Arreglos

Un arreglo es una variable que puede contener un conjunto de valores, cada valor es almacenado en un miembro o elemento del arreglo y puede ser accesado usando el nombre del arreglo y el índice. El índice puede ser un número o una cadena, si el índice es una cadena, el arreglo es llamado un arreglo asociativo.

Ejemplo 15:

```
#script awk que muestra como manipular arreglos
#
BEGIN {
    printf "Script que usa un arreglo"
}
{
    t[NR] = $1
}
END {
    for ( i=1 ; i<=NR ; i++)
    {
        print t[i]
    }
}
```

2.4 Bibliografía

- Rod Manis, Marc H. Meyer. Howard W, “The UNIX Shell. Programming Lenguaje”, SAMS &Co, 1988.
- H. M. Deitel, “Sistemas Operativos”, Addison Wesley, 1999.
- Kenneth H. Rosen, Richard R. Rosinski, James M. Farber, Douglas A. Host , “UNIX. Sistema V”, Mc Graw Hill, 1997.
- Peter Holsberg, “UNIX Desktop Guide To Tools”, SAMS, 1992.
- David Medinets, “UNIX Shell. Programming Tools”, Mc Graw Hill, 1999.
- Arnold D. Robbins, “Effective AWK Programming”, Free Software Foundation, 1997.