APUNTES DE PROGRAMACION AVANZADA EN UNIX

Fecha de elaboración: enero de 2006

Autor: Profesor David Kanagusico Hernández

CONTENIDO

I IMPORTANCIA DE LA PROGRAMACIÓN EN SHELL. 4
1 Principales características. 4
2 ¿Dónde encontrar KSH? 6
3 El shell como lenguaje de programación. 7
4 El shell y los CGI's. 8
II EL SHELL DE BOURNE. 8
1 Introducción. 8
2 Redirección. 8
3 Variables. 10
4 Asignación y herencia de variables. 10
5 Interpretación de nombres de variables. 11
6 Substitución de variables. 11
7 Funciones. 12
8 Aritmética. 14
III PROGRAMACIÓN EN SH. 15
1 Introducción. 15
2 El shell como lenguaje de programación. 15
3 at, batch, cron. 18
4 Variables especiales. 21
5 Parámetros posicionales. 24
6 Comando shift. 24
7 Substitución del comando. 24
8 Comando test. 26
9 Estructuras de control. 26
Secuencias condicionales : if .. fi : 26
Secuencia condicional case .. esac. 28
Bucles FOR. 30
Bucles WHILE. 32
Operadores AND / OR. 34
Comandos trap/exec/exit. 35
IV PROGRAMACIÓN EN CSH. 37
1 Variables del shell. 37
2 Asignación y substitución de variables. 37
3 Aritmética y operadores para expresiones. 37
4 Estructuras de control. 37
If 38
Switch 38
foreach 38
while 38
V EL EDITOR DE TEXTO ED. 39
1 Introducción. 39
2. Comandos de Edición 39
2. Editando por contexto 42

3. Aplicación de ejemplo 43
VI EXPRESIONES REGULARES. 44
1 Definición. 44
2 Concordancias. 45
3. Carácteres especiales 45
4. Definiendo Rangos 46
5. Agrupando, referenciando y extrayendo 46
5.1 Agrupación 46
5.2 Referencias 46
5.3 Extracción 47
6. Expresiones opcionales y alternativas. 47
6.1 Opcionales 47
6.2 Alternativas 47
7. Contando expresiones 47
8. Expresiones repetidas 48
VII GREP. 48
1 Filtros. 48
2 Familia grep. 48
3 grep. 49
4 fgrep y egrep. 50
VIII EL EDITOR DE FLUJO SED. 51
1 Introducción. 51
2 Listas de comandos. 51
3 Selección de lineas 53
IX AWK. 53
1 Introducción. 53
2 Registros y campos. 54
3. Parámetros en tiempo de ejecución 55
4. Campos de entrada 56
5. Variables 56
6. Operadores 57
7. Instrucciones de control 58
8. Funciones 59

I IMPORTANCIA DE LA PROGRAMACIÓN EN SHELL.

1 Principales características.

La shell es el programa más importante para la mayoría de los usuarios y administradores de UNIX; con la excepción del editor de textos y del menú de administración, posiblemente es con el que más tiempo se trabaja.

La shell es el lenguaje de comandos de UNIX ; es un programa que lee los caracteres tecleados por los usuarios , los interpreta y los ejecuta.

A diferencia de otros intérpretes más estáticos en otros sistemas operativos , aquí existe además un completo lenguaje de programación que permite adaptar de manera relativamente sencilla el sistema operativo a las necesidades de la instalación.

Una vez que un usuario común se ha registrado cara al sistema con su login y su password , se le cede el control a la shell , la cual normalmente ejecuta dos ficheros de configuración , el general (/etc/profile) y el particular(<directorio_del_usuario>/.profile). Una vez aquí , la propia shell despliega el literal "inductor de comandos" , que normalmente será :

$

ó

#

Y el cursor en espera de que alguien escriba algún comando para ejecutar.

Para comprender la manera en la cual la shell ejecuta los comandos hay que tener en cuenta las circunstancias siguientes :

- Tras sacar en pantalla el indicador $, espera a que se le introduzca algo, lo cual será interpretado y ejecutado en el momento de pulsar <INTRO>.

- La shell evalúa lo que hemos escrito buscando primero si contiene un carácter "/" al principio. En caso que sea así, lo toma como un programa y lo ejecuta.

Si no , examina si se trata de una función (o un alias, en el caso de la ksh). Una función es una secuencia de comandos identificada por un nombre unívoco , y se verá más adelante. En caso de no encontrar ninguna con ése nombre , busca a ver si se trata de un comando interno

(exit, exec , trap , etc) ó palabra reservada (case,do,done,if,for .. etc) , para ejecutarlo ó pedir más entrada. Si ninguna de éstas condiciones es cierta , la shell piensa que lo que hemos escrito es un comando , y lo busca dentro de los directorios contenidos en la variable de entorno PATH . Si no está , saca un mensaje del tipo "XXXX : not found" , siendo XXXX lo que hemos escrito.

Mejor verlo con un ejemplo : supongamos que escribimos alguna aberración del tipo :

```
$ hola
```

Suponiendo que la variable PATH contenga los directorios /bin,/usr/bin y /etc , la shell busca el comando "/bin/hola" , "/usr/bin/hola" y "/etc/hola". Ya que obviamente no existe , la contestación será :

```
sh : hola : not found.
```

La shell utiliza UNIX para la ejecución de procesos, los cuales quedan bajo su control. Podemos definir un proceso aquí como un programa en ejecución. Ya que UNIX es multitarea , utiliza una serie de métodos de "tiempo compartido" en los cuales parece que hay varios programas ejecutándose a la vez , cuando en realidad lo que hay son intervalos de tiempo cedidos a cada uno de ellos según un complejo esquema de prioridades .

Cuando la shell lanza un programa , se crea un nuevo proceso en UNIX y se le asigna un número entero (PID) entre el 1 y el 30.000 , del cual se tiene la seguridad que va a ser unívoco mientras dure la sesión. Lo podemos ver ejecutando el comando **"ps"** , el cual nos da los procesos activos que tenemos asociados a nuestro terminal.

Un proceso que crea otro se le denomina proceso padre. El nuevo proceso , en éste ámbito, se le denomina proceso hijo. Este hereda casi la totalidad del entorno de su padre (variables ,etc)., pero sólo puede modificar su entorno, y no el del padre.

La mayoría de las veces, un proceso padre se queda en espera de que el hijo termine, esto es lo que sucede cuando lanzamos un comando ; el proceso padre es la shell , que lanza un proceso hijo (el comando). Cuando éste comando acaba , el padre vuelve a tomar el control , y recibe un número entero donde recoge el código de retorno del hijo (0=terminación sin errores , otro valor=aquí ha pasado algo).

Cada proceso posee también un número de "grupo de procesos" . Procesos con el mismo número forman un sólo grupo, y cada terminal conectado en el sistema posee un sólo grupo de procesos. (Comando ps -j) . Si uno de nuestros procesos no se halla en el grupo asociado al terminal , recibe el nombre de proceso en background (segundo plano).

Podemos utilizar algunas variantes del comando **"ps"** para ver qué procesos tenemos en el equipo:

ps : muestra el número de proceso (PID) , el terminal , el tiempo en ejecución y el comando. Sólo informa de nuestra sesión.

ps -e : de todas las sesiones.

ps -f : full listing : da los números del PID,del PPID(padre),uso del procesador y tiempo de comienzo.

ps -j : da el PGID (número de grupo de los procesos - coincide normalmente con el padre de todos ellos)

Este comando puede servirnos para matar ó anular procesos indeseados. Se debe tener en cuenta que cada proceso lleva su usuario y por tanto sólo el (ó el superusuario) pueden matarlo.

Normalmente , si los programas que componen el grupo de procesos son civilizados , al morir el padre mueren todos ellos siempre y cuando el padre haya sido "señalizado" adecuadamente. Para ello , empleamos el comando **kill -<número de señal> PID**, siendo PID el número del proceso ó del grupo de procesos.

Los números de señal son :

-15 : TERM ó terminación. Se manda para que el proceso cancele ordenadamente todos sus recursos y termine.

-1 : corte

-2 : interrupción.

-3 : quit

-5 : hangup

-9 : kill : la más enérgica de todas pero no permite que los procesos mueran ordenadamente.

2 ¿Dónde encontrar KSH?

Ya que , como hemos explicado anteriormente , la shell es un programa , existen varios , cada uno con sus características particulares . Veamos algunas de ellas:

- Bourne shell (/bin/sh) : Creada por Steven Bourne de la AT&T . Es la más antigua de todas y , por tanto , la más fiable y compatible entre plataformas. Esta es en la que se basan las explicaciones posteriores.

- Korn shell (/bin/ksh) : Creada por David G. Korn de los laboratorios Bell de la AT&T. Más moderna, toma todos los comandos de la Bourne y le añade varios más así como varias características de edición interactiva de comandos, control de trabajos y mejor rendimiento en términos de velocidad que la anterior. Existen dos versiones , una primera "maldita" y la actual , de 16/11/1988. (podemos averiguar qué versión tiene la nuestra ejecutando el comando "what /bin/ksh") Pero no es oro todo lo que reluce, cuando se trata de situaciones extremas ó complicadas donde la ksh falla, la de Bourne normalmente está más "blindada".

- C Shell , desarrollada por Bill Joy en la Universidad de California y , por tanto , más extendida entre UNIX BSD. Bastante más críptica que la de Bourne , incorpora no obstante sustanciales mejoras.

Estructura de las órdenes - Metacaracteres.

Cuando escribimos cualquier comando y pulsamos <INTRO> , es la shell y no UNIX quien se encarga de interpretar lo que estamos escribiendo y ordenando que se ejecute dicho comando. Aparte de los caracteres normales , la shell interpreta otros caracteres de modo especial: un grupo de caracteres se utiliza para generar nombres de ficheros sin necesidad de teclearlos explícitamente.

Cuando la shell está interpretando un nombre , los caracteres * ? [ ] se utilizan para generar patrones. El nombre que contenga alguno de éstos caracteres es reemplazado por una lista de los ficheros del directorio actual cuyo nombre se ajuste al patrón generado.

Las reglas de generación de patrones son :

***** Vale cualquier cadena de caracteres .

? Vale un carácter cualquiera.

[..] Vale cualquiera de los caracteres que coincida con los que estén entre corchetes.

Ejemplo . Supongamos que en nuestro directorio actual tenemos los siguientes ficheros :

```
$ ls

tonto

tonta

diario

mayor
```

Veamos cada una de las salidas correspondientes a las reglas anteriores :

`$ ls *` (valen todos)

```
tonto

tonta

diario

mayor
```

`$ ls *o` (todos, pero tienen que acabar en "o")

```
tonto

diario
```

`$ ls tont?` (que empiecen por tont y cualquier otro carácter)

```
tonto

tonta
```

`$ ls tont[oa]` (que empiecen por tont y el siguiente sea "o" ó "a")

3 El shell como lenguaje de programación.

El shell posee algunas de las capacidades mas avanzadas de cualquier interprete de comandos de su tipo, aunque su sintaxis no es tan elegante ni consistente como otros lenguajes de

programación, su poder y flexibilidad es comparable con estos. De hecho, el shell puede ser usado como un ambiente completo para escribir prototipos de software.

4 El shell y los CGI's.

Common Getaway Interface). Interface Común de Pasarela. Interface de intercambio de datos estándar en WWW a través del cual se organiza el envío de recepción de datos entre visualizadores y programas residentes en servidores WWW.

II EL SHELL DE BOURNE.

1 Introducción.

El shell traduce los comandos en línea del usuario en instrucciones del sistema operativo, es importante recalcar que el shell en si mismo no es UNIX, es solo la interfaz para UNIX. UNIX es uno de los primeros sistemas operativos en hacer la interface del usuario independiente del sistema operativo.

2 Redirección.

Para cada sesión , UNIX abre tres ficheros predeterminados , la entrada estándar, la salida estándar y el error estándar, como ficheros con número 0, 1 y 2. La entrada es de donde un comando obtiene su información de entrada ; por defecto se halla asociada al teclado del terminal. La salida estándar es donde un comando envía su resultado; por defecto se halla asociada a la pantalla del terminal; el error estándar coincide con la salida estándar, es decir, con la pantalla.

A efectos de modificar éste comportamiento para cualquier fin que nos convenga , la shell emplea 4 tipos de redireccionamiento :

< Acepta la entrada desde un fichero.

> Envía la salida estándar a un fichero.

>> Añade la salida a un archivo existente. Si no existe , se crea.

| Conecta la salida estándar de un programa con la entrada estándar de otro.

Intentar comprender éstas explicaciones crípticas puede resultar cuanto menos, confuso ; es mejor pensar en términos de "letras" :

< En vez de coger líneas del teclado , las coge de un fichero. Mejor , adquirir unos cuantos conceptos más para entender u ejemplo.

> Las letras que salen de un comando van a un fichero. Supongamos que ejecutamos el comando ls usando esto :

```
$ ls > cosa
```

$ (.. parece que no ha pasado nada ...)

```
$ ls

tonto

tonta

diario

mayor

cosa
```

$ (.. se ha creado un fichero que antes no estaba - cosa ...)

```
$ cat cosa

tonto

tonta

diario

mayor

cosa
```

$ (.. y contiene la salida del comando "ls" ...)

>> En vez de crear siempre de nuevo el fichero , añade las letras al final del mismo.

| El comando a la derecha toma su entrada del comando de la izquierda. El comando de la derecha debe estar programado para leer de su entrada ; no valen todos.

Por ejemplo , ejecutamos el programa "cal" que saca unos calendarios muy aparentes . Para imprimir la salida del cal , y tener a mano un calendario sobre papel , podemos ejecutar los siguientes comandos :

$ cal 1996 > /tmp/cosa (el fichero "cosa" contiene la salida del "cal")

$ lp /tmp/cosa (sacamos el fichero por impresora).

Hemos usado un fichero intermedio. Pero , ya que el comando "lp" está programado para leer su entrada , podemos hacer (más cómodo, y ahorramos un fichero intermedio) :

```
$ cal 1996 | lp
```

El uso de la interconexión (pipa) exige que el comando de la izquierda lance información a la salida estándar. Por tanto , podríamos usar muchos comandos, tales como cat , echo , cal , banner , ls , find , who .Pero , el comando de la derecha debe estar preparado para recibir información por su entrada ; no podemos usar cualquier cosa . Por ejemplo , sería erróneo un comando como :

```
ls | vi
```

(si bien ls si saca caracteres por su salida estándar , el comando "vi" no está preparado para recibirlos en su entrada).

Y la interconexión no está limitada a dos comandos ; se pueden poner varios , tal que así:

```
cat /etc/passwd | grep -v root | cut -d":" -f2- | fold -80 | lp
```

(del fichero /etc/passwd sacar por impresora aquellas líneas que no contengan root desde el segundo campo hasta el final con un ancho máximo de 80 columnas).

Normalmente, cualquier comando civilizado de UNIX devuelve un valor de retorno (tipo ERRORLEVEL del DOS) indicando si ha terminado correctamente ó bien si ha habido algún error ; en el caso de comandos compuestos como e s éste , el valor global de retorno lo da el último comando de la pipa ; en el último ejemplo , el retorno de toda la línea sería el del comando "lp".

3 Variables.

Las variables son lugares que almacenan datos, usualmente en forma de cadenas de caracteres, y sus valores pueden ser obtenidos precediendo sus nombres con el signo de dólar ($). Algunas variables, llamadas variables de ambiente, son nombradas de manera convencional en letras mayúsculas y son inicializadas con el comando **export**.

4 Asignación y herencia de variables.

Se definen valores para las variables con declaraciones de la forma **varname=value**.

```
$ hatter=mad

$ echo "$hatter"

mad
```

5 Interpretación de nombres de variables.

Algunas variables de ambiente son predefinidas por el shell en el momento de entrar al sistema.

Algo especialmente importante, son las variable con parámetros posicionales. Estas mantienen los argumentos de los comandos en línea cuando estas son invocadas. Los parámetros posicionales tienen los nombres **1, 2, 3,** etc., esto significa que sus valores son denotados por **$1, $2, $3,** etc. Además existe el parámetro posicional **0**, cuyo valor es el nombre del script.

Dos variables posicionales contienen todos los parámetros posicionales (excepto el parámetro posicional 0): * y @. La diferencia entre ellos es sutil pero importante:

"$*"** es una cadena simple que cosiste en todos los parámetros posicionales y **"$@" es igual a **"$1" "$2" ... "$N"**, donde N es el numero de parametros posicionales.

6 Substitución de variables.

El shell también reconoce variables alfanuméricas, a las cuales un valor de cadena puede ser asignado. No deben existir espacios entre el signo igual (=) y para conocer el contenido de la variable se hace de la siguiente forma:

```
Letras="cadenas"
$ echo $Letras
cadenas
$
```

Más de una asignación es posible puede aparecer en una sentencia, tomando en cuanta que las asignaciones se ejecuten de derecha a izquierda.

```
W=abc
$ C2=$W
```

La cadena abc será asignada a la variable W y después el valor de la variable W será asignado a la variable C2.

```
$ echo $W $C2
abc abc
$
```

Para concatenar el valor de una variable se puede encerrar el nombre de la variable entre llaves ({y}) anteponiendo el signo de pesos ($). Ejemplo:

```
$ agp='esto es una cadena'
$ echo "${agp}miento de cadenas"
esto es un encadenamiento de cadenas
$
```

7 Funciones.

De manera similar a las utilizadas en lenguajes convencionales , dentro de una shell se pueden especificar funciones y pasarle parámetros. La sintaxis de una función sería :

```
nombre_funcion()

{

... comandos ...

}
```

Las variables de entorno de la función son las mismas que las de la propia shell-script ; debemos imaginar que la propia función se halla "incluida" en la porción de código desde donde la invocamos . Por tanto , una variable definida en una función queda definida en la shell y viceversa .

La excepción se produce con los parámetros posiciones ; el $1 , $2 .... cambian su sentido dentro de las funciones , en las cuales representan los parámetros con los que se la ha llamado.

Veamos para variar un ejemplo :

```
# programa de prueba - llamar con parámetros

echo "Hola $1"

pinta()

{

echo "Hola en la función $1"

}
```

pinta "gorgorito"

Veamos el resultado de la ejecución :

```
# sh prueba "probando"
```

Hola probando

Hola en la función gorgorito

La variable -$1- , dentro del programa toma el valor del primer parámetro (probando) , y dentro de la función , el parámetro que se le ha pasado a la función (gorgorito). Una vez la función termina , e l $1 vale lo mismo que al principio.

La función nos puede devolver códigos de retorno utilizando la cláusula "return <codigo de error>".

- Ejecución en segundo plano : & , wait y nohup.

Si un comando en la shell termina en un ampersand -&- , la shell ejecuta el comando de manera asíncrona , es decir , no espera a que el comando termine .

La sintaxis para ejecutar un comando en segundo plano es :

comando &

Y la shell muestra un numerito por la pantalla , indicativo del número del proceso que se la lanzado. Hay que tener en cuenta que ése proceso es hijo del grupo de procesos actual asociado a nuestro terminal ; significa que si apagamos el terminal ó terminamos la sesión , tal proceso se cortará.

Para esperar a los procesos en segundo plano , empleamos el comando "wait" que hace que la shell espere a que todos sus procesos secundarios terminen.

Siguiendo con lo de antes , hay veces que ejecutamos :

comando_lentisimo_y_pesadisimo &

Y queremos apagar el terminal e irnos a casa a dormir ; a tal efecto , existe el comando "nohup" (traducción : no cuelgues , es decir , aunque hagas exit ó apagues el terminal , sigue) , que independiza el proceso en segundo plano del grupo de procesos actual con terminal asociado.

Lo que esto último quiere decir es que UNIX se encarga de ejecutar en otro plano el comando y nosotros quedamos libres de hacer lo que queramos.

Una duda : al ejecutar un comando en background , la salida del programa nos sigue apareciendo en nuestra pantalla , pero si el comando nohup lo independiza , que pasa con la salida ? La respuesta es que dicho comando crea un fichero ll amado "nohup.out" en el directorio desde donde se ha lanzado , que contiene toda la salida , tanto normal como de error del comando.

Ejemplo sobre cómo lanzar comando_lentisimo_y_pesadisimo :

```
# nohup comando_lentisimo_y_pesadisimo &

Sending output to nohup.out

12345

#
```

El PID es 12345 , y la salida del comando la tendremos en el fichero "nohup.out" , el cual es acumulativo ; si lanzamos dos veces el nohup , tendremos dos salidas en el fichero.

8 Aritmética.

La aritmética es denotada encerrando una expresión aritmética dentro de ((...)). Las expresiones aritméticas utilizan los operadores, precedencias y funciones de librería matemática del lenguaje ANSI C.

La sentencia expr evalúa una expresión y la muestra en la salida estándar. La expresión normalmente consiste de dos números ó variables de contenido numérico y un operador de suma , resta , multiplicación ó división.

Son válidos los comandos siguientes:

expr 100 "+" 1 # saca en pantalla 101

expr 100 "-" 1 # saca en pantalla 99

expr 100 "*" 2 # OJO CON LAS COMILLAS DOBLES- Previenen a la shell de sustituciones.

Bueno , todo el mundo sabe lo que es 100 * 2 , no?.

expr 100 "/" 2

Por tanto , podemos escribir :

pepe=`expr 10 "*" 5` # y la variable pepe vale 50.

ó incluso :

pepe=0

pepe=`expr $pepe "+" 1`

Esto último es bastante menos evidente . Para comprenderlo , hay que creerse que la shell ejecuta lo siguiente :

- Al principio , $pepe vale 0.

- En cualquier expresión de asignación , PRIMERO se calcula el resultado y DESPUES se ejecuta la asignación. Por tanto, lo primero que la shell hace es "expr 0 + 1".

- El "1" resultante va a sacar por la salida estándar. Pero como hemos puesto las comillas de ejecución , se asigna a pepe.

- Al final , $pepe vale 1.

Pues ya se pueden ejecutar bucles con contadores. Considerese el siguiente programa :

cnt=0

while [$cnt -lt 50]

```
do

cnt=`expr $cnt "+" 1`

echo "Vuelta numero $cnt"

done
```

Se autoexplica.

III PROGRAMACIÓN EN SH.

1 Introducción.

Un script, que es un archivo que contiene comandos shell, es un programa shell.

2 El shell como lenguaje de programación.

La shell , además de interpretar y ejecutar comandos , tiene primitivas de control de ejecución de programas tales como sentencias condicionales y bucles.

La interpretación del lenguaje se lleva a cabo prácticamente en tiempo real; a medida que va interpretando va ejecutando.

Los programas , como se ha mencionado antes , se interpretan en tiempo de ejecución. Por tanto , la codificación de una shell-script es sumamente sencilla en el sentido en el que basta con escribir en un fichero de texto l os comandos y ejecutarlo.

- Variables.

Dentro de una shell , existen las variables de entorno que hayamos definido anteriormente , bien en la misma , en otra ó en los ficheros profile de inicialización. Además de éstas , existen otras que detallamos a continuación :

$0 : Su contenido es el nombre de la shell-script que estamos ejecutando.

$1 , $2 : Primer y segundo parámetro posicional.

$# : Número de parámetros que han pasado a la shell.

$* : Un argumento que contiene todos los parámetros que se han pasado ($1 , $2 ...) menos el $0.

$? :Número donde se almacena el código de error del último comando que se ha ejecutado.

$$:Número de proceso actual (PID)

$! :Ultimo número de proceso ejecutado.

:COMENTARIO : Todo lo que haya a la derecha de la almohadilla se toma como comentario.

Ejemplo : Supongamos que hemos escrito la siguiente shell-script llamada "prueba.sh" :

echo "La script se llama $0"

echo "Me han llamado con $# argumentos"

echo "El primero es $1"

echo "Y todos son $*"

echo "Hasta luego lucas!"

Y la podemos ejecutar de dos maneras :

1) Directamente :

```
# sh prueba.sh uno dos
```

2) Dando permisos y ejecutando como un comando:

```
# chmod 777 prueba.sh

# prueba.sh uno dos
```

La salida :

Me han llamado com 2 argumentos

El primero es uno

Y todos son uno dos

Hasta luego lucas

Hemos visto que los comandos se separan por líneas , y se van ejecutando de forma secuencial. Podemos , no obstante , poner varios comandos en la misma línea separandolos por punto y coma ';'.

Además , podemos agrupar comandos mediante paréntesis , lo cual permite ejecutarlos en un subentorno (las variables que usemos no nos van a interferir en nuestro proceso "padre")

```
# ( date ; who ) | wc -l
```

Normalmente , ejecutar una shell implica crear un proceso hijo , y el proceso padre (normalmente , nuestra sesión inicial de shell) espera a que dicho proceso acabe para continuar su ejecución (si

nosotros lanzamos un pro grama shell-script , hasta que éste no acaba (hijo) , no nos aparece en pantalla el inductor de comandos '#' (padre)).

Por definición , en UNIX un proceso hijo , al rodar en un espacio de datos distinto , hereda varias cosas del padre , entre ellas todas las variables de entorno ; pero por ello , no puede modificar el entorno del padre (si modificamos en una shell script el contenido , por ejemplo , de "TERM" , al acabar dicha shell y volver al padre la variable continúa con su valor original. Hay situaciones en las cuales necesitamos que una shell modifique nuestro entorno actual , y a tal efecto se ejecuta con un punto (.) delante de la shell-script.

Es mejor ver éste último aspecto mediante un programa en shell.script : supongamos una shell que se llame "tipoterm" que nos pida el terminal y nos ponga la variable TERM de acuerdo a ésta entrada :

```
# script para pedir el tipo de terminal

echo "Por favor escriba que terminal tiene :"

read TERM

echo "Ha elegido --- $TERM"
```

Si la ejecutamos como

```
# tipoterm
```

al volver al '#' NO se ha modificado la variable! Para que SI la modifique , se llamaría como:

```
# . tipoterm
```

Hay que suponerse al punto como un "include" , que en vez de crear un proceso hijo "expande" el código dentro de nuestra shell actual.

- Comando read

Con el fin de permitir una ejecución interactiva , existe el comando "read <nombre_variable>" , el cual , en el momento de ejecución , espera una entrada de datos por teclado terminada en <INTRO> ; lo que han introducido por el teclado va a la variable especificada.

Supongamos la siguiente shell-script :

```
echo "Como te llamas?"

read nom

echo "Hola $nom"
```

Ejecución :

Como te llamas ?

jose (aquí escribimos "jose" y pulsamos <INTRO>)

Hola jose

el comando "read" , ha cargado "jose" en "nom".

3 at, batch, cron.

Comandos a ejecutar en diferido : at , batch y cron.

Estos tres comandos ejecutan comandos en diferido con las siguientes diferencias :

AT lanza comandos una sola vez a una determinada hora.

BATCH lanza comandos una sola vez en el momento de llamarlo.

CRON lanza comandos varias veces a unas determinadas horas , días ó meses.

Estos comandos conviene tenerlos muy en cuenta fundamentalmente cuando es necesario ejecutar regularmente tareas de administración ó de operación. Ejemplos de situaciones donde éstos comandos nos pueden ayuda r son :

- Necesitamos hacer una salva en cinta de determinados ficheros todos los días a las diez de la mañana y los viernes una total a las 10 de la noche = CRON.

- Necesitamos dejar rodando hoy una reconstrucción de ficheros y apagar la máquina cuando termine (sobre las 3 de la mañana) , pero nos queremos ir a casa (son ya las 8) =AT

- Necesitamos lanzar una cadena de actualización , pero están todos los usuarios sacando listados a la vez y la máquina está tumbada = BATCH

-Comando at <cuando> <comando a ejecutar>

Ejecuta , a la hora determinada , el <comando>. Puede ser una shell-script , ó un ejecutable.

Este comando admite la entrada desde un fichero ó bien desde el teclado. Normalmente , le daremos la entrada usando el "here document" de la shell.

El "cuando" admite formas complejas de tiempo. En vez de contar todas , veamos algunos ejemplos que seguro que alguno nos cuadrará :

* Ejecutar la shell "salu2.sh" que felicita a todos los usuarios , pasado mañana a las 4 de la tarde:

```
# at 4pm + 2 days <<EOF

/usr/yo/salu2.sh

EOF
```

* Lanzar ahora mismo un listado:

```
at now + 1 minute <<EOF

"lp -dlaserjet /tmp/balance.txt"

EOF
```

* Ejecutar una cadena de reindexado de ficheros larguísima y apagar la máquina:

```
at now + 3 hours <<EOF

"/usr/cadenas/reind.sh 1>/trazas 2>&1 ; shutdown -h now"

EOF
```

* A las 10 de la mañana del 28 de Diciembre mandar un saludo a todos los usuarios :

```
at 10am Dec 28 <<EOF

wall "Detectado un virus en este ordenador"

EOF
```

* A la una de la tarde mañana hacer una salvita :

```
at 1pm tomorrow <<EOF

/home/Salvas/diario.sh

EOF
```

De la misma manera que el comando nohup , éstos comandos de ejecución en diferido mandan su salida al mail , por lo que hay que ser cuidadoso y redirigir su salida a ficheros personales de traza en evitación de saturar el directorio /var/mail.

El comando "at" asigna un nombre a cada trabajo encolado , el cual lo podemos usar con opciones para consultar y borrar trabajos :

at -l : lista todos los trabajos en cola , hora y día de lanzamiento de los mismos y usuario.

at -d <trabajo>: borra <trabajo> de la cola.

Ya que el uso indiscriminado de éste comando es potencialmente peligroso , existen dos ficheros que son inspeccionados por el mismo para limitarlo : at.allow y at.deny. (Normalmente residen en /usr/spool/atjobs ó en /var/a t).

at.allow:si existe, solo los usuarios que esten aquí pueden ejecutar el comando "at".

at.deny:si existe, los usuarios que estén aquí no estan autorizados a ejecutar "at".

El "truco" que hay si se desea que todos puedan usar at sin tener que escribirlos en el fichero , es borrar at.allow y dejar sólo at.deny pero vacío.

- Batch .

Ejecuta el comando como en "at" , pero no se le asigna hora ; batch lo ejecutará cuando detecte que el sistema se halla lo suficientemente libre de tareas. En caso que no sea así , se esperará hasta que ocurra tal cosa.

-Cron.

No es un comando ; es un "demonio" , ó proceso que se arranca al encender la máquina y está permanentemente activo. Su misión es inspeccionar cada minuto los ficheros crontabs de los usuarios y ejecutar los comandos que allí se digan a los intervalos horarios que hayamos marcado. Como se acaba se señalar , los crontabs son dependientes de cada usuario.

Se deben seguir los siguientes pasos para modificar , añadir ó borrar un fichero crontab :

1 - Sacarlo a fichero usando el comando "crontab -l >/tmp/mio" , por ejemplo.

2 - Modificarlo con un editor de acuerdo a las instrucciones de formato que se explican a continuación.

3 - Registrar el nuevo fichero mediante el comando "crontab /tmp/mio" , por ejemplo.

mira cada minuto el directorio /var/spool/crontabs y ejecuta los comandos

crontab .

El formato del fichero es el siguiente :

#minutos horas día mes mes día-semana comando (# = comentario)

minutos : de 0 a 59.

horas : de 0 a 23.

día del mes : de 0 a 31.

mes : de 0 a 12.

día semana : de 0 a 6 (0 = domingo , 1 = lunes ...)

Aparte de la especificación normal , pueden utilizarse listas , es decir , rangos de valores de acuerdo con las siguientes reglas :

8-11 : Rango desde las 8 hasta las 11 ambas inclusive.

8,9,10,11 : Igual que antes ; desde las 8 hasta las 11.

Si queremos incrementar en saltos diferentes de la unidad , podemos escribir la barra "/" :

0-8/2 : Desde las 12 hasta las 8 cada 2 horas. Equivale a 0,2,4,6,8 .

El asterisco significa "todas" ó "todo". Por tanto :

*/2 : Cada dos horas.

Ejemplos más evidentes :

```
# ejecutar una salvita todos los dias 5 minutos despues de las 12 de la
noche :

5 0 * * * /home/salvas/diaria.sh 1>>/home/salvas/log 2>&1

# ejecutar a las 2:15pm el primer dia de cada mes:

15 14 1 * * /home/salvas/mensual.sh 1>/dev/null 2>&1

# ejecutar de lunes a viernes a las diez de la noche - apagar la maquina
que es muy tarde

0 22 * * 1-5 shutdown -h now 1>/dev/null 2>&1

# ejecutar algo cada minuto

* * * * * /home/cosas/espia.sh
```

4 Variables especiales.

Una variable de entorno en la shell es una referencia a un valor. Se distinguen dos tipos : locales y globales.

Una variable local es aquella que se define en el shell actual y sólo se conocerá en ese shell durante la sesión de conexión vigente.

Una variable global es aquella que se exporta desde un proceso activo a todos los procesos hijos.

Para crear una variable local :

```
# cosa="ANTONIO ROMERO"
```

Para hacerla global

```
# export cosa
```

Para ver qué variables tenemos:

```
# set

LOGNAME=root
```

```
TERM=vt220

PS1=#

SHELL=/bin/sh
```

(.. salen mas ..)

Una variable se inicializa con la expresión <variable>=<valor> . Es imprescindible que el carácter de igual '=' vaya SIN espacios. Son lícitas las siguientes declaraciones :

```
# TERM=vt220

# TERM="vt220"
```

(TERM toma el mismo valor en ambos casos)

```
# contador=0
```

Una variable se referencia anteponiendo a su nombre el signo dólar '$'. Utilizando el comando 'echo' , que imprime por la salida estándar el valor que se le indique , podemos ver el contenido de algunas de éstas variables :

```
# echo TERM
```

TERM

(!! MUY MAL!! - Hemos dicho que la variable se referencia por $).

```
# echo $TERM
```

vt220

(AHORA SI)

Otra metedura de pata que comete el neófito trabajando con variables :

```
# $TERM=vt220
```

Y el señor se queda tan ancho. Investigando qué és esto , la shell interpreta que le queremos poner algo así como "vt220=vt220" , lo cual es erróneo.

Para borrar una variable , empleamos el comando "unset" junto con el nombre de la variable que queremos quitar , como :

echo $cosa (saca el valor de dicha variable)

ANTONIO ROMERO (el que le dimos antes)

unset cosa (BORRAR la variable)

```
# echo $cosa
```

(ya no tiene nada)

Cuidadito con borrar variables empleadas por programas nativos de UNIX , tales como TERM ! Si borráis ésta variable , el editor "vi" automáticamente dejará de funcionar.

Otro problema que es susceptible de suceder es el siguiente : supongamos una variable denominada COSA y otra denominada COSAS . La shell , en el momento de evaluar la expresión "$COSAS" , se encuentra ante la siguiente disyuntiva :

- Evaluar $COSA y pegar su contenido a la "S" (<contenido de COSA> + "S")

- Evaluar $COSAS , empleando intuición.

Cara a la galería , ambas evaluaciones por parte de la shell serían correctas , pero dependiendo de lo que nosotros queramos hacer puede producir efectos indeseados. A tal fin , en conveniente utilizar los caracteres " ;llave" -{}- para encerrar la variable que queremos expandir. De tal forma , para reflejar "COSA" , escribiríamos :

${COSA}

Y para reflejar "COSAS" ,

${COSAS}

Con lo que tenemos la seguridad de que las variables siempre son bien interpretadas . Las llaves se utilizan SIEMPRE en el momento de evaluar la variable , no de asignarle valores. No tiene sentido hacer cosas como {COSAS}=tontería.

Algunas de las variables usadas por el sistema ó por programas del mismo son:

HOME:: Directorio personal . Usado por el comando "cd" , se cambia aquí al ser llamado sin argumentos.

LOGNAME: Nombre del usuario con el que se ha comenzado la sesión.

PATH: Lista de rutas de acceso , separadas por dos puntos ‘:’ y donde una entrada con un sólo punto identifica el "directorio actual". Son válidas asignaciones como:

PATH=$PATH:/home/pepe:/home/antonio

PS1: Símbolo principal del indicador de "preparado" del sistema. Normalmente , su valor será ‘#’ -o ‘$’.

TERM: Tipo de terminal.

Podemos ver cómo se inicializan las variables consultando los ficheros de inicialización. Estos ficheros son :

/etc/profile: Fichero de inicialización global . Significa que , tras hacer login , todos los usuarios pasan a través del mismo. Inicializa variables como PATH , TERM ....

<directorio usuario>/.profile: Fichero particular , reside en el "home directory" del usuario en cuestión. Es , por tanto , particular para cada uno de ellos y es aquí donde podemos configurar cosas tale s como que les salga un menú al entrar , mostrarles el correo ...

5 Parámetros posicionales.

Algo especialmente importante, son las variable con parámetros posicionales. Estas mantienen los argumentos de los comandos en línea cuando estas son invocadas. Los parámetros posicionales tienen los nombres **1, 2, 3,** etc., esto significa que sus valores son denotados por **$1, $2, $3,** etc. Ademas existe el parámetro posicional **0**, cuyo valor es el nombre del script.

Dos variables posicionales contienen todos los paramentros posicionales (excepto el parámetro posicoinal 0): * y @. La diferencia entre ellos es sutil pero importante:

6 Comando shift.

El comando shift es usado para cambiar argumentos a la izquierda y es usualmete utilizado para obtener acceso a los argumentos que son mayores a >$9. (Realmente, esto es necesario solo con el Bourne Shell desde que el Korn y el Bash permite argumetos >9).

El siguiente ciclo procesara todos los argumentos pasados al shell utilizando el comando shift hasta que no haya argumentos en la izquierda.

```
while [ "$1" ]
do
      process $1
      shift
done
```

7 Substitución del comando.

Dependiendo de cuales sean las comillas utilizadas en una expresión , los resultados son los siguientes:

- Carácter backslash \ :Quita el significado especial del carácter a continuación de la barra invertida.

- Comillas simples ‘ ’ :Quitan el significado especial de todos los caracteres encerrados entre comillas simples.

- Comillas dobles " " :Quitan el significado especial de todos los caracteres EXCEPTO los siguientes : $ (dolar) , \ (backslash) y ` (comilla de ejecución).

- Comillas de ejecución ` ` :Ejecutan el comando encerrado entre las mismas y sustituyen su valor por la salida estándar del comando que se ha ejecutado .

Es mejor , sobre todo en el último caso , ver algunos ejemplos :

- Para sacar un cartel en pantalla que contenga comillas , deberemos "escaparlas" pues , si no , la shell las interpretaría , como en :

echo "Pulse "INTRO" para seguir" (MAL!! - la shell ve 4 comillas y no las sacaría !)

echo "Pulse \"INTRO\" para seguir" (AHORA SI sale bien)

- También , podríamos haber escrito el texto entre comillas simples :

echo ‘Pulse "INTRO" para seguir’ (BIEN como antes)

lo que ocurre es que de ésta manera no se interpretaría nada ; nos podría convenir algo como:

echo ‘Oye , $LOGNAME pulsa "INTRO" para seguir’

y saldría :

Oye , $LOGNAME pulsa INTRO para seguir

Lo cual no vale. Habría que poner :

echo "Oye , $LOGNAME pulsa \"INTRO\" para seguir"

y saldría :

Oye , root pulsa INTRO para seguir.

- En el caso de comillas de ejecución , podemos escribir :

echo "Oye , `logname` pulsa \"INTRO\" para seguir"

(sale bien , la shell sustituye el comando logname por su resultado)

o bien , valdrían expresiones como:

echo "Y estas en el terminal : `tty`"

Y estas en el terminal /dev/ttyp002

Hay que imaginarse , por tanto , que la shell "ve" el resultado del comando en la línea de ejecución.

Valen también , como es lógico , asignaciones a variables :

```
# TERMINAL=`tty`

# echo $TERMINAL

/dev/ttyp001
```

8 Comando test.

El comando **test** determina si el nombre obtenido es un archivo o un directorio, si es de lectura, escritura o ejecutable y cuando dos cadenas de caracteres son mas grandes que, menores que o cuando son iguales:

```
if (test $? -eq 0)
then
      echo "Existe un error en el programa"
      exit (1)
fi
```

compara dos cadenas para saber si son iguales:

```
test "myname ="lja"
return $?
```

Determina si un archivo es de lectura

```
if [-r archivo]
      echo "archivo de lectura"
fi
```

9 Estructuras de control.

Secuencias condicionales : if .. fi :

La sintaxis de ésta sentencia es :

```
if <condicion>

then

..... comandos ....

else

..... comandos ....

fi
```

(la cláusula "else" puede omitirse ; sólo se usará cuando se requiera).

La condición puede escribirse como "test <argumentos>" ó con corchetes. Es imprescindible en este último caso , poner espacios entre los corchetes y los valores.

Posibles condiciones y su sintaxis :

if [<variable> = <valor>] : variable es igual a valor. Ojo con los espacios en '=' .

if [<variable> != <valor>] : variable es distinta a valor.

if [<variable -eq <valor>] : variable es igual a valor . La variable debe contener números. En éste caso , valen las comparaciones siguientes :

-eq : Igual (equal)

-ne : Distinto (not equal)

-ge : Mayor ó igual (Greater or equal).

-le : Menor ó igual (Less or equal).

-lt : Menor que (Less than).

-gt : Mayor que (Greater than).

if [-f <fichero>] : Existe <fichero>. Ojo con los espacios.

if [-d <fichero>] : Existe <fichero> y es un directorio.

if [-s <fichero>] :Existe <fichero> y tiene un tamaño mayor de cero.

if [-x <fichero>] : Existe <fichero> y es ejecutable.

(Hay mas , pero con éstos de momento es suficiente).

En el campo <condición> vale escribir comandos , los cuales se ejecutarán y el valor de la condición dependerá de dos factores :

* Retorno 0 del comando = VERDADERO.

* Retorno != 0 del comando = FALSO.

Ejemplo de ésto último sería el siguiente programa :

```
if grep jose /etc/passwd

then # retorno del comando -grep- ha sido CERO

echo "Jose esta registrado como usuario"

else # retorno del comando grep NO ha sido CERO.

echo "Jose NO esta registrado como usuario"

fi
```

Secuencia condicional case .. esac.

Sintaxis :

```
case <variable> in

<valor> ) <comando> ( la variable es = valor , ejecuta los comandos hasta
';;' )

<comando>

;;

<valor> ) <comando>

<comando>

;;

* ) <comando> ( Clausula "otherwise" ó "default" : Si no se cumple alguna

<comando> de las anteriores ejecuta los comandos hasta ';;' )

;;

esac ( Igual que if acaba en fi , case acaba en esac )
```

Ejemplos: minimenu.sh

```
clear # borrar pantalla

echo "1.- Quien hay por ahi ?" # pintar opciones

echo "2.- Cuanto disco queda ?"

echo "3.- Nada. Salir. "

echo "Cual quieres ? : \c" # el carácter "\c" evita que el echo salte
nueva línea

read opcion # "opcion" vale lo que se ha tecleado en pantalla

case "$opcion" in # IMPORTANTE : Poner la variable como "$opcion"

1) who ;; # pues si el señor pulsa <INTRO> daría error al no valer nada.

2) df;;

3) echo "Adios";;
```

```
*) echo "Opcion $opcion Es Incorrecta" ;;

esac


# programa para paginar un fichero con mensaje al final de cada pantalla

# llamar como "mas <ficheros>"

pg -p `Pagina %d : ` -n $*


# programa para seleccionar interactivamente ficheros

# se usa dentro de una pipe como : cat `pick *` | <el comando que se
quiera , tal que lp>

for j # no tiene error! esta cosa rara permite coger todos los ficheros
del argumento

do

echo "$j ? \c" >/dev/tty

read resp

case $resp in

s*) echo $j ;; # si escribe "s" ó "si"

n*) break ;; # lo mismo

esac

done </dev/tty


# programa para borrar de un directorio los ficheros mas viejos de 7 días

# llamar como "limpia.sh <directorio>"

case $# in

0) echo "Falta argumentos"

exit 1

;;
```

```
esac

cd $1 && { find . -type f -mtime +7 -exec rm -f -- {} \; ; }


# programa que saca el date en formato dia/mes/año hora:minuto:segundo

# llamar como "fecha.sh"

d=`date '+%d/%m/%y %H:%M:%S'`

echo "FECHA Y HORA : $d"

# programa para detectar si alguien inicia sesión

# llamar como "watchfor <nombre_de_usuario>"

# Extraído del "Unix Programming Environment" de Kernighan & Pike

case $# in

0) echo "Sintaxis : watchfor usuario" ;

exit 1 ;;

esac

until who | grep "$1"

do

sleep 60

done
```

Bucles FOR.

```
Sintaxis :

for <variable> in <lista>

do

<.. comandos ..>

done
```

El bloque entre "for" y "done" da tantas vueltas como elementos existan en <lista> , tomando la variable cada uno de los elementos de <lista> para cada iteración . En esto conviene no confundirlo con los for..next existentes en los lenguajes de tipo algol (pascal , basic ...) que varían contadores .

Supongamos un programa que contenga un bucle for de la siguiente manera :

```
for j in rosa antonio

do

echo "Variable = $j"

done
```

Y la salida que produce es :

Variable es rosa

Variable es antonio

Explicación : el bloque ha efectuado dos iteraciones (dos vueltas). Para la primera , la variable -j- toma el valor del primer elemento -rosa- , y para la segunda , -antonio-.

En el campo <lista> podemos sustituir la lista por patrones de ficheros para la shell , la cual expande dichos patrones por los ficheros correspondientes ; de tal forma que al escribir

for j in *

la shell cambia el '*' por todos los ficheros del directorio actual. Por tanto , el siguiente programa :

```
for j in *

do

echo $j

done
```

equivale al comando 'ls' sin opciones - merece la pena detenerse un momento para comprender ésto.

Vale también poner en el campo <lista> comillas de ejecución junto con cualquier comando ; la construcción - for j in `cat /etc/passwd` -, por ejemplo , ejecutaría tantas iteraciones como líneas tuviese dicho fichero , y para cada vuelta , la variable -j- contendría cada una de las líneas del mismo. Por tanto , valdrían expresiones como - for j in `who` - para procesar todos los usuarios activos en el sistema , - for j in `lpstat -o ` - , para procesar todos los listados pendientes , ó - for j in `ps -e` - para tratar todos los procesos de nuestra sesión.

```
# programa para si los ficheros de un directorio ocupan mas de 5 Mb se
truncan.

# llamar como "trunca.sh <directorio>"

case $# in

0) echo "Falta argumento"

exit 1

;;

esac

cd $1 || echo "No puedo cambiar a $1 - saliendo" ; exit 1

for j in *

do

if [ ! -f $j ]

then

continue

fi

siz=`ls -ld $j | awk '{ print $5 }'

if [ $siz -gt 5000000 ]

then

echo "Truncando $j"

>$j

fi

done
```

Bucles WHILE.

Sintaxis :

```
while <condición>
```

```
do

( ... comandos ... )

done
```

Aquí las iteraciones se producen mientras que la <condición> sea verdadera (ó retorno = 0). En caso que sea falsa (ó retorno != 0) , el bucle termina.

La sintaxis de <condición> es igual que en el comando -if- .

Ejemplo :

```
while [ "$opcion" != "3" ]

do

echo "Meta opcion"

read opcion

done
```

ó también , utilizando comandos :

```
echo "Escribe cosas y pulsa ^D para terminar"

while read cosa

do

echo $cosa >> /tmp/borrame

done

echo "Las lineas que has escrito son :"

cat /tmp/borrame


# programa para contar

# llamada : contar.sh <numero>

case $# in

0) echo "Falta argumento"

exit 1
```

```
;;

esac

c=0

while [ $c -le $1 ]

do

echo $c

c=`expr $c "+" 1`

done
```

Explicación : El comando -read- devuelve un retorno VERDADERO (cero) mientras que no se pulse el carácter EOF (^D) ; por tanto , el bucle está indefinidamente dando vueltas hasta dar ése carácter.

Operadores AND / OR.

Una construcción usual en la shell , utilizada principalmente por lo compacto de su código , pero con el inconveniente de que permite oscurecer el código son el operador "OR" -||- y el operador "AND&q uot; -&&- .

El operador "OR" ejecuta el primer comando , y si el código de error del mismo es FALSO (distinto de cero) , ejecuta el segundo comando.

El operador "AND" ejecuta el primer comando , y si el código de error del mismo es VERDADERO (igual a cero) , ejecuta el segundo comando.

Veamos un ejemplo y , por favor , reléase a continuación los dos párrafos anteriores :

cd /home/basura && rm -f *

Explicación : nos cambiamos al directorio indicado. Solamente en el caso de haber tenido éxito , nos cargamos todos los ficheros del directorio actual.

ls /home/basurilla || mkdir /home/basurilla

Explicación : El comando ls falla si no existe el directorio indicado . En tal caso , se crea.

banner "hola" | lp && echo "Listado Mandado" || echo "Listado ha cascado"

Explicación : El comando son dos , el banner y el lp . Si por lo que sea no se puede imprimir , da el mensaje de error ; si va todo bien, sale Listado Mandado .

- Depuración de "shell scripts".

Si bien los métodos utilizados para esto son bastante "toscos" , ha de tenerse en cuenta que la shell NO se hizo como un lenguaje de programación de propósito general . Cuando se requieren depuradores , ha de acudirse bien a lenguajes convencionales ó bien a intérpretes más modernos y sofisticados (y más complicados , por supuesto) , tales como el TCL (Task Control Language) ó el PERL , los cuales si bien son de libre dominio no vienen "de fabrica" en todos los equipos.

Normalmente , emplearemos el comando "set" , el cual modifica algunos de los comportamientos de la shell a la hora de interpretar los comandos :

set -v : Verbose . Saca en la salida de error su entrada (es decir , las líneas del script según los va leyendo , que no ejecutando, pues primero se lee y después se ejecuta , que esto es un intérprete y no hay que olvidarlo).

set -x :Xtrace . Muestra cada comando según lo va ejecutando por la salida de error , antecedido por un "+".

set -n :Noexec . Lee comandos , los interpreta pero NO los ejecuta. Vale para ver errores de sintaxis antes de probarlo de verdad.

set -e :Errexit. Terminar la ejecución inmediatamente si alguno de los comandos empleados devuelve un retorno distinto de VERDADERO (0) y NO se evalúa su retorno . El retorno de un comando se determina evaluado en las siguientes sentencias :

if..fi , while do..done , until do..done.

a la izquierda del operador AND/OR (-||- ó -&&-).

Comandos trap/exec/exit.

Como se vio en un capítulo anterior , cualquier proceso UNIX es susceptible de recibir señales a lo largo de su tiempo de ejecución. Por ejemplo , si el sistema se apaga (shutdown) , todos los procesos recibe n de entrada una señal 15 (SIGTERM) , y , al rato , una señal 9 .Por ahora , recordaremos que solamente la señal 9 (SIGKILL) no puede ser ignorada ni redirigida . Un programa puede cambiar el tratamiento que los procesos hacen respecto de las señales , que suele ser terminar.

Las shell-scripts pueden efectuar cosas dependiendo de la señal que reciban , usando el comando "trap <comandos> <señales>" . Este comando se suele poner al principio de la shell para que siga vigente a lo largo de toda la ejecución.

Así , podemos directamente ignorar una señal escribiendo lo siguiente :

trap "" 15

(Si llega la señal 15 (SIGTERM) , no hagas nada)

O evitar que nos pulsen Control-C , y si lo hacen , se acabó :

trap ‘echo "Hasta luego lucas" ; exit 1’ 2 3 15

(Si llega cualquiera , sacar el mensaje y terminar la ejecución con código de retorno 1).

Ojo con la ejecución de subshells - las señales ignoradas (las del trap ‘’) se heredan pero las demás vuelven a su acción original.

El comando ‘exec <comando>’ , aparte de una serie de lindezas sobre redireccionamientos cuyo ámbito queda fuera de éste manual , reemplaza el proceso de la shell en cuestión con el del comando , el cual debe s er un programa ó otra shell-script. Las implicaciones que esto tiene son que no se vuelve de dicha ejecución. Veamoslo , para no perder la costumbre , con un ejemplo :

Caso 1:

```
echo "Ejecuto el comando ls"

ls

echo "Estamos de vuelta"
```

Caso 2:

```
echo "Ejecuto el comando ls"

exec ls

echo "Estamos de vuelta"
```

En el caso -1- , la shell actual ejecuta un hijo que es el comando "ls", espera a que termine y vuelve , es decir , sigue ejecutando el echo "Estamos de vuelta". Sin embargo , en el caso -2- esto no es suerte, hacer exec implica que el número de proceso de nuestra shell pasa a ser el del comando "ls" , con lo que no hay regreso posible , y por tanto , el echo "estamos de vuelta" , NUNCA podría ejecutarse.

Por tanto , al ejecutar un nuevo programa desde exec , el número de proceso (PID) no varía al pasar de un proceso a otro.

-Comando exit <código de retorno>.

Como ya hemos visto parcialmente , éste comando efectúa dos acciones:

Termina de inmediato la ejecución del shell-script , regresando al programa padre (que lógicamente podría ser otra shell ó directamente el inductor de comandos).

Devuelve al proceso antes citado un código de retorno , que podremos averiguar mirando la variable -$?- .

Al terminar una shell script , el proceso inmediatamente antes de acabar , recibe una señal 0 , útil en ocasiones para ver por dónde hemos salido usando el comando "trap".

IV PROGRAMACIÓN EN CSH.

Bill Joy y los estudiantes de la Universidad de Califormia en el campo de Berkeley crearon una versión del shell llamda C Shell que es muy utilizada por los programadores de C.

1 Variables del shell.

El C Shell ofrece dos tipos de variables: regular (local) o environment (global). El C Shell utiliza **set** y **setenv** para establecer estos dos tipos de variables

```
set variable=valor
```

La sintaxis del C Shell es parecida a la del lenguaje C y ofrece todos los operadores condicoinales (==, >, etc.).

2 Asignación y substitución de variables.

```
set Temp.=/usr/tmp
set month=01
```

3 Aritmética y operadores para expresiones.

```
if ( $variable + 1 > $maximum ) then
      @ var1 += 5
      @ var2 -
endif
```

4 Estructuras de control.

Estructuras de control del C Shell

Estructura	C Shell
IF	if (…)
THEN	then
ELSE-IF	else if
ELSE	else
ENDIF	endif
CASE	switch case:value
	breaksw
	default:
	endsw
FOR	for each
	end
REPEAT	repeat
WHILE	while

If

Las versiones de Bourne y del C Shell del **if** son casi idénticas. El C Shell utiliza los paréntesis “( )” en lugar del paréntesis cuadrado “[ ]” para encerrar la expresión a ser evaluada; **then** debe aparecer en la misma línea que el **if**; y el C Shell utiliza el **endif** en lugar del **if**:

```
if ( -r filename ) then

      cat filename
else
      echo "Introduzca los datos para el filename"
      cat > filename
endif
```

Switch

```
switch $variable
      case 'a' :
            wathever
            breaksw
      case '10:
            wathever
            breaksw
      default:
            default actino
            breaksw
endsw
```

foreach

```
foreach variable (variable (value1 value2 ...)
      action on $variable
end

foreach file ( chapter* )
ed - $file <<eof!
g/shell/s//Shell/g
w
q
eof!
end
```

while

```
while (expression )
      actions
end

set month=1
while (${month} < 12)
      process ${month}
      @ month +=1
end
```

V EL EDITOR DE TEXTO ED.

1 Introducción.

El editor **Ed** es el editor estándar en linea, este puede utilizar comandos tecleados en linea como sigue:

```
ed file <<eof!
/First/
a
These are the times that try men's souls
And mine too for that matter.
.
w
q
eof!
```

Lo escrito encuentra la línea con la primer ocurrencia de la palabra First y agrega las siguientes dos líneas después de cada línea. Esta facilidad es ocasionalmente útil en los comandos del Shell.

2. Comandos de Edición

Reading a File into the Buffer
The r command may be used to read a previously created text file into the editor buffer and it takes the form:

```
r filename
```

Lines in the editor's buffer have line numbers starting from 1 and increasing in steps of 1. These line numbers are not part of the text but are used to refer to particular lines in ed commands. Thus any line may be located by typing its line number in response to an editor prompt. So, if you read in the file Longfellow and type 3 :

```
ed
Editor
>r Longfellow
>3
And, departing, leave behind us
>
```

the third line in the buffer is displayed on the screen. Ed has the concept of a current line and, in most cases, it is the current line that is assumed if line numbers are not specified explicitly. For example, if a p (print) command is typed on its own, the current line is displayed on the terminal screen. Note that the commands r, w and = are among the exceptions to this rule. The address of the current line is represented by a dot and, therefore, the editor command . displays the current line on the terminal screen. Thus the commands . and p, each given on its own, are equivalent in that they both display the current line at the terminal.
There is also a special address $ which represents the last line in the buffer. Thus, if the address $ is typed in response to an editor prompt, the last line in the buffer will be displayed (and become the

current line). Most commands may be prefixed by a line number or by a range of line numbers. For example, typing:

```
2,4w fred
```

will write lines two to four inclusive of the current buffer to the file named fred. If the line range is omitted as in the first example, the entire buffer is written out. Similarly :

```
1,7p
```

will display lines 1 to 7 inclusive of the buffer. The r command reads the contents of the named file into the buffer after the last line if there is already text in the buffer, unless a line number is given explicitly, in which case the file is read in after the addressed line. Thus:

```
.r filename
```

reads the contents of the named file into the buffer after the current line, and:

```
0r filename
```

reads those contents into the beginning of the buffer, after the (non-existent) line zero. After the read is done, the last line read from the named file becomes the current line.

The e Command
In contrast, the edit command **e** always causes the entire contents of the buffer to be deleted before the named file is read in:

```
e filename
```

This allows a series of edits on different files to be done in a single editing session.
Deleting Text from the Buffer The d command deletes lines from the buffer. For example:

```
2,5d
```

will delete lines 2 to 5 inclusive from the buffer. The remaining lines in the buffer will be immediately renumbered so that the old line number 6 will become the new line number 2, and so on. With this in mind, you may find that it is more convenient to edit files from the last line upwards, rather than from the first line downwards as seems more natural at first sight. A d command by itself will delete the current line. The following line then becomes the current line, unless this would be beyond the end of the buffer, in which case the current line is set to the line that immediately preceded the first deleted line.

The = Command
The command = may be used to determine the line number of a particular line. By default, it displays the line number of the last line in the buffer (in other words, typing = is equivalent to typing $=). To determine the line number of the current line, use the command .= .

Displaying Lines of Text
The list command l may be used to display lines with a representation of any non-printable characters contained in the line. It may be used anywhere the print command p could be used. For example:

```
1,$l
```

displays the entire contents of the buffer just as with the p command. Unlike the p command, however, tab characters are represented by >!bs!- so that tabs and spaces may be distinguished, and backspace characters are represented by <!bs!-. Note that these characters are overprints so they will appear as > and < respectively, on most video terminals. This representation makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace character where a DEL was intended to correct a typing error. The l command also folds long lines for display to the screen. Any line that exceeds 72 characters is displayed on more than one line with each line except the last terminated by a backslash \ so you can see it was folded. This is intended to make long lines easier to read on narrow terminals. The l command will represent other non-printing characters by their octal value preceded by a backslash, for example, \07 (bell character), \16 (shift-out character). These characters are usually unwanted and were probably accidentally inserted into the file by pressing inadvertently the CONTROL key or some other special function key. Since these combinations all represent a single character, it is easy to remove them by using the substitute command. Suppose the current line had been listed:

```
.l
this line contains a\07 bell character
```

typing:

```
s/a. b/a b/l
this line contains a bell character
```

would remove the unwanted character without the need to type in the exactly matching control sequence.

Moving Lines

The move command m allows one or more lines to be moved from their current position in the buffer to a position following a specified line. For example:

```
5,20m$
```

moves lines 5 to 20 inclusive to the end of the buffer. To reverse the order of two adjacent lines, the command:

```
m+
```

is sufficient, and is interpreted to mean "move the current line to a position following the line after the current line". After the line or lines have been moved, the last line moved becomes the new current line.

Joining Lines Together

Lines may be joined together by using the j command. For example, the command:

```
j
```

given by itself joins the current line to the line following the current line. No spaces are inserted, so these may need to be substituted in before or after the j command is used. Any number of lines may be joined together merely by specifying the addresses of the first and last lines. Thus:

```
1,$j
```

joins all the lines in the buffer into one long line.

2. Editando por contexto

Finding Text
Except for very small files, locating lines by their line number is not very convenient. One of the most useful facilities of ed is the ability to locate lines containing a given character string. To find and display the next line in the buffer containing a given string, a command of the following form should be given:

```
/character-string/
```

For example:

```
/sublime/
```

will find and display the next line in the buffer containing the string sublime. The search starts at the line following the current line, rather than at the first line in the buffer, and the search is cyclic so that, if the end of the buffer is reached, the editor will go back to the beginning of the buffer and carry on searching to the line preceding the current line if necessary. If the character ? is used to delimit the character string, rather than the / character, as in:

```
?sublime?
```

the search for a line containing sublime starts at the line preceding the current line and proceeds backwards through the buffer, rather than forwards. If a line is found containing the specified character string, that line becomes the current line. If the search is unsuccessful, the editor displays the rather cryptic:

```
??
```

at the terminal, and the current line remains as it was before the command was given. One important property of this character string (which, incidentally, is called a "regular expression") is that certain characters have special meanings. These characters are:

```
$  *  .  [  \  ]  ^
```

and the regular expression delimiters / and ? To use these characters literally in a string, suppressing their special meaning, they should be preceded by a backslash character \ . For example, to find a line containing the string VAL*3 the following command could be used:

```
/VAL\*3/
```

Simple Substitution
Probably the most useful command in the editor is the s (substitute) command which takes the form:

```
s/regular-expression/replacement/
```

For example:

```
s/acorn/oak/
```

will replace, in the current line, the first occurrence of the string acorn with the string oak . The s command may be prefixed with a line number or a line range so, for example:

```
1,50s/oak/ash/
```

will replace the first occurrence of the string oak with the string ash in lines 1 to 50 inclusive.

The ampersand character & has a special meaning when used in the second part of a substitute command (that is, as part of the replacement). As with the other special characters, this special meaning may be suppressed by preceding the & with a back slash character \ .

Advanced Substitution
The p command may be added to the end of the substitute command to display the line immediately after the substitution has been made. So, if the current line contained:

```
the lion, the witch and the wardrobe
```

typing:

```
s/the/that/p
```

would change the first occurrence of "the" to "that" and display the changed line:

```
that lion, the witch and the wardrobe
```

By placing a g (global) command on the end of the substitute command, all the occurrences of "the" may be changed, rather than just the first, so that:

```
s/the/that/gp
```

would then produce:

```
that lion, that witch and that wardrobe
```

Undoing Substitution
The undo command may be used after a substitute command to reverse the effect of the substitution. Thus the command:

```
u
```

should restore the buffer to its state prior to the last s command. If the last s command involved a substitution over several lines, only the last line in which that substitution occurred can be restored.

3. Aplicación de ejemplo

The following is a typical editing session using ed:

```
ed
Editor
>a
The wives of grate men all remind us
We can use the UNIX editor
We can make our wives sublime
And, on parting, leave behind us
Footmarks on the sands of tome
.
```

```
>w Longfellow
159
>1,$p
The wives of grate men all remind us
We can use the UNIX editor
We can make our wives sublime
And, on parting, leave behind us
Footmarks on the sands of tome
>/tome/                                1
Footmarks on the sands of tome
>s/tom/tim/                            2
>p                                     3
Footmarks on the sands of time
>1,$s/wives/lives/                     4
>2                                     5
We can use the UNIX editor
>d                                     6
>     1s/grate/great/
>p
The lives of great men all remind us
>/on parting/                          7
And, on parting, leave behind us
>s//departing/                         8
>1,$p
The lives of great men all remind us
We can make our lives sublime
And, departing, leave behind us
Footmarks on the sands of time
>$s/marks/prints/                      9
>w Longfellow
131
>q
```

Notes:

1. Locate the string tome and display the line.
2. Correct tome to time .
3. Print out the corrected line.
4. Substitute the first occurrence of wives with lives on each line.
5. Make line 2 the current line and display it.
6. Delete the current line.
7. Locate a line containing the string on parting and display the line.
8. Replace on parting by departing. (Note that the null regular expression // defaults to the last used regular expression.)
9. In the last line of the buffer, substitute prints for marks.

VI EXPRESIONES REGULARES.

1 Definición.

Los patrones de búsqueda especifica patrones de búsqueda y sustitución. Las expresiones regulares están formadas por la utilización de letras y números en conjunción con caracteres especiales que actúan como operadores. Pueden ser de gran ayuda para encontrar y filtrar

información en archivos. Las utilerías de unix que mas utilizan las expresiones regulares son ed, sed, awk y las varias formas del grep

2 Concordancias.

La expresión regular suele funcionar con que solo coincida un carácter de los muchos listados. Pero en ocasiones queremos que la coincidencia sea total.

Imaginemos que nos hemos dejado todas las tildes de palabras acabadas en ion. Se nos ocurre usar
/[ion]/ como patron de busqueda.

Pero esto daría concordancia positiva con Sol -por la o- con noche -por la n- ... etc.

Para que la concordancia sea de cada caracter y en el orden adecuado, necesitamos el operador Punto "."
Por ejemplo [.ion] solo concordaria con cosas como camion o salsa lionesa

Si bueno, la "ion" de l**ion**esa no esta a final de palabra ... y eso tambien hemos de retocarlo, pero todo a su tiempo.

Aunque se os podria ocurrir que algo del tipo [.ion[^a-zA-Z]] tal vez funcionase ... ya veremos ;)

Por cierto que el punto "." no concuerda con \n \r \f \0 (newline, return, line feed, null respectivamente)

Veamos algunos ejemplos de concordancias.

Se desea todas las cadenas de tres caracteres que empiezan con la letra r y terminan con la letra n

r.n

Utilizar una expresión regular con los paréntesis cuadrados

[A-Z] Todas las palabras en mayúsculas.
[a-z] Todos los caracteres en minúsculas.
[0-9] Todos los digitos.
[A-Za-z] Todos los caracteres del alfabeto.

Hace una concordancia con la palabra Win

^Win

3. Carácteres especiales

[ ] cochetes
() parentesis
{} llaves
- guión

+ más
* asterisco
. Punto
^ circumflejo
$ dolar
? interrogante cerrado
| tuberia unix
\ barra invertida
(se usa para tratar de forma normal un caracter especial)
/ barra del 7

Mención aparte para / puesto que es el simbolo que se usa para indicar la búsqueda. El resto son todo modificadores y se pueden usar sin restricciones.

4. Definiendo Rangos

/[a-z]/ letras minúsculas
/[A-Z]/ letras mayúsculas
/[0-9]/ números
/[,'¿!¡;:\.\?]/ caracteres de puntuación
-la barra invertida hace que no se consideren como comando ni en punto ni el interrogante
/[A-Za-z]/ letras del alfabeto (del ingles claro ;)
/[A-Za-z0-9]/ todos los caracteres alfanumericos habituales
-sin los de puntuación, claro-
/[^a-z]/ El simbolo ^ es el de negación. Esto es decir TODO MENOS las letras minúsculas.
/[^0-9]/ Todo menos los números.
Para definir otros rangos, no dudeis en usar el operador de rangos "-" por ejemplo de la h a la m [h-m] ¿vale?

5. Agrupando, referenciando y extrayendo

5.1 Agrupación

En ocasiones nos interesa usar el resultado de una parte del patrón en otro punto del patrón de concordancia.
Tomemos por ejemplo una agenda en la que alguien puso su teléfono en medio de su nombre.(Pepe 978878787 Gonzalez) Queremos extraer el teléfono de esa frase y guardarlo para introducirlo en otro sitio. O para usarlo como patrón de búsqueda para otra cosa.
Lo primero que hay que hacer es agrupar. Agrupar permite que el resultado del patrón se almacene en una especie de registro para su uso posterior. El **operador de agrupamiento son los parentesis ( )**
Los registros se llaman 1, 2, 3, ... 9 según el orden del agrupamiento en el patrón.
Ejemplo (del libro de perl): /Esto es ([0-9]) prueba/
si el texto fuente es: Esto es 1 prueba.
El valor 1 seria almacenado en el registro $1

5.2 Referencias

Tenemos una serie de elementos agrupados, a los cuales se les ha asignado un valor. Ahora queremos usar ese valor en algún otro punto del patrón de concordancia. Pues para eso existen las referencias.
\1 representa la referencia al grupo 1

Ejemplo (del libro de perl): /([0-9]) \1 ([0-9])/
si el texto es : 3 3 5
El resultado es: $1 --primer grupo-- vale 3
\1 --hace referencia al resultado del primer grupo-- vale 3
$2 --segundo grupo-- vale 5
En este caso hubiese habido concordancia, no asi si el texto hubiese sido 3 5 3

5.3 Extracción

La extracción es simplemente usar el **$**1 en los mensajes que se generan despues de haber usado la expresion regular.

6. Expresiones opcionales y alternativas.

Si os habeis fijado hasta ahora todo lo que hemos visto han sido condiciones mas bien del tipo AND (Y)
Si hay un numero Y hay un texto Y no hay otro numero ... blablabla. Siempre Y, es evidente que nos faltan las condiciones de tipo opcional y alternativo.

6.1 Opcionales

el símbolo que se emplea para indicar que una parte del patrón es **opcional es el interrogante. ?**
/[0-9]? Ejemplo/
Esto concuerda tanto con textos del tipo
1 ejemplo, como con
ejemplo

6.2 Alternativas

No confundamos alternativas con opciones, la alternativa es el equivalente a la OR no lo era asi el operador opcional. Veamos la diferencia. Por cierto, el simbolo de la **alternativa es |**
/[0-9]? (EJEMPLO|ejemplo)/
a: 1 ejemplo
b: 1 EJEMPLO
c: 1
d: EJEMPLO
a,b,d concuerdan con el patron. Sin embargo c no concuerda con el patrón! Porque ejemplo o EJEMPLO son alternavitas válidas, pero tiene que haber uno de los 2. No ocurre lo mismo con la opcionalidad, el 1 puede estar o no estar. Esa es la principal diferencia.

7. Contando expresiones

A estas alturas te estas preguntando qué mas se puede hacer con una expresión. Pues la verdad es que ya queda poca cosa, pero lo de contarlas! vamos ... eso es imprescindible.
El operador para esta ocasión son las llaves {m,n}
Donde m, n son 2 enteros positivos con n mayor o igual que m. Esto se lee asi ... la expresion debe cumplirse al menos m veces y no mas de n veces.
Podemos usar {n}o bien {n,} para indicar que queremos que se repita exactamente n veces o que se repita n veces o más respectivamente.
De tal forma que el patrón /go{1,4}l/
a: gool
b: goooooooool
c: gooool
concordaria con a y c pero no asi con c

8. Expresiones repetidas

Tenemos el **operador * asterisco** que representa que el patrón indicado debe aparecer **0 o mas veces** y el **operador + suma** que representa que el patrón indicado **debe aparecer 1 o más veces (pero almenos 1)**
Mucho ojo con el operador asterisco!!! leeros el articulo relacionado sobre conceptos avanzados de las expresiones regulares antes de usarlo o no me hago responsable de los resultados.

VII GREP.

1 Filtros.

Comando: grep <cadena a buscar> <fichero>

Misión: buscar apariciones de palabras ó textos en ficheros.

Ejemplo:

```
$ grep root /etc/passwd # busca las líneas que contienen "root" en el
fichero.

root:Ajiou42s:0:1:/:/bin/sh
```

Otro:

```
$ ps -e | grep -v constty # busca qué procesos no ruedan en la consola
```

La familia de grep , así como otros comandos tales como de (editor de líneas), sed (editor batch) y vi , manejan expresiones regulares para la búsqueda de secuencias de caracteres. Una expresión regular es un a expresión que especifica una secuencia de caracteres. Un ejemplo de esto es el comando : ls -l | grep "^d" ; el "^d" es una expresión regular que equivale a decir "si la letra -d- está al principio de la línea" ; la construcción anterior nos lista sólo los directorios.

2 Familia grep.

Las expresiones regulares conviene esconderlas de la shell encerrándolas entre comillas.

Algunos montajes de expresiones regulares son :

<.> (Punto) :cualquier carácter distinto al del fin de línea.

[abc] :la letra a , la b ó la c.

[^abc] :cualquier letra distinta a a , b ó c.

[a-z] :cualquier letra de la -a- a la -z- .

Una letra seguida de un asterisco -*- equivale a cero ó más apariciones de la letra. Por tanto , la expresión .* equivale a decir "cualquier cosa". Por tanto, el comando :

```
ls -l | grep '\.sh.*"
```

lista todos los ficheros que terminen en .sh mas los terminados en .sh <otras letras> . Nótese que el punto inicial lo hemos desprovisto de su significado "escapándolo" con un backslash -\- .

^ : al principio de una expresión regular equivale a "desde el principio de la línea".

$: al final de una expresión regular equivale a "hasta el final de la línea".

Ejemplos de ésto :

ls -l | grep '\.sh$' : saca los ficheros que SOLO acaben en .sh

Este tema se complica todo lo que se quiera . Veamos algunas expresiones regulares bastante usadas en comandos tipo sed (que veremos más tarde) para percatarse de ello :

[^] : cualquier letra diferente de espacio.

[^]* : una palabra cualquiera (varias letras no blancas).

[^]* * : una palabra seguida de un número incierto de blancos.

^[^]* * : la primera palabra de la línea y a todos los blancos que la siguen.

3 grep.

programa para listar sólo subdirectorios

llamar como "dir.sh" <argumentos opcionales>

```
ls -l $* | grep '^d'
```

```
# programa para ver quien esta conectado
```

```
# llamar como "ju <usuario> <usuario> ..."
```

```
for j in $*
```

```
do
```

```
donde=`who | grep $j | sed 's/^[^ ]* *\([^ ]*\).*/\1/p'`
```

```
if [ "$donde" ]
```

```
then
```

```
echo "$j conectado en $donde"

else

echo "$j no esta conectado"

fi

done
```

4 fgrep y egrep.

fgrep muestra las líneas de un fichero que coincidan con una lista de cadenas fijas, separadas por saltos de línea, cualquiera de las cuales puede ser coincidente.

egrep muestra las líneas de un fichero que coincidan con una determinada expresión regular extendida.

Hay dos versiones de grep que optimizan la búsqueda en casos particulares:
fgrep (fixed grep, o fast grep) acepta solamente una cadena de caracteres, y no una expresión regular, aunque permite buscar varias de estas cadenas simultáneamente;
egrep (extended grep), que acepta expresiones regulares extendidas con los operadores + ? | y paréntesis.
fgrep no interpreta metacaracteres, pero puede buscar muy eficientemente muchas palabras en paralelo, por lo que se usa mucho en búsquedas bibliográficas; egrep acepta expresiones más complejas, pero es más lento; grep es un buen compromiso entre ambos.
fgrep martes dias
busca la cadena martes en el archivo dias.

En fgrep y egrep puede indicarse la opción -f buscar.exp, donde buscar.exp es un archivo que contiene la expresión a buscar: cadenas simples para fgrep, expresiones regulares para egrep, separadas por nueva línea; las expresiones se buscan en paralelo, es decir que la salida serán todas las líneas que contengan una cualquiera de las expresiones a buscar.
Crear un archivo buscar.fgrep con las cadenas "tes" y "jue", una por línea. El comando
fgrep -f buscar.fgrep dias
extrae del archivo dias las líneas que contienen estas cadenas.

El comando grep soporta fgrep y egrep como opciones -F y -E, respectivamente.
grep -F -f buscar.fgrep dias
egrep "tes|jue" dias
grep -E "tes|jue" dias
obtienen el mismo efecto del comando anterior.
egrep "([0-9]+ab)*1234" archivo
busca cadenas comenzadas opcionalmente por un dígito y los caracteres ab, todo el paréntesis 0 o más veces, y hasta encontrar la cadena 1234.

Escribir grep -E es similar a egrep, aunque no idéntico; egrep es compatible con el comando histórico egrep; grep -E acepta expresiones regulares extendidas y es la versión moderna del comando en GNU. fgrep es idéntico a grep -F.

VIII EL EDITOR DE FLUJO SED.

1 Introducción.

Comando: sed ‘comandos de sed’ <ficheros>

Misión efectúa cambios en ‘fichero’ , enviando el resultado por la salida estándar. Pueden guardarse los resultados de éstas operaciones en un fichero como siempre , redireccionando la salida , tal que - sed ‘expresion’ fichero >/tmp/salida -.

Utilizaremos sed normalmente cuando sea necesario hacer cambios en ficheros de texto ; por ejemplo , nos puede valer para cambiar determinada variable en una serie de ficheros de profile de usuario , etc.

La explicación completa de éste comando exigiría mucho detalle ; veamos solamente algunos ejemplos de operaciones con sed :

* Cambiar , en el fichero /home/pepito/.profile , la expresión TERM=vt100 por TERM=vt220 . Usamos el comando ‘s/expresion_a_buscar/expresion_a_cambiar/g’ (s=search,g=global) :

```
sed 's/TERM=vt100/TERM=vt220/g' /home/pepito/.profile >/tmp/j && mv
/tmp/j /home/pepito/.profile
```

(si el comando ha sido OK , entonces movemos el /tmp/j).

* Meter tres blancos delante de todas las líneas del fichero /tmp/script.sh :

```
sed 's/^/ /' /tmp/script.sh >/tmp/j && mv /tmp/j /tmp/script.sh
```

(el ^ identifica el principio de cada línea).

* (mas raro): Cepillarse todos los ficheros de un directorio :

```
ls | sed 's/^/rm -f /' | sh
```

(del ls sale el fichero ; el sed le pone delante "rm -f" y el sh lo ejecuta.

2 Listas de comandos.

If you use SED at all, you will quite likely want to know these commands.

`#'

[No addresses allowed.] The # "command" begins a comment; the comment continues until the next newline. If you are concerned about portability, be aware that some implementations of SED (which are not POSIX.2 conformant) may only support a single one-line comment, and then only when the very first character of the script is a #. Warning: if the first two characters of the SED script are #n, then the -n (no-autoprint) option is forced. If you want to put a comment in the first line of your script and that comment begins with the letter `n' and you do not want this behavior, then be sure to either use a capital `N', or place at least one space before the `n'.

`s/*regexp*/*replacement*/*flags*'

(The / characters may be uniformly replaced by any other single character within any given s command.) The / character (or whatever other character is used in its stead) can appear in the *regexp* or *replacement* only if it is preceded by a \ character. Also newlines may appear in the *regexp* using the two character sequence \n. The s command attempts to match the pattern space against the supplied *regexp*. If the match is successful, then that portion of the pattern space which was matched is replaced with *replacement*. The *replacement* can contain *n* (*n* being a number from 1 to 9, inclusive) references, which refer to the portion of the match which is contained between the *n*th \(and its matching \). Also, the *replacement* can contain unescaped & characters which will reference the whole matched portion of the pattern space. To include a literal \, &, or newline in the final replacement, be sure to precede the desired \, &, or newline in the *replacement* with a \. The s command can be followed with zero or more of the following *flags*:

`g'

Apply the replacement to *all* matches to the *regexp*, not just the first.

`p'

If the substitution was made, then print the new pattern space.

`*number*'

Only replace the *number*th match of the *regexp*.

`w *file-name*'

If the substitution was made, then write out the result to the named file.

`I'

(This is a GNU extension.) Match *regexp* in a case-insensitive manner.

`q'

[At most one address allowed.] Exit SED without processing any more commands or input. Note that the current pattern space is printed if auto-print is not disabled.

`d'

Delete the pattern space; immediately start next cycle.

`p'

Print out the pattern space (to the standard output). This command is usually only used in conjunction with the -n command-line option. Note: some implementations of SED, such as this one, will double-print lines when auto-print is not disabled and the p command is given. Other implementations will only print the line once. Both ways conform with the POSIX.2 standard, and so neither way can be considered to be in error. Portable SED scripts should thus avoid relying on either behavior; either use the -n option and explicitly print what you want, or avoid use of the p command (and also the p flag to the s command).

`n'

If auto-print is not disabled, print the pattern space, then, regardless, replace the pattern space with the next line of input. If there is no more input then SED exits without processing any more commands.

`{ *commands* }'

A group of commands may be enclosed between { and } characters. (The } must appear in a zero-address command context.) This is particularly useful when you want a group of commands to be triggered by a single address (or address-range) match.

3 Selección de lineas.

Addresses in a SED script can be in any of the following forms:

`*number*'
: Specifying a line number will match only that line in the input. (Note that SED counts lines continuously across all input files.)

`*first*~*step*'
: This GNU extension matches every *step*th line starting with line *first*. In particular, lines will be selected when there exists a non-negative *n* such that the current line-number equals *first* + (*n* * *step*). Thus, to select the odd-numbered lines, one would use 1~2; to pick every third line starting with the second, 2~3 would be used; to pick every fifth line starting with the tenth, use 10~5; and 50~0 is just an obscure way of saying 50.

`$'
: This address matches the last line of the last file of input.

`/*regexp*/'
: This will select any line which matches the regular expression *regexp*. If *regexp* itself includes any / characters, each must be escaped by a backslash (\).

`\%*regexp*%'
: (The % may be replaced by any other single character.) This also matches the regular expression *regexp*, but allows one to use a different delimiter than /. This is particularly useful if the *regexp* itself contains a lot of /s, since it avoids the tedious escaping of every /. If *regexp* itself includes any delimiter characters, each must be escaped by a backslash (\).

`/*regexp*/I'

`\%*regexp*%I'
: The I modifier to regular-expression matching is a GNU extension which causes the *regexp* to be matched in a case-insensitive manner.

If no addresses are given, then all lines are matched; if one address is given, then only lines matching that address are matched.

An address range can be specified by specifying two addresses separated by a comma (,). An address range matches lines starting from where the first address matches, and continues until the second address matches (inclusively). If the second address is a *regexp*, then checking for the ending match will start with the line *following* the line which matched the first address. If the second address is a *number* less than (or equal to) the line matching the first address, then only the one line is matched.

Appending the ! character to the end of an address specification will negate the sense of the match. That is, if the ! character follows an address range, then only lines which do *not* match the address range will be selected. This also works for singleton addresses, and, perhaps perversely, for the null address.

IX AWK.

1 Introducción.

Awk esta especialmente diseñado para trabajar con archivos estructurados y patrones de texto. Dispone de características internas para descomponer líneas de entrada en campos y comparar estos campos con patrones que se especifiquen. Debido a estas posibilidades, resulta

particularmente apropiopiado para trabajar con archivos que contienen información estructurada en campos, como inventarios, listas de correo y otros archivos de bases de datos simples.

La implementación GNU, **gawk**, fue escrita en 1986 por Paul Rubin y Jay Fenlason, por consejos de Richard Stallman, John Woods también contribuyó con parte del código. En 1988 y 1999, David Trueman, con Ayuda Arnold Robbins, trabajaron duramente para hacer gawk compatible con el nuevo awk.

2 Registros y campos.

En el programa awk tipico, toda la entrada se lee de la entrada estándar (normalmente el teclado) o de los archivos cuyos nombres se especifican en la linea de comando awk. Si se especifican archivos de entrada, awk lee los datos del primer archivo hasta que alcanza el final del mismo, después lee el segundo archivo hasta que llega al final, y así sucesivamente.

La entrada se lee en unidades llamadas registros, y éstos son procesados por las reglas uno a uno. Por defecto, cada registro es un alinea del archivo de entrada. Cada registro leído es dividido automáticamente en campos, para que puedan ser tratado más fácilmente por la regla.

El lenguaje awk divide sus registros de entrada en campos. Los registros están separados por un carácter llamado el separador de registros. Por defecto, el separador de registros newline. Por lo tanto, normalmente, un registro se corresponde con una línea de texto. Algunas veces puedes necesitar un carácter diferente para separar tus registros. Es posible usar un carácter diferente mediante la llamada a la variable empotrada RS (Record Separator).

El valor RS es una cadena que dice como separar los registros; el valor por defecto es “\n”, la cadena formada por un único carácter newline. Esta es la razón por la cual un registro se corresponde, por defecto, con una línea.

RS puede tener cualquier cadena como valor, pero solamente el primer carácter de la cadena se usará como separador de registros. El resto de caracteres de la cadena serán ignorados. RS es excepcional es este sentido; awk usa el valor completo del resto de sus variables implícitas.

Awk ‘BEGIN { RS = “/”] ; { print $0 }’ Lista-BBS

Cambia el valor de RS a “/”, antes de leer ninguna entrada. Esta es una cadena cuyo primer carácter es una barra; como resultado, los registros se separarán por las barras. Después se lee el fichero de entrada, y la segunda regla en el programa awk (la acción sin patrón) impriem cada registro. Debido a que cada sentencia print añade un salto de linea al final de sui salida, el efecto de este programa awk es copiar la entrada con cada carácter barra cambiado por un salto de linea. Otra forma de cambiar el separador de registros es en la línea dew comandos, usando la característica asignación de variable.

Awk ‘...’ RS=”/” source-file

Esto fija el valor de RS a ‘/’, antes de procesar source-file.

Cuando awk lee un registro de entrada, el registro es automáticamente separado o particionado por el intérprete en piezas llamados campos. Por defecto, los campos son separados por espacioos en blanco, al igual que las palabras de una frase. Los espacios en awk significan

cualquier cadena de uno o más espacios y/o tabuladores; otros caracteres tales como newline, formfeed, etc... que son considerados como espacios en blanco por otros lenguajes no son

3. Parámetros en tiempo de ejecución

Awk sigue, como hemos indicado, un guión previamente escrito por el programador. Está compuesto por plantillas, que indican --mediante una expresión-- qué registros se desea procesar, y --mediante una acción-- qué proceso se pretende realizar con los registros previamente seleccionados.

Se puede escribir el guión directamente en la línea de mandatos, encerrado entre comillas simples, o en un archivo de texto separado. Si se opta por la segunda opción, debe escribirse el nombre del archivo tras el parámetro –f. La "f" debe ser minúscula obligatoriamente. Por tanto, cualquiera de las dos formas mostradas a continuación produce resultados equivalentes:

```
$ awk '{ print $1 }' entrada.txt
$ awk -f primer.awk entrada.txt
```

Más adelante se mostrará cómo *awk* procesa cada registro como una sucesión de "campos" separados por un carácter determinado que indica dónde termina un campo y empieza el siguiente. La situación más habitual es aquella en la cual cada registro contiene una frase y los campos son las palabras de la frase. El carácter delimitador es el espacio que separa cada palabra de la siguiente. Mediante el parámetro –F (ahora mayúscula) se indica a *awk* qué carácter debe considerar como separador de campos.

```
$ awk -F "#" -f primer.awk entrada.txt
```

También se mostrará la capacidad para usar "variables" dentro del guión, igual que en otros lenguajes de programación. Mediante el parámetro –v se puede asignar un valor a una variable desde la línea de mandatos:

```
$ awk –v fecha="16/Dic/2001" -f primer.awk entrada.txt
```

El resto de los parámetros son los nombres de los archivos de datos "entrantes". El proceso se realiza accediendo a los archivos en el orden especificado. Toma el conjunto de los archivos como si fueran uno solo. *Awk* espera recibir por la entrada estándar (*stdin*) los datos que debe procesar. Por ejemplo:

```
$ cat ejemplo.txt | awk –f primer.awk
```

Alternativamente, los parámetros indicados al final de la línea de mandatos se interpretan como nombres de archivo:

```
$ awk –f primer.awk ejemplo.txt
```

Suponga el lector que se desea actuar sobre un archivo de texto como el que se muestra a continuación. Contiene cuatro líneas de texto terminadas por el carácter de salto de línea "\n", que no se muestra en la pantalla, salvo por el efecto de escribir en la línea siguiente:

```
$ cat ejemplo.txt
```

```
La bella y graciosa moza
marchóse a lavar la ropa
la frotó sobre una piedra
y la colgó de un abedul.
$ _
```

4. Campos de entrada

Para utilizar una expresión que no sea trivial se debe introducir el concepto de “campo”. Se considerará cada registro del archivo de entrada como una sucesión de campos delimitados por un carácter dado. Este carácter es, por omisión, el espacio. Como se ha visto anteriormente, se puede indicar a *awk* que considere otro carácter como separador de campos mediante la opción –F (mayúscula) en la línea de mandatos.

Se hace referencia al contenido de cada campo usando el carácter “$” (dólar) seguido del ordinal del campo: $1, $2, $3, etc. Con la notación $0 se hace referencia a la línea entera.

Se puede forzar el proceso del registro carácter a carácter dejando la variable “separador de campos” FS sin contenido. De este modo, en $1 se tendrá el primer carácter de la línea, en $2 el segundo, etc. Incluso en este caso, el campo $0 contendrá la línea completa.

Como quiera que se está anticipando en estos ejemplos el uso de la instrucción *"print"*, se muestra a continuación el guión anterior con las modificaciones necesarias para mostrar en stdout el contenido de la primera y la segunda palabra de cada línea:

```
$ cat ejemplo.txt \
 | awk '1 { print "Primera y segunda palabras ===>", $1, $2, "<=== final
del registro." }'
Primera y segunda palabras ===> La bella <=== final del registro.
Primera y segunda palabras ===> marchóse a <=== final del registro.
Primera y segunda palabras ===> la frotó <=== final del registro.
Primera y segunda palabras ===> y la <=== final del registro.
$ _
```

Se estudiará más adelante el efecto de la coma en la instrucción "*print*". Si no se indica otra cosa, *awk* inserta un espacio por cada coma que aparezca en esta instrucción.

5. Variables

También es necesario en este punto citar la existencia de variables predefinidas en el lenguaje. El concepto tradicional de “variable” (almacén de datos en memoria, caracterizado por un nombre no repetido en el mismo contexto y capaz de contener un dato que puede ser modificado por el programa) se aplica perfectamente en este caso.

El programador puede crear sus propias variables símplemente haciendo referencia a ellas en expresiones. Las variables pueden ser escalares (con un solo nombre y valor) o vectoriales (con nombre, subíndice y valor) El subíndice puede ser una cadena arbitraria, lo cual permite crear tablas asociativas. Se emplea la notación “nombre[subíndice]” para referirse a un elemento de la

tabla. También se admite la definición de variables de naturaleza matricial, con nombre, varios subíndices y varios valores, si se usa la notación “nombre[subíndice1, subíndice2, subíndice3]”.

Las siguientes variables predefinidas están relacionadas con la información entrante:

FS (Field separator): contiene el carácter que indica a awk en qué punto del registro acaba un campo y empieza el siguiente. Por omisión es un espacio. Se puede indicar un carácter alternativo mediante una instrucción de asignación como FS = “/”. Si se deja vacío, cada lectura se realiza dejando un carácter en cada campo.

NF (Number of fields): contiene el número total de campos que contiene el registro que se está leyendo en cada momento.

RS (Record separator): contiene el carácter que indica a awk en qué punto del archivo acaba un registro y empieza el siguiente. Es “\n” por omisión.

NR (Number of record): contiene el número de orden del registro que se está procesando en cada momento.

Del mismo modo, se dispone de variables relativas a la información de salida, como las siguientes:

OFS (Output FS): contiene el separador de campos para la salida generada por awk. La instrucción print inserta en la salida un carácter de separación cada vez que aparece una coma en el código. Por ejemplo:

```
$ echo "manejo un apple" | awk –v OFS="\t" ‘1 {print $1, $2, $3}'
manejo un apple
$ _
```

ORS (Output RS): Contiene el carácter que awk escribirá al final de cada registro. Es “\n” por omisión. Obsérvese que en esta modificación del ejemplo anterior la entrada contiene DOS registros, dada la aparición del salto de línea en el centro de la cadena de caracteres entre comillas. Al modificar el carácter separador de la salida se genera un solo registro:

```
$ echo –e "manejo un apple\ncaro y elegante" \
 | awk –v OFS = "\t" –v ORS = “\t”  ‘1 {print $1, $2, $3}'
manejo un apple caro y elegante
$ _
```

6. Operadores

Son aplicables los operadores comunes en C para realizar operaciones aritméticas sobre variables con valores numéricos, y para concatenar los valores contenidos en variables con valores alfanuméricos. Así, se tiene:

+	Suma
-	Resta
*	Multiplicación
/	División

%	Resto
^	Exponenciación
Esp	Concatenación

También se dispone de los operadores comunes

!	Negación
A ++	Incrementar positivamente en una unidad
A --	Incrementar negativamente en una unidad
A += N	Incrementar positivamente en N unidades
A -= N	Incrementar negativamente en N unidades
A *= N	Multiplicar por N
A /= N	Dividir por N
A %= N	Reemplazar por el resto de la división por N
A ^= N	Sinceramente, no sé qué demonios hace este operador
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual a
!=	Distinto de
?:	Formato alternativo de decisión (if/else)

7. Instrucciones de control

Algunas instrucciones del lenguaje C están presentes en *awk* de manera simplificada. Se dispone de la construcción condicional *"if/else"*, que se codifica de la manera habitual:

```
if( expression ) statement [ else statement ]
```

En algunas implementaciones se puede complicar hasta añadir varias instrucciones mediante anidamiento de acciones:

```
{
  print "Procesando registro", NR;
 if( $1 > 0 )
 {
  print "positivo";
  print "El primer campo tiene:", $1;
 }
 else
 {
  print "negativo";
```

```
 }
}
```

La primera forma de la construcción *"for"* se maneja igual que en el lenguaje C, es decir:

```
for( expression; expression; expression ) statement
```

donde la primera expresión suele poner valor inicial al índice. En la segunda se controla hasta qué momento la ejecución debe continuar. La tercera marca el nivel de incremento de la variable índice.

```
{
 for( i = 0; i < 10; i ++ )
 {
  j = 2 * i;
  print i, j;
 }
}
```

En su segunda forma, la construcción *"for"* recorre los elementos de una matriz *(array)*. Dado el carácter avanzado de este concepto, quedará para un segundo artículo su manejo.

La construcción *"do/while"* se emplea en la forma habitual. En el caso actual se realiza un sencillo contador de cero a diez:

```
BEGIN { i = 0; do {print i; i ++} while ( i < 10 ) }
```

La instrucción *"break"* permite abandonar un bucle anticipadamente. Para descartar el resto del código del bucle y empezar una nueva iteración se usa *"continue"*. La instrucción *"next"* descarta todas las plantillas que siguen, y pasa a realizar el proceso del siguiente registro. Finalmente, la instrucción *"nextfile"* termina anticipadamente el proceso de un archivo de entrada para pasar al siguiente.

8. Funciones

Las funciones predefinidas del lenguaje responden al concepto habitual en otros lenguajes de programación. La llamada a una función se resuelve en tiempo de ejecución. Cuando se evalúa una expresión, se realiza la sustitución de la llamada a la función por el valor que ésta devuelve. Por ejemplo:

```
$ awk 'BEGIN { print "El seno de 30 grados es", sin( 3.141592653589 / 6
); }' /dev/null
El seno de 30 grados es 0.5
$ _
```

Se dispone de las funciones matemáticas y trigonométricas exp, log, sqrt, sin, cos y atan2. Además, son de mucha utilidad las siguientes:

length( )	Longitud del parámetro en bytes

rand( )	Número al azar entre cero y uno
srand( )	Inicia la semilla de generación al azar
int( )	Devuelve el parámetro convertido en un entero
substr(s,m,n)	Devuelve la subcadena de s en m con longitud N
index(s,t)	Posición de s donde aparece t, o cero si no está.
match(s,r)	Posición de s en que se cumple la expresión r.
split(s,a,fs)	Devuelve s en elementos separados por fs
sub(r,t,s)	Cambia en s la cadena t por r. Es $0 por omisión
gsub(r,t,s)	Igual, pero cambia todas las t en la cadena
sprintf(f,e,e…)	Devuelve cadena de formato f con las “e”
system(cmd)	Ejecuta cmd y devuelve el código de retorno
tolower(s)	Devuelve s convertida en minúsculas
toupper(s)	Devuelve s convertida en mayúsculas
getline	Fuerza una lectura de fichero