PRACTICA III Intérpretes de comandos

En la sesión anterior explicamos algunas características básicas de los intérpretes de comandos (redirección de entrada y salida, trabajos en ***background***, etc). Estas características son comunes a todos los intérpretes de comandos en UNIX. Sin embargo, hay numerosas herramientas y facilidades disponibles en cada intérprete, pero tanto su funcionamiento como su sintaxis difieren de un shell a otro. En esta sesión conoceremos los dos intérpretes de comandos más usados: C Shell y Bourne Shell. En realidad, en las instalaciones Linux se utilizan las versiones evolucionadas de ambos shells: ***tcsh*** y ***bash***, respectivamente. De hecho, si miramos en el directorio ***/bin***, los ficheros ***csh*** y ***sh*** no son más que soft links a ***tcsh*** y ***bash*** respectivamente, de forma que aunque ejecutemos el comando ***csh***, realmente estaremos llamando al ***tcsh***. De cualquier forma estos dos shell mejorados son compatibles descendentemente con sus hermanos menores, de forma que los comandos válidos en un ***csh*** son correctamente ejecutados en un ***tcsh***. Otro shell bastante comodo y amigable, que incorpora buenas cualidades tanto del ***sh*** como del ***csh***, es el ***zsh***, disponible en casi todas las distribuciones de linux.

Intérprete de comandos: El C Shell

El C Shell es el intérprete de comandos que más usaremos en nuestras prácticas. Su nombre viene del parecido intencionado de su sintaxis con la del lenguaje de programación C. Esto quiere decir que el C Shell no es sólo un mecanismo para meter una secuencia de comandos sueltos. En realidad, el C Shell es un lenguaje de programación interpretado, es decir, el intérprete de comandos va ejecutando cada sentencia inmediatamente después de leerla. Esto quiere decir que hasta ahora hemos estado usando una mínima parte de sus posibilidades, ya que lo hemos empleado simplemente para invocar comandos de UNIX. Más adelante veremos como se pueden hacer programas con C Shell.

Además de su faceta de lenguaje de programación, el diseño del C Shell no olvida tampoco que la mayor parte del tiempo se empleará para introducir comandos desde el teclado. Por eso, C Shell también tiene una serie de ayudas para facilitar el trabajo durante las sesiones interactivas, como son repetición y modificación de comandos anteriores, definición de ***alias*** para comandos complejos muy frecuentes, uso de variables para almacenar cadenas de texto empleadas con frecuencia y para parametrizar comandos cuyo resultado deba depender del valor de una variable, etc.

Inicio del C Shell

Cuando se pone en ejecución un C Shell, el programa lee y ejecuta los comandos contenidos en los ficheros ***.cshrc*** y ***.login*** (hay que recordar que los ficheros que comienzan por un punto no se listan con ***ls*** a menos que se use ***ls -a***). El C Shell es suficientemente inteligente para distinguir si él mismo es el primer shell ejecutado al iniciar la sesión, o si ha sido invocado desde otro shell. En este último caso, solamente lee el fichero ***.cshrc***[1] . Estos ficheros se usan para configurar el entorno de trabajo de cada usuario. El usuario pone en ***.cshrc*** los comandos que necesita ejecutar en cada invocación del intérprete de comandos. En ***.login*** pone solo aquellos que solo deben ser ejecutados una vez en cada sesión, y al inicio de ésta. Por ejemplo, si se pone en ***.cshrc*** un mensaje de bienvenida, nos llegaremos a

1. En UNIX existe una convención para los nombres de los ficheros de inicialización de las aplicaciones. Todos ellos se crean en el directorio inicial (***$home***) del usuario de forma que cada uno pueda tener su configuración privada de cada aplicación. Además, su nombre comienza por un punto, de forma que el usuario no vea estos ficheros normalmente (tras un uso continuado, puede llegar a haber cientos de ficheros de configuración correspondientes a todas las aplicaciones que el usuario ha usado alguna vez en su paso por el sistema, de forma que sería muy incómodo verlos con frecuencia). Por último, el nombre del fichero siempre incluye el nombre de la aplicación a la que pertenece y termina con las letras ***'rc'***, que son las iniciales de ***run commands***, es decir, los comandos usados al arrancar (***run***) un programa. Por ejemplo, si queremos configurar el programa ***Mail*** y además ejecutar alguna de sus opciones de forma automática cada vez que lo arranquemos, tendremos que editar el fichero ***.mailrc*** en nuestro directorio inicial.Igual ocurre con el debugger (***dbx***) y el fichero ***.dbxrc***.

hartar de verlo aparecer en pantalla cada vez que ejecutamos un nuevo shell o cuando ejecutamos un ***script***[1]. Resumimos en la siguiente tabla los scripts que se ejecutan al entrar en nuestra sesión dependiendo del shell que ejecutemos:

Shell	Del sistema (globales)		Privados (personales)	
	Siempre	Login shell	Siempre	Login shell
csh	/etc/csh.cshrc	/etc/csh.login	~/.cshrc	~/.login
tcsh	/etc/csh.cshrc	/etc/csh.login	~/.tcshrc[a]	~/.login
sh	--	/etc/profile	--	~/.profile
bash	--	/etc/profile	~/.bash_rc	~/.bash_profile
zsh	/etc/zshrc	/etc/zlogin	~/.zshrc	~/.zlogin

a. Si no existe, el tcsh intenta leer el fichero ~/.cshrc.

Es decir, todos los usuarios que ejecuten el entrar (login) el csh o tcsh, ejecutarán el script situado en /etc/csh.login y /etc/csh.cshrc si estos dos ficheros existen. Aparte, cada usuario puede configurar su entorno de trabajo mediante los ficheros personales .tcshrc o .cshrc y .login, ubicados en su directorio HOME personal.

Variables

Tanto el C Shell como el Bourne Shell son capaces de definir variables. El nombre de la variable puede ser una cadena alfanumérica de cualquier longitud. Hay dos modalidades para definir una variable (¡cuidado con los espacios!):

– ***`set variable=valor`*** : esta forma se usa para definir variables ***locales***;

– ***`setenv variable valor`*** : esta forma se usa para definir variables ***de entorno***.

La diferencia entre variables locales y variables de entorno tiene que ver con la implementación de UNIX. Las variables de entorno tienen la peculiaridad de que son heredadas por los procesos creados. Esto quiere decir que al ejecutar un comando desde el shell, el nuevo comando recibe una copia de las variables de entorno del shell y puede consultar su valor por medio de la llamada al sistema ***`getenv(NOMBRE_VARIABLE)`***. Si definimos una variable como local, no aparecerá en los procesos creados desde este shell[2]. Existe el acuerdo de poner nombre en minúsculas a las variables locales y mayúsculas a las de entorno (¡cuidado porque el shell las diferencia!). No es obligatorio, pero tiene la ventaja de que si más adelante vemos un nombre de variable en mayúscula podremos saber que es de entorno sin necesidad de investigar como fue definida.

Para usar el valor de una variable se emplea la sintaxis ***`$variable`***, es decir, su nombre precedido de un signo '***`$`***'. Lo que hace el shell es simplemente sustituir esa expresión por el valor de la variable, incluso en medio de una cadena de texto. Si el nombre de la variable está seguido de letras o números, se emplean la sintaxis ***`${variable}`*** para delimitar el nombre de la variable, como por ejemplo: ***`set cosa=caramelo;`*** "Me encantan los ${cosa}s." --> "Me encantan los caramelos."

1. Fichero que contiene un programa escrito con el lenguaje del Shell. Guardan cierto parecido con los ficheros ***.BAT*** de MSDOS.
2. Hay que aclarar que no existen variables globales a todos los procesos del sistema. Tanto las variables locales como las de entorno son parte del contexto privado de cada proceso. El hecho de que las de entorno sean heredadas (mediante copia) solo significa que el proceso hijo recibe un conjunto inicial de variables definidas en su propio entorno. El nuevo proceso puede modificarlas sin afectar a ningún otro ya que trabaja con sus propias variables.

```
lac20_$ set a="hola"
lac20_$ setenv B "adios"
lac20_$ echo $a
hola
lac20_$ echo $B
adios
lac20_$ echo $a, que tal estas.
hola, que tal estas.
lac20_$ ls $B.dat
Can not find file adios.dat
```

Variables locales al shell con significado especial

Hay un serie de variables que son usadas por el shell para configurar su funcionamiento. Si se modifican su valores se puede conseguir adaptar el entorno de trabajo al usuario. A continuación se describen las más importantes:

history Se usa para controlar el número de comandos anteriores que recuerda el shell. Inicialmente 0.

home Es el directorio al que vamos cuando se ejecuta el comando ***cd***. Su valor por defecto es el directorio inicial del usuario, pero se puede cambiar.

pwd Contiene el directorio actual en que se encuentra el shell. Su valor cambia cada vez que se usa el comando ***cd***.

path Es la lista de directorios en la que se buscan los nombres de los comandos invocados. Se separan por un espacio en blanco. Ej: ***set path=($path $home/bin)***, añade el subdirectorio ***bin*** del usuario a la lista.

status Contiene el estado en que finalizó el ultimo comando ejecutado por el shell. 0 quiere decir que terminó correctamente. Cualquier otro valor indica algún error.

prompt Especifica la cadena que se imprime cada vez que está esperando un nuevo comando. Hay caracteres con significado especial para incluir el directorio actual, el nombre de la máquina, el nombre del usuario, etc.

La descripción de todas estas variables está en el manual del C Shell (***man csh***). Además, se pueden listar todas las variables locales con el comando ***set***.

Variables globales con significado especial

Hay una serie de variables que son usadas por bastantes aplicaciones para averiguar la configuración que deben usar. Todas estas variables son de entorno, ya que las heredan del proceso que haya lanzado la aplicación. Normalmente definimos estas variables en el shell antes de lanzar las aplicaciones (suelen definirse en el fichero ***.login*** o ***.cshrc***). Para ver todas las variables de entorno definidas en un shell usamos el comando ***env***.

PATH Tiene exactamente la misma utilidad que la variable local ***path*** explicada anteriormente, con la diferencia de que esta es una variable de entorno (heredable) y que su sintaxis es un poco distinta. Los directorios se separan por los dos puntos y no por un espacio. Por ejemplo: ***setenv PATH ${PATH}:${home}/bin***

Aunque parece que puede haber un conflicto entre la variable local ***path*** y la de entorno ***PATH***, no lo hay. El shell cambia el valor de una cuando se altera el de la otra de forma automática, es decir, que da igual cual de las dos definamos.

TERM Indica el tipo de terminal de texto que estamos usando. Como hay bastantes tipos, las aplicaciones tienen que saber el modelo de terminal para seleccionar los comandos adecuados para mover el cursor, borrar la pantalla, identificar las teclas de cursor, etc. El valor inicial en cada sesión lo fija automáticamente el sistema en cada conexión, y lo suele hacer bien. En muy contados casos, tendremos que cambiar su valor si no se corresponde con la realidad. Los tipos mas frecuentes son ***vt100*** (un terminal asíncrono de DIGITAL que se convirtió en estándar de facto de terminal de texto) y ***xterm*** (el estándar de terminal de las ventanas de texto de X11).

PRINTER o ***LPDEST*** Estas variables identifican la cola de impresión por defecto a la que se manda un trabajo. Si la variable no está definida, se usa la cola por defecto definida en el sistema.

History

Para poder reusar comandos tecleados anteriormente el C Shell ofrece un mecanismo llamado ***history***. Consiste en que guarda en una lista los últimos ***n*** comandos introducidos. El número de comandos que guarda se define en la variable ***history***.

```
set history=50
```

Para ver el listado de los comandos guardados set puede teclear el comando ***history***. Cada comando tiene un número delante. Este número sirve para identificar el comando a la hora de recuperarlo. Para invocar -es decir, ejecutar- un comando anterior hay bastantes opciones posibles:

!número Invoca el comando que lleva delante el número indicado. Es el número que aparece en el listado del comando ***history***.

!! Invoca el último comando.

!-n Invoca el enésimo comando contando a partir del último. Ej: ***!-2*** invoca el penúltimo comando.

!cadena Invoca el último comando que comience por la cadena indicada.

^viejo^nuevo Ejecuta el último comando sustituyendo la cadena ***viejo*** por la cadena indicada en ***nuevo***.

También se pueden añadir algunos modificadores a cualquiera de las opciones anteriores.

:p Recupera el comando indicado, pero no lo ejecuta. Simplemente lo añade como el último comando invocado. Ej: ***!15:p*** añade el comando número 15 como último comando. Esto es útil para sustituir una cadena a continuación con ***^viejo^nuevo***.

:número Selecciona tan solo el argumento enésimo del comando indicado.

Como puede verse en el ejemplo siguiente, la sustitución del history puede combinarse con otras palabras (antes y después) que formarán parte del nuevo comando. El C Shell siempre muestra en el terminal el comando resultante antes de ejecutarlo.

Alias

El C Shell ofrece un mecanismo para poder crear nuevos comandos o bien redefinir el funcionamiento de otros ya existentes. La definición de alias permite crear un nombre que será

```
lac20_$ history
     4 ls
     5 set history=5
     6 vi hola
     7 cat hola
     8 history
lac20_$ !4
ls
prueba.c    cpu_bound    cpu_bound.c
lac20_$ !s
set history=5
lac20_$ ^5^10
set history=10
lac20_$ !ca:p
cat hola
lac20_$ rm !!:1 prueba.c
rm hola prueba.c
lac20_$
```

equivalente a la cadena asociada. Si por ejemplo echamos de menos un comando ***dir*** que nos liste los ficheros con todos sus atributos, podemos definirlo:

```
alias dir 'ls -lF'
```

Cuando aparezca el comando ***dir*** al principio de una nueva línea, el shell lo susbstituirá por la cadena ***ls -lF***. Para hacer las definiciones en cada sesión, hay que incluirlas en el fichero ***.cshrc***. El comando ***alias*** sin argumentos nos lista todos los alias definidos hasta ahora. Se puede eliminar un alias con el comando ***unalias nombre_alias***.

Si queremos cambiar el comportamiento de un comando, definimos un alias con su mismo nombre. Por ejemplo:

```
alias rm 'rm -i'
```

Para poder invocar al comando original en lugar del alias podemos escribir un '\' delante (p.e. ***rm**). Cada vez que escribimos ***rm*** se sustituye por ***rm -i*** que nos obliga a confirmar el borrado de cada fichero.

```
lac20_$ alias
dir ls -lF
h history
j jobs
ls ls -F
pd pushd
pop popd
rm rm -i
who who | sort -r
lac20_$ unalias rm
lac20_$
```

Comandos internos y externos

La mayoría de los comandos que podemos usar son externos al intérprete. Es decir, son ficheros ejecutables que se encuentran en los directorios ***/bin***, ***/usr/bin***, ***/usr/local/bin***, etc. Por tanto, estos comandos están disponibles en C Shell, y en cualquier otro shell, y sus opciones no varían cuando cambiamos de intérprete de comandos. Sin embargo, hay algunos comandos que forman parte del C Shell -***builtin commands***- y por tanto no estarán disponibles en Bourne Shell. Si uno no está seguro de

si un comando es interno o externo, puede recurrir al comando ***whereis***. Este comando busca todos los ficheros relacionados con el nombre dado que se encuentren en directorios del sistema. Entre otros muestra los ejecutables (si existen), ficheros ***include*** del compilador de C y ficheros del manual (man).

```
lac20_$ whereis pwd
pwd: /bin/pwd /usr/bin/pwd /usr/include/pwd.h /usr/man/man1/pwd.1
lac20_$ whereis cd
lac20_$
```

A diferencia de ***pwd*** que está en ***/usr/bin/pwd***, ***cd*** es un comando interno ya que no aparece ningún ejecutable. Algunos de los comandos internos de C Shell son:

– alias
– cd
– echo
– history
– login
– logout
– setenv
– shift
– source
– time
– unset

Programación en Bourne Shell

El Bourne Shell (***/bin/sh***) es a la vez un intérprete de comandos y un lenguaje de programación interpretado. Soporta la definición de variables, y sentencias de control estructuradas tales como ***while***, ***for***, ***if then else***. Los programas de Bourne Shell son simples ficheros de texto ASCII. Se llaman ***shell scripts***. Para que un fichero de texto se pueda ejecutar como un script es necesario activar su permiso de ejecución (flag x) con el comando ***chmod***.

Variables

La definición de variables es muy parecida a la que vimos en C Shell, aunque cambia su sintaxis. La sintaxis para definirlas es ***nombre_variable=valor***. Por defecto todas las variables definidas son locales. Si se quiere hacer que una variable sea de entorno, y por tanto heredable, se usa el comando ***export nombre_variable***.

```
lac20_% nuevo=Indigo; export nuevo
lac20_% echo nuevo
nuevo
lac20_% echo $nuevo
Indigo
lac20_% nuevo='tambien se permiten espacios entre comillas'
lac20_% echo $nuevo
tambien se permiten espacios entre comillas
lac20_%
```

Parámetros

Un shell script puede acceder a los argumentos pasados en la línea de comandos cuando se ejecuta. Los argumentos son accesibles en las variables $1, $2, ..., $9, que contienen el primer argumento, segundo, tercero, y así sucesivamente. $0 contiene el nombre del script actual. $# da el número de argumentos. $* contiene todos los argumentos juntos. No existe forma de referenciar el décimo argumento y siguientes. Para ello hay que usar el comando ***shift*** tal como se explica a continuación.

```
lac20_% cat prueba
echo el comando se llama $0
echo y tiene $# argumentos
echo que son $*
echo el primero es \($1\) y el segundo es \($2\)
lac20_% prueba a b c d e f
el comando se llama prueba
y tiene 6 argumentos
que son a b c d e f
el primero es (a) y el segundo es (b)
lac20_%
```

Si se usa el comando ***shift*** en un script, se desplazan todos los argumentos hacia la izquierda. Es decir, El valor anterior de $1 es descartado; el antiguo valor de $2 ahora se encuentra en $1; el antiguo valor de $3 pasa a encontrarse en $2 y asi sucesivamente. El número de argumentos devuelto por $# ahora se ha decrementado. Cuando tenemos que acceder a un número indeterminado de argumentos se suele usar un bucle donde se accede a $1 y a continuación se pasa al siguiente usando ***shift***.

```
lac20_% cat procesa
while test $# -gt 0; do
  echo $1
  shift
done
```

Variables del Bourne Shell

Set pueden listar las variables definidas con el comando ***set***. A continuación se describen las más importantes:

HOME	Es el directorio al que vamos cuando se ejecuta el comando ***cd***. Su valor por defecto es el directorio inicial del usuario, pero se puede cambiar.
pwd	Contiene el directorio actual en que se encuentra el shell. Su valor cambia cada vez que se usa el comando ***cd***.
PATH	Es la lista de directorios en la que se buscan los nombres de los comandos invocados. Los directorios se separan por dos puntos. Ej: ***PATH=.:/bin:/usr/bin***.
PS1	Especifica la cadena que se imprime cada vez que está esperando un nuevo comando (prompt). Hay caracteres con significado especial para incluir el directorio actual, el nombre de la máquina, el nombre del usuario, etc.
PS2	Contiene el prompt secundario, es decir, el que se imprime cuando un comando tiene más de una línea (como las sentencias de control).

IFS	Internal field separator. Es el carácter que se usa para separar los argumentos de la línea de comandos. Por defecto se usa el espacio para separar los argumentos, pero se puede cambiar.
TERM	Indica el tipo de terminal para esta sesión. Ver la misma variable en C Shell.

Estructuras de control de Bourne Shell

Las estructuras de control básicas son:

```
if condición;
then
     comandos
[elif condición
then
     comandos]
[else
     comandos]
fi


for variable in lista_valores
do
       comandos
done
```

```
while condición;
do
       comandos
done


case variable in
       expresión1)
             comandos1;;
       expresión2)
             comandos2;;
       ......
       *)
             comandos3;;
esac
```

Las condiciones que aparecen en estas estructuras consisten en un comando. El shell ejecuta ese comando y comprueba el estado devuelto al terminar. Si el comando da error, entonces se considera falsa la condición. Si termina correctamente, se considera cierta. En los puntos donde se pueden poner comandos, puede aparecer una secuencia de comandos separados por ';'. Para escribir condiciones que incluyan comparaciones entre cadenas, números, consultas sobre la existencia de ficheros o sus permisos, etc, existe un comando llamado ***test***.

```
lac20_% echo "Los ficheros modificables son:"; for f in *
> do
> if test -w $f; then echo $f; fi
> done
Los ficheros modificables son:
prueba.c
t.dat
lac20_%
lac20_% cat coche
echo "Que coche tienes?"
read coche
if test $coche = "BMW"
then echo "Tienes un cochazo"
else echo "Comprate otro coche!!"
fi
lac20_% coche
Que coche tienes?
BMW
Tienes un cochazo
lac20_%
```

```
lac20_% cat traductor
while true
do
        echo -n "* "
        read linea
        case "$linea" in
                pantalla|monitor)
                        echo screen ;;
                coche)
                        echo car ;;
                casa)
                        echo house ;;
                FIN)
                        exit 0;;
                *)
                        echo "Palabra desconocida!" ;;
        esac
done
lac20_% traductor
* casa
house
* coche
car
* monitor
screen
* pantalla
screen
* hola
Palabra desconocida!
* FIN
lac20_%
```

Asignación de variables

Hasta ahora no hemos asignado nada más que constantes, pero se pueden asignar expresiones donde una parte sea el resultado de sustituir una variable o incluso la salida producida por la ejecución de otro comando. Las reglas son las siguientes. Para asignar una expresión que contenga espacios, necesitamos ponerla entre comillas. Ahora bien, el shell (tanto Bourne como C Shell) atribuye significados distintos a los tres tipos de comillas (´ ` y ").

´ Comilla izquierda: la expresión que contiene se toma de forma literal. No se tiene en cuenta el significado de ningún carácter especial contenido en ella. Por ejemplo se pueden poner paréntesis o incluso el carácter '&' o ';' sin que se interpreten como separadores de comandos y sin necesidad de precederlos de un carácter '\'.

" Comillas dobles: El contenido se toma de forma literal, es decir, se pueden introducir caracteres especiales. Sin embargo, las expresiones ***$variable*** contenidas dentro sí son sustituidas por su valor. Las expresiones entre comillas derechas también sigue teniendo significado especial.

` Comilla derecha: El contenido se interpreta como un comando con sus argumentos y se ejecuta. A continuación se sustituye esta expresión por el resultado de la ejecución del comando.

```
lac20_% color=azul
lac20_% echo ´Me gusta el color $color.´
Me gusta el color $color
lac20_% echo "Me gusta el color $color."
Me gusta el color azul.
lac20_% echo "Los ficheros ‘ls /tmp‘ están en el directorio tmp"
Los ficheros t.h yo.c fvwm12945 están en el dirctorio tmp
lac20_%
```

Existe también un comando llamado ***expr*** que permite especificar una expresión y evaluarla. Este comando se puede usar para asignar a una variable el resultado de una operación.

```
i=‘expr $i - 1‘
```

Scripts para otros lenguajes o shells

Existen otros intérpretes de comandos en un sistema Unix. Aparte del resto de los shells (***csh***, ***tcsh***, ***zsh***, ...) que tienen una sintaxis de programación ligeramente distinta a la del ***sh***, también podemos escribir scripts para lenguajes como ***perl*** o ***awk*** (con una sintaxis totalmente distinta). Es posible especificar en la primera línea de un script de que tipo es éste. Por ejemplo, si la primera línea del script es ***#! /bin/csh***, el script primero lanzará un ***csh*** pasándole el resto del script para que lo ejecute. En general, puedes preceder el ***#!*** en la primera línea por cualquier nombre de intérprete disponible (usando el path completo) y los parámetros que necesite para interpretar el resto del script, por ejemplo /bin/tcsh, /bin/awk, /bin/perl, etc. Si el script tiene únicamente un “#” en la primera línea, o no tiene nada, el script se ejecuta en un shell de Bourne.

Recuerda consultar la lista de dudas frecuentes en la página web de la asignatura. Contiene explicaciones complementarias que aclaran algunos puntos que pueden ser difíciles de entender.

Ejercicios

1. Usando C Shell, escriba un comando para añadir un alias que modifique el funcionamiento de ***rm*** para que pida siempre confirmación al borrar un fichero y de ***cp*** para que pida siempre confirmación al sobreescribir un fichero. Haga permanentes estos alias en su sesión escribiendo los comandos anteriores en los ficheros de inicialización correspondientes a C Shell. Recuerde que estos ficheros no tienen por qué existir en su directorio $HOME si no los ha creado todavía.
2. Repita el ejercicio anterior usando Bourne Shell en lugar de C Shell. Recuerde que para usar un shell distinto basta con ejecutar el comando correspondiente (sh o csh) en la línea de comandos. Recuerde que tanto la sintaxis como los ficheros de inicialización de Bourne Shell son distintos a los de C Shell.
3. Compruebe lo que hace el siguiente script llamado ***escoger*** (ver cuadro de la siguiente página) al pasarle cualquier lista de argumentos. ¿Cuál es la misión del comando ***read***? Investigue el manual (recuerda que si read es un comando interno del Bourne Shell, no tiene página propia de man, sino que tienes que buscarlo en la página man del comando bash, que es bastante larga).
4. Se quiere usar ***escoger*** para poder seleccionar manualmente los ficheros sobre los que se quiere ejecutar un comando determinado. Por ejemplo, usamos el comando ***cat `escoger *`***. ¿Por qué se obtiene una salida en pantalla tan confusa? Sugerencia: prueba a ejecutar antes el comando ***echo `escoger *`*** para ver qué argumentos recibirán los comandos cat y echo como resultado de la ejecución del comando contenido entre las comillas.
5. ¿Podría hacerse que ***escoger*** preguntara al usuario justo antes de ejecutar el comando especificado para cada fichero indicado en lugar de hacerlo como se ha visto en el ejercicio anterior? (Pista: reescribir el comando ***escoger*** para pasarle también como primer argumento el comando que se ha de ejecutar con todos los nombres de ficheros pasados como argumentos tras

```
lac20_% cat escoger
for i in $*
do
  echo $i \?
  read RESP
  if test $RESP = y;
  then echo $i
  fi
done
```

el comando. Es decir, las llamadas al comando serían ahora de la forma: "***escoger cat ****" o "***escoger ls ****" o "***escoger rm ****", etc...)

6. Escriba un script que imprima la lista de los argumentos recibidos, pero sin usar la variable ***$****. Sugerencia: repasa la finalidad del comando shift en un script. El script debe ser capaz de procesar más de 9 argumentos.
7. Escriba un script que imprima la lista de los argumentos que reciba desde el último hasta el primero, es decir, que imprima la lista de argumentos en orden inverso al pasado. Sugerencia: se puede usar una variable como si fuera una lista añadiendo elementos al principio o al final mediante las expresiones
   a="$var $a"*** o ***a="$a $var"
   donde ***$var*** puede ser cualquier variable. El script debe ser capaz de procesar más de 9 argumentos.
8. Investigue las opciones del comando ***test***. Escriba un script que para cada argumento que se le pase (un número indeterminado de argumentos) indique el tipo de fichero (directorio, fichero, etc).
9. Antes de hacer este ejecercicio, añada el directorio actual al PATH con el comando
   setenv PATH .:${PATH}
   Escriba un script llamado ***cal*** que acepte como argumentos los meses del año escritos en español y que invoque al comando ***cal*** de Unix. Encontraremos un problema ya que nuestro comando se invoca a si mismo en lugar de al comando de Unix. ¿Cómo se debe invocar el comando ***cal*** de Unix desde el script para evitar la ambigüedad? ¿Cómo debes llamar al script evitando llamar al comando ***cal*** de Unix? ¿Crees que es buena idea ponerle a un script el nombre de un comando del sistema ya existente?