Introducción a `bash` y lenguajes de *scripting*

Iria Estévez Ayres
Departamento de Ingeniería Telemática
Universidad Carlos III de Madrid

Índice

1. Intérpretes de comandos

Un intérprete de comandos o *shell* puede describirse de manera simplificada como un programa que utiliza la consola en modo carácter y ejecuta comandos en secuencia, mostrando la salida de cada comando por el dispositivo de salida: la pantalla, una ventana de terminal o un fichero. Estos comandos pueden ser introducidos interactivamente por un usuario sentado al teclado, pero también pueden agruparse en un fichero, lo que constituye un *script*.

Los *scripts* se utilizan para automatizar tareas, por ejemplo en el proceso de arranque. El modo interactivo es el modo preferido de trabajo en UNIX, y como se verá más adelante ofrece una potencia extraordinaria.

Existen multitud de intérpretes de comandos diferentes, pero son sólo dos los que gozan de uso masivo: *tcsh* y *bash*. En modo interactivo, el intérprete de comandos le indicará que está esperando instrucciones, mostrando lo que se llama prompt: una texto o cadena de caracteres que puede adoptar formas diferentes puesto que es configurable. El carácter más a la derecha del prompt será habitualmente $ o #, dependiendo de si el usuario actual es un usuario normal o es el administrador. Una vez tecleado el comando, pulse la tecla [Enter] para indicar a la *shell* que comience la ejecución del comando. Una vez hecho esto, los pasos que sigue el intérprete de comandos son:

1. Comprueba si es un comando interno, es decir, una orden que forma parte del propio intérprete de comandos. Si es así, ejecuta la orden.

2. Si no es un comando interno, se asume que el comando introducido es un comando externo, es decir, un programa. El intérprete de comandos busca un programa con el nombre indicado en los directorios enumerados en la variable de entorno PATH, ejecutando el programa encontrado. Si no encuentra ningún programa, indica que el comando no existe.

3. Tras la ejecución del comando, sea interno o externo, se vuelve a mostrar el prompt y el ciclo se repite.

A diferencia de los programas de MSDOS, en UNIX no se buscan de forma predeterminada los programas en el directorio actual, por lo que si se desea ejecutar un programa situado en el directorio actual y éste no está incluido en la variable de entorno PATH deberá indicar explícitamente la localización del programa dentro del sistema de archivos, que en la práctica implica preceder el nombre del programa con la ruta "./". Los comandos pueden invocarse con parámetros cuyo significado concreto depende de cada comando. Es práctica estándar que los parámetros que no son rutas comiencen con el carácter "-".

2. La *shell*: bash

2.1. Completado automático de comandos

bash dispone de mecanismos de ayuda a la introducción de comandos. Uno de ellos es el completado automático de comandos. Cuando esté tecleando una ruta, pulsando la tecla [Tab] bash intentará completar el nombre introducido. Por ejemplo, si el directorio actual contiene los subdirectorios:

```
info include local src
```

y se desea cambiar al directorio local, se puede teclear cd l y a continuación pulsar [Tab]. La l se expande automáticamente a local. Esto es posible porque la l es un prefijo único de nombre en el directorio actual. Si se quisiera ir al directorio info, se puede teclear cd i y [Tab]. La shell responde emitiendo un pitido y completando la i hasta in. bash no puede saber a cuál de los dos directorios info o include se refiere el usuario. Añadiendo una f, el prefijo inf ya es único y al pulsar [Tab] se completa finalmente a info.

A pesar de que en este último ejemplo no parece que nos haya beneficiado, el completado automático es muy usado en la práctica y especialmente apropiado para los nombres largos habituales en ficheros UNIX. El completado automático utiliza fuentes de nombres apropiadas para lo que esté tecleando en ese momento. Si está introduciendo un nombre de comando (la primera palabra tras el prompt) el comando será completado con los nombres de ficheros ejecutables visibles en los directorios indicados en la variable de entorno PATH. El resto de palabras serán interpretadas como rutas, ya sean absolutas o relativas. Se completará por los directorios y ficheros presentes en la ruta.

2.2. Comodines

Otra manera de facilitar la introducción de comandos es el uso de comodines, que son caracteres con un significado especial:

- "$*$" : cualquier conjunto de caracteres.
- "?" : cualquier carácter, sólo uno.
- "[. . .]" cualquiera de los caracteres entre corchetes.

Cuando se incluye algún comodín como parte de un nombre de fichero o ruta, bash expande ese nombre con todos los nombres de ficheros y directorios abarcados por los comodines. Por ejemplo, para borrar todos los archivos del directorio actual se puede usar el comando rm *. La *shell* expandirá $*$ a todos los nombre presentes en el directorio actual, y será como si hubiésemos tecleado individualmente todos los nombres de los ficheros nosotros mismos. Más ejemplos:

- ls l* mostrará ficheros que comiencen por l.
- ls fich0? mostrará ficheros tales como fich01, fich02, . . .
- ls *.tar mostrará todos los ficheros terminados en .tar.

2.3. Historial

bash recuerda los comandos introducidos. Se puede revisar la lista con las teclas de movimiento vertical del cursor. La tecla de subir recorre la lista de comandos en orden de más a menos reciente. Para repetir el último comando introducido, basta con pulsar la tecla de subir y el comando aparece como si lo hubiésemos tecleado.

También se puede consultar el historial, utilizando el comando history, que nos presentará la lista de completa de los últimos comandos introducidos.

2.4. Redirección de entrada/salida

Todos los comandos de consola disponen de tres conductos de entrada/salida (file descriptor) pre–abiertos, conocidos como la entrada estándar, salida estándar y salida de error estándar.

La entrada estándar está conectada al teclado, y las salidas a la pantalla o ventana del terminal.

Los comandos leen las órdenes del usuario por la entrada estándar y generan salida de visualización por la salida estándar o, excepcionalmente, por la salida de error estándar.

A través de bash se puede actuar sobre estos cauces. Se puede redirigir la entrada y la salida a un fichero, y también se puede conectar la salida de un comando a la entrada de otro, estableciendo una cadena de procesos en lo que se conoce como una tubería o pipe.

Para redirigir la salida de un comando, se debe utilizar la sintaxis >nombre donde nombre es un nombre de fichero en el que se volcará la información de salida, que de otro modo hubiera ido a la pantalla. Si el fichero existe, se pierde el contenido anterior.

Por ejemplo:

```
$ ls
documentos mifichero miprograma
$ ls >salida
$ cat salida
documentos mifichero miprograma
```

Si se desea conservar el contenido previo del fichero, añadiendo al final del mismo la salida producida por el comando, se debe utilizar la sintaxis ≫nombre. Si utilizamos el fichero del ejemplo anterior:

```
$ ls
documentos mifichero miprograma salida
$ ls >>salida
$ cat salida
documentos mifichero miprograma
documentos mifichero miprograma salida
```

Para redirigir la entrada a un comando, utilice la sintaxis <nombre donde nombre es un nombre de fichero cuyo contenido servirá para alimentar el conducto de entrada que de otro modo hubiera provenido del teclado.

La redirección de entrada es especialmente útil en las llamadas tuberías (o *pipelines*): cadenas de procesos en las que la salida del anterior se conecta a la entrada del siguiente. Para conectar de esta manera dos procesos, sepárelos por el carácter —. Por ejemplo:

```
$ ls -l
drwxr-xr-x 17 iria users 1024 Sep 24 2000 documentos
-rw-r--r-- 1 iria users 375525 Dec 10 00:38 mifichero
-rwxr-xr-x 1 iria users 5342 Dec 10 00:38 miprograma
$ ls -l | wc -l
3
```

En este caso la salida generada por el comando ls -l sirve de entrada al comando para contar líneas wc -l, cuya salida es mostrada por pantalla. Con este otro *pipeline* conseguimos encontrar en el historial de comandos todas las ocurrencias de la cadena "mv".

```
$ history | grep mv
  122  cp mvog.jpg /tmp/
  487  mv billete-e.txt personal/iberia/
  499  mv solicitud-* FormacionFondo/
  511  history | grep mv
```

Esta manera de encadenar comandos permite generar órdenes muy potentes a partir de simples comandos.

2.5. Control de trabajos

El control de trabajos (job control) es una característica que permite controlar la ejecución de varios procesos simultáneamente en el mismo terminal o consola.

De los procesos en ejecución (no suspendidos) sólo uno está conectado a la entrada de consola; de este proceso se dice que está en primer plano (*foreground*), y del resto se dice que están en segundo plano (*background*).

Los comandos se ejecutan normalmente en primer plano, el proceso arrancado toma el control de la consola y no lo devuelve a la *shell* hasta que finaliza. Se puede suspender el proceso en primer plano pulsando la combinación de teclas [Ctrl+Z], recuperando así la *shell* el control, mostrando de nuevo el prompt. Por ejemplo:

```
$ top
[se muestra la pantalla interactiva de top]
[Ctrl+Z]
[1]+ Stopped top
$
```

Se puede arrancar directamente un comando en segundo plano, tecleando el nombre del comando y el carácter &.
Por ejemplo:

```
$ emacs &
$
```

Arranque el editor emacs en segundo plano y la *shell* recupera el control

Para cambiar un proceso a primer plano utilice el comando fg [jobspec] y bg [jobspec] para pasar un proceso suspendido a segundo plano. Para ver el estado de los trabajos, use el comando jobs. El identificador de trabajo mostrado por jobs es el que se indica en [jobspec], no el PID. Para escribir especificaciones de trabajo use la sintaxis %jobid, donde jobid es el identificador de trabajo mostrado por jobs.

2.6. Substitución de comandos

La sustitución de comandos (command substitution) es una característica de la *shell* que nos permite usar la salida de un comando para componer nuevos comandos. Veamos un ejemplo:

```
$ pwd
/home/iria
$ export DIRECTORIOACTUAL=‘pwd‘
$ echo $DIRECTORIOACTUAL
/home/iria
```

El segundo comando contiene una novedad: ‘pwd‘. La sintaxis ‘. . . ‘ introduce la substitución de comandos: la salida producida por el comando encerrado entre la pareja de acentos graves substituye al comando y se procesa como si hubiésemos tecleado esos parámetros nosotros mismos. En nuestro caso, el comando

```
$ export DIRECTORIOACTUAL=‘pwd‘
```

es equivalente a

```
$ export DIRECTORIOACTUAL=/home/iria
```

El acento grave se encuentra en la tecla de acentuar vocales, justo a la derecha de la tecla [P]. Para teclearlo, pulse la tecla de acentuar vocales seguida de un espacio.

2.7. Expansión de la *shell*

Antes de procesar una línea de comando, el *shell* realiza un conjunto de operaciones consistentes en expandir o reemplazar ciertos símbolos por sus valores. Este conjunto de expansiones se realizan en un cierto orden y por tanto afectan a cómo se obtiene la línea final que se ejecuta.

Ejemplo: la línea echo $PWD no se procesa tal cual por el intérprete sino que previa a su interpretación se aplica el proceso de expansión de variables que reemplaza $PWD por su valor.

Una segunda expansión que efectúa el *shell* es la expansión del símbolo $\sim$ cuando se encuentra al comienzo de un parámetro. Si va seguido de una palabra y luego del carácter / , se expande al valor del directorio raíz del usuario con login dado. En caso que aparezca al comienzo de palabra y justo antes de / , se expande al valor del directorio raíz del usuario ejecutando el comando.

Ejemplos:

- El fichero $\sim$user/p1/ficheros/data/f1 se expande tal que $\sim$user se reemplaza con el valor de su directorio raíz.
- La expresión $\sim$/p1/ficheros/data/f1 se expande tal que $\sim$ se reemplaza con el directorio raíz del usuario que esté ejecutando el *script* (en este caso coincide con el valor de la variable de entorno HOME).

El *shell* ejecuta un total de ocho tipos de expansiones: llaves, tilde, parámetros, variables, sustitución de comando por resultado, división en palabras y expansión de nombre de fichero (en este orden). El problema que puede aparecer es respecto al orden de ejecución de estas expansiones. De las dos expansiones mencionadas anteriormente, la expansión de ∼ precede a la de variables. Supongamos que la variable con nombre user contiene el valor *usuario1* y el directorio raíz del usuario con nombre *usuario1*, es /home/usuario1. Ante esta observación, el siguiente comando:

```
unix$echo ~$user
~usuario1
unix$
```

no muestra la cadena /home/usuario1 puesto que primero se realiza la expansión de ∼ y luego la de la variable. Por tanto, al tratar de expandir ∼$user, la *shell* no encuentra tal usuario y al expandir la variable, se reemplaza por su valor y por tanto el comando imprime usuario1.

La pregunta que se plantea es la siguiente: ¿Cómo conseguir que el *shell* vuelva a realizar una segunda expansión?

Para ello se permite la utilización del comando eval que simplemente expande todo aquello que se le pasa como parámetro y a continuación lo ejecuta como un comando. Mediante este comando se puede conseguir que el intérprete haga 2 expansiones, una al obtener los parámetros del eval, y una segunda al ejecutar el comando. Sabiendo esto, y volviendo a nuestro ejemplo, el comando:

```
unix$eval echo ~$user1
/home/usuario1
unix$
```

sí produce el resultado esperado de /home/usuario1 puesto que en la primera expansión se obtiene echo ∼usuario1 y en la segunda (fruto de la ejecución del eval) se expande ∼usuario1 a /home/usuario1.

3. Introducción a los *scripts* de *shell*

Un *script* es un fichero de texto que contiene una sucesión de comandos de shell. Estos comandos pueden ejecutarse como si se trataran de un programa utilizando el propio intérprete o *shell*. Existen dos maneras de conseguir esto:

1. Con el siguiente comando:

   ```
   unix$sh nombre_del_fichero
   ```

   donde sh es el nombre del intérprete que queremos que ejecute el script, en el caso de Linux, podemos usar *bash* o directamente *sh* que es un alias de la primera.

2. Directamente, incluyendo como primera línea del script:

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

(o la ruta al intérprete que queramos que ejecute los comandos) y asignándole al *fichero–script* permisos de ejecución (x) con el comando chmod.

Para facilitar la legibilidad de los ficheros de scripts, se pueden introducir comentarios en el fichero por medio del carácter #. Todo aquello que se escriba a partir de este carácter y hasta el final de línea es ignorado por el *shell*.

Recordemos que un *script* en ejecución es un proceso en Unix que se deriva del proceso *shell* que lo lanza. Lo importante de esto es que ese proceso hijo hereda el entorno (variables de entorno, directorio de trabajo por defecto, etc...) del padre.

Los comandos y sintaxis aquí comentados son validos para la *shell* sh y compatibles. tcsh y otras utilizan una sintaxis distinta y no compatible al $100\,\%$.

3.1. Manejo de Variables

En un *script* de *shell* tenemos tres tipos de variables:

1. **Variables de entorno**: Se heredan de la *shell padre*. Por ejemplo: PATH, MANPATH, UID, PWD, etc.

2. **Variables predefinidas**: Son los argumentos de línea de comandos que se le han pasado al *script* al ejecutarlo.

3. **Variables definidas en el propio script**: Dado que la *shell* tiene un lenguaje de programación propio, nos permite definir variables para hacer programas complejos.

Para acceder a una variable de entorno se utiliza la sintaxis $NOMBRE, donde NOMBRE es el nombre de la variable. El operador $ precediendo al nombre hace que éste se reemplace por su valor en una expresión.

Ejemplo: Para mostrar en pantalla el contenido de la variable de entorno PATH podemos escribir el siguiente script:

```
#!/bin/sh
echo $PATH;
```

El comando echo imprime lo que se le pase como parámetro (en este caso una cadena) por el canal de salida estándar. Podemos escribir el parámetro a este comando utilizando comillas para imprimir texto simple y variables, por ejemplo: echo "Valor de la variable es $PATH".

Para ver las variables de entorno definidas en una *shell* en la que estamos introduciendo comandos podemos utilizar el comando set sin ningún argumento.

3.2. Estructuras de Control: Condicionales y Bucles

La *shell* nos aporta un lenguaje de programación que, además de variables, permite gestionar la ejecución condicional a través de la estructura if–then–else–fi. La sintaxis concreta es:

```
if <condición>
then
        # Los comandos de este bloque se ejecutaran si <condición>
        # devuelve el estatus 0 (verdadero).
else
        # Los comandos de este bloque se ejecutaran sólo si <comando>
        # devuelve el estatus diferente de 0 (falso).
fi
```

Para obtener la condición se puede utilizar el comando test (un comando de la propia *shell*) que devuelve verdadero o falso a cierto tipo de preguntas de un catálogo.

Ejemplo: El comando test -x fichero devuelve verdadero si tenemos permisos de ejecución sobre el fichero dado. Para una lista detallada de las posibles condiciones que se pueden evaluar con el comando test ver su página de manual.

Si la condición que queremos evaluar tiene más de dos posibles valores, típicamente el valor de una variable, se utiliza la sentencia case–esac, cuya sintaxis es:

```
case $variable
in

valor1)
        # Los comandos de este bloque se ejecutaran si la variable
        # tiene el valor valor1
      ;;
valor2)
        # Los comandos de este bloque se ejecutaran si la variable
        # tiene el valor valor2
      ;;
     #########
valorN)
        # Los comandos de este bloque se ejecutaran si la variable
        # tiene el valor valorN
      ;;
     *)
        # Bloque por defecto
        # Se ejecuta si la variable no tiene ninguno de los valores anteriores
      ;;
esac
```

La *shell* ofrece varias opciones para implementar bucles, la más común de ellas es el bucle for–do–done cuya sintaxis es:

```
for <variable> in <lista>
do
  # Comandos a ejecutar
done
```

Para cada iteración la variable $variable toma el siguiente valor de la lista dada. En ausencia de una lista se utilizan los argumentos de la línea de comandos. Un sencillo *script* que crea una copia de seguridad de dos archivos se puede escribir como:

```
#!/bin/sh
for fichero in practica.doc practica.gif
do
  cp $fichero $fichero.bak
done
```

En este ejemplo suponemos que existen dos archivos llamados practica.doc y practica.gif. El *script* creará una copia de cada uno anexando la extensión .bak al final.

Una característica útil de este tipo de bucle es que se pueden acceder a los ficheros del directorio de trabajo utilizando caracteres comodín como $*$ y ?. Así, el siguiente *script* imprime por pantalla el nombre de todos los ficheros del directorio actual:

```
#!/bin/sh
for fichero in *
do
  echo $fichero
done
```

Otra estructura que nos ofrece la *shell* para implementar bucles es la estructura while–do–done cuya sintaxis es:

```
while <condicion>
do
  # Comandos a ejecutar
done
```

Se evalúa la condición de finalización de bucle antes de realizar cada iteración.

3.3. Lectura de usuario: el comando read

El comando read lee una línea de la entrada estándar y la guarda en una variable. Es un comando interno del script. Este conjunto de sentencias piden una entrada al usuario, y después la muestra por pantalla:

```
#!/bin/bash
read n
echo $n
```

Si se desea que se presente un determinado prompt, se debe utilizar la opción –p.

```
#!/bin/bash
read -p "Dime algo: " n
echo $n
```

Si se ejecuta este script, se obtiene la siguiente salida:

```
unix$Pide.sh
Dime algo: algo
algo
unix$
```

3.4. Argumentos de un script

Unix permite pasar datos a un *script* al ejecutarlo, como argumentos en la línea de comandos, y proporciona un conjunto de variables predefinidas para acceder a ellos:

- **$#**: Número de argumentos

- **$*** o **$@**: Texto con todos los argumentos introducidos (en una única cadena), separados por espacios. No incluye el nombre del script
- **$0**, **$1**, **$2**...: Cada uno de los argumentos, según el orden en que fueron introducidos. $0 es el nombre del propio script, $1 el primer argumento...

Por ejemplo:

```
unix$ miScript.sh Hola mundo
```

- Número de argumentos: $# = 2
- Argumentos: $* = $@ = "Hola mundo"
- Argumentos (individualmente):
  - $0 = "miScript.sh"
  - $1 = "Hola"
  - $2 = "mundo"
  - $3 = (cadena vacía, no se ha introducido nada)

Nótese que las comillas (") se utilizan simplemente para enmarcar el texto almacenado en la variable, el valor real de la variable no las incluye.

El *shell* provee una operación para manipular los argumentos que recibe el *script*. El comando shift se utiliza para descartar de la lista de argumentos el primer elemento y desplazar el resto de elementos hacia el comienzo de la cadena. Este comando, si se ejecuta sin argumentos, descarta el primer elemento, pero se le puede dar un argumento entero especificando el número de elementos a desplazar. Así pues, si el segundo elemento de $* era la palabra text, tras ejecutar el comando shift, esta palabra ocupa la primera posición del array.

El siguiente *script* imprime la secuencia de argumentos que recibe:

```
while [ $# -ne 0 ]
do
  echo $1
  shift
done
```

3.5. Invocación de otros programas desde un script

Hay cierto tipo de datos sobre el sistema que se pueden obtener mediante la invocación de otros programas. Tal es el caso del programa hostname que devuelve el nombre del ordenador, o uname que con sus diferentes opciones proporciona información sobre el tipo de ordenador, el tipo de sistema operativo, la versión. Otro ejemplo es el programa whoami que devuelve el nombre de usuario con el que se ha abierto la sesión de trabajo (como su propio nombre indica).

3.6. Opciones de un script

Cuando se crea un fichero de *script*, generalmente se proveen un conjunto de opciones que permiten ejecutar la tarea pertinente pero con ligeras modificaciones. Por ejemplo: un *script* imprime mensajes notificando el comando que está ejecutando, pero se desea a veces que se ejecute sin imprimir ningún mensaje. Esto se consigue con una opción que permite controlar el nivel de mensajes que se imprimen.

Análogamente, esta idea se aplica a otros aspectos de los *scripts* como por ejemplo los nombres de ficheros que se procesan o en los que se depositan resultados. El *script* puede depositar el resultado de un cálculo en un fichero con nombre result.txt, pero a la vez, el *script* permite la especificación de un nombre de fichero en caso de que se quiera cambiar.

Estas opciones, como se ha visto en múltiples comandos Unix, se escriben a continuación del nombre del programa, y suelen ir precedidas del símbolo "−". Algunas opciones necesitan un dato a continuación de la opción (como, por ejemplo, si se especifica el fichero de salida), y otras simplemente necesitan la opción (por ejemplo para activar los mensajes de depuración de un script).

Ejemplo: Considérese un *script* para ensamblar un fichero con código ensamblador (para código en C, es más natural usar `make` y un `Makefile`). Posibles funciones a parametrizar serían: el nombre del fichero a ensamblar, si se incluye la opción para usar el depurador (-gstabs), si se guardan los errores en un fichero, nombre del fichero en el que se guardan los errores, etc. Además, podemos dotar al *script* de otras opciones que reducen su funcionalidad. La opción -help muestra por pantalla la documentación de cómo utilizar el *script* pero sin realizar ninguna tarea. Análogamente, la opción -version tan sólo muestra por pantalla la versión actual.

A nivel interno del *script*, todas estas opciones y sus valores están almacenadas en el array de parámetros representado por la variable "$*". Cuando se diseña un *script*, la estructura genérica que se utiliza para procesar las opciones consta de los siguientes pasos:

1. Definición de los valores por defecto: Se asignan a diferentes variables los valores que deben de tomar en el caso de que no se especifique la opción para su modificación.

2. Procesado secuencial de las opciones: Se implementa un bucle que itera sobre los valores de los parámetros recibidos por el *script* y modifica los valores de las opciones especificadas. Se deben utilizar las sentencias while y case.

3. Comprobación de que las opciones se han utilizado de forma consistente: En ocasiones los scripts tienen opciones que influyen unas en las otras. Esto es, que la utilización de una opción hace el uso de una segunda irrelevante. El *script* debe de detectar estas posibles inconsistencias.

4. Ejecución de la tarea pertinente.

3.7. Comando grep

El comando grep (o sus variantes egrep y fgrep) procesa un fichero en busca de una palabra dada. El fichero se procesa línea a línea, y se imprimen aquellas líneas que contienen dicha palabra.

Ejemplo 1. Agenda Utilizar el programa grep para procesar un fichero que contiene los datos de una simple agenda.

Dada una fecha, el comando grep puede buscar en el fichero de datos que contiene la información de la agenda todos los eventos que tendrán lugar en esa fecha.

Se asume que cada evento está almacenado en el fichero agenda.txt mediante una línea que comienza por la fecha del evento con formato dia/mes/año (los tres campos con dos dígitos cada uno) seguida de una breve descripción. Por ejemplo:

```
31/12/04 Cena en casa de mis suegros
```

Se asume que el usuario escribe los eventos a recordar con este formato en el fichero con nombre agenda.txt.

Se pide: Escribir el *script* cita que obtiene del usuario una fecha y devuelve los eventos planeados para dicha fecha. El *script* debe seguir pidiendo fechas hasta que el usuario pulsa Control-D como señal de final de entrada.
Ejemplo de invocación:

```
unix$ agenda
Fecha: 31/12/04
31/12/04 Cena en casa de mis suegros
Fecha: 01/01/05
01/01/05 Viaje
Fecha: 03/01/05
Fecha: 05/01/05
05/01/05 Retorno de viaje
Fecha:
unix$
```

Solución:

```
#!/bin/sh

#
# Iterar mientras el usuario teclee entrada
#
echo -n "Fecha: "
while read date
do
  grep $date agenda.txt
  echo -n "Fecha: "
done
```

3.8. Comando sed

El programa `sed` procesa una por una las líneas de un fichero y permite la aplicación de comandos sencillos a cada una de las líneas. Una posible aplicación de este comando es la creación de ficheros a partir de otros a través de la substitución.

3.9. Comando awk

El programa awk es un buscador y procesador de patrones en un fichero. Dado un fichero, el programa procesa el fichero línea a línea. El programa además divide cada línea del fichero en campos y permite especificar comandos a aplicar sobre cada campo. En la página del manual se detallan las opciones posibles así como la manera de proveer estos comandos.

El comando awk tiene una opción (–F) para especificar el carácter que separa los diferentes campos de una línea. Además, awk permite especificar comandos. Por ejemplo, se puede utilizar print para imprimir un determinado campo. Si no se especifica ningún fichero, se analiza la entrada estándar.

Ejemplo:

```
awk -F: '{print $1, $6, $7}' /etc/passwd
```

Procesa el fichero /etc/passwd línea a línea, separando cada línea en campos delimitados por el carácter :, imprimiendo, para cada una de las líneas, los campos 1 ($1), 6 ($6) y 7 ($7), separados por un espacio (comando '{print $1, $6, $7}')

El comando que ejecuta awk (print) recibe como argumentos los campos separados por awk. Por eso se utilizan las variables predefinidas ($1, $2...) Observa que se utilizan comillas simples (') para referirnos a estas variables sin que se sustituyan por su valor.

3.10. Localizar y procesar ficheros: comando find

El comando Unix find se utiliza para buscar en una estructura de directorios, ficheros que cumplan un cierto criterio. Las opciones que este comando recibe permiten especificar, no sólo el directorio a partir del cual se comienza la búsqueda (descendiendo por los subdirectorios) y los ficheros a buscar, sino que también permite especificar un comando a ejecutar cada vez que se encuentra un fichero. Todas estas opciones están claramente documentadas en la página del manual del comando.

Notas sobre el comando find:

- En la ejecución del comando find, el conjunto de ficheros que comienza por la letra F se puede especificar con la expresión F*, pero para que el comando lo procese se debe escribir entre apóstrofes '.
- Tal como se explica en la página del manual del comando find, si se especifica un comando a ejecutar sobre cada fichero encontrado, dicho comando debe estar delimitado por un punto y coma (;) al final. El fichero que se ha encontrado (su nombre) se puede incluir en el comando mediante los caracteres {}, pero deben ser precedidos por el carácter de escape
  para que puedan ser procesados correctamente. Lo mismo sucede con el carácter ; al final.

3.11. Uso de funciones en *scripts*

Muchas veces en un programa se realiza muchas veces la misma operación, por ejemplo, presentar la ayuda cada vez que se produce un error o el cálculo de un resultado. Para facilitar la comprensión del código y hacerlo más modular se utilizan funciones y procedimientos. En bash las funciones se codifican **antes** del programa principal y devuelven sus resultados a través de la salida estándar.
Ejemplo: un programa que calcule la suma $2 + 3$ con una función y presente el resultado por pantalla.

```
#!/bin/bash

Suma() {
m=`expr 2 + 3`
echo $m
}

echo "Calculo la suma "
Suma
```

Una función también puede recibir parámetros, así $1 será el primer parámetro de la función, $2 el segundo...
Ejemplo: modificamos el programa anterior para que la función acepte cualquier número.

```
#!/bin/bash

Suma() {
m=`expr $1 + $2`
echo $m
}

echo "Calculo la suma "
Suma 23 45
```

Podemos usar el resultado de una función dentro de nuestro programa, asignándole a una variable del programa principal el resultado de su ejecución.

Ejemplo: modificamos el programa anterior para que sea el programa principal quien muestre por pantalla el resultado de la suma.

```
#!/bin/bash

Suma() {
m=`expr $1 + $2`
echo $m
}

echo "Calculo la suma "
n=`Suma 23 45`
echo La suma es: $n
```