Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

PRÁCTICAS DE I.S.O.

2º de Grado en Ingeniería Informática

Boletín de prácticas 2 – Programación de *shell scripts* en Linux (1/3)

Curso 2012/2013

Índice

1. Concepto de shell en Linux

Un shell es un intérprete de órdenes, y un intérprete de órdenes es un programa que procesa todo lo que se escribe en el terminal. Básicamente, permite a los usuarios interactuar con el sistema operativo para darle órdenes. En otras palabras, el objetivo de cualquier intérprete de órdenes es procesar las órdenes o ejecutar los programas que el usuario teclea.

El prompt es una indicación que muestra el intérprete para anunciar que espera una orden del usuario. Cuando el usuario escribe una orden, el intérprete la ejecuta. Dicha orden puede ser interna o externa. Las órdenes internas son aquellas que vienen incorporados en el propio intérprete, como, por ejemplo, `echo`, `cd`, o `pwd`. Las externas, por el contrario, son programas separados como, por ejemplo, todos los programas que residen en los directorios `/bin` (como `ls`), `/usr/bin` (como `cut`), `/sbin` (como `fsck`) o `/usr/sbin` (como `lpc`).

En el mundo UNIX/Linux existen tres grandes familias de shells: `sh`, `csh` y `ksh`. Se diferencian entre sí, fundamentalmente, en la sintaxis de sus órdenes internas (las que implementa el propio shell) y en su interacción con el usuario. En estas prácticas nos centraremos en el uso del shell `bash`, una variante *libre* de la familia `sh`.

Por defecto, cada usuario tiene asignado un shell, establecido en el momento de creación de su cuenta, y que se guarda en el fichero `/etc/passwd`. El shell asignado a un usuario se puede cambiar de dos maneras: editando manualmente dicho fichero (lo cual sólo puede hacer el administrador del sistema), o bien con el programa `chsh` (que lo puede ejecutar el propio usuario). Los shells están en los directorios `/bin` y `/usr/bin`[1]. Por ejemplo, para hacer que el shell por defecto sea `/bin/bash` se ejecutaría:

```
chsh -s /bin/bash
```

Una de las principales características del shell es que puede programarse usando ficheros de texto a partir de órdenes internas y programas externos. Además, el shell ofrece construcciones y facilidades para hacer más sencilla su programación. Estos ficheros de texto se llaman scripts, shell scripts o guiones shell. Tras su creación, estos guiones shell pueden ser ejecutados tantas veces como se desee. Una posible definición podría ser la siguiente: "*Un guión shell es un fichero de texto ejecutable que contiene una secuencia de órdenes ejecutables por el shell*".

Un guión shell puede incluir comentarios. Para ello se utiliza el carácter # al inicio del texto que constituye el comentario. Además, en un guión shell se puede indicar el shell concreto con el que se debe interpretar o ejecutar, indicándolo en la primera línea de la siguiente forma (el carácter # no es un comentario en este caso):

```
#!/bin/bash
```

La programación de guiones shell es una de las herramientas más apreciadas por todos los administradores y muchos usuarios de UNIX/Linux ya que permite automatizar tareas complejas y/o repetitivas, y ejecutarlas con una sola llamada al script, incluso de manera automática a una hora preestablecida sin intervención humana. A continuación se muestran una serie de ejemplos de distintas tareas que se suelen automatizar con scripts:

- Tareas administrativas definidas por el propio sistema o por su administrador. Por un lado, algunas partes de los sistemas UNIX/Linux son guiones shell. Para poder entenderlos y modificarlos es necesario tener alguna noción sobre la programación de scripts. Por otro lado, el administrador del sistema puede necesitar llevar a cabo tareas de mantenimiento del sistema de manera regular. El uso de guiones shell permite automatizarlas fácilmente en la mayor parte de los casos.
- Tareas tediosas, incluso aquellas que sólo se van a ejecutar una o dos veces, para las que no importa el rendimiento obtenido durante su realización pero sí que se terminen con rapidez.
- Integración de varios programas independientes para que funcionen como un conjunto de forma sencilla.

[1] El fichero `/etc/shells` contiene una lista de los shells disponibles.

- Desarrollo de prototipos de aplicaciones más complejas que posteriormente se implementarán en lenguajes más potentes.

Conocer a fondo el shell aumenta tremendamente la rapidez y productividad a la hora de utilizarlo, incluso sin hacer uso de guiones (es decir, utilizándolo simplemente desde el *prompt* del sistema).

Los guiones shell pueden utilizar un sinnúmero de herramientas como:

- Órdenes del sistema internas o externas, por ejemplo, órdenes como `echo`, `ls`, etc.
- Lenguaje de programación del shell, por ejemplo, construcciones como `if/then/else/fi`, etc.
- Programas y/o lenguajes de procesamiento en línea como `sed` o `awk`.
- Programas propios del usuario escritos en cualquier lenguaje de programación.

Si un guión shell no resulta suficiente para lo que queremos hacer, existen otros lenguajes interpretados mucho más potentes como Perl, TCL o Python.

El intérprete de órdenes seleccionado para realizar estas prácticas es el Bourne-Again Shell o `bash`, cuyo ejecutable es `/bin/bash`. El resto del contenido de este documento está centrado en este intérprete de órdenes.

2. Funcionamiento del shell

Suponiendo que tenemos el siguiente guión shell,

```
#!/bin/bash
clear
date
```

al ejecutarse el proceso que se sigue es el siguiente:

1. El shell `/bin/bash` hace un `fork`.
2. El proceso padre espera mientras no termina el nuevo proceso hijo.
3. El proceso hijo hace un `fork` y un `exec` para ejecutar la orden `clear`, y a continuación ejecuta un `wait` para esperar a que termine la ejecución de `clear`.
4. Una vez que ha terminado la orden `clear`, el proceso hijo repite los mismos pasos pero esta vez ejecutando la orden `date`.
5. Si quedasen órdenes por ejecutar se seguiría el mismo procedimiento.
6. Cuando finaliza el proceso hijo, el proceso padre reanuda su ejecución.

3. Variables y parámetros

3.1. Variables

Cada shell tiene unas variables ligadas a él, a las que el usuario puede añadir tantas como desee. Para dar un valor a una variable `variable` se usa la sintaxis:

```
variable=valor
```

Nótese que no puede haber espacios entre el nombre de la variable, el signo = y el valor. Por otra parte, si se desea que el valor contenga espacios, es necesario utilizar comillas.

Para obtener el valor de una variable hay que anteponerle a su nombre el carácter $. Por ejemplo, para visualizar el valor de una variable:

```
echo $variable
```

Un ejemplo del uso de las variables sería:

```
$ mils="ls -l" # Se crea una nueva variable
$ mils         # No hace nada porque busca el ejecutable
               # mils que no existe
$ $mils        # Ejecuta la orden "ls -l"
$ echo $mils   # Muestra el contenido de la variable mils,
               # es decir, "ls -l"
```

Las variables se dividen en dos tipos:

- **Variables locales**: no son heredadas por los procesos hijos del shell actual cuando se realiza un `fork`.
- **Variables de entorno**: heredadas por los procesos hijos del shell actual cuando se ejecuta un `fork`.

La orden `export` convierte una variable local en variable de entorno:

```
$ export mils      # Convierte la variable mils en variable de entorno
$ export var=valor # Crea la variable, le asigna "valor"
                   # y la exporta a la vez
```

La orden `set` muestra todas las variables (locales y de entorno) mientras que la orden `env` muestra sólo las variables de entorno. Con la orden `unset` se pueden restaurar o eliminar variables o funciones. Por ejemplo, la siguiente instrucción elimina el valor de la variable `mils`:

```
$ unset mils
```

Además de las variables que puede definir el programador, un shell tiene definidas, por defecto, una serie de variables. Las más importantes son:

- `PS1`: prompt primario. El siguiente ejemplo modifica el prompt, utilizando diferentes colores para el nombre del usuario, el *host* y el directorio actual:

  ```
  $ PS1='\[\033[31m\]\u@\h\[\033[0m\]:\[\033[33m\]\w\[\033[0m\] $ '
  ```

- `PS2`: prompt secundario.
- `LOGNAME`: nombre del usuario.
- `HOME`: directorio de trabajo (`home`) del usuario actual que la orden `cd` toma por defecto.
- `PWD`: directorio actual.
- `PATH`: rutas de búsqueda usadas para ejecutar órdenes o programas. Por defecto, el directorio actual no está incluido en la ruta de búsqueda. Para incluirlo, tendríamos que ejecutar:

  ```
  $ PATH=$PATH:.
  ```

- `TERM`: tipo de terminal actual.
- `SHELL`: shell actual.

Las siguientes variables son muy útiles al programar los guiones shell:

- `$?`: esta variable contiene el valor devuelto por la última orden ejecutada. Es útil para saber si una orden ha finalizado con éxito o ha tenido problemas. Un `0` indica que la orden se ha ejecutado con éxito, otro valor indica que ha habido errores.
- `$!`: identificador de proceso (PID) de la última orden ejecutada en segundo plano.
- `$$`: el PID del shell actual (comúnmente utilizado para crear nombres de ficheros únicos).
- `$-`: las opciones actuales suministradas para esta invocación del shell.
- `$*`: todos los argumentos del guión shell comenzando por el `$1`. Cuando la expansión ocurre dentro de comillas dobles, se expande a una sola palabra con el valor de cada parámetro separado por el primer carácter de la variable especial `IFS` (habitualmente un espacio). En general, `$*` es equivalente a `$1c$2c...`, donde `c` es el primer carácter del valor de la variable `IFS`. Si `IFS` no está definida, el carácter `c` se sustituye por un espacio. Si `IFS` es la cadena vacía, los parámetros se concatenan sin ningún separador.
- `$@`: igual que la anterior excepto cuando va entrecomillada. Cuando la expansión ocurre dentro de comillas dobles, cada parámetro se expande a una palabra separada, esto es, `$@` es equivalente a `$1 $2...`

3.2. Parámetros

Como cualquier programa, un guión shell puede recibir parámetros en la línea de órdenes para procesarlos durante su ejecución. Los parámetros recibidos se guardan en una serie de variables que el script puede consultar cuando lo necesite. Los nombres de estas variables son:

```
$1 $2 $3 ... ${10} ${11} ${12} ...
```

La variable `$0` contiene el nombre con el que se ha invocado el script, `$1` contiene el primer parámetro, `$2` contiene el segundo parámetro,...

A continuación se muestra un sencillo ejemplo de un guión shell que muestra los cuatro primeros parámetros recibidos:

```
#!/bin/bash
echo El nombre del programa es $0
echo El primer parámetro recibido es $1
echo El segundo parámetro recibido es $2
echo El tercer parámetro recibido es $3
echo El cuarto parámetro recibido es $4
```

La orden `shift` mueve todos los parámetros una posición a la izquierda, esto hace que el contenido del parámetro `$1` desaparezca y sea reemplazado por el contenido de `$2`, que `$2` sea reemplazado por `$3`, etc.

La variable `$#` contiene el número de parámetros que ha recibido el script. Como se indicó anteriormente `$*` o `$@` contienen todos los parámetros recibidos. La variable `$@` es útil cuando queremos pasar a otros programas algunos de los parámetros que nos han pasado.

Un ejemplo sencillo de un guión shell que muestra el nombre del ejecutable, el número total de parámetros, todos los parámetros y los cuatro primeros parámetros es el siguiente:

```
#!/bin/bash
echo El nombre del programa es $0
echo El número total de parámetros es $#
echo Todos los parámetros recibidos son $*
echo El primer parámetro recibido es $1
shift
echo El segundo parámetro recibido es $1
```

```
shift
echo El tercer parámetro recibido es $1
echo El cuarto parámetro recibido es $2
```

3.3. Reglas de evaluación de variables

A continuación se describen las reglas que gobiernan la evaluación de las variables de un guión shell:

- `$var`: valor de `var` si está definida, si no nada.
- `${var}`: igual que el anterior excepto que las llaves contienen el nombre de la variable a ser evaluada.
- `${var-thing}`: valor de `var` si está definida, si no `thing`.
- `${var=thing}`: igual que el anterior excepto cuando `var` no está definida en cuyo caso el valor de `var` pasa a ser `thing`.
- `${var?message}`: valor de `var` si está definida, si no imprime el mensaje en el terminal.
- `${var+thing}`: `thing` si `var` está definida, si no nada.

El siguiente ejemplo muestra cómo podemos usar una variable asignándole un valor en caso de que no esté definida:

```
$ echo El valor de var1 es ${var1}
  # No está definida, no imprimirá nada

$ echo El valor de la variable es ${var1=5}
  # Al no estar definida, le asigna el valor 5

$ echo Su nuevo valor es $var1
  # Su valor es 5
```

Pero si lo que queremos es usar un valor por defecto, en caso de que la variable no esté definida, sin inicializar la variable:

```
$ echo El valor de var1 es ${var1}
  # No está definida, no imprimirá nada

$ echo El valor de la variable es ${var1-5}
  # Al no estar definida, utiliza el valor 5

$ echo El valor es $var1
  # Su valor sigue siendo nulo, no se ha definido
```

Por otro lado, si lo que queremos es usar el valor de la variable y, en caso de que no esté definida, imprimir un mensaje:

```
$ echo El valor de var1 es ${var1}
  # No está definida, no imprimirá nada

$ echo El valor de la variable es ${var1? No está definida...}
  # Al no estar definida, se muestra en pantalla el mensaje

$ echo El valor es $var1
  # Su valor sigue siendo nulo, no se ha definido
```

Este último ejemplo nos muestra cómo utilizar un valor por defecto si una variable está definida:

```
$ var1=4  # Le asigna el valor 4

$ echo El valor de var1 es ${var1}
  # El valor mostrado será 4

$ echo El valor de la variable es ${var1+5}
  # Al estar definida, se utiliza el valor 5

$ echo El valor es $var1
  # Su valor sigue siendo 4
```

3.4. Arrays

El shell permite que se trabaje con arrays (o listas) unidimensionales. Un array es una colección de elementos del mismo tipo, dotados de un nombre, que se almacenan en posiciones contiguas de memoria. El primer subíndice del primer elemento del array es 0, y si no se utiliza subíndice, se considera también que se está referenciando a dicho elemento. No hay un tamaño máximo para un array, y la asignación de valores se puede hacer de forma alterna.

La sintaxis para crear e inicializar un array es la siguiente:

```
nombre_array=(val1 val2 val3 ...) # Crea e inicializa un array
nombre_array[x]=valor             # Asigna un valor al elemento x
```

Para acceder a un elemento del array se utiliza la siguiente sintaxis:

```
${nombre_array[x]} # Para acceder al elemento x
${nombre_array[*]} # Para consultar todos los elementos
${nombre_array[@]} # Para consultar todos los elementos
```

La diferencia entre usar * y @ es que `${nombre_array[*]}` crea una única palabra con todos los elementos del array mientras que `${nombre_array[@]}` crea palabra distintas para cada elemento del array.

Para conocer el tamaño en bytes de un elemento dado del array se utiliza la sintaxis `${#nombre_array[x]}`, donde x es un índice del array. De hecho, esa misma expresión vale también para saber la longitud de una simple variable normal (por ejemplo, `${#var}`). Si lo que nos interesa, por el contrario, es saber el número total de elementos del array, entonces emplearemos las expresiones `${#nombre_array[*]}` o `${#nombre_array[@]}`.

Nótese la diferencia entre las siguientes órdenes:

```
$ aux=`ls`
$ aux1=(`ls`)
```

En el primer caso, la variable `aux` contiene la salida de `ls` como una cadena de caracteres. En el segundo caso, al haber utilizado los paréntesis, `aux1` es un array, y cada uno de sus elementos es uno de los nombres de fichero devueltos por la orden `ls`. Si en el directorio actual tenemos los ficheros `a.latex`, `b.latex`, `c.latex`, `d.latex`, `e.latex` y `f.latex`, observe el resultado de ejecutar las órdenes anteriores:

```
$ ls
  a.latex b.latex c.latex d.latex e.latex f.latex
$ aux=`ls`
$ echo $aux
  a.latex b.latex c.latex d.latex e.latex f.latex
$ echo ${aux[0]}
  a.latex b.latex c.latex d.latex e.latex f.latex
```

```
$ aux1=(`ls`)
$ echo ${aux1[0]}
  a.latex
```

4. Caracteres especiales y de entrecomillado

Los mecanismos de protección se emplean para quitar el significado especial para el shell de ciertos caracteres especiales o palabras reservadas. Pueden emplearse para que caracteres especiales no se traten de forma especial, para que palabras reservadas no sean reconocidas como tales y para evitar la evaluación de variables.

Los metacaracteres `* $ | & ; ( ) { } < > espacio` y `tab` tienen un significado especial para el shell y deben ser protegidos o entrecomillados si quieren representarse a sí mismos. Hay 3 mecanismos de protección: el carácter de escape, las comillas simples y las comillas dobles[2].

Una barra inclinada inversa (o invertida) no entrecomillada (\) es el carácter de escape (no confundir con el código ASCII cuyo valor es 27 en decimal), el cual preserva el valor literal del siguiente carácter que lo acompaña, con la excepción de <nueva-línea>. Si aparece la combinación \<nueva-línea> y la barra invertida no está entre comillas, la combinación \<nueva-línea> se trata como una continuación de línea (esto es, se quita del flujo de entrada y no se tiene en cuenta). Por ejemplo, sería equivalente a ejecutar `ls -l`:

```
$ ls \
> -l
```

Encerrar caracteres entre comillas simples (`' '`) preserva el valor literal de cada uno de ellos entre las comillas. Una comilla simple no puede estar entre comillas simples, ni siquiera precedida de una barra invertida.

```
$ echo 'caracteres especiales: *, $, |, &, ;, (,),{,},<,>, \, ", `'
caracteres especiales: *, $, |, &, ;, (,),{,},<,>, \, ", `
```

Encerrar caracteres entre comillas dobles (`" "`) preserva el valor literal de todos los caracteres, con la excepción de `$`, `` ` `` y `\`. Los caracteres `$` y `` ` `` mantienen su significado especial dentro de comillas dobles. La barra invertida mantiene su significado especial solamente cuando está seguida por uno de los siguientes caracteres: `$`, `` ` ``, `"`, `\` o <nueva-línea>. Una comilla doble puede aparecer entre otras comillas dobles precedida de una barra invertida.

```
$ echo "caracteres especiales: *, \$, |, &, ;, (,),{,},<,>, \\, \", \`"
caracteres especiales: *, $, |, &, ;, (,),{,},<,>, \, ", `
```

Los parámetros especiales `$*` y `$@` tienen un significado especial cuando están entre comillas dobles (véanse los apartados 3.1 y 3.2).

Las expresiones de la forma `$'cadena'` se tratan de forma especial. Las secuencias de escape con barra invertida de `cadena`, si están presentes, son reemplazadas según especifica el estándar ANSI/ISO de C, y el resultado queda entre comillas simples:

- `\a`: alerta (campana).
- `\b`: espacio-atrás o retroceso.
- `\e`: carácter de escape (ESC).
- `\f`: nueva página.
- `\n`: nueva línea.

[2]Las comillas simples y dobles son las que aparecen en la tecla que hay a la derecha del 0 y en la tecla del 2, respectivamente.

- \r: retorno de carro.
- \t: tabulación horizontal.
- \v: tabulación vertical.
- \\: barra invertida.
- \xnnn: carácter cuyo código es el valor hexadecimal nnn.

Encerrar una cadena entre comillas invertidas (` `), o bien entre paréntesis precedida de un signo $, supone forzar al shell a ejecutarla como una orden y devolver su salida:

```
`orden`
   ó
$(orden)
```

Este proceso se conoce como sustitución de órdenes. A continuación se muestran varios ejemplos:

```
$ aux=`ls -lai` # Ejecuta ls -lai y almacena el resultado en aux
$ echo $aux     # Muestra el contenido de aux
$ fecha=$(date) # Ejecuta date y almacena el resultado en fecha
$ echo $fecha   # Muestra el contenido de fecha
```

Téngase en cuenta que el shell, antes de ejecutar una orden, procesa todos los caracteres especiales (en función de los mecanismos de protección), expande las expresiones regulares[3], y realiza la sustitución de órdenes:

```
$ echo *       # Muestra todos los ficheros del directorio actual
$ var=`ls`     # Primero ejecuta ls -la y luego almacena el resultado en var
$ echo $var    # Muestra el contenido de var, esto es, equivale a echo *
$ echo '$var'  # Imprime $var
$ echo "$var"  # Muestra el contenido de var, esto es, equivale a echo *
$ echo `date`  # Primero se ejecuta date y luego echo
```

5. Estructuras de control

5.1. Condiciones: `if` y `case`

En un guión shell se pueden introducir condiciones, de forma que determinadas órdenes sólo se ejecuten cuando éstas se cumplen. Para ello se utilizan las órdenes `if` y `case`, con la siguiente sintaxis:

```
if  «expresión»  # Habitualmente un test
then
      órdenes a ejecutar si se cumple la primera condición
elif  «expresión»
then
      órdenes a ejecutar si se cumple la segunda condición
      # (el bloque elif y sus órdenes son opcionales)
...
else
      órdenes a ejecutar en caso contrario
      # (el bloque else y sus órdenes son opcionales)
fi
```

La expresión a evaluar por `if` puede ser un `test`, una lista de órdenes (usando su valor de retorno), una variable o una expresión aritmética, esto es, básicamente cualquier orden que devuelva un código en `$?`.

Un ejemplo del funcionamiento de la orden `if` sería:

[3] La generación de nombres de ficheros se basa habitualmente en expresiones regulares tal y como se describe en la utilización de las órdenes `find` y `grep` del primer boletín de prácticas.

```
if grep -q main prac.c
then
        echo encontrada la palabra clave main
else
        echo no encontrada la palabra clave main
fi
```

La sintaxis de la orden `case` es la siguiente:

```
case $var in
v1)     ..
..
;;
v2|v3)  ..
..
;;
*)      ..                # Caso por defecto
;;
esac
```

`v1`, `v2` y `v3` son expresiones regulares similares a las utilizadas como comodines para los nombres de los ficheros.

Un ejemplo de su funcionamiento podría ser:

```
case $var in
     1)   echo La variable var es un uno
     ;;
     2)   echo La variable var es un dos
     ;;
     *)   echo La variable var no es ni un uno ni un dos
     ;;
esac
```

5.2. Bucles condicionales: `while y until`

También es posible ejecutar bloques de órdenes de forma iterativa dependiendo de una condición. La comprobación puede realizarse al principio (`while`) o al final (`until`). La sintaxis es la siguiente:

```
while    «expresión»   # Mientras la expresión sea cierta...
do
    ...
done

until    «expresión»   # Mientras la expresión sea falsa...
do
    ...
done
```

Un ejemplo del funcionamiento de ambas órdenes sería:

```
# Muestra todos los parámetros
while [ ! -z $1 ]
do
      echo Parámetro: $1
      shift
done

# También muestra todos los parámetros
until [ -z $1 ]
do
      echo $1
      shift
done
```

5.3. Bucles incondicionales: `for`

Con la orden `for` se ejecutan bloques de órdenes, permitiendo que en cada iteración una determinada variable tome un valor distinto. La sintaxis es la siguiente:

```
for var in lista
do
    uso de $var
done
```

Por ejemplo:

```
for i in 10 30 70
do
     echo Mi número favorito es $i # toma los valores 10, 30  y 70
done
```

Aunque la lista de valores del `for` puede ser arbitraria (incluyendo no sólo números, sino cualquier otro tipo de cadena o expresión), a menudo lo que queremos es generar secuencias de valores numéricos al estilo de la instrucción *for* de los lenguajes de programación convencionales. En este caso, la orden `seq`, combinada con el mecanismo de sustitución de órdenes (véase el apartado 4) puede resultarnos de utilidad. Por ejemplo:

```
for i in $(seq 0 5 25)
do
     # uso de $i que toma los valores 0, 5, 10, 15, 20 y 25
done
```

5.4. Menús de opciones: `select`

Con la orden `select` podemos solicitar al usuario que elija una opción de una lista. La sintaxis de la orden `select` es:

```
select opcion in [ lista ] ;
do
     # bloque de órdenes
done
```

`select` genera una lista numerada de opciones al expandir la lista `lista`. A continuación, presenta un prompt (`#?`) al usuario pidiéndole que elija una de las posibilidades, y lee de la entrada estándar la opción elegida. Si la respuesta dada es uno de los números de la lista presentada, dicho número se almacena en la variable `REPLY`, la variable `opcion` toma el valor del elemento de la lista elegido, y se ejecuta el bloque de órdenes. Si la respuesta es no válida, se vuelve a interrogar al usuario, y si es EOF, se finaliza. El bloque de órdenes se ejecuta después de cada selección válida, mientras no se termine, bien con `break` o bien con EOF. El valor de salida de `select` será igual al valor de la última orden ejecutada.

Un ejemplo sería el siguiente:

```
select respuesta in "Ver contenido directorio actual" \
    "Salir"
do
    echo Ha seleccionado la opción: $respuesta
    case $REPLY in
    1) ls .
    ;;
    2) break
    ;;
    esac
done
```

En pantalla aparecería:

```
1) Ver contenido directorio actual
2) Salir
#?
```

Si se selecciona la primera opción, 1, se mostraría el mensaje: "Ha seleccionado la opción: Ver contenido directorio actual", se ejecutaría la orden `ls` en el directorio actual y volvería a pedir la siguiente selección. Si por el contrario se pulsase un 2, seleccionando la segunda opción, aparecería el mensaje: "Ha seleccionado la opción: Salir" y se saldría del `select`.

5.5. Ruptura de bucles: `break y continue`

Las órdenes `break` y `continue` sirven para interrumpir la ejecución secuencial del cuerpo de un bucle. La orden `break` transfiere el control a la orden que sigue a `done`, haciendo que el bucle termine antes de tiempo. La orden `continue`, por el contrario, transfiere el control a `done`, haciendo que se evalúe de nuevo la condición, es decir, la ejecución del bucle continúa en la siguiente iteración. En ambos casos, las órdenes del cuerpo del bucle siguientes a estas sentencias no se ejecutan. Lo normal es que formen parte de una sentencia condicional.

Un par de ejemplos de su uso serían:

```
# Muestra todos los parámetros, si encuentra una "f" finaliza
while [ $# -gt 0 ]
do
      if [ $1 = "f" ]
      then
            break
      fi
      echo Parámetro: $1
      shift
done

# Muestra todos los parámetros, si encuentra una "f"
# se lo salta y continúa el bucle
while [ $# -gt 0 ]
do
      if [ $1 = "f" ]
      then
          shift
          continue
      fi
      echo Parámetro: $1
      shift
done
```

6. Ejemplos de guiones shell

1. El siguiente programa `llamar`, muestra su número PID y después llama a un programa llamado `num`, a través de la orden (.). Cuando `num` termina su ejecución, la orden (.) devuelve el control al programa que lo llamó, el cual muestra el mensaje.

   Guión llamar

   ```
   echo "$0 PID = $$"
   . num
   echo "se ejecuta esta línea"
   ```

   Guión num

   ```
   echo "num PID = $$"
   ```

   Como vemos, la orden (.) ejecuta un proceso como parte del proceso que se está ejecutando (`llamar` y `num` tienen el mismo número de proceso). Cuando el nuevo programa termina la ejecución, el proceso actual continúa ejecutando el programa original. El programa `num` no necesita permiso de ejecución.

2. Programa que evalúa la extensión de un fichero. Si ésta se corresponde con "txt", copia el fichero al directorio `~/copias`. Si es otra la extensión o no hace nada o presenta un mensaje.

```
case $1 in
*.txt)
       cp $1 ~/copias/$1
       ;;
*.doc | *.bak)
       # Tan sólo como ejemplo de otras extensiones
       ;;
*)
       echo "$1 extensión desconocida"
       ;;
esac
```

3. Ejemplo `break` y `continue`: este programa utiliza las órdenes `break` y `continue` para permitir al usuario controlar la entrada de datos.

```
while true        #bucle infinito
do
     echo "Introduce un dato "
     read respuesta
     case "$respuesta" in
     "nada")  # no hay datos
              break
              ;;
     "")  # si es un retorno de carro se continúa
          continue
          ;;
     *)  # proceso de los datos
         echo "se procesan los datos"
         ;;
     esac
done
```

4. Ejemplo de un menú:

```
while true
do
    clear
    echo "
          Ver directorio actual...........[1]
          Copiar ficheros.................[2]
          Editar ficheros.................[3]
          Imprimir fichero................[4]
          Salir del menú..................[5]"
    read i
    case $i in
    1)    ls -l|more; read z
          ;;
    2)    echo "Introduzca [desde] [hasta]"
          read x y
          cp $x $y
          read x
          ;;
    3)    echo "¿Nombre de fichero a editar?"
          read x;
          vi $x
          ;;
    4)    echo "¿Nombre de fichero a imprimir?"
          read x
          lpr $x
          ;;
    5)    clear; break
```

```
            ;;
        esac
    done
```

Este mismo ejercicio podría ser resuelto utilizando la orden `select`:

```
select opcion in "Ver directorio actual" \
  "Copiar ficheros" \
  "Editar ficheros" \
  "Imprimir fichero" \
  "Salir del menú"
do
  case $REPLY in
  1)    ls -l|more; read z
        ;;
  2)    echo "Introduzca [desde] [hasta]"
        read x y
        cp $x $y
        read x
        ;;
  3)    echo "¿Nombre de fichero a editar?"
        read x;
        vi $x
        ;;
  4)    echo "¿Nombre de fichero a imprimir?"
        read x
        lpr $x
        ;;
  5)    clear; break
        ;;
  esac
done
```

5. Este ejemplo lee dos números del teclado e imprime su suma (usando las órdenes `read`, `printf` y `let`).

```
#!/bin/bash
printf "Introduzca un número \n"
read numero1
printf "Introduzca otro número \n"
read numero2
let respuesta=$numero1+$numero2
printf "$numero1 + $numero2 = $respuesta \n"
```

6. Escribir un guión shell que, dado el `username` de un usuario, nos devuelva cuántas veces esa persona está conectada. (Usa: `who`, `grep`, `wc`).

```
#!/bin/bash
veces=`who | grep -w ^$1 | wc -l`
echo "$1 está conectado $veces veces"
```

7. Supongamos que queremos cambiar el sufijo de todos los archivos `*.tex` a `*.latex`. Haciendo `mv *.tex *.latex` no funciona (¿por qué?), pero sí con un guión shell.

```
#!/bin/bash
for f in *.tex
do
     nuevo=$(basename $f tex)latex
     mv $f $nuevo
done
```

8. Lo siguiente es un sencillo reloj que va actualizándose cada segundo, hasta ser matado con Ctrl-C:

```
while true do
    clear;
    echo "===========";
    date +"%r";
    echo "===========";
    sleep 1;
done
```

7. Ejercicios propuestos

1. Haga un shell script llamado `priult` que devuelva los argumentos primero y último que se le han pasado. Si se llama con:

   ```
   priult hola qué tal estás
   ```

   debe responder:

   ```
   El primer argumento es hola
   El último argumento es estás
   ```

   Mejorar este shell script para tratar los casos en los que se llame con 0 o 1 argumento, indicando que no hay argumento inicial y/o final.

2. Cree un shell script llamado `fecha_hora` que devuelva la hora y la fecha con el siguiente formato:

   ```
   Son las hh horas, xx minutos del día dd de mmm de aaaa
   ```

   donde `mmm` representa las iniciales del mes en letra (ENE, FEB, MAR, ..., NOV, DIC).

3. Cree un shell script llamado `tabla` que, a partir de un número que se le pasará como argumento, obtenga la tabla de multiplicar de ese número. Si se llama con:

   ```
   tabla 5
   ```

   debe responder:

   ```
   TABLA DE MULTIPLICAR DEL 5
   ==========================
        5 * 1 = 5
        5 * 2 = 10
        ...
        5 * 9 = 45
        5 * 10 = 50
   ```

   Mejore el shell script para que se verifique que sólo se le ha pasado un argumento y que éste es un número válido entre 0 y 10.

4. Haga un shell script llamado `cuenta_tipos` que devuelva el número de ficheros de cada tipo que hay en un directorio, así como los nombres de estos ficheros. Tendrá un único argumento (opcional) que será el directorio a explorar. Si se omite el directorio se considerará que se trata del directorio actual. Devolverá 0 (éxito) si se ha llamado de forma correcta y 1 (error) en caso contrario. La salida será de esta forma:

   ```
   La clasificación de ficheros del directorio XXXX es:
   Hay t ficheros de texto (ASCII text): X1, X2, ... Xt
   Hay dv ficheros de dispositivo de bloques (block special): X1, X2, ... Xdv
   Hay d directorios (directory): X1, X2, ... Xd
   Hay e ficheros ejecutables (executable) : X1, X2, ... Xe
   ```

   (Pista: usar la orden `file`)

5. Cree un shell script llamado `infosis` que muestre la siguiente información:

   - Un saludo de bienvenida del tipo:

```
Hola usuario uuu, está usted conectado en el terminal ttt
```

donde `uuu` y `ttt` son, respectivamente, el nombre de usuario y el terminal desde el que se ejecuta la orden.

- La fecha y la hora actuales, usando para ello el ejercicio número 2.
- Una lista con los usuarios conectados.
- Una lista de los procesos del usuario que se están ejecutando en ese momento.

(Pista: usar las ordenes `who` y `whoami`)

8. Bibliografía

- Página de manual del intérprete de órdenes bash (`man bash`).
- *Unix shell patterns*, J. Coplien *et alt.*
- *Programación en BASH - COMO de introducción*, Mike G. (traducido por Gabriel Rodríguez).
- *Shell scripting*, H. Whittal.
- *El libro de UNIX*, S. M. Sarwar *et alt.*