Fundamentos de Sistemas Operativos

segunda práctica

Explotación programada de "sistemas operativos de tipo UNIX" mediante el desarrollo de programas en lenguajes de programación de tipo "intérprete de comandos de tipo UNIX"

(shell scripts)

Escuela Politécnica Superior de Alcoy
Curso 2021 – 2022

Segunda práctica: shell scripts

Objetivos

- Adquirir competencia en la explotación programada de “sistemas operativos de tipo UNIX” mediante el desarrollo de programas en lenguajes de programación de tipo “intérprete de comandos de tipo UNIX”.
- Asimilar el concepto “intérprete de comandos de tipo UNIX” y los principios de explotación UNIX “redirección de entrada/salida”, “combinación de comandos” y “lenguaje de programación intérprete de comandos”.

Aspectos ejercitados

- Desarrollo de shell scripts.
- Uso de variables en shell scripts.
- Uso de estructuras de control en shell scripts.
- Uso del comando `test` en combinación con la estructura alternativa `if`.
- Uso de la sentencia `exit`.
- Uso de diversas utilidades UNIX.

Contenido

Introducción

En esta introducción, se recuerdan conceptos básicos de sistemas operativos en general y de “sistemas operativos de tipo UNIX” en particular, se recoge la idea de complementar la funcionalidad mínima atribuida a un intérprete de comandos básico, hasta convertirlo en un lenguaje de programación de alto nivel, y se presenta el uso de variables y estructuras de control en “intérpretes de comandos de tipo UNIX”.

1.1.- Los “intérpretes de comandos de tipo UNIX”.

En general, los intérpretes de comandos determinan la forma en la que se perciben y usan los sistemas informáticos. Además, en “sistemas operativos de tipo UNIX”, los “intérpretes de comandos de tipo UNIX” determinan la forma en que se edifica nueva funcionalidad, por lo que cobran, si cabe, mayor importancia.

En esta sección, se recuerda el concepto de intérprete de comandos, en general, la filosofía de explotación subyacente al diseño del sistema operativo UNIX y el concepto de “intérprete de comandos de tipo UNIX”, en particular.

1.1.1.- Concepto de *intérprete de comandos*.

En general, hablamos de un “intérprete de comandos” cuando nos referimos a un servicio, de sistema operativo, típicamente identificado con un programa, cuyo propósito fundamental es interactuar con el usuario para poner a su disposición aquellos servicios, de sistema operativo, para los cuales se ha previsto que puedan ser accedidos a través del mismo. De manera que, en la medida en que el intérprete de comandos determina la forma en la que se solicitan los servicios, se convierte en un componente fundamental de la interfase usuario_humano – sistema_operativo, la cual se complementa con las interfases de los diferentes servicios, del sistema operativo, que, accedidos a través del intérprete de comandos, interactúan directamente con el usuario.

En un sistema operativo, el servicio “intérprete de comandos” no tiene porque ser único, siendo susceptible de ser implementado con diversas tecnologías de interacción con el usuario (textual, gráfica, oral…).

1.1.2.- Filosofía de explotación UNIX.

El diseño del sistema operativo UNIX responde a una forma específica de entender como explotar los recursos de un sistema informático y, en particular, a una forma de entender como reutilizar esfuerzo y edificar funcionalidad. Dicha forma de entender, dicha filosofía de explotación, se concreta en tres ideas fundamentales:

- Debería de ser posible el cambiar (redireccionar) el origen y/o el destino de la información que procesa un comando, en el momento de ordenarlo.
- Debería de ser posible el "combinar" el efecto de varios comandos.
- Debería de ser posible el utilizar un intérprete de comandos "a modo de lenguaje de programación".

Redirección de entrada/salida.

En general, el núcleo algorítmico que proporciona determinada funcionalidad es razonablemente independiente de la procedencia o destino de la información que procesa. Así, por ejemplo, el núcleo de un algoritmo que ordena una lista de palabras es independiente de la procedencia o destino de la misma (el terminal, un fichero...). Si fuera posible cambiar el origen y/o destino de la información que procesa un comando, en el momento de ordenarlo, el esfuerzo invertido, en el desarrollo del comando, podría reutilizarse en diferentes situaciones en las que los orígenes y/o destinos de la información procesada podrían ser diferentes.

Redireccionar la entrada es cambiar el origen de la información que procesa un comando, en el momento de ordenarlo.

Redireccionar la salida es cambiar el destino de la información que procesa un comando, en el momento de ordenarlo.

Combinación de comandos.

Una necesidad, podría ser, la de obtener un listado del contenido de un directorio ordenado alfabéticamente. Otra, la de obtener una relación de procesos ordenada por identificador de proceso. En un escenario de explotación convencional, ambos servicios requerirían de la implementación de algoritmos de ordenación. Alternativamente, el principio de explotación "combinación de comandos" (o "encadenado de comandos") sugiere la posibilidad de implementar la función de ordenación una sola vez. El listado del directorio ordenado se obtendría "combinando" el efecto de un comando que sería capaz de proporcionar el listado del directorio, sin ordenar, con el efecto del comando que es capaz de ordenar. Análogamente, la relación de procesos ordenada se obtendría "combinando" el efecto de un comando que sería capaz de proporcionar la relación de procesos, sin ordenar, con el efecto del comando que es capaz de ordenar. Además, la función de ordenación podría ser reutilizada en infinidad de otras ocasiones.

Lenguaje de programación intérprete de comandos.

La funcionalidad mínima atribuida a la parte del sistema operativo conocida como intérprete de comandos es la de interactuar con el usuario para poner a su disposición, en principio, de forma interactiva, un comando a la vez, aquellos servicios, de sistema operativo, para los cuales se ha previsto que puedan ser accedidos a través del mismo. A partir de aquí, la idea de proporcionar una "lista de comandos", por ejemplo en forma de contenido de un fichero de texto, "a modo de comando" cuyo efecto debería ser el de los diferentes que aparecen en la lista aplicados en secuencia, resulta inmediata. Con dicha funcionalidad extendida, los "comandos", que aparecerían en las listas, se asimilan a "instrucciones" de un lenguaje de programación y las "listas de comandos" a "programas", por lo que la notación entendida por el intérprete de comandos empieza a parecerse a un lenguaje de programación.

Para complementar y potenciar la idea de usar el intérprete de comandos a modo de lenguaje de programación, en los "intérpretes de comandos de tipo UNIX", se introduce, además, la posibilidad de utilizar variables y estructuras de control.

1.1.3.- Concepto de "intérprete de comandos de tipo UNIX".

La función principal de un intérprete de comandos, también en UNIX (shell), es la de interactuar con el usuario para poner a su disposición los servicios básicos proporcionados por el sistema operativo. Sin embargo, dada la filosofía de explotación subyacente al diseño del sistema operativo UNIX, un "intérprete de comando de tipo UNIX" debe, además, permitir expresar y articular la redirección de la entrada/salida, permitir expresar y articular la combinación de comandos, y ejecutar programas escritos en el lenguaje de programación que él mismo define.

Otras características típicas, de "intérpretes de comandos de tipo UNIX", serían: la diferenciación entre comandos internos y externos, la posibilidad de ejecutar comandos en modo *background*, el uso de variables de entorno como forma de controlar el comportamiento de determinados comandos, el mantenimiento de históricos de comandos como forma de agilizar la interacción con el usuario, o el reconocimiento de metacaracteres (comodines o wildcards), usados para expresar patrones de nombres de ficheros.

La aproximación a la explotación de sistemas informáticos que representa el concepto de "intérprete de comandos de tipo UNIX" no es única. Sin embargo, resulta extraordinariamente potente, aunque requiere de cualificación. Contrasta con la otra forma de explotación típica, basada en intérpretes de comandos con soporte gráfico y dispositivos táctiles o ratón, más propia de interfases pensadas para usuarios no cualificados.

Una misma idea, la de un intérprete de comandos que es a su vez un lenguaje de programación, es susceptible de ser implementada, incluso para un mismo sistema, de diversas formas. Por lo que, típicamente, para un mismo "sistema operativo de tipo UNIX", existen diferentes implementaciones de "intérpretes de comandos de tipo UNIX". Es por ello que, con frecuencia, la expresión "intérprete de comandos de tipo UNIX" se utiliza en su forma plural.

Para "sistemas operativos de tipo UNIX", el "intérprete de comandos de tipo UNIX" más básico es el *Bourne shell* (`sh`), que suele estar disponible en todas las instalaciones. Otros "intérpretes de comandos de tipo UNIX", usuales, serían: el *Korn shell* (`ksh`), el shell con sintaxis parecida al lenguaje de programación C (`csh`) y el *Bourne-Again SHell* (`bash`), totalmente compatible con el `sh` al tiempo que incorpora características del `ksh` y `csh`.

1.2.- Shell scripts.

El principio de explotación UNIX "lenguaje de programación intérprete de comandos" sugiere usar el intérprete de comandos "a modo de lenguaje de programación". Bajo dicha aproximación, los "comandos", que reconoce el intérprete de comandos, se asimilan a "instrucciones de un lenguaje de programación" y "listas de comandos" a "programas". Además, en "intérpretes de comandos "de tipo UNIX", se introduce la posibilidad de utilizar variables y estructuras de control.

En esta sección, se fija el concepto de programa escrito en "lenguaje de programación intérprete de comandos" (*shell script*) y se describe la forma de pasar parámetros (a shell scripts) y ejecutar shell scripts, en UNIX.

1.2.1.- Concepto de shell script.

Se denomina *shell script* a un fichero de texto que, "a modo de programa", especifica los comandos que deben ser ejecutados, por determinado intérprete de comandos (shell), para alcanzar determinada funcionalidad.

Ejemplo: el contenido de un shell script, almacenado en el fichero *mi_ls,* que personalizaría el uso del comando `ls`, podría ser:

```
#!/bin/bash
# mi_ls
ls -l
```

Las líneas que empiezan por el carácter `#` son comentarios. Sin embargo, la primera línea (`# !/bin/bash`) indica, además, el shell en particular en que debe ejecutarse el contenido que se detalla a continuación.

1.2.2.- Ejecución de shell scripts en “sistemas de tipo UNIX”.

Todos los “intérpretes de comandos de tipo UNIX,” típicamente disponibles en “sistemas operativos de tipo UNIX”, comparten una misma forma de operar. Si, en el comando en el que se solicita su ejecución (la del intérprete de comandos), no se especifica ningún argumento, “entienden” que la interacción (entrada de comandos) tendrá lugar a través de la entrada/salida estándar (normalmente vinculada al terminal o la consola, pero no siempre). Si, por el contrario, en el comando en el que se solicita su ejecución (la del intérprete de comandos), se especifica un argumento, “entienden” que éste es el nombre del fichero del que deben leer los comandos que deben ejecutar. Por último, una vez están en ejecución, son capaces de “entender” que deben de localizar determinado shell script y desencadenar el efecto de que se interprete su contenido (“ejecutarlo”), en cuyo caso, desde un punto de vista de mecanismos de protección, es necesario que, para dicho shell script, esté habilitado el permiso de ejecución.

De modo que, dada la forma en la que, en general, se comportan los “intérpretes de comandos de tipo UNIX”, en “sistemas operativos de tipo UNIX”, la ejecución de un shell script puede conseguirse de tres formas distintas:

Redireccionando la entrada.

Pidiendo al shell, con el que interactuamos, que cree un nuevo proceso, en el que ponga en ejecución al shell de nuestra elección, provocando que, para el mismo, la entrada, en lugar de proceder de la consola o del terminal, proceda del fichero shell script que contiene los comandos que pretendemos ejecutar (esto es, redireccionando la entrada del shell que se pone en ejecución al fichero shell script que contiene los comandos que pretendemos ejecutar).

Ejemplo:

```
$  bash < mifich
```

Proporcionando argumentos.

Pidiendo al shell, con el que interactuamos, que cree un nuevo proceso, en el que ponga en ejecución al shell de nuestra elección, pasándole, como argumento, el nombre del fichero shell script que contiene los comandos que pretendemos ejecutar.

Ejemplo:

```
$  bash mifich
```

A modo de comando externo.

Pidiendo al shell, con el que interactuamos, que, él mismo, localice al fichero shell script, que contiene los comandos que pretendemos ejecutar, y que desencadene el efecto de que se interprete su contenido. Para ello, proporcionamos al shell, a modo de nombre de comando externo, el nombre del fichero shell script cuya ejecución nos interesa. Desde un punto de vista de mecanismos de protección, además, es necesario que, para dicho shell script, esté habilitado el permiso de ejecución.

Ejemplo:

```
$  chmod +x mifich
$  mifich
```

La idea de "…desencadenar el efecto de que se interprete el contenido del shell script…" se concreta en que, en realidad, no es en el contexto del mismo proceso en el que se ejecuta el shell al que pedimos la localización del shell script que nos interesa en el que tiene lugar la interpretación del contenido de éste, sino en un nuevo shell, acorde al tipo de shell script, creado, por el shell con el que interactuamos, para dicho propósito.

La localización de shell scripts, cuya ejecución se solicita a modo de comandos externos, está sujeta al mismo protocolo de localización de los ficheros ejecutables en los que se implementan comandos externos, el cual se controla a través de la variable `PATH`.

1.2.3.- Paso de parámetros a "shell scripts de tipo UNIX".

En el diseño de shell scripts, que han de ser ejecutados por un "intérprete de comandos de tipo UNIX", se puede hacer uso de la posibilidad de que estos puedan ser invocados con argumentos (o parámetros actuales o reales), los cuales pueden ser referenciados, en el texto del propio script, como `$0`, `$1`, `$2`, `$3`, etc (prámetros formales).

Concretamente, `$0` referencia al nombre del propio shell script, `$1` referencia al primer argumento, suministrado junto al nombre, `$2` al segundo, etc. Además, `$*` referencia todos los argumentos (en conjunto). Y `$#` se refiere al número total de argumentos suministrados.

Ejemplo: contenido del shell script *argumentos*.

```
#!/bin/bash
# argumentos
echo Orden tecleada: $0 $*
echo Número de argumentos: $#
echo Nombre de la orden: $0
echo Argumento 1: $1
echo Argumento 2: $2
echo Argumento 3: $3
```

Ejemplo: Invocación del shell script *argumentos* y resultado de su ejecución.

```
$  argumentos pera uva limon
Orden tecleada: argumentos pera uva limon
Número de argumentos: 3
Nombre de la orden: argumentos
Argumento 1: pera
Argumento 2: uva
Argumento 3: limón
```

1.3.- Uso de variables en “intérpretes de comandos UNIX”.

Para mejorar la capacidad expresiva de los “intérpretes de comandos de tipo UNIX”, cuando se usan a modo de lenguajes de programación, éstos permiten hacer uso de variables en shell scripts. Dichas variables, no se declaran, se nombran cuando se les asigna valor por primera vez y son todas del mismo tipo: cadena de caracteres alfanuméricos.

1.3.1.- Creación de variables.

En “intérpretes de comandos de tipo UNIX”, para crear una nueva variable, es suficiente con hacer uso de su nombre, en el momento en que se necesita, en la parte izquierda de la sentencia de asignación (denotada por el símbolo =) en la que, por primera vez, se le asigna valor. El símbolo = debe utilizarse sin dejar espacios a su alrededor.

Ejemplo:

```
#!/bin/bash
# creacion_variable
   ...
nombre=Pepe
   ...
```

1.3.2.- Modificación del valor de una variable.

En “intérpretes de comandos de tipo UNIX”, para modificar el valor de una variable, es suficiente con hacer uso de su nombre en la parte izquierda de la sentencia de asignación (denotada por el símbolo =) en la que, por n-ésima vez, se modifica su valor. El símbolo = debe utilizarse sin dejar espacios a su alrededor.

Ejemplo:

```
#!/bin/bash
# modificación_variable
   ...
nombre=Pepe
   ...
nombre=Juan
   ...
```

1.3.3.- Consulta del valor de una variable.

En “intérpretes de comandos de tipo UNIX”, para referenciar, consultar o hacer uso del valor de una variable hay que anteponer el operador $ al nombre de la variable, el cual debe utilizarse sin dejar espacios a su alrededor.

Ejemplo:

```
#!/bin/bash
# referenciado_variable
   ...
nombre=Pepe
   ...
echo $nombre
   ...
```

1.3.4.- Uso de entrecomillado (quoting).

En general, en el ámbito del desarrollo de notación para lenguajes de programación, llega a hacerse necesaria la consideración de elementos sintácticos sencillos que permitan la "matización" del significado de determinadas construcciones sintácticas (entendido en un sentido amplio, *entrecomillado* o *quoting*). Así, por ejemplo, podría diseñarse, que la letra a hace referencia a la variable de nombre a, que la letra a entrecomillada entre comillas simples 'a' se refiera al valor literal constante de tipo carácter a, y que la letra a entrecomillada entre comillas dobles "a" se refiera al valor literal constante de tipo cadena de caracteres (de longitud uno y valor a). También en el caso de desarrollo de notación para "intérpretes de comandos de tipo UNIX" se dan necesidades del estilo de las sugeridas en el ejemplo, pero con sus usos específicos. En el caso de "intérpretes de comandos de tipo UNIX" resultan de aplicación los siguientes usos de entrecomillado.

comillas simples:

Permiten la inclusión de espacios en blanco y metacaracteres en el literal constante que representa el valor de una cadena de caracteres alfanuméricos.

Ejemplo:

```
$ saludo='Hola; que tal !'
```

comillas dobles

Permiten expresar que el operador $ debe ser interpretado, por ejemplo, al construir el valor que se debe asignar a una variable:

Ejemplo: para entender el efecto de su uso, probar:

```
$  echo ' Alumno : - $nombre $apellido -'
$  echo " Alumno: - $nombre $apellido -"
```

símbolo $()

Permite usar al resultado de un comando como valor, por ejemplo, a la hora de asignar valor a una variable.

Ejemplo: para entender el efecto de su uso, probar:

```
$  hoy=date
$  echo $hoy
$  hoy=$(date)
$  echo $hoy
```

1.4.- Estructuras de control en intérpretes de comandos UNIX.

Además de la *composición secuencial* de comandos (lista de comandos que deben ejecutarse uno a continuación del otro, en secuencia), los "intérpretes de comandos de tipo UNIX" permiten el uso de *estructuras de control de flujo* análogas a aquellas disponibles en lenguajes de programación *imperativos estructurados* convencionales.

1.4.1.- La estructura alternativa `if`.

En lenguajes de programación de tipo "intérprete de comando de tipo UNIX", la *sentencia alternativa* `if` permite la ejecución condicional de comandos.

```
if comando_de_control
  then
       comandos
  else
       comandos_alternativos
fi
```

Sin embargo, mientras que en lenguajes imperativos convencionales la ejecución condicional expresada en una estructura alternativa está controlada por una *expresión booleana* (que evalúa cierto o falso), en el caso de los lenguajes de programación de tipo "intérprete de comandos de tipo UNIX" dicha ejecución condicional se vincula a la circunstancia de que el comando que controla la estructura "termine o no correctamente".

1.4.2.- El estado de terminación de un comando.

Se dice que un comando "termina correctamente" (o con éxito) cuando termina su ejecución sin generar ningún mensaje de error, en cuyo caso su "estado de terminación" vale 0. En caso contrario, su estado de terminación evaluará algún valor distinto de 0.

Además, el estado de terminación del último comando ejecutado por un "intérprete de comandos de tipo UNIX" se almacena en la *variable de entorno* de nombre ?

Ejemplo: si suponemos que el fichero que tiene por nombre *existe* existe, y que *noexiste* representa a un fichero que no existe, los comandos siguientes permiten observar diferentes estados de terminación de comandos.

```
$  cp noexiste fich
   ...
$  echo $?
$  cp existe fich
$  echo $?
```

Ejemplo: el shell script *mi_copia*, que se presenta a continuación, desencadena la ejecución del comando `cp` y comprueba su estado de terminación.

```
#!/bin/bash
# mi_copia
if cp $1 $2 2> /dev/null
   then
          echo " $1 ha sido copiado a $2 "
   else
          echo " $1 no ha podido ser copiado a $2 "
fi
```

1.4.3.- La sentencia `exit`

La *sentencia* `exit` fuerza la terminación inmediata de la ejecución de un shell script, de modo que no se ejecutan las sentencias que siguen a `exit`. El estado de terminación del shell script puede proporcionarse como parámetro para `exit` (la sentencia `exit -1` provocaría un estado de terminación, para la ejecución del shell script en que aparece, igual a -1).

Ejemplo: el shell script *mi_copia*, que se muestra a continuación, proporciona un estado de terminación distinto de cero en el caso en que el comando *cp* no termine correctamente:

```
#!/bin/bash
# mi_copia
if cp $1 $2 2> /dev/null
   then
         echo " $1 ha sido copiado a $2 "
         exit 0
   else
         echo " $1 no ha podido ser copiado a $2 "
         exit -1
fi
```

1.4.4.- El comando `test`.

El *comando externo* (programa ejecutable) `test` permite evaluar expresiones, por lo que resulta de utilidad utilizado conjuntamente con la sentencia `if`.

Ejemplo: en el shell script *mi_copia*, que se muestra a continuación, se comprueba que el número de argumentos es acorde a la operación que se intenta llevar a cabo, antes de intentar llevarla a cabo.

```
# !/bin/bash
# mi_copia
if test $# -ne 2
  then
        echo Uso de mi_copia: mi_copia f1 f2
        echo Observe que mi_copia necesita dos argumentos
        exit 1
  else
        if cp $1 $2 2> /dev/null
              then
                    echo " $1 ha sido copiado a $2 "
                    exit 0
              else
                    echo " $1 no ha podido ser copiado a $2 "
                    exit -1
        fi
fi
```

El comando `test` admite, además, lo que se conoce como la *notación abreviada* `[]`. Haciendo uso de dicha notación abreviada, el ejemplo de uso, del comando test, presentado con anterioridad, queda de la siguiente forma.

Ejemplo:

```
#!/bin/bash
# mi_copia
if [ $# -ne 2 ]
  ...
fi
```

En general, el comando *test* puede evaluar los siguientes tipos de expresiones:

Expresiones numéricas:

La forma general de las expresiones numéricas es:

```
N <primitiva> M
```

donde M y N representan valores numéricos.

-eq	test termina correctamente si N y M son iguales.
-ne	test termina correctamente si N y M son distintos.
-gt	test termina correctamente si N es mayor que M.
-lt	test termina correctamente si N es menor que M.
-ge	test termina correctamente si N es mayor o igual que M.
-le	test termina correctamente si N es menor o igual que M.

primitivas para expresiones numéricas

Expresiones alfanuméricas:

Las formas generales de las expresiones alfanuméricas son:

```
  <primitiva> S
S <primitiva> R
```

donde S y R representan cadenas alfanuméricas.

S = R	test termina correctamente si S y R son iguales.
S != R	test termina correctamente si S y R son distintas.
-z S	test termina correctamente si la longitud de la cadena S es 0.
-n S	test termina correctamente si la longitud de la cadena S es distinta de 0.

primitivas para expresiones alfanuméricas

Ejemplo:

```
#!/bin/bash
# empiezan_por
LISTADO=$(ls $1* 2> /dev/null)
if [ -z "$LISTADO" ]
    then
          echo “No existen ficheros que empiece por $1”
    else
          echo “Existe algún fichero que empieza por $1”
fi
```

Expresiones con ficheros:

La forma general de las expresiones con ficheros es:

```
<primitiva> F
```

donde F representa a un fichero.

-s	test termina correctamente si el fichero existe y no está vacío.
-f	test termina correctamente si el fichero existe y es regular.
-d	test termina correctamente si el fichero existe y es un directorio.
-r	test termina correctamente si el fichero existe y tiene permiso de lectura
-w	test termina correctamente si el fichero existe y tiene permiso de escritura
-x	test termina correctamente si el fichero existe y tiene permiso de ejecución

primitivas para expresiones con ficheros

Expresiones compuestas:

El comando `test` contempla la evaluación de expresiones compuestas ensambladas a partir del uso de los operadores lógicos “and” y “or”, denotados por `-a` y `-o`, respectivamente.

Ejemplo:

```
#!/bin/bash
# append
if test -w $2 -a -r $1
    then
          cat $1 >> $2
    else
          echo "No se puede añadir $1 a $2"
fi
```

1.4.5.- La estructura iterativa `for`.

A diferencia de lo que típicamente ocurre con *estructuras iterativas* de nombre *for*, en lenguajes de programación *imperativos estructurados* convencionales, en los que dicha estructura suele expresar iteración desde un valor inicial hasta otro final, actualizando la variable de control de la estructura en cada iteración según un paso especificado, en el caso de los lenguajes de programación de tipo "intérpretes de comandos de tipo UNIX", la *sentencia* `for` expresa iteración sobre una lista de valores, asignando, en cada iteración, el valor correspondiente, de dicha lista, a la variable de control asociada a la estructura de control. Su sintaxis es la siguiente:

```
for variable in lista_de_valores
do
   comandos
done
```

Ejemplo: en el siguiente shell script, la estructura iterativa *for* se usa para visualizar diferentes partes del contenido de un directorio.

```
#!/bin/bash
# lista
for i in /usr/bin/a* /usr/bin/b* /usr/bin/c*
do
   ls $i
done
```

Diferentes bucles `for` difieren, fundamentalmente, en la forma en que se proporciona la lista de valores. Algunas posibilidades, para dicha lista, serían:

La lista de argumentos:

La lista de valores, sobre los que itera la estructura `for`, puede ser la lista de argumentos proporcionados en la invocación del shell script.

Ejemplo:

```
#!/bin/bash
# ejemplo_for
for i in $*
do
    ls /usr/bin/$i*
done
```

La lista de nombres de ficheros del directorio de trabajo por defecto:

La lista de valores, sobre los que itera la estructura `for`, puede ser las lista de nombres de los ficheros nombrados en el directorio de trabajo por defecto.

Ejemplo:

```
#!/bin/bash
# ejemplo_for
for k in *
do
    cp $k $k.bak
done
```

La lista de valores resultantes de la ejecución de un comando:

La lista de valores, sobre los que itera la estructura `for`, puede ser la lista de valores resultante de la ejecución de un comando.

Ejemplo:

```
#!/bin/bash
# ejemplo_for
for j in $(ls)
do
    echo $j
done
```

Ejercicios de autoevaluación

1 Escribe un shell script que compare si dos cadenas, proporcionadas como argumentos, son iguales.

2 Escribe un shell script llamado *copia_lineas* que copie las m primeras líneas de un fichero (pasado como primer argumento) al final de otro fichero (pasado como segundo argumento), siendo "m" el tercer argumento a suministrar. Por ejemplo, si se desea copiar las 5 primeras líneas del fichero *fich1* al final del fichero *fich2* se debería introducir la orden: *$ copia_lineas fich1 fich2 5.*

Indicaciones: usar el comando *head*

3 Modifica el shell script *copia_lineas* para que compruebe que el número de argumentos suministrado es exáctamente tres, mostrando un mensaje de error en caso contrario.

4 Modifica el shell script *copia_lineas* para que compruebe, que el primer argumento se corresponde con un fichero regular existente, y que el fichero pasado como primer argumento contiene, al menos, *m* líneas. Recuerda que el valor de *m* se suministra, como tercer argumento, al shell script.

Indicaciones: Usar el comando *wc*

5 Escribe un shell script tal que dado un fichero (pasado como primer argumento) y una palabra (pasada como segundo argumento), muestre por pantalla las líneas de ese fichero que contengan la palabra buscada.

Indicaciones: usar el comando *grep*

6 Escribe un fichero llamado *amigos* que contenga 10 nombres de personas que conozcas (nombres y apellidos). Escribe otro fichero llamado *telefonos* que contenga los telefonos de esos amigos (nombre y teléfono), sin incluir los apellidos, en distinto orden al del listado de *amigos*. A continuación, escribe un shell script que:

a. Ordene de forma alfabética los ficheros *amigos* y *teléfonos*
b. Genere un nuevo fichero *agenda* con la información de los amigos y teléfonos, donde en cada línea se tenga: Nombre Apellidos Teléfono
c. Genere un nuevo fichero llamado *listado* que contenga únicamente los apellidos de los amigos.

Indicaciones: usar los comandos *sort*, *paste* y *cut*

7 En el directorio /proc existen varios subdirectorios (cuyo nombre es un número) que representan a los procesos que existen en el sistema. Dicho número se corresponde con el PID de esos procesos. En cada subdirectorio existen diversos ficheros (state, status, cmdline, etc.) en los que se guarda información relativa al proceso con PID igual al nombre del subdirectorio.

Escribe un shell script llamado *informe* tal que, dado un número de PID de un proceso, como argumento, muestre por pantalla el estado de dicho proceso. Dicha información se encuentra en la línea State del fichero *status*.

Indicaciones: usar el comando *awk*

8 Modifica el shell script *informe* para que muestre por pantalla una información similar a la asiguiente (sustituyendo los puntos suspensivos por el valor adecuado).

```
El proceso con PID=.... está en estado..... y ejecutando la orden .......
```

Indicaciones: La información acerca del comando deberá extraerla del fichero de nombre *cmdline* que se encuentra en el directorio /proc/PIDproceso. Usar el comando *tr* para reemplazar los caracteres nulos, de la cadena almacenada en *cmdline*, por blancos, para así obtener el nombre de la orden y sus argumentos en un formato legible.

9 Modifica el shell script *informe* para que compruebe que se introduce un único parámetro y que se trata de un proceso existente en el sistema.

10 Escribe un shell script llamado *sistema* que muestre el estado y la orden de ejecución de los procesos activos del sistema, invocando al shell script *informe.*

Soluciones a los ejercicios de autoevaluación

1 Escribe un shell script que compare si dos cadenas, proporcionadas como parámetro, son iguales.

```
#!/bin/bash
# cadenas_iguales
if test $1 = $2
   then
       echo Las cadenas son iguales
   else
       echo Las cadenas son distintas
fi
exit 0
```

2 Escribe un shell script llamado *copia_lineas* que copie las m primeras líneas de un fichero (pasado como primer argumento) al final de otro fichero (pasado como segundo argumento), siendo “m” el tercer argumento a suministrar.

```
#!/bin/bash
# copia_lineas
head -$3 $1 >> $2
```

3 Modifica el shell script *copia_lineas* para que compruebe que el número de argumentos suministrado es exáctamente tres, mostrando un mensaje de error en caso contrario.

```
#!/bin/bash
#copia_lineas
if [ $# -ne 3 ]
   then
       echo Número de argumentos incorrecto
       echo Uso: copia_lineas origen destino m
       exit 1
   else
       head -$3 $1 >> $2
fi
```

4 Modifica el el shell script *copia_lineas* para que compruebe, que el primer argumento se corresponde con un fichero regular existente, y que el fichero pasado como primer argumento contiene, al menos, *m* líneas. Recuerda que el valor de *m* se suministra, como tercer argumento, al shell script.

```
#!/bin/bash
# copia_lineas
if [ -f $1]
   then
      num_lineas=$(cat $1 | wc –l )
      if [ $num_lineas –lt $3 ]
         then
            echo “El fichero $1 tiene menos de $3 lineas”
            exit 1
         else
           head -$3 $1 >> $2
      fi
   else
      echo “El fichero $1 no es un fichero regular”
      exit 1
fi
```

5 Escribe un shell script tal que dado un fichero (pasado como primer argumento) y una palabra (pasada como segundo argumento), muestre por pantalla las líneas de ese fichero que contengan la palabra buscada.

```
#!/bin/bash
# busqueda
grep $2 $1
```

6 Escribe un shell script que:

a. Ordene de forma alfabética los ficheros *amigos* y *teléfonos*

b. Genere un nuevo fichero *agenda* con la información de los amigos y teléfonos

c. Genere un nuevo fichero llamado *listado* que contenga únicamente los apellidos de los amigos.

```
#!/bin/bash
# ejercicio_7
sort amigos > amigos_ord
sort teléfonos > telefonos_ord
paste amigos_ord telefonos_ord | cut –d" " –f 1,2,4 > agenda
cut –d" " –f 2 amigos_ord > listado
```

7 Escribe un shell script llamado *informe* que, dado un número de PID de un proceso como argumento, muestre por pantalla el estado de dicho proceso

```
#!/bin/bash
# informe
awk '/State/ {print $3}' /proc/$1/status
```

8 Modifica el shell script *informe* para que muestre por pantalla una información similar a la asiguiente (sustituyendo los puntos suspensivos por el valor adecuado).

```
El proceso con PID=.... está en estado..... y ejecutando la orden .......
```

Indicaciones: La información acerca del comando deberá extraerla del fichero de nombre *cmdline* que se encuentra en el directorio /proc/PIDproceso. Usar el comando *tr* para reemplazar los caracteres nulos, de la cadena almacenada en *cmdline*, por blancos, para así obtener el nombre de la orden y sus argumentos en un formato legibleel shell script *informe* para que muestre por pantalla una información similar a la siguiente…

```
#!/bin/bash
# informe
estado=$(awk '/State/ {print $3}' /proc/$1/status)
orden=$(cat /proc/$1/cmdline | tr "\000" " " )
echo "El proceso con PID=$1 esta en estado $estado"
echo " y ejecutando la orden $orden"
```

9 Modifica el shell script *informe* para que compruebe que se introduce un único parámetro y que se trata de un proceso existente en el sistema.

```
#!/bin/bash
# informe
if [ $# -ne 1 ]
  then
    echo Se requiere un argumento. Uso: informe PID
    exit 1
  else
    if [ -d /proc/$1 ]
      then
        estado=$(awk '/State/ {print $3}' /proc/$1/status)
        orden=$(cat /proc/$1/cmdline | tr "\000" " " )
        echo "El proceso con PID=$1 esta en estado $estado"
        echo "y ejecutando la orden $orden"
        exit 0
      else
        echo El PID no se corresponde con un proceso
        exit 1
    fi
fi
```

10 Escribe un shell script llamado *sistema* que muestre el estado y la orden de ejecución de los procesos activos del sistema, invocando al shell script *informe.*

```
#!/bin/bash
# sistema
for i in /proc/[1-9]*
do
   pid=$(echo $i | tr /proc/ "\000")
   ./informe $pid
Done
```

Ejercicios para evaluación

A continuación se propone un conjunto de ejercicios que el alumno podrá realizar y, en su caso, entregar al profesor.

Algunos ejercicios a realizar dependen del número de DNI del alumno. Dicho número consta de 8 dígitos. Anotarlos, como se indica en el ejemplo, y adjuntarlos a la documentación presentada sobre las prácticas.

Ejemplo:

Digito	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
DNI	4	4	8	6	8	5	2	7

DNI alumno:

Digito	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
DNI								

Cada vez que, en un ejercicio, se indique *dígito_i* (siendo *i* un número entre 1 y 8), la expresión deberá ser sustituida por el valor consignado en la casilla, del DNI del alumno, correspondiente al dígito i. Así, según los datos mostrados, en el ejemplo de DNI, si en un ejercicio se pide crear *dígito_3* directorios distintos, se deberá crear 5 directorios distintos (pues en el DNI del ejemplo tenemos el valor 5 correspondiendo al dígito 3). Si, al sustituir *dígito_i*, obtenemos 0 tomar 1 en su lugar.

Recordatorio: En el directorio */proc* existen una serie de subdirectorios cuyo nombre es un número. Ese número se corresponde con el PID de los procesos que existen en el sistema. Cada uno de esos subdirectorios contiene una serie de ficheros con diversos atributos e información sobre el proceso correspondiente. Por ejemplo, en el fichero *status* se guarda, entre otros datos, el nombre del proceso, su estado, su PID, su PPID, su propietario (UID) y el grupo al que pertenece (GID).

Trabajo previo 1: Crear un directorio, llamado *segunda_practica*, y crear, y nombrar en él, *digito_6* + 1 ficheros, con información variada.

Trabajo previo 2: Crear un fichero, llamado *mis_asignaturas*, que contenga los datos de las asignaturas que cursas durante el primer cuatrimestre. Como primera línea del fichero, debes incluir:

```
CODIGO     ASIGNATURA      PROFESOR     DIA     HORA_COMIENZO
```

Cada línea del fichero debe tener un formato similar al siguiente (observar que se emplean dos puntos como separadores entre campos):

```
SO1 : Sistemas Operativos : Estefanía Argente : Miercoles : 9:30
```

EJERCICIOS

1. Escribir un *shell script* tal que, dados varios ficheros, como argumentos, muestre, para cada uno de ellos, su nombre y el número de líneas que contiene. Probarlo, con los ficheros creados en el *trabajo previo 1*, anotando los resultados obtenidos.

2. Escribir un *shell script* tal que, dados *digitio_3* +1 ficheros, como argumentos, compruebe que, efectivamente, se han proporcionado *digitio_3* +1 argumentos, y genere un nuevo fichero, llamado *total*, con el contenido de todos de los ficheros concatenado.

3. Escribir un *shell script* tal que, dados *digitio_5* +1 ficheros, como argumentos, compruebe que, efectivamente, se han proporcionado *digitio_5* +1 argumentos, compruebe que cada uno de los ficheros proporcionados existe, y genere un fichero, llamado *total*, con el contenido de todos los ficheros concatenado.

4. Escribir un *shell script* tal que, dadas *digitio_1* +1 palabras, como argumentos, compruebe que, efectivamente, se han proporcionado *digitio_1* +1 argumentos, y determine si todas las palabras proporcionadas son iguales entre sí.

5. Escribir un *shell script* tal que, dadas *digitio_7* +1 palabras, como argumentos, compruebe que, efectivamente, se han proporcionado *digitio_7* +1 argumentos, y muestre las palabras proporcionadas ordenadas alfabéticamente.

6. Escribir un *shell script* que liste aquellos ficheros que, nombrados en */usr/bin*, tienen nombre que empieza por alguna de las letras correspondientes a las iniciales de tu nombre y apellidos. Por ejemplo, un alumno llamado “Enrique García” deberá listar los archivos cuyo nombre empieza por la letra “e” o por la letra “g”.

7. Escribir un *shell script* tal que, a partir del fichero creado en el *trabajo previo 2*, genere un nuevo fichero, llamado *midia*, con el código y el nombre de las asignaturas a las que asistes el *digito_2* día de la semana. Si *digito_2* es mayor que 5, o bien no asistes a ninguna asignatura ese día, toma las asignaturas que cursas los miércoles.

8. Diseñar una orden que muestre los subdirectorios de */proc* cuyo nombre empieza por un dígito comprendido entre *digito_1* y *digito_3*. Seleccionar uno de esos subdirectorios y anotar el contenido del fichero *status,* nombrado en él.

9. Escribir un *shell script*, llamado *miproceso*, tal que, dado el PID de un proceso, como argumento, compruebe que se ha proporcionado un único parámetro, compruebe que dicho valor corresponde a un proceso que existe, asigne a una variable el estado y a otra el PPID del proceso, y muestre dichos resultados siguiendo un formato similar al sugerido en el ejemplo:

```
$ miproceso 2534
Proceso PID = 2534, estado (sleeping), cuyo padre es PID = 1239
```

10. Escribe un *shell script*, de nombre *sistema*, que muestre todos los procesos del sistema que comiencen entre *digito_1* y *digito_3* y, para cada uno de ellos, muestre su estado y su PPID.