Práctica 3

Programación en Bash

Contenidos

3.1. Introducción 3-2
3.2. Un primer *script* 3-2
3.3. Variables y argumentos 3-3
3.4. Estructuras de control 3-5
3.4.1. if...else...elif...fi 3-6
3.4.2. for...in...do...done 3-9
3.4.2. while...do...done 3-10
3.5. Otras herramientas 3-12
3.6. Practiquemos un poco 3-13

3.1. Introducción

El intérprete `bash`, además de la ejecución de órdenes Linux y de programas del sistema de ficheros, sirve de *intérprete* a un pequeño lenguaje de programación que incluye un conjunto de estructuras de control de flujo. En él podemos definir variables, leer argumentos, imprimir, utilizar bucles y condicionales, etc. La programación en `bash` nos permite crear programas *scripts* para automatizar procesos complejos que integran multitud de órdenes, otros scripts, y programas en general, en muchos casos, escritos en diferentes lenguajes de programación. En esta práctica, pretendemos introducirte en la escritura de programas para intérpretes de órdenes y, en particular, para `bash`.

3.2. Un primer script

Vamos a empezar escribiendo un primer programa sencillo, que cuando lo ejecutemos nos dirá en qué directorio estamos y la fecha. Para ello, crearemos con un editor cualquiera (como XEMACS) un archivo de nombre `fecha.sh`[1] y escribiremos en él las siguientes líneas:

```
#!/bin/bash
# fecha.sh: Ejemplo sencillo de programacion en bash
echo 'Estas en el directorio:'
pwd
echo 'y la fecha es:'
date
```

Analicemos con detalle este primer programa. La primera línea le dice a Linux que utilice el intérprete `bash` para ejecutar este *script*. Suponemos que el programa `bash` se encuentra en el directorio `/bin`, aunque ésto es algo que deberías comprobar antes en tu propio ordenador.

La segunda línea no es más que un comentario. Podemos incluir comentarios en cualquier punto del *script* con una almohadilla (`#`). Esto excluye la primera línea del *script*, que es la que le indica al sistema que utilice `bash`, aunque empiece con `#`.

En la tercera línea se utiliza el programa `echo`, que envía una cadena a la salida estándar (o imprime el valor de una variable, como veremos más adelante). Por lo tanto, esta orden únicamente imprimirá en pantalla el texto `Estas en el directorio`. En la cuarta línea se ejecuta la orden `pwd` para averiguar cuál es nuestro directorio de trabajo, que es aquel desde el que ejecutamos el *script*. Las dos últimas líneas del programa son ahora más sencillas de entender.

Ahora ya estamos casi a punto de poder ejecutar este *script*. Pero antes debemos darle permiso de ejecución (por ahora, es sólo un archivo de texto). Para ello, podemos utilizar la orden `chmod`, que ya estudiamos en la práctica anterior:

```
$ chmod u+x fecha.sh
```

Por último, podemos ya proceder a ejecutar el programa. No te olvides, de referirte al programa como `./fecha.sh`, ya que puede que no exista el directorio '`.`' en tu `PATH`. A continuación, se muestra el resultado de la ejecución del programa:

```
$ ./fecha.sh
Estas en el directorio:
/users/alumni/?/al0xxxxx
y la fecha es:
jue feb 27 17:55:01 CET 2003
```

[1] Es habitual utilizar la extensión `.sh` para los *script* de *shell*.

Además de mostrar el resultado final de la ejecución del *script*, podemos ejecutarlo paso a paso (`bash` es un intérprete, ¿recuerdas?) ejecutando `bash -x programa.sh`. Esto puede ser muy útil si queremos analizar la ejecución de un *script* con detalle para encontrar algún error. En ese caso, se muestra en pantalla la línea de `bash` con un signo más y a continuación el resultado de su ejecución. En el caso del programa anterior, la ejecución de `bash -x fecha.sh` daría lugar a la siguiente salida:

```
$ bash -x fecha.sh
+ echo 'Estas en el directorio:'
Estas en el directorio:
+ pwd
/users/alumni/?/al0xxxxx
+ echo 'y la fecha es:'
y la fecha es:
+ date
jue feb 27 17:55:01 CET 2003
```

Ejercicios

► **1** Escribe con XEMACS el programa descrito al principio de la práctica y guárdalo con el nombre `fecha.sh`. Dale permiso de ejecución y ejecútalo desde un terminal con `./fecha.sh`.

► **2** Repite la ejecución con la orden `bash -x fecha.sh`.

3.3. Variables y argumentos

Al igual que con cualquier otro lenguaje de programación, en *scripts* de `bash` podemos utilizar variables para almacenar valores. La orden de asignación es:

```
nombre=valor
```

Debemos hacer ahora una serie de consideraciones particulares de la asignación de variables en `bash`:

- En `bash`, las variables se consideran cadenas de caracteres. Por lo tanto, no existen los *tipos* de variables entero, real, etc.
- Para utilizar una variable debemos anteceder a su nombre el carácter `$`. Así, si definimos la variable `var=5`, podemos mostrar su valor por pantalla con la orden `echo $var`.
- Evita insertar algún espacio antes o después del signo '=' de la orden de asignación. Es más que probable que ocurra un error de ejecución.

Además de las variables definidas por el programador, existen una serie de variables especiales para referirse a los argumentos recibidos por el *script*. Así, `$1` hace referencia al primer argumento, `$2` al segundo, y así sucesivamente. `$0` es el propio nombre del programa, `$#` el número de argumentos que recibe el *script* y `$*` el conjunto completo de argumentos. Todas las variables definidas en una sesión de `bash` son locales a dicha sesión, por lo que al terminar la ejecución de un *script* se pierden las variables definidas.

Ejercicios

► **3** Fíjate en el siguiente programa, de nombre `var.sh`, y trata de entender su funcionamiento.

```
#!/bin/bash
# var.sh: Programa para entender el uso de variables.

cd
DIR=prueba
mkdir $DIR
cd $DIR
echo 'El nombre del programa es' $0 >foo
echo 'Ha recibido' $# 'argumentos' >>foo
echo 'El primer argumento es' $1 >>foo
echo 'El segundo argumento es' $2 >>foo
echo 'El conjunto de argumentos es' $* >>foo
```

▶ 4 Escribe el programa anterior, dale permisos de ejecución y ejecútalo desde una terminal con la siguiente orden: `./var.sh A B C`. Comprueba que se ha creado un directorio `prueba` y que en su interior existe un fichero `foo` con el resultado que esperabas.

▶ 5 Ejecuta en el terminal la orden `echo $DIR`. ¿No deberías obtener como resultado `prueba`?

Además de ejecutar programas desde el *script* y enviar el resultado a la salida estándar, podemos guardar la salida como una variable. Para ello debemos incluir la orden entre comillas invertidas (‘) o dentro de `$()`. Por ejemplo, la orden

```
FICH_C=‘ls *.c‘
```

almacena en la variable `FICH_C` la salida de la orden `ls *.c`.

Pero las comillas invertidas no son las únicas de las que disponemos en `bash`. También podemos utilizar comillas dobles (") y apóstrofes (’): ambos sirven para definir cadenas de caracteres. Por ejemplo, podemos definir un directorio de nombre `Mi directorio` (fíjate en que hay un espacio) con la orden `mkdir 'Mi directorio'`, aunque también podemos utilizar comillas dobles (`mkdir "Mi directorio"`). Para acceder al directorio con `cd` deberíamos incluir de nuevo el nombre entre apóstrofes o comillas dobles (p.e., `cd 'Mi directorio'`) [2].

Entonces, ¿qué diferencia hay entre las comillas dobles y los apóstrofes? La respuesta es que las comillas dobles *permiten la expansión de variables, mientras que los apóstrofes no.* Lo entenderás mejor con el siguiente *script*:

```
#!/bin/bash
# comillas.sh
x=5
echo "Utilizando comillas dobles, el valor de x es $x"
echo 'Utilizando apostrofes, el valor de x es $x'
```

cuya salida al ejecutarse es:

```
$ comillas.sh
Utilizando comillas dobles, el valor de x es 5
Utilizando apostrofes, el valor de x es $x
```

Ejercicios

▶ 6 Bájate el archivo `heapsort.tar` de la página *web* de prácticas y desempaquétalo en `/users/alumni/?/al0xxxxx`.

[2] También podemos representar el espacio en blanco con los caracteres `\` (barra invertida + espacio). Así, la orden `cd Mi\ directorio` sería igualmente válida.

▶ 7 Escribe y ejecuta este *script* en el directorio `heapsort`, tratando de entender su funcionamiento. ¿Para qué se redirecciona la salida de error?

```
#!/bin/bash
# cuenta.sh : Cuenta el numero de ficheros .c y .h en un directorio

FICH_C=‘ls *.c 2>error | wc -w‘
FICH_H=‘ls *.h 2>error | wc -w‘
echo ’Número de ficheros .c =’ $FICH_C
echo ’Número de ficheros .h =’ $FICH_H
```

▶ 8 Escribe y ejecuta el *script* `comillas.sh` mostrado anteriormente para comprender la diferencia en el uso de comillas dobles y apóstrofes.

3.4. Estructuras de control

Al igual que en otros lenguajes de programación, en `bash` se dispone de estructuras para controlar el flujo del programa. Algunos ejemplos de estas estructuras son:

- `if/else`
- `for`
- `case`
- `select`
- `while`
- `until`

Estudiar con detalle todas estas estructuras está más allá de los objetivos de este curso, por lo que nos limitaremos a tratar sólo algunas de ellas.

3.4.1. if...else...elif...fi

Una de las estructuras más comunes es la condicional `if/else`, que nos va a permitir realizar acciones determinadas siempre que se cumplan una serie de condiciones. La sintaxis de esta estructura es la siguiente:

```
if condición_1
then
            órdenes_1
elif condición_2
then
            órdenes_2
.................
else
            órdenes_n
fi
```

Existen múltiples alternativas a la hora de establecer una condición para el control de flujo. Una de ellas es la verificación del código de error devuelto por una orden o programa al finalizar su ejecución. Por definición el código de error de una orden que ha terminado de forma satisfactoria es 0, lo cuál es considerado por el intérprete como que la condición se cumple. En caso contrario, la orden devolverá un código distinto de 0 y el intérprete considerará que la condición no se cumple. En un *script*, podemos obtener el código de error de la última orden ejecutada con `$?`.

Ejercicios

▶ **9** Prueba a ejecutar una orden correcta en un terminal (p.e., `pwd`) y después `echo $?`. Repite esta última acción tras una orden errónea (como `cat pru`).

Sin embargo, la forma más habitual de establecer condiciones es el uso de la instrucción `test`. Esta instrucción acepta una serie de opciones que son de gran utilidad a la hora de comprobar existencia de ficheros y directorios, comprobación de permisos, etc. Por ejemplo, la orden

```
test -f /etc/passwd
```

comprueba si existe un fichero `passwd` en el directorio `/etc`, devolviendo un código verdadero o falso. En la siguiente tabla se indican otras opciones que puede recibir la orden `test`:

Opción	Acción
`-e nombre`	Comprueba si `nombre` existe
`-d nombre`	Comprueba si `nombre` es un directorio
`-f nombre`	Comprueba si `nombre` es un fichero
`-L nombre`	Comprueba si `nombre` es un enlace simbólico
`-r nombre`	Comprueba si `nombre` posee permiso de lectura
`-w nombre`	Comprueba si `nombre` posee permiso de escritura
`-x nombre`	Comprueba si `nombre` posee permiso de ejecución
`fichero1 -ot fichero2`	Cierto si `fichero1` es más antiguo que `fichero2`
`fichero1 -nt fichero2`	Cierto si `fichero1` es más reciente que `fichero2`

Por ejemplo, el siguiente *script* comprueba si el argumento que recibe es un directorio. En caso contrario, comprueba si el archivo existe o no.

```
#!/bin/bash
# dir.sh : Comprueba si el argumento es un directorio

if test -d $1
then
      echo $1 'es un directorio'
elif test -e $1
then
      echo $1 'no es un directorio'
else
      echo $1 'no existe'
fi
```

La orden `test` puede utilizarse con otra sintaxis, en la que se incluye entre corchetes las opciones y argumentos del programa y no se utiliza la palabra `test`. Esta se usa muy habitualmente en la programación de *shell*. De esta forma, el anterior *script* se escribiría como se indica a continuación:

```
#!/bin/bash
# dir.sh : Comprueba si el argumento es un directorio

if [ -d $1 ] ; then
    echo $1 'es un directorio'
elif [ -e $1 ] ; then
    echo $1 'no es un directorio'
else
    echo $1 'no existe'
fi
```

Fíjate en que debe existir un espacio en blanco separando los corchetes de la expresión en su interior. En este caso, además, hemos incluído la orden `then` en la misma línea que la orden `if` y `elif`, aunque para ello debemos separar las órdenes con punto y coma (;).

Por último, si queremos negar una condición disponemos del operador de negación `!`. Así si queremos comprobar que el primer argumento que recibe el *script* no es un directorio, escribiríamos:

```
if [ ! -d $1 ]
```

Ejercicios

▶ **10** Escribe y ejecuta el *script* `dir.sh` (dándole previamente permiso de ejecución) de forma que obtengas los tres mensajes.

▶ **11** Escribe y ejecuta el siguiente *script*, que realiza una copia de un archivo comprobando previamente que exista el original y no la copia. Fíjate en que se han anidado dos estructuras `if`.

```
#!/bin/bash
# copy.sh : copia un archivo

if [ -f $1 ] ; then
    if [ -e $2 ] ; then
       echo $2 'ya existe.'
    else
       cp $1 $2
    fi
elif [ ! -e $1 ] ; then
    echo $1 'no existe.'
else
    echo $1 'no es un archivo regular.'
fi
```

El *script* del último ejercicio funcionará correctamente siempre que el usuario le pase dos argumentos al mismo, es decir, que lo ejecute como

```
copy.sh nombre1 nombre2
```

En caso contrario, el resultado no será el esperado. Sería interesante comprobar antes de realizar la copia de archivos que el *script* ha recibido dos argumentos. Para ello, debemos hablar previamente de los operadores de comparación, que ya conocerás de otros lenguajes de programación. En el caso de `bash`, los operadores de comparación son los que se muestran en la siguiente tabla:

Operador	Comparación
`-lt`	Menor que
`-le`	Menor o igual que
`-eq`	Igual que
`-gt`	Mayor que
`-ge`	Mayor o igual que
`-ne`	Distinto que

Teniendo en cuenta estos operadores de comparación, se muestra a continuación el *script* `copy.sh` modificado para que imprima un mensaje de error y termine si el número de argumentos (`$#`, ¿recuerdas?) es distinto de dos. Fíjate en que la orden `exit` termina el *script* (el argumento 1 devuelve un código de error).

```
#!/bin/bash
# copy.sh : copia un archivo

if [ $# -ne 2 ]; then
    echo 'Este programa requiere dos argumentos'
    exit 1
fi

if [ -f $1 ] ; then
    if [ -e $2 ] ; then
       echo $2 'ya existe.'
    else
       cp $1 $2
    fi
elif [ ! -e $1 ] ; then
    echo $1 'no existe.'
else
    echo $1 'no es un archivo regular.'
fi
```

Ejercicios

▶ **12** Realiza la modificación indicada en el *script* `copy.sh` y ejecútalo con un sólo argumento.

3.4.2. for...in...do...done

Hasta ahora hemos utilizado los *scripts* para procesar ficheros individuales, pero ¿y si queremos repetir la misma operación sobre un conjunto de ficheros? En ese caso resulta muy útil el uso de bucles, y en particular el bucle `for`, cuya sintaxis es la siguiente:

`for` *var* `in` *lista*
`do`
órdenes
`done`

donde *var* es la variable que toma en cada iteración del bucle uno de los elementos de *lista*. La lista puede indicarse de forma explícita, separando los elementos por espacios,

```
for i in coche casa hipoteca
do
    echo 'Tengo' $i
done
```

obteniéndola con un listado de archivos (fíjate en que no es necesario utilizar `ls`)

```
for i in *.txt
do
    echo $i 'es un archivo de texto.'
done
```

o ejecutando el bucle para cada argumento proporcionado al programa

```
for i in $*
do
    echo $i
done
```

Veamos un ejemplo de aplicación de un bucle `for`. En ocasiones, es necesario renombrar un conjunto de archivos que cumplen una determinada condición. En este ejemplo vamos a escribir un *script* que cambia la extensión de un tipo determinado de archivos. Este *script* (que llamaremos `renombra.sh`) recibirá dos argumentos: la extensión de los archivos que queremos cambiar, y su nueva extensión. En este ejemplo, utilizaremos una orden Linux que veremos en la siguiente sección: la orden `basename`. Por el momento, basta con que sepas que

```
basename archivo.ext ext
```

nos devuelve 'archivo.', es decir, elimina la extensión que recibe como segundo argumento. Fíjate ahora en el *script* `renombra.sh`, y trata de entender su funcionamiento.

```
#!/bin/bash
# renombra.sh : cambia la extensión de un tipo de ficheros.
# las extensiones que se pasan como parámetros no incluirán el '.'

if [ $# -ne 2 ]; then
    echo 'Este programa requiere dos argumentos'
    exit 1
fi

for i in *.$1
do
    nombre=`basename $i $1`
    mv $nombre$1 $nombre$2
done
```

Ejercicios

▶ **13** Escribe el *script* `renombra.sh` y dale permiso de ejecución. Utilízalo para cambiar la extensión de tus archivos `.sh` a `.shell`. Después devuélveles su extensión original.

3.4.3. while...do...done

La estructura `while` repite un conjunto de instrucciones mientras se cumple una condición, a diferencia del `for` que itera sobre los elementos de una lista. Cada iteración comienza con la evaluación de la condición. Si es verdadera, se ejecutan las instrucciones del cuerpo del `while`, entre las cuales debe existir alguna que pueda modificar la evaluación de la condición. Si la condición es falsa o se ejecuta un `break` interno, el `while` termina. Su sintaxis es:

```
while [ var op valor ]
do
        órdenes
done
```

donde *var* es la variable de control de flujo y `op` es un operador de comparación. Veamos un *script* sencillo que hemos denominado `lista.sh`:

```
#!/bin/bash
# lista.sh : Lista archivos de extensiones dadas por el usuario
# Termina cuando se escribe la extensión "fin"

read -p "Teclea una extensión: "  ext
while [ $ext != "fin" ]
do
    ls *.$ext
    read -p "Teclea otra extensión: "  ext
done
echo "Has finalizado el programa"
```

El ejemplo anterior lee repetidamente cadenas de caracteres, y lista aquellos ficheros que la tengan como extensión. Pero, ¿qué sucede si no existen ficheros de esa extensión? La orden `ls` imprimirá un mensaje de error "poco estético" por la salida estándar de error. La siguiente versión trata adecuadamente esta posibilidad:

```
#!/bin/bash
# lista.sh : Lista archivos de extensiones dadas por el usuario
# Termina cuando se escribe la extensión "fin"

read -p "Teclea una extensión: "  ext
while [ $ext != "fin" ]
do
   lista=$(ls *.$ext 2>/dev/null)
   cant=$(echo $lista | wc -w)
   if [ $cant -gt 0 ]
   then
      echo "Los ficheros .$ext son:"  $lista
   else
      echo "No hay ficheros con la extensión \".$ext\""
   fi
   read -p "Teclea otra extensión: "  ext
done
echo "Has finalizado el programa"
```

Las órdenes que aparecen en esta nueva versión fueron vistas en la práctica anterior. Si tienes alguna duda, consulta el material "Órdenes Linux".

Ejercicios

▶ **14** Escribe el script `lista.sh` (la segunda versión) y comprueba que funciona correctamente.

Otra construcción muy frecuente con el `while` es aquella en la que la variable de control contiene valores numéricos. Veamos este segundo ejemplo:

```
#!/bin/bash
# cuenta.sh : Imprime los números de 0 a un valor MAX pasado
# como parámetro

max=$1
cuenta=0
while [ $cuenta -lt $max ]; do
    echo $cuenta
    cuenta=$[cuenta+1]
done
```

Observa en la línea en la que se actualiza `cuenta`, una de las sintaxis correctas para evaluar expresiones aritméticas en `bash`. Una segunda forma válida es `cuenta=$((cuenta+1))` y, una tercera, es a través de la orden `let cuenta=cuenta+1`. Más adelante volverémos sobre esta última.

Ejercicios

▶ **15** Escribe el script `cuenta.sh` y verifica que funciona como esperas.

▶ **16** Modifica el script `cuenta.sh` de forma que reciba 3 parámetros (que deberás comprobar), siendo el primero de ellos un texto que usaremos como nombre base, y los dos restantes dos números, el primero menor que el segundo. Dados los siguientes datos '`result 3 6`', el nuevo `cuenta.sh` deberá crear los subdirectorios `result3`, `result4`, y `result5`. Deberás realizar las comprobaciones necesarias para no intentar crear un subdirectorio que ya exista.

3.5. Otras herramientas

Ya hemos comentado que en un *script* podemos utilizar cualquiera de las órdenes Linux estudiadas en la unidad anterior. Sin embargo, existen una serie de programas que no hemos visto hasta ahora y nos van a facilitar la escritura de programas en `bash`. Vamos a comentar ahora alguno de ellos.

- **basename**: Ya utilizamos esta orden en un ejemplo anterior. Su función es proporcionar el nombre llano del archivo que recibe como argumento, eliminando la ruta hasta el mismo (si existe) y su extensión (si se indica como segundo argumento). A continuación aparecen un par de ejemplos de su utilización:

```
$ basename /etc/hosts.allow
hosts.allow
$ basename /etc/hosts.allow allow
hosts.
```

- **dirname**: Orden complementaria a `basename`, cuando recibe el nombre completo de un archivo (ruta incluída) devuelve sólo la ruta.

```
$ dirname /etc/hosts.allow
/etc
```

- **read**: Lee una línea de entrada y asigna las palabras a las variables que sigan al `read`. Puede especificarse un mensaje que aparecerá en el terminal antes de introducir la entrada con la orden `-p mensaje`. El siguiente ejemplo lee dos palabras y las guarda en las variables NOMBRE y APELLIDO:

```
read -p "Teclee nombre y apellido: "  NOMBRE APELLIDO
```

- **let**: Esta orden se utiliza mucho en *scripts* para realizar operaciones aritméticas con variables numéricas. Fíjate en el siguiente ejemplo, en el que se muestra el *script* `lsdirs.sh`, que realiza un listado de los directorios del directorio actual y cuenta los que hay con ayuda de una variable contador `num`:

```
#!/bin/bash
# lsdirs.sh : Lista los directorios del directorio actual.

num=0
for i in *
do
   if [ -d $i ]; then
      echo $i
      let num=num+1
   fi
done
echo 'Total:' $num 'directorio(s)'
```

Ejercicios

▶ 17 Ejecuta las órdenes que se indican en los ejemplos anteriores, y escribe y ejecuta el *script* `lsdirs.sh` en tu directorio de usuario.

3.6. Practiquemos un poco

Te planteamos a continuación la programación de una serie de *scripts*, que deberás pensar, escribir y ejecutar para comprobar su correcto funcionamiento.

Ejercicios

▶ 18 Escribe un *script* de nombre `basura.sh`, que enviará/recuperará archivos a/de la papelera. Para ello, debe crear primero (si no existe ya) la papelera, que puede ser un directorio oculto `.deleted` en tu directorio de usuario `/users/alumni/?/al0xxxxx`. `basura.sh` podrá recibir una lista de nombres de fichero, para cada uno de los cuales comprobará:

- Si el archivo se encuentra en el directorio actual, en cuyo caso lo enviará a la papelera.
- Si el archivo se encuentra en la papelera, en cuyo caso lo recuperará moviéndolo al directorio actual.
- Si el archivo no existe.

▶ 19 Escribe un *script* que reciba dos parámetros: un directorio y una palabra. El *script* debe mostrar por pantalla el archivo de ese directorio que contenga más líneas con la palabra que le hemos pasado. Puedes comprobar la ejecución con el directorio `heapsort` y la palabra `heap`.

▶ 20 Escribe un *script* que muestre por pantalla cúantos de sus parámetros corresponden a nombres de ficheros y cuántos a nombres de directorios. Además deberá decir la talla media de los ficheros que se le han pasado (recuerda que puedes medir el tamaño en bytes con la orden `wc -c`.).

▶ 21 Escribe un *script* que muestre un listado de todos los ficheros del directorio actual con un tamaño mayor al que se pasa como argumento.

▶ **22** Escribe un *script* de nombre `sumadir` que sume los tamaños de todos los ficheros del directorio pasado como argumento, dando un mensaje de error si el argumento pasado no es un directorio o no existe.

▶ **23** Escribe un *script* de nombre `sumatodo` que muestre la suma de los tamaños de todos los ficheros del directorios actual y también la suma de los archivos que contienen cada uno de sus subdirectorios en un primer nivel. Deberás utilizar el *script* `sumadir` del ejercicio anterior.
