Reporte de Actividad 4

Valenzuela Terán Jonás

March 1, 2018

Introducción

La terminal del sistema operativo Linux ofrece muchas funcionalidades, que tal vez sea menos conveniente que su contra parte gráfica. especialmente para tareas repetitivas. En esta actividad se tuvo un primer acercamiento a comandos nuevos y útiles dentro de la terminal, y se pusieron en práctica para observar su funcionamiento.

Comandos utilizados

cat

Se utiliza para leer archivos completamente, y mostrarlo en pantalla, o guardarlo en otro archivo. Es posible leer varios archivos a la vez utilizando "*".

```
cat datos.mat > respaldo.mat
```

Se muestra un ejemplo, donde se lee un archivo, y los datos son pasados a un respaldo.

chmod

Funciona para cambiar los permisos de un archivo para diferentes tipos de usuario (dueño, grupo y desconocido), esto es, habilitar o des habilitar la lectura, edición y ejecución del archivo.

```
chmod 755 script.sh
```

El modo corresponde a 3 dígitos, para dueño, grupo y desconocido respectivamente, cada uno es un número binario de 3 dígitos, que corresponden a read write execute, 0 es permiso denegado y 1 es permitido, el número que resulta, se representa en decimal para el argumento.

echo

Escribe un argumento en pantalla o en un archivo, se utiliza normalmente para mostrar información en pantalla, o escribir información reducida a un archivo.

```
echo Hello World
```

grep

Sirve para buscar cadenas de texto en un archivo, siendo útil para solo mostrar la información de interés.

```
grep 'temperatura' sondeos.txt
```

El ejemplo tendrá como resultado todas las cadenas de texto con la palabra temperatura y su ubicación.

less

Se utiliza para leer el contenido de un archivo, pagina por página, con la ventaja de solo cargar el contenido que se ve, siendo más rápido con archivos de gran tamaño.

```
less sondeos.txt
```

Al introducir el ejemplo en la termina, se mostrará el contenido del archivo en la misma terminal, con la habilidad de navegar en el página por página.

ls

Al usarse sin argumento, se despliega en una lista los archivos contenidos en el directorio actual
ls -l
Además, con el uso de banderas, permite mostrar información adicional de cada archivo, como los permisos, y fecha de edición o creación.

wc

Es abreviatura de "Word Count", muestra el número de lineas, de palabras y de caracteres/bytes que contiene el/los archivo(s) en el argumento.

```
wc notas1.txt notas2.txt
```

El ejemplo mostrará una tabla, conteniendo la información de las 3 categorías mencionadas de cada uno, así como del total (suma) de los 2.

Redirectores: |, >

Funcionan para redirigir la salida de un comando o programa, normalmente cambiar la salida de pantalla, a un archivo.

```
cat datos.mat > respaldo.mat
```

Retomando el primer ejemplo, toma los datos leidos del archivo datos.mat y los guarda en respaldo.mat

```
ls -l | less
```

El pipe "|" pasa como argumento de la salida del comando ls -l, a less, resultando en una lectura página por página de todos los archivos en el directorio.

Síntesis de las notas de Steve Parker Shell Script Tutorial

1. Introducción

El propósito del texto es brindar un tutorial introductorio acerca de shell script, mostrando las posibilidades y utilidades de este. Existen algunos tipos de shell, pero el tutorial se enfoca en Borune y Bourne Again shells. Los archivos script de este tipo tiene formato sh, y comienza siempre con:

```
#!/bin/sh
# Esto es un comentario
```

Los comentarios son marcados por #. Para poder ejecutar el archivo, se debe habilitar el permiso de ejecutarlo, con el comando chmod desde la terminal.

2. Filosofía

Programación en shell script no ha sido totalmente bien recibido, ya que en comparación, la velocidad de un programa como C, es mayor, además, existen muchos script de mala calidad, por la facilidad de su creación. Para crear uno con calidad, se debe crear estructura comprensible, clara, y evitar comandos innecesarios, el objetivo es optimizar.

3. Un primer script

Similar al ejemplo de la introducción:

```
#!/bin/sh
# Comentario
echo Hello World        # También un comentario!
```

La línea con la que empiezan los archivos le dice a Unix que el archivo será ejecutado por /bin/sh, que es la ubicación del Borune shell. Como vimos, # denota los comentarios, y pueden ser ubicados después del código en la misma línea. Echo tiene como salida sus argumentos, en este caso, 2. Para poder ejecutar el archivo, se cambia el permiso de la siguiente manera:

```
chmod 755 first.sh
```

Esto hará ejecutable el archivo first.sh para el creador. Es importante entender como se comporta el argumento de echo, lo ideal es insertar una cuerda de texto, encerrada por "".

4. Variables (parte1)

En casi todos los lenguajes de programación se hace uso de las variables, un nombre simbólico que se le da a una porción de memoria a la que se le asignan valores, pueden ser leídos y modificados. Un ejemplo para asignar a una variable un valor y mostrarlo:

```
#!/bin/sh
MY_MESSAGE="Hello World"
echo $MY_MESSAGE
```

La segunda línea asigna la cuerda de texto "Hello World" a la variable MY_MESSAGE, la tercera lo imprime en pantalla. Es importante cuidar los espacios y ponerlos dentro las comillas, ya que estos separan los argumentos y programas. Las variables pueden recibir números, no es necesario declararla. El siguiente ejemplo muestra el uso de variables dentro de cuerdas de texto:

```
#!/bin/sh
echo What is your name?
read MY_NAME
echo "Hello $MY_NAME - hope you're well."
```

$ es usado para indicar el uso de una variable, incluso dentro de cuerdas de texto, es considerado un carácter de escape. Cabe notar que el comando read automáticamente inserta comillas alrededor de su entrada.

Cuando se intenta usar una variable que no ha sido asignada a un valor, se mostrará vacía.

```
#!/bin/sh
echo "MYVAR is: $MYVAR"
MYVAR="hi there"
echo "MYVAR is: $MYVAR"
```

En el ejemplo, la primer salida es vacía, mientras que la segunda es "hi there". Eso ocurre aunque asignemos un valor a la variable desde la terminal antes de correr el script, para pasar valores desde la terminal a los programas, se utiliza el comando export. El mismo problema se presenta al querer recuperar valores de un programa a la terminal, para eso debemos correr el programa con . ./ que también funciona para importar variables de la terminal.

```
$ MYVAR=hello
$ echo $MYVAR
hello
$ . ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
$ echo $MYVAR
hi there
```

Para hacer uso de la variable modificaciones ligeras, es conveniente hacer referencia a la variable con

```
#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called ${USER_NAME}_file"
touch "${USER_NAME}_file"
```

5. Comodín

Estos son usados con regularidad en el ambiente de Unix, ayuda a evitar realizar procesos repetitivos, por ejemplo, copiar todos los archivos de /tmp/a a /tmp/b/

```
$ cp /tmp/a/* /tmp/b/
$ cp /tmp/a/*.txt /tmp/b/
$ cp /tmp/a/*.html /tmp/b/
```

La segunda y tercera línea solo realizarán la acción para archivos con la terminación apropiada.

6. Caracteres de escape

Ciertos caracteres son significativos para shell, como las comillas, que afectan como los espacios dentro de una cuerda de texto son interpretados, si queremos insertar comillas dentro de una cuerda de texto, se debe acompañar de

Lo mismo ocurre para $ que marca variables y para
que hace que el carácter sea tomado literalmente.

7. Ciclos

Ciclos son muy útiles, nos permiten repetir una tarea múltiples veces, con la posibilidad de hacerlo con pequeños cambios. Existen 2 tipos de ciclos, for y while.

Ciclos de for hacen iteraciones hasta que una lista indicada de valores termina.

```
#!/bin/sh
for i in 1 2 3 4 5
do
  echo "Looping ... number $i"
done
```

```
#!/bin/sh
for i in hello 1 * 2 goodbye
do
  echo "Looping ... i is set to $i"
done
```

El primer ejemplo imprime el argumento cambiando i desde 1 hasta 5, el segundo imprime en i los nombres de los archivos en el directorio, a excepción del principio y del final que es asignado 1 y 2.

Los ciclos while repiten el código hasta que una condición es o no cumplida.

```
#!/bin/sh
INPUT_STRING=hello
while [ "$INPUT_STRING" != "bye" ]
do
  echo "Please type something in (bye to quit)"
  read INPUT_STRING
  echo "You typed: $INPUT_STRING"
done
```

El ciclo se repite hasta que el usuario inserta bye. Un recurso útil es usar while read f, que usa casos, el siguiente ejemplo lee cada linea, e imprime el lenguaje del saludo introducido:

```
#!/bin/sh
while read f
do
  case $f in
hello) echo English ;;
howdy) echo American ;;
gday) echo Australian ;;
bonjour) echo French ;;
"guten tag") echo German ;;
*) echo Unknown Language: $f
;;
   esac
done < myfile
```

8. Prueba

Prueba es usado en casi todos los scripts, la manera simbólica de representarlo es [. o [, es un programa tal como ls, y debe ser rodeado de espacios. Es llamado comúnmente por if y while, es utilizado para comparar valores. Cabe notar que se debe tener cuidado y no nombrar un programa test.

```
if [ $foo = "bar" ]
then
  # if-code
else
  # else-code
fi
```

El ejemplo muestra la sintaxis general de una estructura if, if y then deben estar en lineas diferentes, o pueden ser separadas por punto y coma ; optimizando espacio. Para comparar números, es importante verificar que la entrada no sea una cuerda de texto, ya que este no puede se tratado como número. La diagonal
funciona para partir código de una sola línea en 2, y hacer el texto en ciertas situaciones, mas claro de leer. fi es if al revés, indica el fin de la estructura.

```
#!/bin/sh
if [ "$X" -lt "0" ]
then
  echo "X is less than zero"
fi
if [ "$X" -gt "0" ]; then
  echo "X is more than zero"
fi
[ "$X" -le "0" ] && \
      echo "X is less than or equal to  zero"
[ "$X" -ge "0" ] && \
      echo "X is more than or equal to zero"
[ "$X" = "0" ] && \
```

```
      echo "X is the string or number \"0\""
[ "$X" = "hello" ] && \
      echo "X matches the string \"hello\""
[ "$X" != "hello" ] && \
      echo "X is not the string \"hello\""
[ -n "$X" ] && \
      echo "X is of nonzero length"
[ -f "$X" ] && \
      echo "X is the path of a real file" || \
      echo "No such file: $X"
[ -x "$X" ] && \
      echo "X is the path of an executable file"
[ "$X" -nt "/etc/passwd" ] && \
      echo "X is a file which is newer than /etc/passwd"
```

```
#!/bin/sh
X=0
while [ -n "$X" ]
do
  echo "Enter some text (RETURN to quit)"
  read X
  if [ -n "$X" ]; then
    echo "You said: $X"
  fi
done
```

Se muestra un ejemplo donde se usa la prueba en una estructura while.

9. Caso

Son útiles en el caso de necesitar insertar una cadena larga de if.

```
#!/bin/sh

echo "Please talk to me ..."
while :
do
  read INPUT_STRING
  case $INPUT_STRING in
hello)
echo "Hello yourself!"
;;
bye)
echo "See you again!"
break
;;
*)
echo "Sorry, I don't understand"
;;
  esac
done
echo
echo "That's all folks!"
```

Compara el valor de una variable, y realiza el código donde cumple la condición, *) es ejecutado cuando ningún caso es cumplido. La estructura, al igual que if, es terminada con esac, case al revés.

10. Variables (parte 2)

Existe un conjunto de variables que son predeterminados, no pueden ser asignados otros valores, estos contienen información útil acerca del ambiente en el que corre el programa. $0 es el *basename*

del programa que fue llamado, $1 hasta $9 con los primeros 9 parámetros adicionales con los que fue llamado el programa. $@ contiene todos los parámetros, $* igual pero sin preservar espacios y comillas, $# contiene el número de parámetros con los que fue llamado.

```
$ ./var3.sh hello world earth
I was called with 3 parameters
My name is ./var3.sh
My first parameter is hello
My second parameter is world
All parameters are hello world earth
```

El ejemplo es un script que muestra $#, $0, $1, $2 y $@.

$? contiene el valor exit de el ultimo comando run, el cual debe ser 0 si todo ha resultado bien, es útil para manejar errores de manera más conveniente.

$$ es el PID (Identificador de procesos) del shell corriendo actualmente, útil para crear archivos temporales, cuando más de una instancia de un mismo programa se corren, y necesita sus archivos temporales,

$! es el PID de el proceso anterior del background.

IFS es el *Internal Field Separator*. El valor predeterminado es SPACE TAB NEWLINE.

11. Variables (parte 3)

Las llaves son muy útiles para evitar confusión con variables:

```
foo=sun
echo $fooshine     # $fooshine es indefinido
echo ${foo}shine   # muestra la palabra "sunshine"
```

Para que una variable acepte valores predeterminados:

```
#!/bin/sh
echo -en "What is your name [ ‘whoami‘ ] "
read myname
if [ -z "$myname" ]; then
  myname=‘whoami‘
fi
echo "Your name is : $myname"
```

El código mostrado tomará el nombre del usuario predeterminado (UID) si presiona RETURN como entrada. Esto también puede lograrse de la siguiente manera:

```
echo -en "What is your name [ ‘whoami‘ ] "
read myname
echo "Your name is : ${myname:-‘whoami‘}"
```

Whoami es el comando que imprime el nombre de usuario (UID), pero puede ser cambiado por cualquier cuerda de texto, esto ayuda asegurar que la variable siempre tomará un valor, y no será vacía o nula.

12. Programas externos

Programas externos se usan frecuentemente en scripts de shell, como los comandos de Unix. ‘ indica que el texto encerrado en el va a correrse como comando.

```
$ MYNAME=‘grep "^${USER}:" /etc/passwd | cut -d: -f5‘
$ echo $MYNAME
Steve Parker
```

En el ejemplo, se muestra que esto nos permite asignar a la variable, el valor de salida producido por el comando encerrado, puede ser útil también cuando la salida de una tarea lenta será usada múltiples veces:

```
#!/bin/sh
HTML_FILES=‘find / -name "*.html" -print‘
echo "$HTML_FILES" | grep "/index.html$"
echo "$HTML_FILES" | grep "/contents.html$"
```

13. Funciones

Es relativamente fácil crear funciones en shell, y más conveniente aún, librería con funciones para crear múltiples scripts. Las funciones de shell abarcan lo que otros programas llamarían funciones, procedimientos, subrutinas, modulos, etc. Las funciones pueden: cambiar el estado de variables, terminar el script, terminar la función y devolver el valor a el comando que lo llama y hacer echo a la salida a stdout. Un script conteniendo una función se ve así:

```
#!/bin/sh
# A simple script with a function...

add_a_user()
{
  USER=$1
  PASSWORD=$2
  shift; shift;
  # Having shifted twice, the rest is now comments ...
  COMMENTS=$@
  echo "Adding user $USER ..."
  echo useradd -c "$COMMENTS" $USER
  echo passwd $USER $PASSWORD
  echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}

###
# Main body of script starts here
###
echo "Start of script..."
add_a_user bob letmein Bob Holness the presenter
add_a_user fred badpassword Fred Durst the singer
add_a_user bilko worsepassword Sgt. Bilko the role model
echo "End of script..."
```

La función comienza cuando se indica el () después del nombre, después, lo contenido en es el código de la función, que es ignorado hasta ser llamado.

Es posible recuperar información de como fue llamada la función, utilizando las variables predeterminadas dentro de la función:

```
#!/bin/sh

myfunc()
{
  echo "I was called as : $@"
  x=2
}

### Main script starts here

echo "Script was called with $@"
x=1
echo "x is $x"
myfunc 1 2 3
echo "x is $x"
```

Una función no puede cambiar los valores con los que fue llamados, puede hacerlo cambiando las variables, no los parámetros. Además, pueden ser recursivas, es decir, llamar otras funciones, aquí un ejemplo:

```
#!/bin/sh

factorial()
{
  if [ "$1" -gt "1" ]; then
    i=‘expr $1 - 1‘
    j=‘factorial $i‘
    k=‘expr $1 \* $j‘
    echo $k
  else
    echo 1
  fi
}


while :
do
  echo "Enter a number:"
  read x
  factorial $x
done
```

Como se mencionó antes, es conveniente crear librerías de funciones, se pueden usar para definir variables comunes, un ejemplo:

```
# common.lib
# Note no #!/bin/sh as this should not spawn
# an extra shell. It’s not the end of the world
# to have one, but clearer not to.
#
STD_MSG="About to rename some files..."

rename()
{
  # expects to be called as: rename .txt .bak
  FROM=$1
  TO=$2

  for i in *$FROM
  do
    j=‘basename $i $FROM‘
    mv $i ${j}$TO
  done
}
```

Mostramos 2 funciones que se relacionan con la librería

```
#!/bin/sh
# function2.sh
. ./common.lib
echo $STD_MSG
rename .txt .bak
```

```
#!/bin/sh
# function3.sh
```

```
. ./common.lib
echo $STD_MSG
rename .html .html-bak
```

El ejemplo siguiente muestra un script que hace referencia a 2 funciones, y notifica si ocurrió un error:

```
#!/bin/sh

adduser()
{
  USER=$1
  PASSWORD=$2
  shift ; shift
  COMMENTS=$@
  useradd -c "${COMMENTS}" $USER
  if [ "$?" -ne "0" ]; then
    echo "Useradd failed"
    return 1
  fi
  passwd $USER $PASSWORD
  if [ "$?" -ne "0" ]; then
    echo "Setting password failed"
    return 2
  fi
  echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}

## Main script starts here

adduser bob letmein Bob Holness from Blockbusters
ADDUSER_RETURN_CODE=$?
if [ "$ADDUSER_RETURN_CODE" -eq "1" ]; then
  echo "Something went wrong with useradd"
elif [ "$ADDUSER_RETURN_CODE" -eq "2" ]; then
   echo "Something went wrong with passwd"
else
  echo "Bob Holness added to the system."
fi
```

Se utiliza ADDUSER_RETURN_CODE=$? para guardar $?, y ser utilizado en las pruebas.

14. Consejos

En Unix, todo es controlado por texto o una interface de comandos, con *nix, "todo es un archivo", por lo tanto, se tiene un sinfín de procedimientos que podemos automatizar.

grep es un comando muy útil para el programador, permite buscar en archivos información y devolver sólo la de interés. Un ejemplo:

```
#!/bin/sh
steves=‘grep -i steve /etc/passwd | cut -d: -f1‘
echo "All users with the word \"steve\" in their passwd"
echo "Entries are: $steves"
```

El código mostrará los usuarios que contienen steve en su contraseña, sin embargo, si existe más de una entrada, grep devuelve información sobre la localización de cada coincidencia, para esto, podemos usar el comando cut:

```
$> grep -i steve /etc/passwd
steve:x:5062:509:Steve Parker:/home/steve:/bin/bash
```

```
fred:x:5068:512:Fred Stevens:/home/fred:/bin/bash
$> grep -i steve /etc/passwd |cut -d: -f1
steve
fred
```

El resultado en el script mostrará los resultados horizontal separados por espacio, para convertir los espacios en lineas nuevas, se utiliza el comando tr.

```
#!/bin/sh
steves=‘grep -i steve /etc/passwd | cut -d: -f1‘
echo "All users with the word \"steve\" in their passwd"
echo "Entries are: "
echo "$steves" | tr ’ ’ ’\012’
```

También puede cambiar texto de minúscula a mayúscula, y viceversa:

```
#!/bin/sh
steves=‘grep -i steve /etc/passwd | cut -d: -f1‘
echo "All users with the word \"steve\" in their passwd"
echo "Entries are: "
echo "$steves" | tr ’ ’ ’\012’
```

Existen 2 ejecutables que nos pueden ayudar considerablemente para ciertas situaciones, uno de ellos es *awk*, para abrirlo se necesitan cargar 110k de memoria. Si se necesita asignar a una variable el número de lineas que contiene un archivo, wc puede hacer el trabajo bien, pero como la salida contiene espacios, es difícil recuperar el número de la cuerda de texto. Usando awk, obtenemos exactamente el número que buscamos:

```
NO_LINES=‘wc -l file | awk ’{ print $1 }’‘
```

La otra herramienta útil es sed, se necesitan cargar 52k de memoria para utilizarlo. sed es corto de *stream editor*, a diferencia de grep que trabaja por líneas, sed puede modificar variables y palabras en líneas, por ejemplo:

```
echo ${SOMETHING} | sed s/"bad word"//g
```

Esto remueve la frase *bad word* de la variable.

Telnet es una técnica útil, que no es muy utilizada en la actualidad, ayuda a mostrar información del usuario.

```
#!/bin/sh
host=127.0.0.1
port=23
login=steve
passwd=hellothere
cmd="ls /tmp"

echo open ${host} ${port}
sleep 1
echo ${login}
sleep 1
echo ${passwd}
sleep 1
echo ${cmd}
sleep 1
echo exit
```

15. Referencia rápida

Se proporciona una guía rápida del significado no obvio de algunos comandos, ya que no es fácil encontrarlos todos usando motores de búsqueda. Incluye manejo de procesos, argumentos y condiciones de test de scripts shell.

```
Command/Description/Example
& Run the previous command in the background ls &
&& Logical AND if [ "$foo" -ge "0" ] && [ "$foo" -le "9"]
|| Logical OR if [ "$foo" -lt "0" ] || [ "$foo" -gt "9" ] (not in Bourne shell)
^ Start of line grep "^foo"
$ End of line grep "foo$"
= String equality (cf. -eq) if [ "$foo" = "bar" ]
! Logical NOT if [ "$foo" != "bar" ]
$$ PID of current shell echo "my PID = $$"
$! PID of last background command ls & echo "PID of ls = $!"
$? exit status of last command ls ; echo "ls returned code $?"
$0 Name of current command (as called) echo "I am $0"
$1 Name of current command's first parameter echo "My first argument is $1"
$9 Name of current command's ninth parameter echo "My ninth argument is $9"
$@ All of current command's parameters (preserving whitespace and quoting) echo "My arguments are
$* All of current command's parameters (not preserving whitespace and quoting) echo "My arguments
-eq Numeric Equality if [ "$foo" -eq "9" ]
-ne Numeric Inquality if [ "$foo" -ne "9" ]
-lt Less Than if [ "$foo" -lt "9" ]
-le Less Than or Equal if [ "$foo" -le "9" ]
-gt Greater Than if [ "$foo" -gt "9" ]
-ge Greater Than or Equal if [ "$foo" -ge "9" ]
-z String is zero length if [ -z "$foo" ]
-n String is not zero length if [ -n "$foo" ]
-nt Newer Than if [ "$file1" -nt "$file2" ]
-d Is a Directory if [ -d /bin ]
-f Is a File if [ -f /bin/ls ]
-r Is a readable file if [ -r /bin/ls ]
-w Is a writable file if [ -w /bin/ls ]
-x Is an executable file if [ -x /bin/ls ]
( ... ) Function definition function myfunc() { echo hello }
```

16. Shell Interactivo

Se recomienda usar bash como herramienta interactiva, es disponible en muchos sabores de *nix, y es intuitivo. Bash contiene herramientas para buscar el historial de instrucciones insertadas, las flechas verticales mueven una lista la lista del historial de comandos previos, ctrl+r funciona para hacer una búsqueda en reversa.

Para repetir comandos que fueron corridos anteriormente, y se conoce como inician, se puede hacer:

```
bash$ ls /tmp
(list of files in /tmp)
bash$ touch /tmp/foo
bash$ !l
ls /tmp
(list of files in /tmp, now including /tmp/foo)
```

Es posible hacer *ksh* más útil, agregando comandos de historial, ya sea en modo *vi* o *emacs*. Para iniciar una sesión ksh:

```
csh% # oh no, it's csh!
csh% ksh
ksh$ # phew, that's better
```

```
ksh$ # do some stuff under ksh
ksh$ # then leave it back at the csh prompt:
ksh$ exit
csh%

csh% ksh
ksh$ set -o vi
ksh$ # You can now edit the history with vi-like commands,
  # and use ESC-k to access the history.
```

El ejemplo anterior habilita la edición del historial con comandos en modo vi. Si presionas ESC, luego k, y presionando k múltiples veces, puedes regresar en el historial de comandos previosm se puede usar el modo vi de la manera siguiente:

```
ksh$ touch foo
  ESC-k (enter vi mode, brings up the previous command)
  w (skip to the next word, to go from "touch" to "foo"
  cw (change word) bar (change "foo" to "bar")
ksh$ touch bar
```

Bibliografía

Steve Parker. (2017). Shell Scripting Tutorial. 20 de febrero, 2018, de Shell Scripting Sitio web:
https://www.shellscript.sh/index.html

(2017). cat (Unix). 20 de febrero, 2018, de Wikipedia Sitio web:
https://en.wikipedia.org/wiki/Cat_(Unix)

(2017). less (Unix). 20 de febrero, 2018, de Wikipedia Sitio web:
https://en.wikipedia.org/wiki/Less_(Unix)

(2017). chmod (Unix). 20 de febrero, 2018, de Wikipedia Sitio web: https://en.wikipedia.org/wiki/C

srijan. (2017). How to use grep to search for strings in files on the shell. 20 de febrero, 2018,

Apéndice

- ¿Qué fue lo que más te llamó la atención en esta actividad?

  Que la terminal tiene funcionalidades como un programa

- ¿Qué consideras que aprendiste?

  A manejar archivos de manera conveniente y rápida

- ¿Cuáles fueron las cosas que más se te dificultaron?

  Aprender la nueva sintaxis, como el comportamiento de espacios y caracteres especiales

- ¿Cómo se podría mejorar en esta actividad?

  El único inconveniente que encuentro es que para poder practicar los comando, es necesario tener el sistema operativo adecuado, lo cual no es accesible para todos fuera del centro de computo, pero al familiarizarnos más con el ambiente, podremos disponer de el.

- ¿En general, cómo te sentiste al realizar en esta actividad?

  Sentí que la guía esta completa, con todos los conceptos e información accesible, clara y guiada.