Practica 4: Introducción a la programación de los intérpretes de comando

Barajas Ibarria Juan Pedro

Hermosillo, Sonora a 21 de Febrero del 2018

76256 Empalme, Son. Observations at 2017

1. Reporte

1.1. Introducción

Los sistemas operativos UNIX y otros sistemas derivados, como lo es Linux y macOS, se apoyan con un intérprete de comando (Shell), quien es juega el papel de intermediario entre el usuario y el sistema operativo. Hay toda una familia de intérpretes de comando para los sistemas operativos.

Entre ellos están: C Shell (/bin/csh), Bourne Shell (/bin/sh), Korn Shell (/bin/ksh/), Bourne Again Shell (/bin/bash), y otros.

En el caso de la mayoría de las variantes de Linux y macOS, por default viene el intérprete /bin/bash.

En esta actividad, exploraremos las distintas formas de interactuar y hacer programas (scripts) con el Bourne Shell (/bin/sh) y el Bourne Again Shell /bin/bash. En cierta forma en las actividades anteriores, hemos estado haciendo programas (scripts) para los lenguajes interpretados LaTeX y Python.

1.2. Análisis de datos

En esta actividad se realizo una introducción al interprete de comando shell, donde utilizamos el siguiente script ejemplo para descargar automáticamente los datos de un sondeo atmosférico anual [3], en este caso fue del 2017.

```
#!/bin/bash

# Archivo "script1.sh"
# Script para bajar automáticamente los datos de sondeo atmosférico de la
# Universidad de Wyoming (http://weather.uwyo.edu/upperair/sounding.html)
# para un rango de tiempo. Sustituir el valor de la estación de interés:
# Caso: Empalme, Sonora.
#Número de estación: 76256


STATION=76256

# Despues de editar este archivo, ejecuta el comando: chmod 755 script1.sh
# Para ejecutar el script: ./script1.sh

# Definimos el separador de valores de las variables, en este caso es ":"
IFS=":"

# Por ejemplo nos interesan bajar los datos del año 2017
LISTYs="2017"
```

```
# Lista de meses por días
LISTM31="01:03:05:07:08:10:12"
LISTM30="04:06:09:11"
LISTM28="02"

for j in $LISTYs ; do

# Meses 31 dias
for i in $LISTM31 ; do
/usr/bin/wget "http://weather.uwyo.edu/cgi-bin/sounding?region=naconf&TYPE=TEXT
    \%3ALIST&YEAR=$j&MONTH=$i&FROM=0100&TO=3112&STNM=$STATION"
# Reposa 8 segundos
        /bin/sleep 8
done
# Meses 30 dias
for i in $LISTM30 ; do
/usr/bin/wget "http://weather.uwyo.edu/cgi-bin/sounding?region=naconf&TYPE=TEXT
    %3ALIST&YEAR=$j&MONTH=$i&FROM=0100&TO=3012&STNM=$STATION"
# Reposa 8 segundos
        /bin/sleep 8
done
# Feb. 28 dias
for i in $LISTM28 ; do
/usr/bin/wget "http://weather.uwyo.edu/cgi-bin/sounding?region=naconf&TYPE=TEXT
    %3ALIST&YEAR=$j&MONTH=$i&FROM=0100&TO=2812&STNM=$STATION"
# Reposa 8 segundos
        /bin/sleep 8
done
done
```

Después de descargar los 12 archivos de datos de cada uno de los meses del año proseguimos a utilizar los siguientes comandos [1] los cuales nos ayudaron a limpiar los datos y dejar solo lo que necesitamos.

- **less**: El cual nos permitirá visualizar el archivo con todo su contenido pero sin poder editarlo.
- **cat** : El cual nos dio el contenido de los archivos descargados.

- **grep**: El cual nos permitirá realizar búsquedas mas intensivas en archivos, es muy versátil para buscar detalles y ordenar entradas.
- **chmod**: El cual nos ayudo a modificar los permisos de lectura de los archivos, por ejemplo en estos scripts utilizamos

  ```
  -chmod 755 script1.txt
  ```

  El cual nos dio los permisos para poder ejecutarlo.
- **echo**: El comando interno echo es una de las instrucciones más simples de la shell. Se encarga de repetir o desplegar en la salida estándar cualquier argumento que se le indique(inclusive comodines), para posteriormente saltar una línea.
- **ls**: Esta ordena el contenido de los directorios. en este caso nos enlisto los archivos con su permiso de uso.

  ```
  ls -alg
  ```

- **wc:** Es un comando que permite realizar diferentes conteos desde la entrada estándar, ya sea de palabras, caracteres o saltos de líneas.

  ```
  wc -l <archivo> número de lineas
  wc -c <archivo> número de bytes
  wc -m <archivo> imprime el número de caracteres
  wc -L <archivo> imprime la longitud de la línea más larga
  wc -w <archivo> imprime el número de palabras
  ```

- **redictores:**

  1. |: El cual funciona para cambiar el tipo de lectura de *grep*, que en vez de hacerlo horizontal lo hará vertical hacia abajo.
  2. **>**: el cual nos permitió redirigir datos de un archivo o muchos a otro único.

  ```
  egrep -v 'PRES|hPa' sondeos.txt | egrep '76225|Showalter|LIFT|SWEAT|K|
      Totals|CAPE|CINS|LFCT|CAPV|Temp|Pres|thick|Precip' > df2017.csv
  ```

1.3. Resultados

Como resultado obtuvimos un archivo de texto el cual compila los datos que recolectamos con la función *egrep*. En el cual se encuentran compilados.

1.4. Conclusión

En conclusión con la actividad podemos decir que la herramienta de interprete de comandos SHELL es sumamente útil ya que nos ahorro muchísimo tiempo ya sea en descarga como en manejo y preparación de datos para su siguiente etapa de limpieza y poder interpretarlos al final.

2. Síntesis

Shell Scripting Tutorial

2.1. Introducción

El tutorial [2] se escribe para ayudar a las personas a comprender algunos de los conceptos básicos de la programación de scripts de shell. Steve Bourne escribió el shell Bourne que aparecioó en la version Seventh Edition Bell Labs Research de Unix.

2.2. Filosofía

La programación de script de shell tiene una mala impresión entre algunos administradores de sistemas de Unix. Esto es normalmente por esto:

- La velocidad en la que un programa se ejecuta en comparación con un programa C
- Se puede llegar a escribir muchos scripts de mala calidad por su simple tipo de trabajo por lotes.

Gracias a esto existe cierta discriminación asociada con la creación de buenos scripts de Shell. Esto nos dice que el diseño claro hace la diferencia entre scripts, podemos tomar en cuenta estas dos cosas:

1. Una secuencia de comandos simples con mucha frecuencia puede convertir el script en algo muy grande y complejo.

2. Si nadie entiende como funciona entonces usted sera el único que tendrá que cargar con esto.

Todo esto en conjunto nos lleva a algo muy bueno : nunca te sientas demasiado cómodo con tus scripts de shell; por su propia naturaleza la fuente no puede terminar, puede llegar incluso a recibir concejos de alguien mas.

2.3. Primer script

Para el primer script shell, solo escribiremos un script que diga *"Hello world"*. En la primera linea del primer ejemplo le dice a Unix que el archivo debe ser ejecutado por /bin/sh. Esta es la ubicación estándar del Shel Bourne en casi todos los sistemas Unix. La segunda es un comentario y en la tercera esta ejecutando un comando con dos parametros o argumentos: el primero es **"Hello"**; y el segundo es **"world"**. Para esto se tiene que ejecutar **chmod 755 first.sh** para hacer que el archivo de texto sea ejecutable.

- ```
  #!/bin/sh
  # This is a comment!
  echo Hello World # This is a comment too!
  ```

- ```
  #! / bin / sh # ¡Este es un comentario! echo "Hello W
  orld" # ¡Esto también es un comentario!
  ```

2.4. Variables (parte 1)

Echemos un vistazo a nuestro primer ejemplo Hello World. Esto podría hacerse usando variables. Tenga en cuenta que no debe haber espacios alrededor del signo -": VAR=valuefunciona;VAR = valueno funciona En el primer caso, el shell ve el símbolo -z trata el comando como una asignación de variable. En el segundo caso, el shell supone que VAR debe ser el nombre de un comando e intenta ejecutarlo.

Esto asigna la cadena "Hello World.ª la variable y MY MESSAGE luego echo establece el valor de la variable.

Tenga en cuenta que necesitamos las comillas alrededor de la cadena Hello World. Mientras que podríamos salirse con la suya echo Hello World porque echo tomará cualquier cantidad de parámetros, una variable solo puede contener un valor, por lo que una cadena con espacios debe ser citada para que el intérprete sepa que debe tratarlo todo como uno solo. De lo contrario, el shell intentará ejecutar el comando World después de asignar MY MESSAGE es igual a Hello.

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

- ```
  #! / bin / sh echo ¿Cuál es tu nombre ? lee el eco de MY_NAME "Hello
  $ MY_NAME - Espero que estés bien".
  ¿Cuál es tu nombre?
  ```

2.5. Comodines

Esta sección es realmente solo para hacer que las viejas celdas grises piensen cómo se ven las cosas cuando estás en un guión de shell, prediciendo cuál es el efecto de usar diferentes sintaxis.

2.6. Caracteres de escape

Ciertos personajes son importantes para el caparazón; hemos visto, por ejemplo, que el uso de caracteres de comillas dobles (") afecta la forma en que se tratan los espacios y los caracteres TAB.

La mayoría de los caracteres (*, ', etc) no se interpretan (es decir, que se toman literalmente) por medio de la colocación entre comillas dobles ( ). Se toman como están y se pasan al comando que se llama.

- ```
  $ echo "Esto es \\ una barra diagonal
  inversa" Esto es \ una barra diagonal inversa
  ```

- ```
  $ echo "Esto es \" una cita y esto es \\ una barra diagonal
  inversa " Esto es " una comilla y esta es \ una barra diagonal inversa
  ```

2.7. Bucle

La mayoría de los lenguajes tienen el concepto de bucles: si queremos repetir una tarea veinte veces, no queremos tener que escribir el código veinte veces, con un ligero cambio cada vez.

Como resultado, tenemos **fory whilebucles** en el shell Bourne. En el segundo ejemplo lo que ocurre es que las instrucciones de eco y lectura se ejecutarán indefinidamente hasta que escriba "byeçuando se le solicite.

Los dos puntos (:) siempre se evalúan como verdaderos; mientras que usar esto puede ser necesario algunas veces, a menudo es preferible usar una condición de salida real(tercer ejemplo).

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

- ```
  #!/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
  ```

- ```
  #!/bin/sh
  while :
  do
    echo "Please type something in (^C to quit)"
    read INPUT_STRING
    echo "You typed: $INPUT_STRING"
  done
  ```
  ```

2.8. Test

La prueba es utilizada por prácticamente todos los guiones de shell escritos. Puede que no parezca así, porque a testmenudo no se llama directamente. testes más frecuentemente llamado as [. Es un enlace simbólico test, solo para hacer que los programas de shell sean más legibles. También es normalmente un shell incorporado (lo que significa que el intérprete de comandos interpretará el significado test, incluso si su entorno Unix está configurado de manera diferente):

- ```
  $ type [
  [ is a shell builtin
  $ which [
  /usr/bin/[
  $ ls -l /usr/bin/[
  lrwxrwxrwx 1 root root 4 Mar 27 2000 /usr/bin/[ -> test
  $ ls -l /usr/bin/test
  -rwxr-xr-x 1 root root 35368 Mar 27  2000 /usr/bin/test
  ```

La prueba es una utilidad de comparación simple pero poderosa. Para obtener más detalles, ejecute **man** test en su sistema.

- ```
  #!/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"
```

2.9. Case

La sintaxis es bastante simple: la **case** línea en sí tiene siempre el mismo formato, y significa que estamos probando el valor de la variable INPUT STRING.

Las opciones que entendemos están listadas y seguidas por un corchete derecho, como hello) y bye).

Esto significa que si INPUT STRING coincide hello, se ejecuta esa sección de código, hasta el punto y coma doble. Si INPUT STRING coincide bye, se imprime el mensaje de despedida y sale el ciclo. Tenga en cuenta que si quisiéramos salir del script por completo, usaríamos el comando en exit lugar de break.

La tercera opción aquí, la *), es la condición catch-all predeterminada; no es obligatorio, pero a menudo es útil para la depuración, incluso si creemos saber qué valores tendrá la variable de prueba.

-
```
#!/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!"
```

2.10. Variables (part 2)

Ya hay un conjunto de variables establecidas para usted, y la mayoría de ellas no pueden tener valores asignados. Estos pueden contener información útil, que el script puede utilizar para conocer el entorno en el que se está ejecutando.

El primer conjunto de variables que veremos son $0 .. $9 y $#. La variable $ 0 es el nombre base del programa como se lo llamó. $1 ..$9 son los primeros 9 parámetros adicionales con los que se invocó el script.

La variable $@ es todos los parámetros $1 .. whatever. La variable $*, es similar, pero no conserva ningún espacio en blanco, y las comillas, por lo que .Archivo con espacios"se convierte en .Archivoconespacios". Esto es similar a echolo que vimos en A First Script . Como regla general, usa $@ y evita $*. $# es la cantidad de parámetros con los que se invocó el script.

- ```
  #!/bin/sh
  echo "I was called with $# parameters"
  echo "My name is $0"
  echo "My first parameter is $1"
  echo "My second parameter is $2"
  echo "All parameters are $@"
  ```
  ```

```
#!/bin/sh
while [ "$#" -gt "0" ]
do
  echo "\$1 is $1"
  shift
done
```

2.11. Variables (part 3)

Considere el siguiente fragmento de código que solicita al usuario la entrada, pero acepta los 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"
```

Esto podría hacerse mejor utilizando una función de variable de shell. Mediante el uso de llaves y el uso especial ": -", puede especificar un valor predeterminado para usar si la variable no está configurada:

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

2.12. Programas externos

Los programas externos a menudo se usan en scripts de shell; hay algunas órdenes internas (echo, which, y test son comúnmente incorporados), pero muchos comandos útiles son en realidad utilidades Unix, tales como **tr**, **grep**, **expry** y **cut**.

El backtick (‘) también se asocia a menudo con comandos externos. Debido a esto, discutiremos primero el contragolpe. El backtick se usa para indicar que el texto adjunto se debe ejecutar como un comando. Esto es bastante simple de entender. Primero, use un intérprete interactivo para leer su nombre completo desde **/etc/passwd**:

- ```
  grep "^${USER}:" /etc/passwd | cut -d: -f5
  ```

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

2.13. Funciones

Una característica que a menudo se pasa por alto de la programación de guiones de shell de Bourne es que puede escribir fácilmente funciones para usar en su secuencia de comandos. Esto generalmente se hace de una de dos maneras; con un script simple, la función simplemente se declara en el mismo archivo como se llama.

Una función puede devolver un valor en una de cuatro formas diferentes:

1. Cambiar el estado de una variable o variables
2. Use el **exit** para finalizar el script de shell
3. Utilice el **return** para finalizar la función y devolver el valor proporcionado a la sección de llamada del script de shell
4. echo output to stdout, que será capturado por la persona que llama al igual que $c = ‘expr \$a + \$b‘$ está atrapado

- ```
  #!/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..."
```

Recursivas

Las funciones pueden ser recursivas: aquí hay un ejemplo simple de una función factorial:

```
#!/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
```

Tal como se prometió, ahora discutiremos brevemente el uso de bibliotecas entre scripts de shell. Estos también se pueden usar para definir variables comunes, como veremos.

- ```
  # 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
  }
  ```

- ```
  #!/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
```

Referencias

[1] Ryan Chadwick, 2018.

[2] Steve Parker. The linux shell scripting tutorial. `shellscript.sh`, 2000-2018.

[3] Department of Atmospheric Science University of Wyoming. Weather sounding. `http://weather.uwyo.edu/upperair/sounding.html`, 2017.

Apéndice

1. ¿Qué fue lo que más te llamó la atención en esta actividad?
    - El nuevo lenguaje de programación que manejamos.
2. ¿Qué consideras que aprendiste?
    - Aprendí aun mas herramientas para la optimización de procesos computacionales.
3. ¿Cuáles fueron las cosas que más se te dificultaron?
    - El nuevo entorno y lenguaje de programación.
4. ¿Cómo se podría mejorar en esta actividad?
    - Darle mas tiempo para asimilar todos los detalles sobre el lenguaje.
5. ¿En general, cómo te sentiste al realizar en esta actividad?
    - Bien, interesado.