Informe_Trabajo_Práctico_Bash

Trabajo Práctico N°1.

Universidad Tecnológica Nacional.

Facultad Regional Buenos Aires.

Sintaxis y Semántica de los Lenguajes.

Curso: K2054.

Profesor: Ing. Pablo D. Mendez.

Tema: Expresiones Regulares y Expresiones Regulares Extendidas en Bash..

Grupo N° 201.

Integrantes.

#	Nombre	Apellido	Mail Institucional	Usuario GitHub	Legajo
1	Joan Franco	Canossa	jcanossa@frba.utn.edu.ar	lucasserral	204.062-1
2	Lucas Gabriel	Serral	lserral@frba.utn.edu.ar	jfr4nc0	203.464
3	Uriel	Moreno	umoreno@frba.utn.edu.ar	UriCode	176.344-1

Fecha de entrega: 05/10/2023

Link al Repositorio: https://github.com/lucasserral/SSL-grupo

Consigna.

1. Crear un usuario en GitHub https://github.com/ con el correo institucional frba. Crear un repositorio. Complete la planilla: https://docs.google.com/spreadsheets/d/1JSY2o7uRNJzwAJmo6yR0j2_swm7E_xfOshGX-UjoTsU/edit?usp=sharing con los datos del grupo Dentro del repositorio deberá subir todos los archivos que compongan la entrega de este trabajo dentro de una carpeta llamada “TP 1”. Para desarrollar este punto, ver el apéndice de este documento.
2. Debe entregar un único script que resuelva los siguientes puntos:
    1. Reemplace cada punto del archivo “breve_historia.txt” por punto y salto de línea generando un nuevo archivo.
    2. Borre todas las líneas en blanco.
    3. Cree un nuevo archivo: “breve_historia_2.txt” con el resultado de las operaciones a y b (redireccionamiento de la salida estándar).
    4. Del archivo “breve_historia.txt”, liste todas las oraciones que contengan la palabra “independencia” sin distinguir mayúsculas y minúsculas.
    5. Muestre las líneas que empiecen con “El” y terminen con “.” del archivo “breve_historia.txt”.
    6. Sobre el mismo archivo del punto anterior, Indique en cuántas oraciones aparece la palabra “peronismo”. Puede usar la opción -c para contar.
    7. Muestre la cantidad de oraciones que contienen la palabra “Sarmiento” y la palabra “Rosas”.
    8. Muestre las oraciones que tengan fechas referidas al siglo XIX.
    9. Borre la primera palabra de cada línea. Utilice substitución con sed. La sintaxis para sustituir la primera palabra de cada línea por “nada” sería: $sed “s/^[[a-zA-Z]]*\b//g” nombre_archivo (La “s” indica sustitución; entre los dos primeros /.../ está la expresión regular que queremos reemplazar, en este caso “/^[[a-zA-Z]]*\b”; entre el segundo y el tercer “//” se indica la expresión por la cual será reemplazada, en este caso por la palabra vacía. Finalmente la “g” indica que el cambio será en todo el archivo.
    10. Escriba un comando que enumere todos los archivos de una carpeta que contengan extensión “.txt”. (tip: pipe con el comando ls).
3. Investigue y explique, dando ejemplos cómo se utilizan los siguientes elementos en Bash:
   - Variables.
   - Sentencias condicionales.
   - Sentencias cíclicas.
   - Subprogramas.

   Dé ejemplos de cada una.

Investigación de elementos Bash.

Variables.

Principalmente las variables de Bash son de tipo cadenas de caracteres, pero también podemos encontrar variables de tipo entero con los que podríamos realizar operaciones aritméticas. Como en todo lenguaje de programación las variables contienen información relevante que es asignada por el usuario para luego ser utilizada a lo largo del programa

Parámetros posicionales:

Encargados de recibir los argumentos para un script y los parámetros de una función. Sus nombres son 1, 2, 3, …, etc. Las variables son accedidas mediante el símbolo $, por lo que si queremos acceder a las variables posicionales deberíamos anteceder el signo seguido del nombre de la variable. $1, $2, $3, …, etc. También existe el parámetro posicional $0 que es el encargado de almacenar el nombre del script que la ejecuta.
Por ejemplo:

```
echo "El script $0"

echo "Recibe los argumentos $1, $2, $3, $4"
```

Si el script anterior lo guardamos en un fichero con permiso x activado podremos ejecutarlo así:

```
$ saludo Hola Mundo SSL

"El Script .\Saludo"

"Recibe los argumentos Hola, Mundo, SSL"
```

Como el argumento $4 no fue pasado al script este da lugar a una cadena vacía que no se imprime.

Variables $*, $@, $#

- $# Almacena el numero de argumentos o parámetros recibidos.
- $* y $@ tienen un comportamiento similar entre si. devuelven los argumentos que recibe el script o función. cuando no se entrecomillan su funcionamiento es el mismo, pero cuando se entrecomillan con las comillas dobles su comportamiento cambia. Por ejemplo en el siguiente script llamado "DevuelveParametros"

```
for i in $@; do echo "\$@ $i"; done

for i in $*; do echo "\$* $i"; done
```

```
for i in "$@"; do echo "\"\$@\" $i"; done

for i in "$*"; do echo "\"\$*\" $i"; done
```

Al ejecutarlo notamos la diferencia

```
$ ./recibeparametros hola mundo maravilloso

$@ hola

$@ mundo

$@ maravilloso

$* hola

$* mundo

$* maravilloso4

"$@" hola

"$@" mundo

"$@" maravilloso

"$*" hola mundo maravilloso
```

Una diferencia notable cuando usamos comillas dobles es que en $* podemos cambiar el símbolo separador de los argumentos en una variable de entorno que identificamos como IFS (Internal Field Separator). En cambio, con $@, siempre el separador será un espacio simple . El ejemplo siguiente de un script llamado "MuestraParametros"

```
IFS = '|'

echo "El script $0 recibe $# argumentos: $*"

echo "El script $0 recibe $# argumentos: $@"
```

Al ejecutarlo por consola obtenemos lo siguiente

```
$ MuestraParametros Hola Mundo

El script MuestraParametros recibe 2 argumentos: Hola|Mundo

El script MuestraParametros recibe 2 argumentos: Hola Mundo
```

La ultima diferencia la encontramos en como son interpretadas las variables que contienen espacios. Pueden ser malinterpretadas, por lo que siempre se recomienda que esten

entrecomilladas las variables @* y $@. En el siguiente ejemplo veremos su efecto ante variables con espacios. Ej de script llamado "Argumentos"

```
function cuentaArgumentos {
  echo "Se recibieron $# Argumentos"
}

cuentaArgumentos $*

cuentaArgumentos $@

cuentaArgumentos "$*"

cuentaArgumentos "$@"
```

Al ejecutarlo obtendremos el siguiente resultado:

```
$ Argumentos "Sintaxis y Semantica" "de los Lenguajes"

Se recibieorn 6 Argumentos

Se recibieron 6 Argumentos

Se recibieron 2 Argumentos

Se recibieron 1 Argumentos
```

"$*" tiene el efecto de convertir todos los argumentos en un único token.
"$@" cada argumento es un token sin importar de que hallan espacios
$* y $@ un token por cada palabra encontrada, sin importar que el argumento este entre comillas.

Expansión de variables usando llaves

Hay dos usos comunes para el uso de llaves en las variables ${variable}.
Si por ejemplo quisiéramos mostrar nuestro nombre como Nombre_Apellido nos encontramos con un problema.

```
$nombre  = Uriel

$apellido = Moreno

$ echo "$nombre_$apellido"

Moreno
```

Lo que sucede es que Bash esta intentando encontrar la variable $nombre_ y al no encontrarla no imprime nada, pero esto se soluciona con las llaves

```
$nombre  = Uriel

$apellido = Moreno

$ echo "${nombre}_$apellido"

Uriel_Moreno
```

Por ultimo las llaves también se utilizan para extender las variables posicionales. Si quisiéramos tener la variable posicional $10, debemos entrecerrarlo entre llaves pues Bash lo interpreta como la variable $1 seguido de un 0. Por lo tanto la forma correcta de definirlo seria ${10}

Sentencias condicionales.

Las sentencias de control condicionales tienen el siguiente formato:

```
if condicion
then
sentencias
elif condicion
then
sentencias
else
sentencias
fi

#Otra forma de visualizarlo

if condicion; then
  sentencias
elif condicion; then
  sentencias
else
  sentencias
fi
```

La sentencia if comprueba el código de terminación de un comando en la condición (En Unix los comandos terminan con un código numérico que indica si el comando tuvo éxito o no). Si este es 0, la condición se evalúa como cierta.
El código de terminación puede consultarse pues disponemos de la variable ?, y accedemos a su valor mediante $?. Esta variable debe ser leída luego de ejecutar algún comando.
Disponemos también de operadores lógicos para ser utilizados dentro de la condición de la sentencia if. estos son && (and) , || (or) y ! (not).

```
#Ejemplo condicional con comandos y &&

if cd /tmp && cp 001.tmp $HOME; then
  echo "la condicion de la sentencia fue satisfecha"
else
  echo "La condicion de la sentencia no se ha satisfecho"
fi

#Ejemplo condicional con comandos y ||

if cp 001.tmp $HOME || cp 002.tmp $HOME; then
  echo "La condicion de la sentencia fue satisfecha"
fi

#Ejemplo condicional con comando y !

if cp /tmp/001.tmp $HOME; then
  echo "La condicion fallo con exito"
fi
```

En el primer ejemplo si la primer condición falla, deja de evaluar las siguientes pues no tiene sentido seguir evaluando. Similar pasa con el segundo ejemplo, si la primer condición se satisface correctamente, deja de evaluar las siguientes condiciones pues no tiene sentido seguir evaluándolo. Por ultimo en el tercer ejemplo se ingresa a las sentencias cuando el comando falla y el código de terminación se niega con el operador lógico !, volviéndolo verdadero.
Para comparar cadenas lo hacemos de manera lexicográficamente, para ello contamos con los operadores:

Operador	Verdadero cuando
str1 = str2	Las cadenas son iguales
srt1 != str2	Las cadenas son distintas
str1 < str2	str1 es menor lexicográficamente a str2
str1 > str2	str1 es mayor lexicográficamente a str2
-n str1	str1 es no nula y tiene longitud mayor a cero
-z str1	sgtr1 es nula

Para comparar enteros tenemos otros operadores

Operador	Verdadero Cuando
-lt	Less Than
-le	Less Than or Equal
-eq	EQual
-ge	Greater Than
-gt	Greater Than or Equal

Operador	Verdadero Cuando
-ne	Not Equal

Por ultimo podemos ver operadores de comparación de ficheros.

Operador	Verdadero cuando
-a fichero	El fichero existe
-b fichero	El fichero existe y es un dispositivo de bloque
-c fichero	El fichero existe y es un dispositivo de carácter
-d fichero	El fichero existe y es un directorio
-e fichero	Equivalente a: -a fichero
-f fichero	El fichero existe y es un fichero regular
-g fichero	El fichero existe y tiene activo el bit setgid
-G fichero	El fichero existe y es poseído por el grupo ID efectivo
-h fichero	El fichero existe y es un enlace simbólico
-k fichero	El fichero existe y tiene activo el bit stricky
-L fichero	El fichero existe y es un enlace simbólico
-N fichero	El fichero existe y fue modificado desde la ultima lectura
-O fichero	El fichero existe y es poseído por el grupo ID efectivo
-p fichero	El fichero existe y es un pipe o named pipe
-r fichero	El fichero existe y podemos leerlo
-s fichero	El fichero existe y no esta vacío
-S fichero	El fichero existe y es un socket
-u fichero	El fichero existe y tiene activo el bit setuid
-w fichero	El fichero existe y tenemos permiso de escritura
-x fichero	El fichero existe y tenemos permiso de ejecución o lectura si es un directorio
fich1 -nt fich2	La fecha de modificación del fich1 es mas nueva que la de fich2 (Newer Than)
fich1 -ot fich2	La fecha de modificación del fich1 es mas vieja que la de fich2 (Older Than)
fich1 -ef fich2	fich1 y fich2 son el mismo fichero (Equal File)

Sentencias cíclicas.

El bucle for es mas parecido al estilo de bucles for each de algunos lenguajes. Su sintaxis es

```
for var [in lista]; do
  #sentensias que usan $var
done
```

Bucle for

El bucle for no se repite una cantidad fija de veces sino que se procesan las palabras de una frase una a una. El bucle puede contener comodines, por ejemplo

```
  #Script que muestra informacion detallada de todos los ficheros en el
directorio actual

  for fichero in *; do
    ls -l "$fichero"
  done
```

Bucle while y until

La sintaxis de este bucle es como se presenta a continuación.

```
while comando; do
  #Sentencias del bucle while
done

until comando; do
  #Sentencias del bucle until
done
```

La diferencia entre While y Until es que while se ejecuta mientras que el código de terminación es exitoso y Until se ejecuta hasta que el código de terminación es exitoso. Esto podemos interpretarlo como que while se ejecuta siempre que el comando tenga éxito y until se ejecuta varias veces hasta que el comando tuvo éxito.

```
#Ejemplo de bucles while y until

contador = 0
termina = 10

#while
while [$termina -ge $contador]; do
  echo $contador
  let contador = $contador + 1
done

#until
until [$termina -lt $contador]; do
  echo $contador
  let contador = $contador +1
done
```

Subprogramas.

Los subprogramas o funciones en Bash son una herramienta útil que permite la reutilización del código como también evitar la repetición de bloques del mismo, y esto no solo es una ventaja sino que es una practica recomendada.

Declaración y Uso.

```
una_funcion(){
    echo Hola Mundo
}
```

También puede definirse una función en una sola línea, pero es importante respetar los espacios entre las llaves y los ';' entre instrucciones. Como muestra el siguiente ejemplo.

```
una_funcion_en_linea(){ echo Hola Mundo; echo otro Hola; }
```

Es común declarar funciones con la palabra reservada función. dando mas declaratividad al código.

```
function otra_funcion(){
    echo Hola Mundo
}
```

Para poder hacer uso de la función solo hay que usar su nombre sin los paréntesis ni llaves que se usaron para su declaración.

```
una_funcion
```

Una función no se ejecuta por si misma, es condición que este declarada como cualquier otra variable en el código. y para que se ejecute debemos hacer uso de ella.

Scope de las variables (Variables locales y globales.)

Si bien esto es un tema relacionado a las variables resulta imposible explicarlo sin antes haber mencionado las generalidades de los subprogramas o funciones.
Los parámetros posicionales son locales al script o función y no pueden ser accedidas ni modificadas por otras. Esto quiere decir que si nosotros realizamos un script que hace un llamado a una función que usa un parámetro e intentamos pasar el parámetro al script, este no funcionará.

```
    # Sript "saludo" que hace un llamado a la funcion saludar

    function saludar{
      echo "hola $1"
    }
```

```
saludar
```

El resultado que obtendremos por consola al querer hacer el llamado al script será el siguiente

```
$ saludo Pablo


hola
```

Si quisiéramos que este ejemplo funcionara al hacer el llamado de la función saludar hay que pasarle la variable.

```
# Sript "saludo" que hace un llamado a la funcion saludar

function saludar{
  echo "hola $1"
}

saludar $1
```

El resto de variables que pueden definirse son globales. Son accesibles y modificables desde cualquier función. Por ejemplo en el siguiente script "VariableGlobal"

```
#Ejemplo variable global

function ubicacion {
  variable = 'La variable esta dentro de la funcion'
}

variable = 'La variable esta dentro del script'

echo $variable

ubicacion

echo $variable
```

Al ejecutar el script anterior obtenemos lo siguiente por consola:

```
$ VariableGlobal

La variable esta dentro del script

La variable esta dentro de la funcion
```

Es posible volver local una variable agregándole el modificador 'local'. Pero eso únicamente podemos hacerlo dentro de las funciones y no en los scripts

```
function ubicacion {
  local variable = 'La variable esta dentro de la funcion'
}
```

Devolver valores

Una función puede devolver valores, para poder hacer eso, debemos utilizar la palabra reservada return. Los únicos valores que Bash permite retornar son valores numéricos desde el 0 hasta el 255, comúnmente esos valores se usan para devolver estados de una función siendo 0 el estado de ejecución exitosa y el resto de números distintos estados de error.

```
funcion1(){
    return 5
}

funcion1
echo $?
```

Entonces la forma que podríamos saber si una función que por ejemplo suma dos números los ha sumado correctamente es mediante el uso de variables locales de la siguiente forma.

```
funcion1(){
    resultado='resultado'
}

funcion1
echo $resultado
```

El problema que esto conlleva es que para programas mas extensos esta técnica incrementa considerablemente la complejidad. La segunda alternativa en relación con el scope de las variables es utilizar las variables locales de la función, de esa manera se mantiene un "Ambiente controlado".

```
funcion1(){
    local resultado='resultado'
    echo "$resultado"
}
resultado=$(funcion1)
echo $resultado
```

Reutilización de funciones.

Las funciones pueden ser implementadas incluso cuando se tratan de scrpts distintos. Para ello tenemos el comando "source", que se encarga de ejecutar el script al que se hace referencia.
Por ejemplo si tuviéramos un script mateBasica.

```
#!/bin/bash
suma(){
    local resultado=$(($1+$2))
    echo $resultado
}
resta(){
    local resultado=$(($1-$2))
    echo $resultado
}
producto(){
    local resultado=$(($1*$2))
    echo $resultado
}
```

y lo quisiéramos utilizar la función suma en otro script, entonces:

```
#!/bin/bash
source ./mateBasica
echo $(suma 2 2)
```

Esto es de gran utilidad pues podríamos crear scripts agrupando funciones útiles y ser utilizadas por distintos scripts que vayamos creando.