PROGRAMACIÓN SHELL

El shell es un interprete de comandos que permite al administrador ejecutar determinadas tareas. Es un auténtico Lenguaje de programación que le permite al administrador automatizar y programar tareas. Debido a que son ejecutados línea por línea razón por la cual se les conoce como programas SHELL SCRIPT. El shell de Linux incorpora sentencias de control de flujo, sentencias de asignación, funciones,… etc.

Para crear un shell script simple, mediante un editor cualquiera, en este caso **vi** , se insertan líneas de comando en Un fichero (para manejo estandarizado lo calificaremos con extensión **sh**, puede ir sin calificación o asignársele cualquiera otra. Lo importante es su contenido y la manera de ejecución); también se debe prever dotar al fichero de comandos con los permisos de acceso apropiados (en especial el de e**x**ecute) y después ejecuta el fichero.

Variable(s).- Es la denominación de un espacio en memoria que dentro de un programa será tratado bajo un nombre y propiedades determinadas (atributo numerico, de string,… longitud). Debido a que los pgms poseen un conjunto de instrucciones que se identifican con palabras específicas, es de restricción cuidadosa asignarles estas a aquellas (palabras reservadas a variables). Ejemplo: **echo.**

→ Ejm de un Pgm (script) cuya función es la de asignar a la variable numero un valor cualquiera para el caso se le cargará un contenido de inicio 5 y, luego mostrarla por pantalla.

Por editor cualquiera vi o nano se creará un fichero bajo un nombre calificado con **punto sh** ==> **name.sh**

```
#! /bin/bash
numero=5
echo “el valor de la variable es “  $numero
```

Pgms Shell que como comando, incluye comandos:

Las backquotes (`) entre el comando whoami ilustran el uso de la **sustitución de comandos** (Alt-Gr tecla parentesis cuadrado y llave derechos).

- sustitución de comandos : para incluir la salida de un comando dentro de una línea de comandos de otro comando, encierra el comando cuya salida quieres incluir, entre backquotes (`)

- whoami : displaya el usuario actual

```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ more prueba.sh
#! /bin/bash
echo "1 Fecha y hora:   `date` "
echo "2 Su usuario es:  `whoami` "
echo "3 Su directorio actual es: `pwd` "
echo "4 El mes actual es: `cal` "
echo "5 Consultaré paquetería: `synaptic` "
echo " "
hucaza@hucaza-Aspire-E5-471:~/PGMs$
```

Nota.- Para el comando incluido dentro la clausula echo que seráq mostrado en la ejecuón del shell debe estar entre comillas **`**cmdo**`**, estas comillas se obtienen utilizando teclas: **ALT-G y , tecla → paréntesis cuadrado y llave seguida del carácter a utilizar..**

PASO DE PARÁMETROS.

Es común que los scripts reciban parámetros en el momento de su ejecución desde la línea de comandos a efectos de hacerlos más versátiles, ellos pueden usarse dentro del script como cualquier otra variable Shell.

Los parámetros dentro del shell script son accesibles utilizando las variables:

- **$0** Para identificar el nombre del programa ambiente Shell donde opera el script.
- **$1** Identifica el primer parámetro de entrada
- **$2** Identifica el segundo parámetro de entrada
- **$3** Identificaría el tercer parámetro de entrada
- **$n** Identificaría así sucesivamente el enésimo parámetro de entrada
- **$#** Variable o parámetro de reserva por el script para indicar la cantidad de parámetros que ha recibido el Shell.

Solo se pueden pasar nueve argumentos, pero se puede acceder a mas de nueve usando el comando **shift**. Cada vez que se ejecuta el comando shift el argumento 1 desaparece, el 2 se convierte en el uno, y así sucesivamente hasta el 9 que se convierte en el 8 quedando aquel libre.

Algunas variables establecidas internamente por el shell y que están disponibles para el usuario:

$1 - $9 parámetros posicionales
$0 nombre del comando actual
$# *número de parámetros posicionales*
$? exit status del último comando ejecutado dado como un string decimal. Si todo ha ido bien se retorna cero.
$$ el numero de proceso de este shell, útil para incluirlo en nombres de ficheros para hacerlos únicos.
$! la pid del último comando ejecutado en background.
$- las opciones actuales suministradas para esta invocación del shell.
$* un string que contiene todos los argumentos del shell comenzando por el $1.
$@@ igual que el anterior, excepto cuando va entrecomillado.

Los parámetros dentro del shell script son accesibles utilizando en la denominación de las variables el prefijo **$.**

- El tipo de programa $0
- El primer parámetro $1
- El segundo parámetro $2 … etc
- Se utiliza la variable $# para conocer la cantidad de parámetros que ha recibido el shell.

EJM.
```
#! /bin/bash
echo "El tipo de programa es:          "  $0
echo  "El primer parámetro recibido es  "  $1
echo  " El segundo parámetro recibido es "  $2
echo  "   …   "
echo  " n total se han recibido          "  $#
```

Rutina o pgm shell que incluida en otro pgmSHELL donde espera una cantidad de parámetros determinados, obliga al cumplimiento de esa cantidad esperada:

```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ more prueba.sh
#! /bin/bash
echo "entra al programa"
if [ $# -lt 2 ]; then
   echo "Necesitas pasar dos argumentos."
   echo "sale forzadamente del pgm -por el exit- "
   exit 1
 fi
echo "sale del programa"
hucaza@hucaza-Aspire-E5-471:~/PGMs$
```

El precedente programa exige su ejecución a través del prefijo PUNTO-SLACH ==> (./prueba.sh)

PGMS sencillo en relación con el tratamiento de FECHAS:

ExtracciÓn:

```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ more ExtryndoFECHA.sh
#! /bin/bash
#Almacenar la salida del comando 'date'
dia=$(date)
#Extraer la fecha
nuevo_dia=${dia:0:10}
echo "nuevo_dia: " $nuevo_dia
#Extraer la hora
hora=${dia:11:8}
echo "hora: " $hora
#mostrar el dia
echo "dia: " $dia
hucaza@hucaza-Aspire-E5-471:~/PGMs$
```

```
#!/bin/bash

#Almacenar la salida del comando 'date'
dia=$(date)
#Extraer la fecha
nuevo_dia=${dia:0:10}
echo "nuevo_dia: " $nuevo_dia
#Extraer la hora
hora=${dia:11:8}
echo "hora: " $hora

#Mostrar el dia
echo "dia: " $dia
prueba.sh (END)
```

```
hucaza@hucaza-Aspire-E5-471:~$ . prueba.sh
nuevo_dia:  dom may 31
hora:  10:39:29
dia:  dom may 31 10:39:29 -05 2020
hucaza@hucaza-Aspire-E5-471:~$
```

ENTRADA Y SALIDA DE DATOS POR CONSOLA

Ya visto, la salida de datos con el comando echo y, la entrada, además de efectuarla con el paso de parámetros, también se efectúa con el comando read. Así:

```
#! /bin/bash
echo –n  "Introduce el valor de la variable:  "
# el parámetro  -n se utiliza para evitar el salto de línea      ( ← aquí el promt # opera como comentario )
read numero
echo "El valor introducido es: " $numero
```

Si se desea leer varios valores a la vez,

```
read numero1
read numero2
read numero3
```

O, lo que es lo mismo:

```
read numero1, numero2, numero3
```

Programa SHELL determinado para evaluar comandos, si es válido es ejecutado.

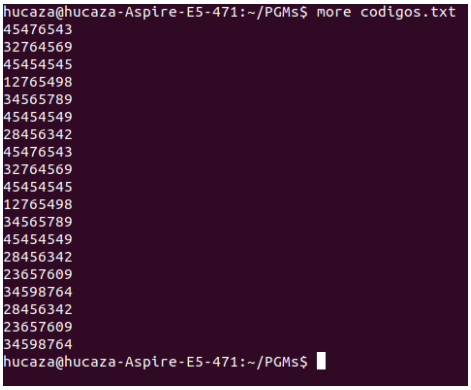
```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ cat prueba.sh
        #! bin/sh
        echo ?enter a command:?
        read command
        eval $command
```

Es de tener en cuenta que además el script puede obtener entradas e igualmente generar salidas a partir de ficheros no directorios, tal como en se podría aplicar mediante tuberías no nombradas (de comandos) cuando a partir de un fichero (**codigos.txt**) a través del comando grep se ignora (o se incluye) condicionamientos; ejemplo, iguales o diferentes a un string "XX", sorteando y obteniendo un único registro a partir del fichero referido (codigos.txt). Veamos:
Se ha creado un fichero texto con editor el cual contiene una serie de codigos estudiantiles simples únicamente, así:

vi **codigos.txt**

```
45476543
32764569
45454545
12765498
34565789
45454549
28456342
45476543
32764569
45454545
12765498
34565789
45454549
28456342
23657609
34598764
28456342
23657609
34598764
```

:wq!



Se quiere a partir de este fichero codigos.txt buscar aquellos que contengan **45**, para mayor comodidad queremos ordenarlos, y como tenemos repeticiones de líneas, queremos eliminar estas repeticiones. Para ello podemos ejecutar lo siguiente:

```
$ grep -i '45' codigos.txt | sort | uniq
```

El comando **grep** realiza la búsqueda, le pasa el resultado al comando **sort** que ordena las líneas, a su vez este resultado se le pasa a **uniq** que descartará las líneas repetidas.

Hacer un comando o programa Shell que nos muestre el contenido del fichero tratado en la tubería: codigos.txt:

vi pgm.sh Modalidad de inserción I:

```
#! /bin/bash
while read t;
do echo $t;
done < codigos.txt
```

Ejecutarlo así**: $. pgm.sh**

```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ . listacodigos.sh
45476543
32764569
45454545
12765498
34565789
45454549
28456342
45476543
32764569
45454545
12765498
34565789
45454549
28456342
23657609
34598764
28456342
23657609
34598764
hucaza@hucaza-Aspire-E5-471:~/PGMs$
```

OPERACIONES ARITMÉTICOS LÓGICAS

Utiliza el comando **expr** para realizar operaciones simples, en menor medida para manipular cadenas; y existe otro comando útil para comparaciones de string (cadenas), ficheros y números: comando **test**

Síntaxis:

$ **expr** arg1 op arg2 [op arg3…]

Ejm:

```
#! /bin/bash
echo -n "introduce un valor: "
read var1
echo -n "introduce un valor: "
read var2
resultado=$(expr $var1 \* $var2)
echo "el resultado de multiplicar valor1 por valor2 es: " $resultado
prueba3.sh (END)
```

Otra funcionalidad adicional de la función expr y que resulta muy interesante a la hora de programar es la generación de números aleatorios:

```
#! /bin/bash
numero=$(expr $RANDOM % 100)
echo "numero: " $numero
```

TABLA DE OPERADORES en script SH

Operador	**Comentario**				
aritméticos					
+	Suma	*	Multiplicación	%	Resto, de la división
_	Resta	/	División		
relacionales					
=	Igualdad	>=	Mayor o igual		
!=	Diferentes	<	Menor		
>	Mayor	<=	Menor o igual		

Operadores lógicos

||	Or lógico
&&	And lógico

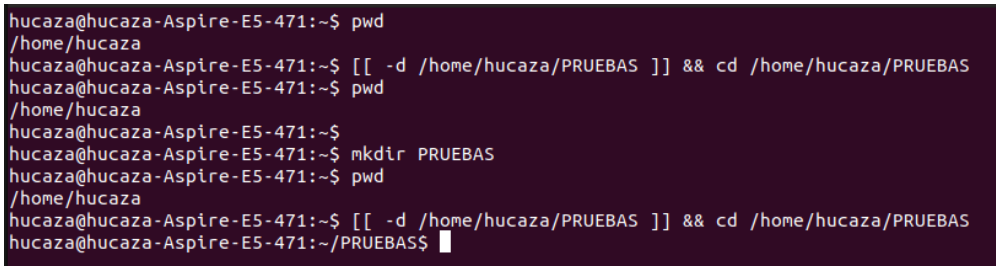
EJM. operador de igualdad (=)

```
#! /bin/bash
echo –n “Introduce un valor: ”
read var1
echo –n “Introduce un valor: ”
read var2
resultado = $(expr1 $var1 = $var2)
echo  “El resultado es:  ”  $resultado
```

EJM. Operador And lógico (&&) por fuera de un pgm.sh:

Basándonos en el frame siguiente, los primeros comandos para observar la efectividad del operador **&& (and lógico)** el fichero PRUEBAS no existe, por lo tanto no ejecuta el comando cd (no se puede trasladar)

En los comandos que le siguen, previamente se crea el fichero directorio PRUEBAS (con mkdir), luego entonces aseguramos que el directorio sí existe lo cual permitiría desplazarnos a ese directorio, el sistema implicitamente muestra donde estamos (pwd:PRUEBAS).



Operador DIVISIÓN enteros:

pgm:

```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ more prueba.sh
#! /bin/bash
echo -n "introduce un valor -dividendo-: "
read var1
echo -n "introduce otro valor -divisor-: "
read var2
resultado=$(expr $var1 / $var2)
echo "resultado de dividir var1 entre var2 es: " $resultado
hucaza@hucaza-Aspire-E5-471:~/PGMs$
```

Ejecución obedeciendo a tratamiento de cantidades enteras:

caso no generador de décimales

PGM:

```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ ./prueba.sh
introduce un valor -dividendo-: 15
introduce otro valor -divisor-: 3
resultado de dividir var1 entre var2 es:  5
hucaza@hucaza-Aspire-E5-471:~/PGMs$
```

caso generador de décimales

PGM:

```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ . prueba.sh
introduce un valor -dividendo-: 15
introduce otro valor -divisor-: 2
resultado de dividir var1 entre var2 es:  7
hucaza@hucaza-Aspire-E5-471:~/PGMs$
```

PARA ESTE CASO CUYA RESPUESTA ES INCORRECTA, podría tornarse en forma correcta00 incluyendo al PGM el operador adecuado % → resto de la división:

pgm:

```
#! /bin/bash
echo -n "introduce un valor -dividendo-: "
read var1
echo -n "introduce otro valor -divisor-: "
read var2
resultado=$(expr $var1 / $var2)
echo "resultado de dividir var1 entre var2 parte ENTERA es: " $resultado
resultado=$(expr $var1 % $var2)
echo "RESTO de la división es: " $resultado
prueba.sh (END)
```

ejecución:

```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ . prueba.sh
introduce un valor -dividendo-: 15
introduce otro valor -divisor-: 2
resultado de dividir var1 entre var2 parte ENTERA es:  7
RESTO de la división es:  1
hucaza@hucaza-Aspire-E5-471:~/PGMs$
```

COMANDO test

Permite evaluar tres tipos de elementos: Archivos, cadenas y, números.

Sintaxis: **test - opción archivo**
test [expresión]

TABLA comando TEST según el tipo de datos

Opción	**Descripción**
ARCHIVOS O DIRECTORIOS	
-f	Siendo un fichero regular (no directorio ni dispositivo) devuelve verdadero (0), si el fichero existe.
-s	Devuelve verdadero (0) si el fichero existe y si su tamaño es mayor que 0.
-r	Devuelve verdadero (0) si el fichero existe y tiene permiso de lectura
-w	Devuelve verdadero (0) si el fichero existe y tiene permiso de escritura
-x	Devuelve verdadero (0) si el fichero existe y tiene permiso de ejecución
-d	**Devuelve verdadero (0) si existe y es un directorio.**

Creación de un pgm para la verificación del comando test que comprueba la existencia de un fichero directorio (PRUEBAS)

```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ more verificaEcia_PRUEBAS-en-hucaza.sh
#! /bin/bash
if test -d /home/hucaza/PRUEBAS
then
cd /home/hucaza/PRUEBAS
pwd
else
echo "no existe el directorio PRUEBAS "
fi
hucaza@hucaza-Aspire-E5-471:~/PGMs$
```

```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ . verificaEcia_PRUEBAS-en-hucaza.sh
no existe el directorio PRUEBAS
hucaza@hucaza-Aspire-E5-471:~/PGMs$
```

Entonces, para probar la lógica del PGM formulado se procede por comando en hucaza crear el directorio PRUEBAS mkdir PRUEBAS en hucaza, y una vez allí creado se verifica el PGM-test para observar que ya no envía el anterior informe sino que procede a situarse (cd) en ese fichero ya que, se tiene la seguridad puesto que allí se creó.
Así:

```
hucaza@hucaza-Aspire-E5-471:~$ mkdir PRUEBAS
hucaza@hucaza-Aspire-E5-471:~$ pwd
/home/hucaza
hucaza@hucaza-Aspire-E5-471:~$ . PGMs/verificaEcia_PRUEBAS-en-hucaza.sh
/home/hucaza/PRUEBAS
hucaza@hucaza-Aspire-E5-471:~/PRUEBAS$
```

Evaluación de ficheros y Directorios
test -f /etc/passwd

PGM y ejecución:

```
hucaza@hucaza-Aspire-E5-471:~/PGMs$ more testf.sh
#! /bin/bash
test   -f  /etc/passwd ;  echo $?
test   -f /etc/hucaza ; echo $?
hucaza@hucaza-Aspire-E5-471:~/PGMs$
hucaza@hucaza-Aspire-E5-471:~/PGMs$ . testf.sh
0
1
hucaza@hucaza-Aspire-E5-471:~/PGMs$
```

Cambia fecha testeando del día deseado para el cambio: (donde el día: 0 Dom, 1 Lun, 2 Mart, 3 Mier, 4 Jue, 5 Sab)

PGM

```
#! /bin/bash
if $(test $(date +"%u") -eq 4)
then
   date --set "2020/12/31 23:59"
fi
```

VALORES NÚMERICOS

-lt ...menor que	-qt ...mayor que	-eq ...igual que
-le ...menor o igual que	-qe ...mayor o igual que	-ne ...no igual a

CONECTORES

-o	OR
-a	AND
!	NOT

Evaluación de cadenas
test [cadena1 = cadena2]
test [cadena1 != cadena2]

Evaluación numérica (válidas para números enteros)
Sintaxis: test número operador número

ESTRUCTURAS DE CONTROL

Permiten cambiar el flujo del programa, en función del estado de las variables.

Condición simple (if)
If condición
then
comandos
else
comandos
fi

Ejemplo de consición simple:

```
#! /bin/bash
echo –n "introduce un valor: "
read var
if (( var < 10))
then
                echo "El valor es menor que 10"
else
                echo "El valor es mayor que 10"
```

fi

CONDICIONES MÚLTIPLES

Es cuando se quiere hacer muchas condiciones sobre el mismo valor

```
case   $variable  in
                    valor1)   comando
                              . .
                               comando; ;
                    valor2)   comando
                              . .
                              Comando; ;
                              . . .
                    *)        comando
                              . .
                              Comando; ;
esac
```

Ejm.

```
#! /bin/bash
echo –n “Introduce un valor: “
read var1

case  $var1  in
1)        echo  “   uno  “;  ;
2)        echo  “   dos  “;  ;
3)        echo  “   tres  “;  ;
4)        echo  “   cuatro  “;  ;
*)        echo  “   opción incorrecta  “;  ;
esac
```

BUCLE FOR

Ejecuta un código un determinado número de veces

```
For  ( (expr1;  expr2;  expr3  ) )
          . . .
Done
```

Ejm que muestra los números del 0 al 5.

```
#! /bin/bash
for   ( (   i  =  0  ;   i  <=  5;   i++  ) )
do
          echo  “  $  “
done
```

También se puede utilizar el for para moverse en una lista de elementos:

```
for  variable in lista elementos
do
          echo  “  variable  “
done
```

BUCLE WHITE

Permite ejecutar un código hasta que no cumpla una determinada condición de salida. Su sintaxis es:

```
White  [    condición  ]
Do
          comando1
          comando2
          . . .
```

Done

Ejemplo:

```
#!   /bin/bash
limite = 5
I= 0;

while  (test  $limite   -gt  $i)
do
          echo  "Valor  $i"
          let  I = $i + 1
done
```

FUNCIONES

Una función es un bloque de código que permite su reutilización fácilmente. El nombre de la función debe reunir la descripción en síntesis de su funcionamiento.

Para definir una función se mediante la palabra reservada **function** seguida del nombre de la función y a continuación se declara la instrucción o conjunto de instrucciones que son ejecutadas cada vez que se realiza la llamada a la función, siempre entre llaves.

Ejem.

```
#!  /bin/bash
function mostrar_mensaje ( ) {
          echo "Hola Mundo!"
}
mostrar_mensaje;
```

1.
```
hucaza@hucaza-VirtualBox:~/Documentos$ more funcion.sh
#! /bin/bash
function mostrar_mensaje () {
      echo "Hola Mundo!"
      }
mostrar_mensaje;
hucaza@hucaza-VirtualBox:~/Documentos$
```

Nota.- Una vez ha sido creada la(s) función(es) pertinentes, ésta(s) desde cualquier otro script independiente puede(n) invocarse obteniendo el resultado esperado de las funciones llamadas.

Ejm. Se crea un script cualquiera, llamémoslo algo.sh y su objetivo simplemente es la invocar la función llamada << mostrar_mensaje >>, veamoslo:

```
vi   algo.sh                    #!  /bin/bash
                                mostrar_mensaje

Al activar algo.sh   . algo.sh
                                Generaría:    Hola Mundo!
hucaza@hucaza-VirtualBox:~/Documentos$    . algo.sh

Hola Mundo!

hucaza@hucaza-VirtualBox:~/Documentos$
```

Función **suma:**

suma.sh:

hucaza@hucaza-VirtualBox:~/Documentos$ more **suma.sh**

```
#! /bin/bash
function suma () {
     resultado=$(expr $a + $b)
     echo " a + b = " $resultado
}
a=5
b=10
suma $a $b;
hucaza@hucaza-VirtualBox:~/Documentos$
```

```
hucaza@hucaza-VirtualBox:~/Documentos$ . suma.sh
 a + b =  15
hucaza@hucaza-VirtualBox:~/Documentos$
```

Ahora, se procederá a activarla mediante la aplicación de parámetros:

```
hucaza@hucaza-VirtualBox:~/Documentos$ more suma2.sh
#! /bin/bash
function suma () {
        resultado=$(expr $a + $b)
        echo " a + b = " $resultado
}
a=$1
b=$2
suma $a $b;
hucaza@hucaza-VirtualBox:~/Documentos$
```

Primero: Si se invoca sin aportársele en la entrada los parámetros esperados entonces NO entrega resultados:

```
hucaza@hucaza-VirtualBox:~/Documentos$ . suma2.sh
expr: error de sintaxis
 a + b =
hucaza@hucaza-VirtualBox:~/Documentos$
```

Segundo: Ahora, si se le aporta parámetros entrega resultados:

```
hucaza@hucaza-VirtualBox:~/Documentos$ . suma.sh  12  45
a + b =  57
hucaza@hucaza-VirtualBox:~/Documentos$
```

También, podrá optar alguna transformación completa en la definición y tratamiento con parámetros:

suma3.sh:

```
hucaza@hucaza-VirtualBox:~/Documentos$ more funcion5.sh
#! /bin/bash
function suma () {
      c=$(expr $1 + $2)
      return $c

}
suma $1 $2
resultado=$c;
echo "..."  $resulado
```

Si se invoca sin parámetros sus resultados serán equívocos pero, si al contrario se ejecuta teniendo en cuenta su codificación ($1, $2 - con parámetros) sus resultados estarán acorde a lo esperado.

RUTINA PARA EL TRATAMIENTO CON DECIMALES.

1.- root@hucaza-VirtualBox:/home/hucaza/PGMS# **apt-get install apcalc**
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  debugedit librpmbuild3 librpmsign3 rpm
Use 'apt autoremove' to remove them.
The following additional packages will be installed:
  apcalc-common
The following NEW packages will be installed:
  apcalc apcalc-common
0 upgraded, 2 newly installed, 0 to remove and 422 not upgraded.
Need to get 777 kB of archives.
After this operation, 4.473 kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Get:1 http://co.archive.ubuntu.com/ubuntu xenial/universe amd64 apcalc-common all 2.12.5.0-1 [467 kB]
Get:2 http://co.archive.ubuntu.com/ubuntu xenial/universe amd64 apcalc amd64 2.12.5.0-1 [310 kB]
Fetched 777 kB in 33s (23,5 kB/s)
Selecting previously unselected package apcalc-common.
(Reading database ... 173991 files and directories currently installed.)
Preparing to unpack .../apcalc-common_2.12.5.0-1_all.deb ...
Unpacking apcalc-common (2.12.5.0-1) ...
Selecting previously unselected package apcalc.
Preparing to unpack .../apcalc_2.12.5.0-1_amd64.deb ...
Unpacking apcalc (2.12.5.0-1) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up apcalc-common (2.12.5.0-1) ...
Setting up apcalc (2.12.5.0-1) ...
root@hucaza-VirtualBox:/home/hucaza/PGMS#

2) Programa donde se aplica la función apcalc caso Decimales, donde se reemplaza la expresión expr OP * por simplemente el asterisco (*):

root@hucaza-VirtualBox:/home/hucaza/PGMS# more pgm2Decimales.sh
#! /bin/bash
aumento1=**0.075** decimales con PUNTO (no coma)
echo -n "valor : "
read valor
resultado=$(calc $valor*$aumento1)
echo "aumento1: " $aumento1
echo "valor: " $valor
echo "resultado es: " $resultado

3) Ejecución root@hucaza-VirtualBox:/home/hucaza/PGMS#
root@hucaza-VirtualBox:/home/hucaza/PGMS# . pgm2Decimales.sh
valor : 12000
aumento1: 0.075
valor: 12000
resultado es: 900
root@hucaza-VirtualBox:/home/hucaza/PGMS#