Tutorial de programación en BASH

Contenidos

Artículos

Referencias

Licencias de artículos

Conceptos e Historia de BASH

Descripción

BASH es un shell de Unix (*intérprete de comandos de Unix*) escrito para el proyecto GNU. Su nombre es un acrónimo de bourne-again shell (otro shell bourne); haciendo un juego de palabras (born-again significa renacimiento) sobre el Bourne shell (sh), que fue uno de los primeros shells importantes de Unix.

BASH es el shell por defecto en la mayoría de sistemas GNU/Linux, además de Mac OS X Tiger, y puede ejecutarse en la mayoría de los sistemas operativos tipo UNIX. También se ha portado a Microsoft Windows por el proyecto Cygwin.

Breve historia

Hacia 1978 el shell Bourne era el shell distribuido con el Unix versión 7. Stephen Bourne, por entonces investigador de los Laboratorios Bell, escribió el shell Bourne original.

Brian Fox escribió el shell bash en 1987.

En 1990, Chet Ramey se convirtió en su principal desarrollador.

El Manual de BASH Scripting Básico para Principiantes

El Manual de BASH Scripting Básico para Principiantes

Presentación

¡Saludos! estimado lector, seguramente te preguntas: "¿Qué clase de 'introducción' es esta?". Bueno, la verdad es que no soy muy original a la hora de redactar introducciones, pero lo bueno siempre está por venir (al menos así dicen por ahí). Bueno, al grano, este libro tiene la finalidad de dar una ligera introducción al mundo del software libre por medio de enseñar lo que se conoce como Shell Scripting, más específicamente enfocado a la shell GNU/BASH; sin embargo del tema en concreto hablaré en el primer capítulo.

Te invito si me estás leyendo en Wikilibros a que corrijas, traduzcas, amplíes (o dignifiques) el contenido del texto en cuestión. O bien, puedes contactarme en mi pagina de discusión y con gusto te tomaré en cuenta.

Nota importante

> Si haces alguna modificación al libro, por favor en la zona de discusión de esta página anota los cambios, siempre que sean significativos. No olvides poner tu firma.

Contenido

' * ' = Significa que este cápitulo aún no ha sido creado.

Muestrario de Scripts

En esta sección podrás encontrar scripts de muestra. Por supuesto, todos liberados bajo la licencia GPLv3.

1. Script instalador de Automatix 2 para *Ubuntu 6.06 a 7.10 y Debian Etch/
2. Script para Bloquear Páginas de Internet (con Zenity)/

Wikilibros Relacionados

- Fundamentos de Programación

Sintaxis

La sintaxis de órdenes de bash es un superconjunto de la sintaxis del shell Bourne. La especificación definitiva de la sintaxis de órdenes de bash, puede encontrarse en el *BASH Reference Manual* distribuido por el proyecto GNU. Esta sección destaca algunas de las características únicas de bash.

Características únicas de BASH

La mayoría de los shell scripts (guiones de órdenes) Bourne pueden ejecutarse por bash sin ningún cambio, con la excepción de aquellos scripts de shell Bourne que hacen referencia a variables especiales de Bourne o que utilizan una orden interna de Bourne.

La sintaxis de órdenes de bash incluye ideas tomadas desde el Korn Shell (ksh) y el C Shell (csh), como la edición de la línea de órdenes, el historial de órdenes, la pila de directorios, las variables $RANDOM y $PPID, y la sintaxis de substitución de órdenes POSIX: $(...).

Cuando se utiliza como un intérprete de órdenes interactivo, bash proporciona autocompletado de nombres de programas, nombres de archivos, nombres de variables, etc, cuando el usuario pulsa la tecla `TAB`.

Los scripts de bash reciben los parámetros que le pasa la shell como $1, $2, ..., $n. Podemos saber cuantos hemos recibido con el símbolo $#.

Ejemplo de uso

Por ejemplo, si nuestro script necesita dos parámetros pondremos:

```
if [ $# -lt 1 ]; then
   echo "Necesitas pasar dos parámetros"
fi
```

Hola Mundo en BASH

Hola mundo en BASH

Para empezar, haré una mención al simplismo "Hola mundo" de BASH, y también lo explicare.

```
#!/bin/bash
# Script de hola mundo
echo hola mundo
```

Salida:

```
hola mundo
```

Explicación

Es muy simple, en realidad. Lo que hacemos aquí al escribir primero "#!/bin/bash", es llamar a nuestra shell BASH, que por lo general se ubica en /bin/bash, pero podría estar en otro lugar, así que si no la tenemos ahí se ejecutara el comando "locate bash" desde nuestra terminal, o bien si no tenemos *locate* podemos usar "which bash" o bien desde nuestro directorio raiz (/) ejecutamos "find bash" para localizar la ruta del programa bash.

Comentario en BASH

Pero ¿el símbolo '#' escrito al inicio de una linea, es un comentario? En realidad sí, como en la segunda linea vemos, eso es un comentario, es decir, algo que no es interpretado y está ahí para que el programador/coder/scripter se ubique y sepa que es lo que hace el código, esto ayuda a tener mejor orden y al corregir el codigo si tiene BUGS. Pero la excepción a esto, es al escribir #!/ruta/de/interprete , esto nos sirve para llamar a nuestro interprete, como *perl*, *bash*, *sh*, *python*, etc, (que son otros lenguajes) y se usa para los lenguajes interpretados, de modo que se ejecuten en donde corresponde.

Imprimiendo en BASH

Por ultimo tenemos el comando `echo`, como en `batch`, que sirve para imprimir texto en la pantalla.

Recomendaciones

BASH funciona igual que nuestra terminal favorita, esto es por que nuestra terminal usa BASH para funcionar, y claro, cuando hacemos scripts es para automatizar procesos (lease INFORMÁTICA). Así es que la sintaxis es muy simple, **un comando por linea**.

Recomendamos **leer** los manuales **y experimentar** en el sistema operativo (claro sin ser root si no sabemos que es lo que hacemos) **para aprender**.

Variables en BASH

Como en todo lenguaje, necesitamos del uso de variables, que nos servirán para todo lo que queramos hacer en determinadas situaciones. Una variable, es una cadena de datos que se almacena en la memoria y que podemos llamar en cualquier momento para darle *X* uso.

Asignando variables

En BASH, las variables se asignan simplemente dando el nombre de la variable y su valor:

```
#!/bin/bash
#Asignando variables
hola=1
```

Como podemos leer aquí, este código asigna el valor 1 a la variable "hola". En BASH las variables son "CASE SENSITIVE" (sensibles), es decir, la variable `Hola` no es lo mismo que `HOLA` ni que `holA`.

Otro de los puntos importantes de las variables en BASH es que no tenemos que asignarles un tipo, es decir, podemos darle a las variables cualquier valor y lo aceptará sin tener que decirle a BASH si es numérico o si son letras.

Llamando a una variable

Invocando variables

Ya sabemos asignar una variable, pero de poco nos sirve si no podemos llamarla.

En BASH, las variables las invocamos simplemente anteponiendo un símbolo de dolar '$' antes del nombre de la variable.

Ejemplo

```
#!/bin/bash
#Asignando variables
hola=1
#Llamando a la variable
$hola
#Mostrando el contenido de la variable
echo $hola
```

Explicación

Si ponemos atención al código, en BASH, las variables simplemente se reemplazan por su valor al llamarlas, de modo que en ellas podemos almacenar *X* texto ó numero, ya sean comandos o lo que sea.

Al llamar a la variable, observamos que simplemente da el valor y lo pasa como una orden al interprete, pero si lo ponemos siguiendo un comando como `echo` entonces este mostrará el contenido de la variable.

Generando un numero aleatorio y enviandolo a una variable

Ejemplo de uso

Un pequeño ejemplo sobre como generar números aleatorios en BASH:

```
#!/bin/bash
#
# Se guarda en la variable el valor generado por $RANDOM,
# el % 5 asegura obtener un numero menor a 5 .
RNM=$RANDOM % 5`
echo $RNM
```

Comandos básicos de una shell

Comandos básicos

Para poder trabajar eficientemente en BASH, es indispensable conocer los comandos más básicos, aquí una pequeña lista que debemos conocer a la perfección:

Comandos básicos para BASH

Comando [Opciones]	Descripción del comando	Ejemplo de uso
cat *fich1* [...*fichN*]	Concatena y muestra un archivos	cat /etc/passwd
cd [*dir*]	Cambia de directorio	cd /tmp
chmod permisos fich	Cambia los permisos de un archivo	chmod +x miscript
chown usuario:grupo fich	Cambia el dueño un archivo	chown nobody miscript
cp fich1...fichN dir	Copia archivos	cp foo foo.backup
diff [-e]arch1 arch2	Encuentra diferencia entre archivos	diff foo.c newfoo.c
du [-sabr] fich	Reporta el tamaño del directorio	du -s /home/
file arch	Muestra el tipo de un archivo	file arc_desconocido
find dir test acción	Encuentra archivos.	find . -name ``.bak *-print*
grep [-cilnv] expr archivos	Busca patrones en archivos	grep mike /etc/passwd
head -count fich	Muestra el inicio de un archivo	head prog1.c
mkdir dir	Crea un directorio.	mkdir temp
mv fich1 ...fichN dir	Mueve un archivo(s) a un directorio	mv a.out prog1
mv fich1 fich2	Renombra un archivo.	mv .c prog_dir
less / more fich(s)	Visualiza página a página un archivo. less acepta comandos vi.	less muy_largo.c
ln [-s] fich acceso	Crea un acceso directo a un archivo	ln -s /users/mike/.profile .
ls	Lista el contenido del directorio	ls -l /usr/bin

pwd	Muestra la ruta del directorio actual	pwd
rm fich	Borra un fichero.	rm foo.c
rm -r dir	Borra todo un directorio	rm -rf prog_dir
rmdir dir	Borra un directorio vacío	rmdir prog_dir
tail -count fich	Muestra el final de un archivo	tail prog1.c
at [-lr] hora [fecha]	Ejecuta un comando mas tarde	at 6pm Friday miscript
cal [[mes] año]	Muestra un calendario del mes/año	cal 1 2025
date [mmddhhmm] [+form]	Muestra la hora y la fecha	date
echo string	Escribe mensaje en la salida estándar	echo ``Hola mundo
finger usuario	Muestra información general sobre un usuario en la red	finger nn @maquina.aca.com.co
id	Número id de un usuario	id usuario
kill [-señal] PID	Matar un proceso	kill 1234
man comando	Ayuda del comando especificado	man gcc, man -k printer
passwd	Cambia la contraseña.	passwd
ps [axiu]	Muestra información sobre los procesos que se están ejecutando en el sistema	ps -ux , ps -ef
who / rwho	Muestra información de los usuarios conectados al sistema.	who

Condicionales y ciclos

BASH incluye lo que cualquier lenguaje de programación necesita, las comprobaciones condicionales y los ciclos.

Comprobantes

> Una **comprobación**(test) es (test en español se traduce como prueba), como su nombre lo dice simplemente una función que se usa para verificar que determinada condición se cumpla (o ~~bien que~~ no);

Mientras que

un **ciclo**

> es un conjunto de instrucciones que se repitan mientras o hasta que la condición se cumpla.

El básico (If-Then)

La primera comprobación que veremos es el señor `if-then`, quizá el más básico.

EN BASH, se usa así:

```
#!/bin/bash
#Test IF

echo ' Adivina el valor numerico de la variable'
read A

if [ $A = 1 ]
then
echo 'Acertaste'
exit 0
fi

echo 'No acertaste'
exit
```

Como podemos notar, en este tipo de comprobación la condición es verificada y si se cumple las ordenes se ejecutan, y sino simplemente las salta.

Ahora bien, si necesitamos más complejidad usamos lo siguiente:

```
#!/bin/bash
#Test IF-ELSE

echo ' Adivina el valor numerico de la variable'
read A

if [ $A = 1 ]
then
echo 'Has acertado'
exit 0
else
echo 'Error, te has equivocado'
exit
fi
```

Y si queremos ponerle mas jugo al asunto:

```
#!/bin/bash
#Test IF-ELSE

echo ' Adivina el valor numerico de la variable'
read A

if [ $A = 1 ]
then
```

```
echo 'Has acertado'
exit 0
else
if [ $A = 2 ]
then
echo 'Estuviste cerca'
fi
fi

exit
```

Notas sobre la sintaxis:

- No olvidemos indicar en el test entre los paréntesis que la comparación de una variable siempre debe de llevar $ (no es mismo comparar si A = 1 que $A = 1).
- El usar mas de dos comprobadores juntos, es decir, querer usar 3 o mas comprobadores en el mismo terminará por fallar, ejemplo:

```
#!/bin/bash
#Test IF-ELSE

echo ' Adivina el valor numerico de la variable'
read A

if [ $A = 1 ]
then
echo 'Has acertado'
exit 0
else
if [ $A = 2 ]
then
echo 'Estuviste cerca'
fi
else
echo 'Erraste'
fi

exit
```

Notemos que la parte en negritas va a causar conflictos a la hora de ejecutar:

```
Bash $ line 16: error de sintaxis cerca de token no esperado `else'
Bash $ line 16: `else'
```

Para que esto funcione en su lugar usaremos la siguiente sintaxis:

```
#!/bin/bash
#Test IF-ELSE

echo ' Adivina el valor numerico de la variable'
read A
```

```
if [ $A = 1 ]
then
echo 'Has acertado'
exit 0
elif [ $A = 2 ]
then
echo 'Estuviste cerca'
else
echo 'Erraste'
fi

exit 0
```

- EL comprobador `IF`, siempre termina con `FI`. Si olvidamos esto el script fallará.
- Colocar el `FI` va al final de cada comprobador.

El clon (Case-Esac)

Sentencia Case-Esac

El conjunto de palabras `Case` ... `Esac` conforman un *selector* en función de un resultado.

Ejemplo

Por ejemplo, supongamos que tenemos una variable *X* la cual puede tomar un valor numérico leído desde teclado:

```
  read x
  case $x in
     1)
        echo "uno"
     ;;
     2)
        echo "dos"
     ;;
     *)
        echo "no se que numero es"
     ;;
  esac
```

Explicación del código

read espera un valor desde el teclado para *X* , 'ingrese un numero'; **Case** llama la variable y según su valor muestra un mensaje en pantalla, si el numero es distinto a 1 o 2 se ejecuta la linea '*' y **esac** es la instrucción de cierre.

Esto también es aplicable a variables alfanuméricas, aquí también presten atención que es sensible a mayúsculas y minúsculas.

Por ejemplo:

```
read opcion
case $opcion in
    s|S)
       echo "pulso la opción SI"
    ;;
    n|N)
       echo "pulso la opción NO"
    ;;
    *)
       echo "desconozco esa opción"
    ;;
esac
```

Explicación del ejemplo

En este caso el *programa* reconoce las mayúsculas y las minúsculas , el (símbolo barra vertical) '|' es una instrucción 'o' (OR) en donde las opciones son 's' ó 'S', lo que determinan si el usuario tecleo alguna de estas letras se ejecuta tal tarea, en este caso imprime en la pantalla la leyenda `pulso la opción SI`.

Este mecanismo es muy útil a la hora de toma de decisiones, por menú o de forma autónoma para tener un medio de control.

Consejos útiles

Un par de tips para esto:

- Las estructura `case-esac` puede estar contenida dentro de otro, por ejemplo de un `if` ó un `while`.
- En estos ejemplos la ejecución de tarea es de una línea pero se pueden poner varias lineas por tareas de decisión o bien llamar un script externo.

El clásico (For)

Sentencia `for-if`

El ciclo `for` es ampliamente usada en programación en la mayoría de los lenguajes, en el cual por cada elemento en determinada variable se repetirán determinadas acciones, esto hasta que lo haya hecho con todos los elementos indicados, entonces el ciclo habrá terminado y solo entonces.

Ejemplo de uso

```
for a in $x
do
        if [ a = algo ]
        then
                echo "Algo... es igual a algo"
        fi
done
```

Uso practico

Para poner un ejemplo fácil, analicemos el siguiente algoritmo:

```
Tengo 10 manzanas.
Por cada manzana que tenga, la tomaré y verificaré que no tenga un gusano:
Si lo tiene, tiraré la manzana,
si no, la guardaré en el refrigerador.
```

Imaginemos que mis 10 manzanas son los 10 archivos en determinada carpeta, llamados respectivamente desde el numero 1 hasta el 10 (sus nombres serían entonces 1, 2, 3, 4... y así hasta llegar al 10 respectivamente, esto va para cada archivo).

Imaginemos después que si el archivo (la manzana) contiene un gusano (la variable $gusano), entonces la variable ya mencionada tendrá un valor igual a 1, y si no, tendrá un valor igual a 0.

Uso practico en BASH

```
#!/bin/bash
# Manzana Parser: El script importa la variable guardada en cada
archivo
#                 y determina si "tiene o no gusano"

## Verificando que los parámetros sean válidos
if [ $# -ne 1 ]
then
        echo "Favor de solo especificar un directorio"
        exit 1;
elif [ ! -d $1 ]
then
        echo "El archivo $1 especificado no es un directorio,
abortando."
```

```
        exit 1;
fi

##Cambiando al directorio especificado

lastdir=(echo $PWD)
cd $1

## Inicializando aplicación

contador=0

for archivo in `ls $1`
do
        if [ -f $archivo ]
        then
                let contador=contador+1
        fi
for manzana in $contador
do
        if [ -f $archivo ]
        then
                source $archivo
                if [ $gusano = 0 ]
                then
                echo "La manzana $archivo no tiene gusano, guardando en
 $HOME/refrigerador"
                mv $archivo $HOME/refrigerador
                else
                echo "La manzana $archivo tiene gusano, eliminando la
manzana"
                rm $archivo
                fi
        fi
done
done

## regresando al directorio anterior|

cd $lastdir

exit 0
```

El ciclo (While)

Sentencia `while-do-done`

While es una instrucción de control que sirve para generar bucles, en los cuales grupos de instrucciones se ejecutan de forma repetida hasta que se cumpla una condición.

Cabe destacar que while, primero comprobará si se cumple la condición y si se cumple y sólo si se cumple entonces pasará a ejecutar secuencialmente las instrucciones contenidas entre `do` y `done`.

Cada vez que se ejecuta un bucle completo (paso de bucle) vuelve a verificar si se cumple la condición antes de volver a ejecutar otro paso de bucle.

Ejemplo de uso

Por ejemplo:

```
while [[ "condición lógica" ]]
do
     acción 1
     acción 2
     acción n
done
```

`do` y `done`

Las acciones entre "do" y "done" se repetirán secuencialmente mientras se cumpla la "condición lógica". Cuando la "condición lógica" deje de ser verdadera, no se ejecutará ninguna "acción". Por ejemplo:

Ejemplo de `do` y `done`

```
limite=5
i=0;

while [[ $limite -gt $i ]]
do
     echo Acción $i ejecutada
     let i=$i+1
done
```

El ejemplo anterior mostraría por pantalla:

```
Acción 0 ejecutada
Acción 1 ejecutada
Acción 2 ejecutada
Acción 3 ejecutada
Acción 4 ejecutada
Acción 5 ejecutada
```

Funciones

Funciones

Muchas veces viene bien tener funciones para evitar repetir código y hacer buenos scripts.

Las funciones se pueden definir de la siguientes formas:

```
function nombre_de_la_funcion {
   # comandos o instrucciones bash.
}
```

Ejemplos

Un ejemplo sería:

Función para limpiar la pantalla

```
function borrarPantalla {
   clear
}
```

Invocando funciones

Para invocar a una función simplemente se ha de usar su nombre.

Creamos las funciones:

```
function limpiarPantalla {
   clear
}
function listarETC  {
   ls /etc
}
function crearDirectorio
{
   mkdir directorio
}
```

Invocamos las funciones:

```
read opcion
case $opcion in
   b|B)
      limpiarPantalla
   ;;
   l|L)
      listarETC
   ;;
   c|C)
      crearDirectorio ;;
esac
```

Opciones (parámetros)

El uso de parámetros en BASH

Al ejecutar un script, tenemos una función muy útil que son las opciones, mejor conocidas como parámetros; es decir, son especificaciones que se le hacen al programa al momento de llamarlo para obtener un efecto diferente.

Ejemplo:

```
bash $ mount -a
```

El `-a` es un parámetro para el programa `mount`, en este caso lo que hace es montar todas las unidades del FSTAB (aunque esto no tiene mucho que ver en este momento con Bash, es tan solo un ejemplo.)

La pregunta es, ¿como implementar esto en nuestro script?

Como usar parámetros en BASH

Bien, si por ejemplo tenemos el siguiente script:

```
# !/bin/bash

RNM=`expr $RANDOM % 11`
if [$1 = $RNM] ; then
echo "Acertaste, el número "$1" es correcto"
else
echo "Has errado"
fi
```

¿Como es posible que funcione si no le hemos pedido que ejecute `read` para obtener el valor de la variable `$1`? En realidad es muy simple, ya que nuestro script lo llamaríamos así (por ejemplo):

```
bash $ ./adivina.sh 6
```

El `6` es el primer parámetro, en Bash pasa directo como la variable `$1`, si hubiéramos asignado mas parámetros se les iría asignando sucesivamente a las variables `$2`, `$3`, etc...

Como manejar los parámetros

Otra interesante función de Bash es que podemos trabajar con los parámetros de otras maneras, por ejemplo miren este script:

```
# !/bin/bash

echo "El nombre del fichero en ejecución es: $0"
echo "El primer parametro es: $1"
echo "El segundo parametro es: $2"
echo "Los parametros son: $*"
echo "La cantidad de parametros pasados es de $# parametros"
exit 0
```

Ahora ejecutamos el script:

```
bash $ ./parámetros.sh primero segundo
El nombre del fichero en ejecución es: ./parámetros.sh
El primer parámetro es: primero
El segundo parámetro es: segundo
Los parámetros son: primero segundo
La cantidad de parámetros pasados es de 2 parámetros
```

Esto amplía nuestros horizontes para realizar eficientes scripts.

Compilar (ofuscar) BASH scripts con C - SHC

Privacidad del código BASH

¿Alguna vez te has fastidiado por que no puedes proteger la intimidad de tu script? No te aflijas, que existe una genial utileria para todos aquellos que desean por alguna razón hacer algo de lo siguiente:

- Proteger su código de ojos mirones
- Evitar que alguien edite el script (útil para importantes scripts de sistema que por alguna razón alguien desee modificar de manera enfermiza)
- Curiosear un poco

Usando SHC

Al grano; la dichosa utileria se llama SHC [1] y es de la autoria de Francisco Javier Rosales García, un profesor de la Facultad de Informática en la Universidad Politécnica de Madrid.

Esta utileria no es en realidad un compilador, sino un "ofuscador" que encripta el código envolviéndolo en C, esto no hace al script mas rápido pero si sirve para 'ocultar' el codigo; después simplemente lo ejecuta y no es fácil hacer Ing. Inversa para obtener el código, sin embargo es posible.

Puedes descargala la utileria de http://www.datsi.fi.upm.es/~frosal/sources/compilarla e instalarla tú mismo, algunas distribuciones de Linux la instalan o están dentro de su repositorio.

Breve instalación de SHC

Se compila con instrucciones no muy genéricas, solo extraes el tarball y en ese directorio ejecutas 'make' y sigues las instrucciones.

O bien, puedes usar el siguiente método:

```
  #!/bin/bash
  # SHC Install Script

  echo 'Este script necesita permisos de root, asegurate de tenerlos
antes de ejecutarlo'
  echo 'Dime el numero de la ultima version de SHC (por ejemplo:
3.8.6). Puedes verificar cual es la ultima version entrando a
http://www.datsi.fi.upm.es/~frosal/sources/'
  read shcversion
  startdir=(pwd)
  wget http://www.datsi.fi.upm.es/~frosal/sources/shc-$shcversion.tgz
```

```
|| ! echo 'Error al descargar las fuentes, saliendo...' | exit 2
  tar xfv shc-$shcversion.tgz  || ! echo 'Error al extraer las fuentes
del tarball, saliendo...' | exit 3
  make  || ! echo 'Error al compilar, saliendo...' | exit 4
  install -D -s shc /usr/bin/shc  || ! echo 'Error al instalar,
saliendo...' | exit 5
  install -D -m 644 shc.1 /usr/man/man1/shc.1  || ! echo 'Error al
agregar pagina de man, saliendo...' | exit 6
  echo 'Instalacion exitosa, saliendo' | exit 0
```

Referencias

[1] http://www.datsi.fi.upm.es/~frosal/sources/shc.html

Combinando BASH con otros lenguajes de scripting

Como sabemos, al trabajar con BASH bajo Linux o algun UNIX tenemos una gran flexibilidad, por lo cual podemos llamar a otro interprete y pasarle un source code con las instrucciones deseadas. Esto permite hacer scripts muy completos.

Una de las formas de hacer esto, por ejemplo con perl, sería lo siguiente:

```
#!/bin/bash
#Llamando a un interprete externo a BASH
echo 'El siguiente texto será mostrado por el interprete de PERL'
perl -e 'print "Este texto es mostrado por un script PERL embebido.\n";'
exit 0
```

O bien, con python:

```
#!/bin/bash
#Llamando al interprete de Python.
echo 'El siguiente es un script de python:'
echo print "Hola, mundo!" | tee $HOME/.testpythonbash.py
python $HOME/.testpythonbash.py
exit 0
```

Otro método interesante es el siguiente:

```
#!/bin/bash
# bash-y-perl.sh

echo "Saludos desde la parte BASH del script."
# Es posible añadir mas comandos BASH aqui.

exit 0
# Fin de la parte BASH del script.

# ================================================== #
```

```
# ========================================================== #

#!/usr/bin/perl
# Esta parte del script se invoca con la opcion -x.

print "Saludos desde la parte PERL del script.\n";
# Podemos añadir mas comandos PERL aqui.

# Fin de la parte PERL del script.
# ========================================================== #
```

Notemos lo que obtenemos al ejecutar nuestro script:

- ---
- bash $ ./bash-y-perl.sh
- Saludos desde la parte BASH del script.
- bash $ ./bash-y-perl.sh -x
- Saludos desde la parte PERL del script.
- ---

Lo importante es experimentar y encontrar un metodo propio.

Fuentes y contribuyentes del artículo

Conceptos e Historia de BASH *Fuente*: http://es.wikibooks.org/w/index.php?oldid=127252 *Contribuyentes*: Oleinad, 1 ediciones anónimas

El Manual de BASH Scripting Básico para Principiantes *Fuente*: http://es.wikibooks.org/w/index.php?oldid=153757 *Contribuyentes*: Ciencia Al Poder, Karekachile, LTSmash, Morza, Oleinad, Shooke, 7 ediciones anónimas

Sintaxis *Fuente*: http://es.wikibooks.org/w/index.php?oldid=171914 *Contribuyentes*: LTSmash, Oleinad, 2 ediciones anónimas

Hola Mundo en BASH *Fuente*: http://es.wikibooks.org/w/index.php?oldid=166071 *Contribuyentes*: LTSmash, Oleinad, 3 ediciones anónimas

Variables en BASH *Fuente*: http://es.wikibooks.org/w/index.php?oldid=127255 *Contribuyentes*: LTSmash, Oleinad

Llamando a una variable *Fuente*: http://es.wikibooks.org/w/index.php?oldid=127256 *Contribuyentes*: LTSmash, Oleinad

Generando un numero aleatorio y enviandolo a una variable *Fuente*: http://es.wikibooks.org/w/index.php?oldid=150402 *Contribuyentes*: Oleinad, Wutsje, 14 ediciones anónimas

Comandos básicos de una shell *Fuente*: http://es.wikibooks.org/w/index.php?oldid=148390 *Contribuyentes*: LTSmash, Oleinad, 3 ediciones anónimas

Condicionales y ciclos *Fuente*: http://es.wikibooks.org/w/index.php?oldid=127259 *Contribuyentes*: LTSmash, Oleinad

El básico (If-Then) *Fuente*: http://es.wikibooks.org/w/index.php?oldid=127260 *Contribuyentes*: LTSmash, Oleinad

El clon (Case-Esac) *Fuente*: http://es.wikibooks.org/w/index.php?oldid=127268 *Contribuyentes*: Oleinad, Omerta-ve, 1 ediciones anónimas

El clásico (For) *Fuente*: http://es.wikibooks.org/w/index.php?oldid=147367 *Contribuyentes*: LTSmash, Oleinad, 2 ediciones anónimas

El ciclo (While) *Fuente*: http://es.wikibooks.org/w/index.php?oldid=127270 *Contribuyentes*: Daniel perella, LTSmash, Oleinad

Funciones *Fuente*: http://es.wikibooks.org/w/index.php?oldid=130017 *Contribuyentes*: Oleinad, 2 ediciones anónimas

Opciones (parámetros) *Fuente*: http://es.wikibooks.org/w/index.php?oldid=127271 *Contribuyentes*: LTSmash, Oleinad, 1 ediciones anónimas

Compilar (ofuscar) BASH scripts con C - SHC *Fuente*: http://es.wikibooks.org/w/index.php?oldid=127272 *Contribuyentes*: LTSmash, Oleinad

Combinando BASH con otros lenguajes de scripting *Fuente*: http://es.wikibooks.org/w/index.php?oldid=85887 *Contribuyentes*: LTSmash

Fuentes de imagen, Licencias y contribuyentes

Archivo:50%.svg *Fuente*: http://es.wikibooks.org/w/index.php?title=Archivo:50%.svg *Licencia*: Public Domain *Contribuyentes*: Siebrand

Licencia