PROGRAMACION A NIVEL DEL SISTEMA OPERATIVO UNIX: BOURNE AGAIN SHELL

1. Esquema general para estudio de un lenguaje de programación

El siguiente es un esquema general que puede usarse para iniciarse en el estudio de cualquier lenguaje de programación. Con este esquema estudiaremos la programación en uno de los shells que posee UNIX: Bourne Again Shell (bash).

I. Características generales del lenguaje
- A. Tipo de lenguaje
  - *1. Interpretado*
  - *2. Compilado*
- B. Herramientas para editar, ejecutar y depurar programas
- C. Compatibilidad entre plataformas

II. Estructura del programa
- A. Forma y orden de los diferentes elementos que lo componen
- B. Estructura de una línea de programa

III. Tipos de datos y variables
- A. Tipificación de los datos
  - *1. Simples*
  - *2. Estructurados*
- B. Variables
  - *1. Nombres y tipos de datos*
  - *2. Variables predefinidas*

IV. Instrucciones básicas
- A. Asignación
- B. Decisión
- C. Repetición
- D. Salto

V. Funciones y procedimientos
- A. Estructura
- B. Paso de parámetros
- C. Invocación

VI. Manejo de archivos
- A. Tipos de archivos
- B. Definición dentro del programa
- C. Instrucciones básicas (Abrir, cerrar, leer, escribir)

VII. Librerías

2. Aplicación del esquema a BASH

I. Características generales del lenguaje

A. Tipo de lenguaje

BASH es un lenguaje INTERPRETADO, esto quiere decir que el código fuente se convierte a código objeto en tiempo de ejecución; en otras palabras, el programa fuente es el mismo ejecutable. Al carecer de compilador, los errores de sintaxis se detectan en tiempo de ejecución.

Para poder ejecutar un programa del shell (o script), se debe escribir el código en un archivo de texto y establecerle a este archivo el permiso de ejecución, por medio del comando `chmod`. Para especificarle al sistema operativo que debe emplear bash para interpretar el archivo, la primera línea del archivo debe ser la siguiente:

```
#!/bin/bash
```

La construcción `#!` se denomina *shebang*, y es empleada por UNIX para determinar el intérprete que se debe usar para correr un archivo en particular.

B. Herramientas para editar, ejecutar y depurar programas
Para editar los scripts de shell se emplea cualquier editor de texto de UNIX. Esto quiere decir que no se cuenta con ayudas para chequeo de la sintaxis, ya que estos editores no tienen ningún mecanismo para validación del código fuente.

No existe ningún tipo de debugger para este lenguaje.

C. Compatibilidad entre plataformas
Un script de shell corre con mínimas modificaciones en cualquier implementación de UNIX que posea BASH. Existe portabilidad hacia Windows, empleando el paquete de comandos UNIX/Linux conocido como Cygwin.

II. Estructura del programa
A. Forma y orden de los diferentes elementos que lo componen
Un programa shell posee la siguiente estructura:

```
función 1
   Código de la función 1
   ...

función 2
   Código de la función 2
   ...

función n
   Código de la función n

cuerpo principal del programa
...
```

B. Estructura de una línea de programa

Una línea de programa puede ser alguna de las seis cosas siguientes:

1. Una instrucción básica de shell
2. Un llamado a una función
3. Comentarios o documentación:

Toda línea que comience con un signo # en la primera columna se considera como un comentario.

4. Un comando del sistema operativo
5. El nombre de un script de shell
6. Un encadenamiento de comandos del sistema operativo. El encadenamiento se hace de alguna de las siguientes maneras:

a) `comando1;comando2; ... ; comandon`

b) `(comando1;comando2; ... ; comandon)` ó

`{comando1;comando2; ... ;comandon}`

c) `comando1 && comando2`

Este tipo de encadenamiento se llama **and-if**. Se basa en el hecho de que todo programa, cuando termina, devuelve un **status de salida** al ambiente que lo llamó. Dicho status de salida puede ser correcto o incorrecto.

En el caso del and-if, *la ejecución del segundo comando se condiciona a que la ejecución del primero devuelva un status de salida correcto*. En caso contrario, el segundo comando no se ejecuta.

Ejemplo:

```
rm archivox && echo "Archivo borrado"
```

Esta línea de comando trata de borrar el archivo. Si lo logra, despliega por pantalla un mensaje.

Si el and-if se combina con los paréntesis y con otros encadenamientos, de la siguiente manera:

```
(comando1; comando2; ...; comandon) && comandox
```

... se condiciona la ejecución del `comandox` al status de salida de todo el conjunto de comandos que va antes del `&&` (los paréntesis unifican el status de salida de todo el conjunto).

En caso de omitir los paréntesis:

```
comando1; comando2; ...; comandon && comandox
```

... la ejecución del `comandox` resulta condicionada únicamente al status de salida del último comando de la serie.

d) `comando1 || comando2`

Este tipo de encadenamiento se llama **xor-if**. Funciona también con base en el status de salida del primer comando, pero en este caso *se condiciona la ejecución del segundo a que el primero termine con status de salida incorrecto*. Si el primer comando termina con status correcto, el segundo no se ejecuta.
Ejemplo:

```
rm archivox 2> /dev/null || echo "Archivo no borrado"
```

Esta línea de comando trata de borrar el archivo. Si no lo logra, despliega un mensaje.

Todo lo dicho para el and-if en cuanto a combinación con otros encadenamientos y el uso de los paréntesis se aplica al xor-if.

III. Tipos de datos y variables
A. Tipificación de los datos
Bash maneja dos tipos de datos simples: La **cadena de caracteres** (o string) y los **enteros** (o datos numéricos)

En cuanto a datos estructurados, la única estructura soportada son los arreglos de una sola dimensión o **vectores**, tanto de cadenas de caracteres como de enteros.

B. Definición de variables
Se emplea la instrucción **`typeset`** para definir las variables y su respectivo tipo de datos. Bash no tiene una sección del código dedicada para definir tipos de datos, el sólo hecho de emplear el comando `typeset` crea las variables.

Para definir una variable tipo cadena: `typeset nombre_variable`
Para definir una variable tipo entero: `typeset -i nombre_variable`
o bien `integer nombre_variable`

Ejemplos:

```
typeset nombre
typeset -i contador
integer acumulador
```

- La definición de variables se puede hacer en cualquier lugar del programa. Sin embargo, para propósitos de claridad, se aconseja declararlas al principio del cuerpo principal del programa y al principio de cada función, para el caso de las variables locales.
- No es necesario declarar las variables tipo cadena de caracteres, pero se aconseja hacerlo.

Para definir un arreglo, se define como una variable de tipo simple con `typeset`; al asignar los elementos el shell varía dinámicamente el tamaño del arreglo.
Ejemplo: Definición de un arreglo para almacenar notas de 40 estudiantes

```
typeset -i nota
...

nota[1]=3
nota[2]=5
...
nota[40]=4
```

C. *Operaciones de asignación y acceso al contenido*
La asignación de variables se puede hacer de cuatro maneras distintas:

a) `variable=valor`
En este caso, el valor asignado a la variable puede ser una cadena de caracteres o un número, dependiendo de la variable.
Ejemplos:

```
nombre=pedro
nombre="Pedro Pérez"
edad=25
```

Nota: La sintaxis de bash exige que *no se dejen espacios* a ambos lados del signo igual.

b) `variable=$variablex`
Para acceder al contenido de otra variable (no sólo en el caso de la asignación, sino en todos los casos) se emplea el caracter de control `$`.
Ejemplo:

```
alumno=$nombre
```

Nótese que esto es distinto a decir `alumno=nombre` ! En el primer caso, la variable `alumno` queda con el mismo valor de la variable `nombre`; en el segundo, la variable `alumno` contendrá la cadena `"nombre"`

c) `variable=$(comando)` ó `variable=`comando``
En este caso, la salida del comando especificado entre los paréntesis o las comillas invertidas (backticks) no se lleva al stdout, sino que queda almacenada en la variable especificada.
Ejemplo:

```
linea=$(ls -l tareax)
linea=`ls -l tareax`
```

En cualquiera de los dos casos anteriores, en la variable `línea` quedará almacenada la información del directorio del archivo `tareax`.

d) `variable=((expresión))` ó `variable=[[expresión]]`

`(( ))` se emplea para delimitar operaciones aritméticas, `[[ ]]` se emplea para delimitar operaciones con cadenas. A la variable se le asigna el resultado de la operación o expresión.

Ejemplos:

```
suma=(( 4+5 ))
total=(( total + suma ))
```

Nota: bash exige que se deje un espacio en blanco después de `((` o `[[`, y un espacio en blanco antes de `))` o `]]`.

D. *Variables predefinidas*

BASH maneja algunas variables que conforman el entorno de trabajo (o environment). Algunas de estas variables son:

`HOME` Contiene la trayectoria al directorio home del usuario.

`TERM` Contiene el tipo de terminal en que se inició sesión.

`PATH` Contiene los nombres de los directorios donde se encuentran comandos ejecutables. Bash busca en estos directorios cuando se ejecuta un comando.

`PS1` Contiene el prompt de trabajo para el usuario (por omisión `$`)

`PS2` Contiene el segundo prompt de comando (por omisión >). Este segundo prompt se despliega en los siguientes casos:

- Cuando se emplean comillas sencillas en un comando, y algún par de ellas no se ha cerrado al pulsar ENTER.
- Cuando se presiona \ antes de ENTER, para partir un comando largo en dos o más líneas.
- Cuando se emplea un comando de shell como **`if`**, **`case`**, etc.

`?` Contiene el status de salida del último comando

Bash maneja lo que se denomina historia de comandos, es decir, es capaz de recordar un número determinado de comandos de los últimos que se han usado, y permite volverlos a llamar. A cada línea de comando que el usuario introduzca se le asigna un número único, y queda almacenada en un archivo. Mediante el uso de algunos comandos se pueden volver a usar esas líneas de comando:

`history` Despliega el listado de las líneas de comando almacenadas.

`r [# línea]` Ejecuta la línea especificada. Sin parámetros, ejecuta la última instrucción digitada.

`fc [# línea]` Lanza un editor de texto y permite modificar la línea de comando especificada. Al grabar y salir del editor, la línea se ejecuta con los cambios. Sin parámetros edita la última línea de comando digitada.

Existen variables del environment relacionadas con la historia de comandos:

`HISTFILE` Contiene el nombre del archivo donde se almacenan los últimos comandos ejecutados.

`HISTSIZE`	Determina cuántos comandos puede recordar la historia (es decir, cuántos se despliegan al invocar el comando `history`).
`FCEDIT`	Determina qué editor de textos se emplea al usar el comando `fc`.

Las variables predefinidas se pueden consultar de una en una usando el signo `$` como con las variables normales. Si se desea un listado de todas las variables que haya definidas en el environment, se usa el comando `env`.

El valor de una variable predefinida se puede cambiar de la siguiente manera:

- Se asigna un nuevo valor a la variable (`nombre_variable=nuevo_valor`)
- Se exporta la variable al environment (`export nombre_variable`)

Las variables no predefinidas también se pueden exportar al environment. La diferencia entre una variable exportada y una que no lo esté consiste en que *las variables exportadas se ven en los subprocesos que pueden ser invocados por el shell*, mientras que las variables que no se exporten no se ven. La razón de esto es que todo subproceso hereda de su proceso padre el environment.

Existen otras variables especiales cuyo contenido no se puede modificar, pero que son de gran utilidad para el programador:

`$?`	Contiene el status de salida del último comando ejecutado
`$$`	Contiene el número de identificación del proceso que se encuentra en ejecución
`$!`	Contiene el número de identificación del último proceso que se haya mandado a ejecutar en background

E. *Asignación y acceso a los elementos de un arreglo.*
Para asignar se emplea la sintaxis usual:

```
variable_arreglo[n]=valor | $variable
```

Para consultar los elementos de un arreglo se emplea la siguiente sintaxis:

`${variable[n]}`	… para acceder a un elemento
`${variable[*]}`	… para acceder a todo el arreglo

IV. Instrucciones básicas
Primero, se analizarán las expresiones para hacer operaciones con números y cadenas, y luego las expresiones lógicas y de comparación.

A. *Expresiones con variables*
a) Operaciones aritméticas
Las operaciones aritméticas se delimitan, como se vio en el apartado anterior, con doble paréntesis `(( ))`.
Los operadores aritméticos que se admiten son:

+ (Suma), - (Resta), * (Multiplicación), / (División entera) y % (Módulo).

Ejemplo: Para incrementar el valor de la variable `suma` en 20 se escribiría:

```
suma=(( suma+20 ))
```

b) Operaciones alfanuméricas

Antes de analizar las operaciones alfanuméricas admisibles, se analizarán los caracteres de acotamiento de los significados especiales. Como se ha podido ver, hay numerosos caracteres que tienen un significado especial. Ellos son:

```
* \ ? > < $ | [ ] " ( ) { } (espacio en blanco)
```

➔ En cualquier cadena de caracteres que esté encerrada entre comillas simples (' ') se ignorarán todos los caracteres especiales.
➔ En cualquier cadena de caracteres que esté encerrada entre comillas dobles (" ") se ignorarán todos los caracteres especiales, a excepción de $, ` ` y \
➔ El backslash (\) anula el significado del caracter especial que le sigue.
➔ Las comillas invertidas ` ` actúan como `$(comando)`. Ver el apartado anterior para más detalles.

Las operaciones alfanuméricas disponibles son:

- Conversión a mayúsculas: `typeset -u cadena`
- Conversión a minúsculas: `typeset -l cadena`

Ejemplo: Leer un nombre del teclado y pasarlo a mayúsculas

```
read nombre
nomb=`typeset -u nombre`
```

- Concatenación: Se especifica una cadena a continuación de la otra

Ejemplo:

```
read nombre; read apellido
nomcomp="$nombre $apellido"
```

c) Operaciones lógicas y de comparación

COMPARACIONES

	Comparación numérica ((expresión))	Comparación alfanumérica [[expresión]]
Igual	==	== ó -eq
Mayor que	>	> ó -gt
Menor que	<	< ó -lt
Mayor que o igual	>=	>= ó -ge
Menor que o igual	<=	<= ó -lt
Diferente de	!=	!= ó -ne

EXPRESIONES LÓGICAS
Permiten encadenar varias expresiones lógicas:

AND	`expr_comp_1 && expr_comp_2`
OR	`expr_comp_1 || expr_comp_2`
NOT	`!expr_comp`

Las expresiones lógicas se pueden hacer tan complejas como se quiera mediante el uso de los paréntesis.

B. *Entrada y salida básicas.*
a) Lectura de valores desde stdin: Se hace por medio del comando `read`.

```
read variable [? prompt]
```

El valor leído de la pantalla se almacena en la variable. Si se especifica el prompt, éste se desplegará en el momento de solicitar el valor.

b) Escritura de valores o variables a stdout: Se hace por medio de la instrucción `echo`.

```
echo [opciones] "texto" | valor | $variable
```

- La opción `-n` evita que el cursor salte a una línea nueva después de imprimir (evita el CR + LF).

Ejemplo:

```
echo -n codigo; read codigo
```

El comando `echo` admite secuencias de control tipo C dentro del texto de los mensajes, cuando se lo acompaña de la opción `-e`:

Secuencia	Significado
`\n`	Salto de línea
`\t`	Salto al siguiente tabulador
`\c`	Evita el salto de línea automático al fin del mensaje
`\f`	Alimentación de página
`\nnn`	(nnn valor en octal) Caracter especial o secuencia de control

Ejemplo: Para el caso de las terminales ANSI, la secuencia de control para activar el vídeo inverso es **`ESC[7m`**, y para volver a visualización normal se usa **`ESC[0m`**. Dado que el caracter del ESC no se puede obtener directamente por el teclado, se puede especificar como su equivalente octal, 033 (27 en decimal). En este ejemplo, el mensaje **Código** saldría en vídeo inverso:

```
echo -e "\033[7m Código: \033[0m\c"
```

C. *Instrucciones de decisión*

a) **Instrucción `if`**. Su sintaxis es la siguiente:

```
if comandos | instrucciones
then
    secuencia de instrucciones
[ else
    secuencia de instrucciones ]
fi
```

Si el status de salida del último comando, instrucción o expresión que se especifica a continuación de la directiva `if` es correcto, se ejecutarán las instrucciones que aparecen a continuación del `then`, de lo contrario, si existe la cláusula `else`, se ejecutarán las instrucciones que van a continuación de ella.

Debe recordarse que las expresiones lógicas devuelven VERDADERO ó FALSO, pero en el caso de usar alguna de ellas en un `if`, se tendrá que hacer que devuelvan status de salida correcto o incorrecto. Para convertir un valor de verdad en un status de salida se emplea la instrucción `test`.

Sintaxis de la instrucción `test`:

a) `(( expresión condicional numérica ))`
b) `[[ expresión condicional alfanumérica ]]`
c) `test expresión condicional alfanumérica`
d) `[[ -f | -w | -r | -x | -d | -s nombre de archivo ]]`
e) `[[ -z | -n nombre de cadena ]]`

Para el caso del numeral d), los significados de los condicionales son los que siguen:

- `-f` devuelve verdadero si el archivo nombrado a continuación existe.
- `-w` devuelve verdadero si el archivo existe y tiene permiso de escritura.
- `-r` devuelve verdadero si el archivo existe y tiene permiso de lectura.
- `-x` devuelve verdadero si el archivo existe y tiene permiso de ejecución.
- `-d` devuelve verdadero si el directorio nombrado existe.
- `-s` devuelve verdadero si el archivo especificado existe y tiene más de 0 bytes.

Para el caso del numeral e), los significados son los siguientes:

- `-z` devuelve verdadero si la variable de cadena existe y contiene la cadena nula.
- `-n` devuelve verdadero si la variable de cadena existe y no es nula.

Ejemplo:

```
suma=0
echo -n "Dato:"
read dato
…
```

```
(( suma = suma + dato ))
if (( suma >= 35 ))
then
 print "Suma =" $suma
else
 print "Total demasiado bajo"
fi
```

b) **Instrucción `case`**. Se utiliza para la selección múltiple. Su sintaxis es ésta:

```
case $variable in
    patrón1)               secuencia de instrucciones;;
    patrón2 | patrón3)     secuencia de instrucciones;;
    patrón4)               secuencia de instrucciones;;
esac
```

La instrucción verifica si la variable coincide con alguno de los patrones de la lista; se ejecuta la secuencia de instrucciones que siga al patrón que coincida primero.
Ejemplo:

```
echo -n "Opción:"
read opc
case $opc in
  "a" | "A")      bash administrar;;
  "c" | "C")      bash contabilizar;;
  "q" | "Q")      exit;;
  *)              echo "Opción inválida";;
esac
```

Nota: La secuencia de instrucciones de cada patrón del case puede ser tan larga como se desee. Lo importante es que la última instrucción de cada secuencia finalice con doble punto y coma (`;;`).

D. *Instrucciones de repetición*
a) **Instrucción `while`**
Su sintaxis es la siguiente:

```
while comandos | instrucciones
do
    secuencia de instrucciones
done
```

La secuencia de instrucciones que se encuentre entre el `do` y el `done` se ejecutará mientras que el último comando de la lista que va entre el `while` y el `do` retorne status de salida correcto. En el momento que dicho comando retorne status de salida incorrecto, el ciclo finalizará.

d) **Instrucción `until`**

Su sintaxis es la siguiente:

```
until comandos | instrucciones
do
    secuencia de instrucciones
done
```

La secuencia de instrucciones que se encuentre entre el `do` y el `done` se ejecutará hasta que el último comando de la lista que va entre el `while` y el `do` retorne status de salida correcto.

e) **Instrucción `for`**

Su sintaxis es la siguiente:

```
for variable in lista_valores
do
    secuencia de instrucciones
done
```

La forma de operación de la instrucción es la siguiente: El ciclo se ejecuta tantas veces como elementos haya en la lista, y para cada iteración, la variable de control toma el valor de cada uno de los elementos de la lista.

Ejemplo: Imprimir y luego borrar los archivos del directorio actual que terminen en "doc":

```
for arch in `ls *doc`
do
    lp $arch
    rm $arch
done
```

f) **Instrucción `break`**

Se emplea para interrumpir un ciclo repetitivo. Al procesarse esta orden, el control del programa pasa a la instrucción que sigue al `done` que indica el fin del ciclo.

g) **Instrucción `continue`**

Se emplea en ciclos repetitivos para interrumpir la iteración actual. Al procesarse esta orden, se ignora el resto de órdenes del ciclo repetitivo y se itera de nuevo.

V. FUNCIONES

Bash soporta únicamente el uso de funciones (No se permiten los procedimientos)

A. Definición de las funciones

- Las funciones deben ubicarse al principio del programa, antes del cuerpo principal de la aplicación.
- Existen dos maneras de definirlas:

a)
```
function nombre_función
{
      definición de variables locales (typeset)
      secuencia de instrucciones
}
```

b)
```
nombre_función()
{
      definición de variables locales (typeset)
      secuencia de instrucciones
}
```

Toda función retorna un status de salida, al igual que los comandos del sistema operativo. El status de salida retornado es siempre correcto, a menos que se especifique lo contrario con la instrucción **return**:

```
return(n)
```

... donde *n* es un valor entero positivo (0 para status de salida correcto y cualquier otro para status incorrecto).

B. Invocación

Para invocar una función, se escribe su nombre en una línea de programa sin los paréntesis.

Si se desea almacenar su status de salida, se puede usar la variable `$?` ó la sintaxis `variable=$(función)`. Sin embargo, debe tenerse en cuenta que esta última ejecuta la función dentro de un subshell, consumiendo por lo tanto más recursos de máquina.

C. Paso de parámetros

Cuando se invoca la función se pueden especificar a continuación de su nombre un grupo de valores separados por espacios. Estos valores se tratarán como parámetros.

Dentro del cuerpo de la función, el uso de los parámetros que se pasen se hace teniendo en cuenta la posición del parámetro. El primer parámetro se llamará `$1`, el segundo `$2`, y así sucesivamente hasta el noveno (`$9`).

Después del noveno parámetro, los demás se acceden como `${n}`, donde n es el número del parámetro. La variable `$#` contiene el número total de parámetros que se pasaron, y las variables `$@` y `$*` contienen la lista de todos los parámetros.

Nota: Este apartado acerca del paso de parámetros aplica también a la invocación de un script de shell desde el prompt del sistema operativo. (En la línea de comando se pueden especificar parámetros, y dentro del script se recibirían igual).

VI. MANEJO DE ARCHIVOS

A. Tipos de archivos

Bash soporta únicamente el uso de archivos tipo texto, donde cada línea se lee como un registro completo hasta el retorno de carro (No soporta delimitación de campos).

B. Definición dentro del programa

Dada la sencillez inherente al manejo de este tipo de archivos, no hace falta definir ninguna estructura para su manejo en el programa.

C. Instrucciones básicas

- **APERTURA**: Se hace con el comando `exec`:

```
exec {fd}{tipo_apertura} nombre_archivo
```

`fd` es el número de descriptor de archivo (tipo C) que se le va a asignar al archivo que se va a abrir. Se recomienda usar un número alto (por arriba de 6) porque los descriptores 0, 1 y 2 están ocupados por `stdin`, `stdout` y `stderr`, y puede haber otros ocupados por otros procesos. Debe recordarse que el descriptor identifica al archivo dentro del programa, y por tanto debe ser único.

El tipo o modalidad de apertura está dado por combinaciones de los signos de redirección:

Sólo lectura	`<`
Sólo escritura	`>`
Adición al final (Append)	`>>`
Lectura y escritura	`<>`

Ejemplos:

Para abrir el archivo `documento` en modalidad de sólo lectura:

```
exec 6< documento
```

Para abrir el archivo `notas` en modalidad de lectura y escritura:

```
exec 7<> notas
```

- **CLAUSURA**: Para cerrar un archivo también se emplea `exec`.

```
exec {fd}<&-
```

- **LECTURA DE UN REGISTRO**: Para leer de un archivo se emplea una variante de la orden `read`:

```
read -u{fd} variable
```

Ejemplo: Leer la siguiente línea del archivo **notas** de los ejemplos anteriores en la variable **línea**:

```
read -u7 linea
```

- **ESCRITURA DE UN REGISTRO**: Para leer de un archivo se emplea una variante de la orden `echo`:

```
echo variable | "texto" >&fd
```

Ejemplo: Escribir el contenido de la variable **frase** en el archivo **documento** de los ejemplos anteriores:

```
echo $frase >&6
```

A continuación, se muestra un ejemplo completo de script en shell con manejo de archivos. Este script despliega un archivo por páginas, de manera similar a como lo hace el comando **more**.

```
#!/bin/bash
################################################################
#                PROGRAMA DE DESPLIEGUE DE UN ARCHIVO
################################################################

################################################################
# DEFINICION DE VARIABLES:

typeset -i conta prueba
typeset linea arch
typeset -r tamano=22

################################################################
if (( $# == 1 ))
then
    arch=$1
else
    echo "Sintaxis del comando: desp.sh nomb_arch"
    exit
fi
if [[ -r $arch ]]
then
    exec 5< $arch
else
    echo "El archivo $arch no existe, o no tiene permiso de lectura
"
    exit
```

```
fi
conta=1
linea=$arch
clear
echo -r "despliegue del archivo  "
while  read -u5 linea
do
   echo "$linea"
   (( conta=conta+1 ))
   if (( conta > tamano ))
   then
      echo -e "\033[7;5m -- MAS -- \033[0m"
      read nada
      conta=1
   fi
done
```

VII. MANEJO DE LIBRERIAS

Bash no dispone de librerías de procedimientos elaborados previamente.