

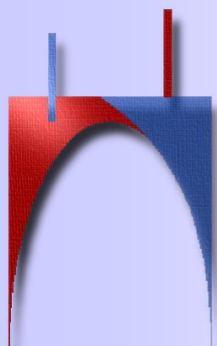
C.F.G.S.

Desarrollo de Aplicaciones Multiplataforma

Desarrollo de Aplicaciones Web

# UD 7

## Sistemas Operativos Multiusuario. Programación de Shell-Scripts



Instituto de Educación Secundaria  
**Santiago Hernández**  
*Informática*

# Introducción

## ➤ **Lenguajes de Programación**

- Cada shell tiene su propio lenguaje
- Los programas escritos para un shell no deben ejecutarse en otros
- Shell de Bourne (sh)
  - El más sencillo y con menos posibilidades.
  - Existe en todos los sistemas unix.
- Shell Bash
  - Todos los programas escritos para Bourne funcionan en él.
  - Permite el uso de funciones.
- Existen otros lenguajes de programación de scripts
  - perl



# Variables

# Variables

## ➤ Definición

- Como en cualquier lenguaje de programación:
  - Estructura con nombre que almacena un valor
- No se define el tipo de datos a almacenar.

## ➤ Entorno

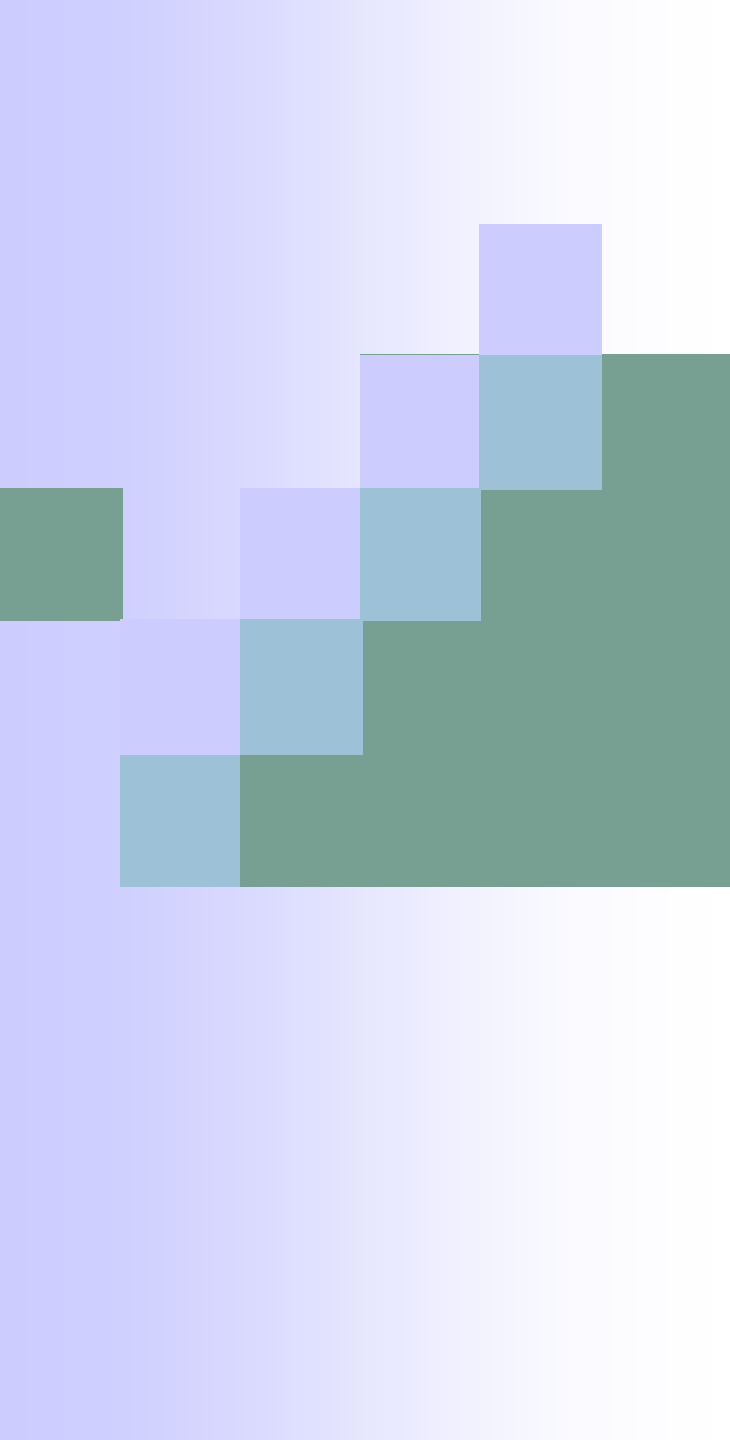
- Conjunto de variables del shell.
- **set**: muestra las variables.

## ➤ Variables del Sistema

- Ver man del shell para información completa.
- Diferentes según el shell utilizado.
- Algunas son asignadas automáticamente.
- Otras son utilizadas por el shell y modificables por el usuario.

## ➤ Variables de Usuario

- Nombre:
  - No puede comenzar con número.
  - Puede tener letras, números y caracteres especiales.
  - Distingue entre mayúsculas y minúsculas.
  - Suelen utilizarse en mayúsculas.
- Asignación de valor
  - `NOMBRE=valor`
  - `read NOMBRE` (para leer el valor de la entrada estándar. Muy útil en redireccionamientos)
- Utilización del valor
  - `${NOMBRE}`
  - `$NOMBRE`
- Borrado de la variable
  - `unset NOMBRE`
- Utilizaciones complejas
  - `${NOMBRE:-palabra}` (si existe `NOMBRE` y no tiene valor nulo se utiliza su valor, en caso contrario se utiliza "palabra")
  - Ver resto de posibilidades en man.



# Invocación de shell-scripts

# Invocación

## ➤ Formas de ejecución

- Dependiendo de que es lo que queramos hacer con los permisos, el entorno o el shell

	Necesita permiso de ejecución	No necesita permiso de ejecución
<b>Crea shell hijo</b>	<b>dir/shell-script</b>  al terminar cierra el shell	<b>sh shell-script</b> (u otro shell)  al terminar cierra el shell
<b>No crea shell hijo</b>	<b>exec dir/shell-script</b> al terminar cierra el shell  En solaris: No necesita permiso de ejecución Hay que poner el nombre completo o relativo del fichero	<b>. shell-script</b> al terminar no cierra el shell==>modifica el entorno  En solaris: Necesita permiso de ejecución Hay que poner el nombre del fichero si no está en el PATH

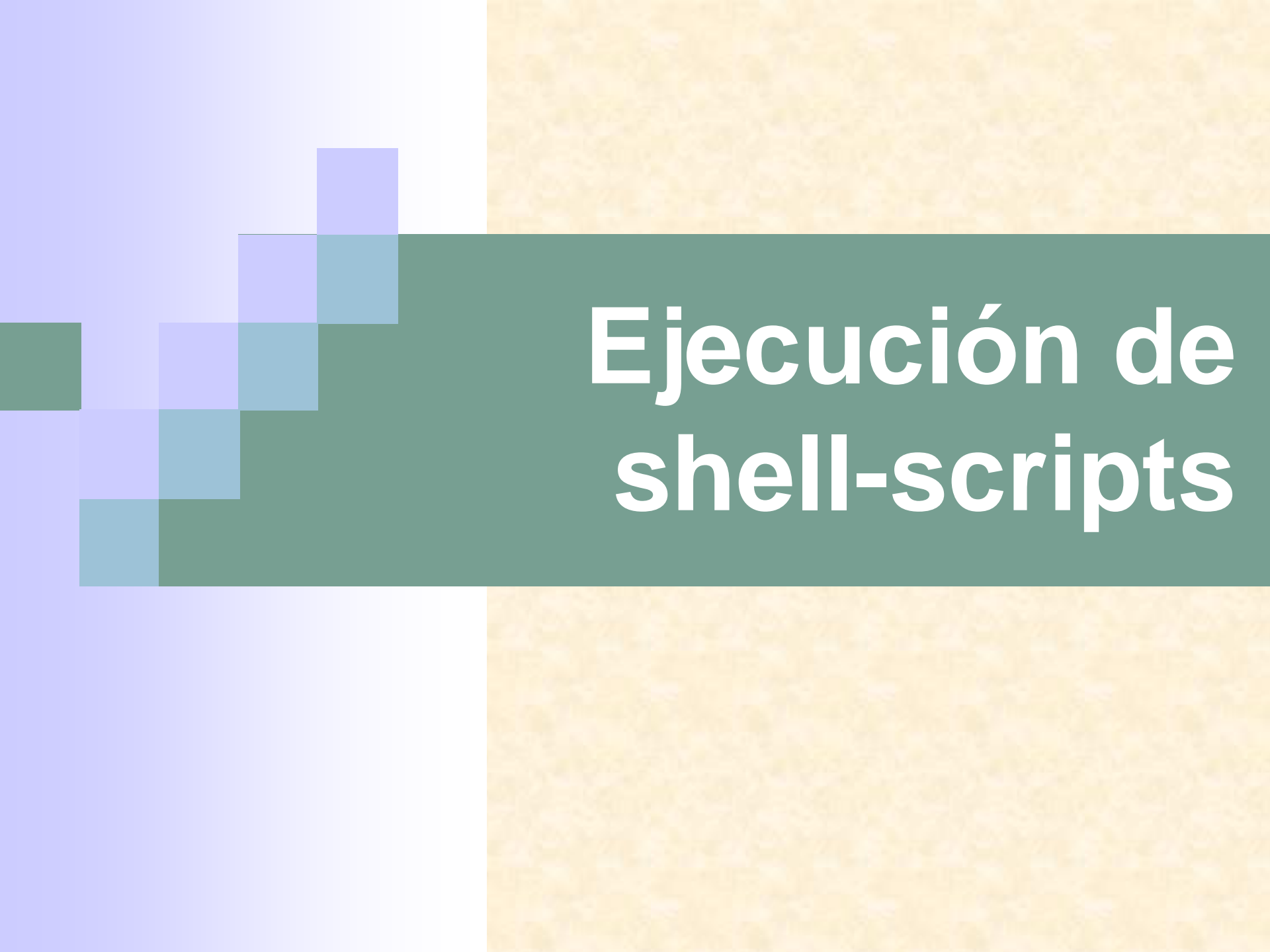
## ➤ Ejecución Normal

- Con permiso de ejecución y creando un shell hijo
- El nombre del directorio no es necesario ponerlo si el shell-script reside en un directorio apuntado por la variable PATH

## ➤ Variable PATH

- Definida en el login por los ficheros de configuración
- Estructura:
  - camino1[:camino2:camino3:...]
- Funcionamiento:
  - Al escribir un nombre en el shell se busca en camino1 si existe un fichero con ese nombre y sobre el que tengamos permiso de ejecución.
  - Si no existe allí, se mirará en camino2 y seguirá así hasta comprobar en todos los indicados
  - Si no existe en ninguno de ellos nos dará un mensaje de error
- Directorio Actual
  - El directorio actual no es un camino que se utilice para buscar un shell-script (ni ningún programa ejecutable)
  - Si se quiere invocar algo que esté en el directorio actual debe indicarse explícitamente:
    - ./shell-script





# Ejecución de shell-scripts

# Ejecución

## ➤ Lenguaje Interpretado

- El shell-script se ejecuta línea a línea.
- Primero se sustituyen todos los valores necesarios (variables, acotaciones...).
- Una vez sustituido se comprueba la sintaxis de la línea
- Si la sintaxis es correcta se ejecuta la línea y se pasa a la siguiente

## ➤ Errores

- Si la sintaxis es incorrecta se produce un error del sistema y se pasa a la línea siguiente.
- El error puede producirse después de la sustitución por lo que puede ser difícil de detectar.
- Hay que tener en cuenta que todas las líneas sin errores se han ejecutado (pueden producirse cambios en el estado del sistema que afecten a una segunda ejecución)



# Definición del entorno

# Definición del entorno

## ➤ Variables de exportación

- Variables exportables: las que se pasan al los shell hijos.
- **export**: muestra el conjunto de variables exportables.
- **export VARIABLE[=valor]**: designa las variables exportables (y les asigna valor si se le indica).

## ➤ Creación del entorno para el shell hijo

- `VARIABLE=valor [VAR2=valor2 ...] shell-script`



# Parámetros Posicionales

# Parámetros posicionales

## ➤ Definición

- Lista de valores que se pasan al shell-script en su invocación:
  - `shell-script [parámetro1 [parámetro2 ...]]`
- Se utilizan dentro del programa como variables de asignación automática.

## ➤ Uso

- **\$0**: nombre del script
- **\$1..\$9**: parámetros del 1 al 9
  - Si no hay parámetros toman valor nulo.
- **\${1}..\${n}**: parámetros del 1 al n
  - no pueden utilizarse en el shell de bourne.
- **shift [x]**: rotación de los parámetros en x posiciones
  - El parámetros \$0 no se modifica.
  - El parámetro \$n toma el valor del \${n+x}.
  - Los valores sustituidos se pierden.
  - Si no existe x se rota un parámetro.
- **set lista**: asigna los valores de la lista a los parámetros posicionales
  - Todos los valores existentes en los parámetros al invocar el shell-script se pierden.
  - Es la única forma de asignar valor a los parámetros posicionales.

## ➤ Variables asociadas

- Se modifican con la utilización de "shift" o "set *lista*"
- **\$#**: número de parámetros posicionales.
- **\$\***: lista de parámetros.
  - "\$\*" equivale a "\$1 \$2 \$3 ..."
- **\$@**: lista de parámetros.
  - "\$@" equivale a "\$1" "\$2" "\$3" ...



# Comando test



# comando test

- **test *expresión***
- **[ *expresión* ]**

- Evalúa *expresión* de forma lógica.
- El resultado de la evaluación se verá reflejado en la variable de estado:
  - Si verdadera: STATUS valdrá 0**
  - Si falsa: STATUS no valdrá 0**
- Si utilizamos variables en la expresión puede ser necesario acotarlas para evitar errores de sintaxis en la sustitución si la variable contiene un valor nulo
- Pueden unirse expresiones lógicas con **-a** y **-o** así como utilizar el símbolo de negación de la expresión (**!**). Si se quieren alterar las precedencias deberemos utilizar paréntesis (acotados).
- Dependiendo del shell podremos utilizar más o menos expresiones.

## ➤ Existencia de ficheros o directorios

- -e elemento si existe (no existe en todos los shell)
- -d directorio si existe
- -f fichero si existe y es ordinario
- -L fichero si existe y es enlace simbólico
  
- -r elemento
- -w elemento si existe y se tiene el permiso indicado sobre él
- -x elemento
  
- -k directorio
- -u fichero si existe y se tiene el permiso especial indicado sobre él
- -g elemento
  
- -s fichero si existe y tiene un tamaño mayor que cero

## ➤ Comparación de cadenas de caracteres

- cadena si no es nula
- -n cadena si no es nula
- -z cadena si es nula
  
- $c1 = c2$  si son iguales
- $c1 \neq c2$  si son diferentes

## ➤ Comparación de números enteros

- $n1 -eq n2$  si  $n1 = n2$
- $n1 -ne n2$  si  $n1 \neq n2$
  
- $n1 -gt n2$  si  $n1 > n2$
- $n1 -ge n2$  si  $n1 \geq n2$
  
- $n1 -lt n2$  si  $n1 < n2$
- $n1 -le n2$  si  $n1 \leq n2$



# Estructuras de programación

# Programación lineal

## ➤ Comentarios

- # Al principio de la línea indica al shell que esa línea es un comentario

## ➤ Definición del shell

- Si la primera línea del shell-script empieza por #! espera que después se le indique el shell con el que se interpretará el shell script.  
p.e.  
    #!/bin/sh  
    #!/bin/bash
- De esta manera el shell-script se ejecutará con unas normas independientemente del shell desde el que se ejecute.

# Estructura condicional: if (simple)

## ➤ Sintaxis

```
if lista1  
then  
    lista2  
fi
```

## ➤ Funcionamiento

- Se ejecuta *lista1*
- Si el resultado de la variable de estado (\$) tras la ejecución de *lista1* es igual a cero entonces se ejecuta *lista2*.

# Estructura condicional: if (completa)

## ➤ Sintaxis

```
if lista1
then
    lista2
else
    lista3
fi
```

## ➤ Funcionamiento

- Se ejecuta *lista1*
- Si el resultado de la variable de estado tras la ejecución de *lista1* es igual a cero entonces se ejecuta *lista2*.
- Si el resultado de la variable de estado tras la ejecución de *lista1* es distinto de cero entonces se ejecuta *lista3*.

# Estructura condicional: if (else anidado)

## ➤ Sintaxis

```
if lista1
then
    lista2
elif lista3
then
    lista4
[...]
fi
```

## ➤ Funcionamiento

- Se ejecuta *lista1*
- Si el resultado de la variable de estado tras la ejecución de *lista1* es igual a cero entonces se ejecuta *lista2*.
- Si el resultado de la variable de estado tras la ejecución de *lista1* es distinto de cero entonces se ejecuta *lista3*.
- Si el resultado de la variable de estado tras la ejecución de *lista3* es igual a cero entonces se ejecuta *lista4*
- Puede continuarse con más **elif**.
- Puede terminarse con un **else**.



# Estructura condicional por patrón: case

## ➤ Sintaxis

```
case palabra in  
  patrón)  
  lista  
;;  
...  
...  
esac
```

## ➤ Funcionamiento

- Se ejecuta la primera lista en que *palabra* coincide con *patrón* (lista es cualquier comando, pipeline o lista de ellos)
- Cuando se encuentra un *patrón* y ejecuta su lista asociada sale del **case**.
- En *patrón* pueden utilizarse metacaracteres (\*;?;[ ])
- Puede utilizarse un *patrón* \* al final para indicar "ninguno de los anteriores"

# Estructura repetitiva: while

## ➤ Sintaxis

```
while lista1
do
    lista2
done
```

## ➤ Funcionamiento

- Se ejecuta *lista1*.
- Si el resultado de la variable de estado tras la ejecución de *lista1* es igual a cero entonces se ejecuta *lista2*.
- Se vuelve a ejecutar *lista1*.
- Se realiza ese bucle **mientras que** la ejecución de *lista1* dé como resultado un valor de la variable de estado igual a cero.

# Estructura repetitiva: until

## ➤ Sintaxis

```
until lista1  
do  
    lista2  
done
```

## ➤ Funcionamiento

- Se ejecuta *lista1*.
- Si el resultado de la variable de estado tras la ejecución de *lista1* es distinto cero entonces se ejecuta *lista2*.
- Se vuelve a ejecutar *lista1*.
- Se realiza ese bucle **hasta que** que la ejecución de *lista1* dé como resultado un valor de la variable de estado igual a cero.

# Estructura repetitiva simple: for

## ➤ Sintaxis

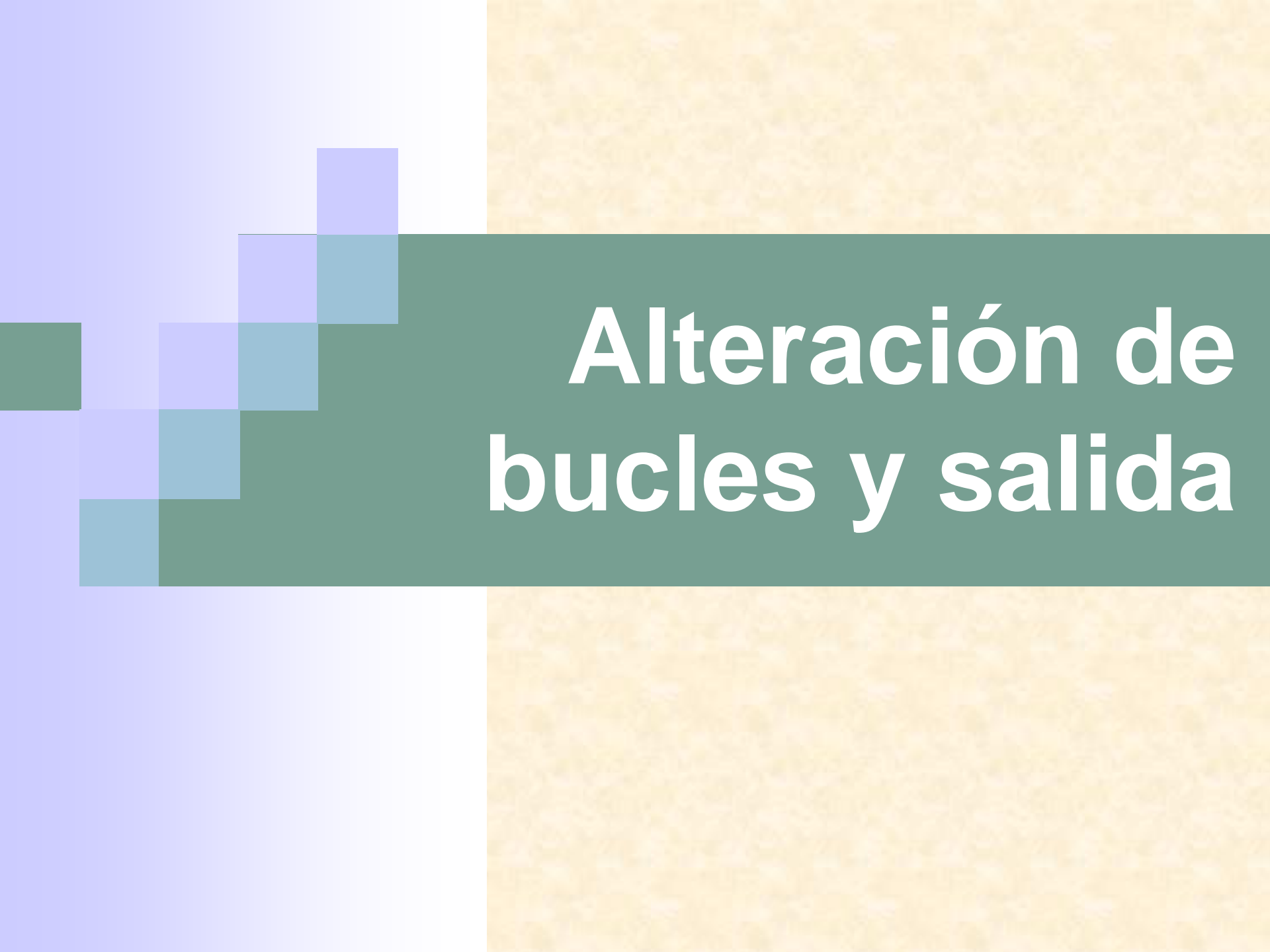
```
for nombre [in palabra .... ]  
do  
    lista  
done
```

## ➤ Funcionamiento

- Cada vez que se ejecuta el **for**, *nombre* toma el valor de la siguiente *palabra* tomada de la lista de in. Si no se pone el **in**, entonces el **for** se ejecuta una vez para cada uno de los parámetros posicionales que existen.
- La ejecución termina cuando no hay más palabras en la lista.

# Estructuras de programación en bash

- Todas las estructuras anteriores funcionan en el shell de bourne pero pueden ser aplicadas directamente en bash.
- En bash pueden ejecutarse en una línea utilizando ; en lugar de los saltos de línea.
- En bash existen otras estructuras de programación:
  - select
  - Segundo uso de la estructura for (similar al for de C)



# Alteración de bucles y salida

# Alteración de bucles y salida

## ➤ **continue**


- Corta la ejecución de la iteración actual del bucle.
- La ejecución del shell-script continúa en la última línea del bucle y se comenzará una nueva fase de iteración (si se cumple la condición adecuada)

## ➤ **break**

- Corta inmediatamente la ejecución de un bucle.
- La ejecución del shell-script continua en la línea siguiente al fin del bucle.

## ➤ **exit [n]**

- Corta la ejecución del shell-script.
- El valor de la variable de estado resultante será  $n$ .
- Si no se indica el valor de  $n$ , la variable de estado valdrá 0.
- Si no aparece la sentencia exit el shell-script terminará cuando se haya ejecutado la última sentencia (o se produzca un error de sintaxis) y el valor de salida será el que corresponda a ésta.



# Operaciones con números y cadenas de caracteres



# comando expr

## ➤ **expr *expresión***

- Evalúa *expresión* y deja el resultado en la salida estándar.
- Si queremos asignar el resultado a una variable:

**VAR=`expr *expresión*`**

- Pueden utilizarse literales o variables pero estas deben contener el valor adecuado a la operación.

## ➤ **Operaciones enteras**

- Operadores: +, -, \*(acotado), /, % (Siguen las reglas de precedencia que pueden alterarse con paréntesis (acotados). Hay que dejar espacios entre los operandos).
- Ejemplos:

**I=`expr \$I + 1`**

**expr 3 \\* \(\$A + \$B \)**

## ➤ operaciones con cadenas de caracteres

- **substr *cadena inicio longitud***

Extrae una subcadena de *cadena* empezando en el carácter *inicio*.  
La cadena empieza a numerarse desde 1.

- **length *cadena***

Devuelve la longitud de *cadena*.

- **index *cadena lista***

Devuelve la posición en la que encuentra el primer carácter de *lista* (*lista* es una cadena de caracteres sin espacios).  
Si no lo encuentra, devuelve 0.

# comando let (bash)

## ➤ **let *expresión\_aritmética***

- Evalúa expresiones aritméticas enteras.
- Si utilizamos variables no es necesario indicarlas con el prefijo "\$".
- Si utilizamos una variable no existente o con valor no numérico la utilizará con valor 0.
- Es un comando incorporado en bash por lo que no se puede utilizar para shell-scripts que utilicen el shell de bourne.
- Uso:

**let VARIABLE=expresión**

o bien

**let VARIABLEoperador**

## ➤ **Operaciones**

- Operadores: +, -, \*, /, % y \*\* (Siguen las reglas de precedencia que pueden alterarse con paréntesis. No hace falta dejar espacios entre los operandos).
- Incrementos y decrementos: VAR++, VAR--
- Ejemplos:  
let l=l+1  
let l++  
let l=5\*\*2\*(V+1)



# Uso de funciones en bash

# Funciones

## ➤ Definición de funciones

```
function nombre () {  
    comandos  
    ....  
}
```

- La definición de la función debe estar antes de su uso.
- Para dejar el shell-script más ordenado situaremos todas las definiciones de las funciones al principio del archivo.

## ➤ Utilización

- En cualquier línea del shell-script como si fuese un comando más:  
*nombre*  
línea\_cualquiera *`nombre`* resto\_de\_línea
- Se le pueden pasar parámetros que serán utilizados dentro de la función como parámetros posicionales (no afectará a los parámetros posicionales del shell-script)