# Uso de bash (parte 2)

# Tuberías

- Las **tuberías** (o ***pipes***), indicada con el metacaracter “**|**”, son un mecanismo que permite **conectar la salida (stdout)** de un comando con **la entrada (stdin) de otro**. Por ejemplo:

  ```
  > cat archivo.txt | wc -l
  ```

  La tubería captura el stdout de cat (el contenido de archivo.txt) y lo transfiere al stdin de “wc -l”, programa que cuenta las líneas del archivo.

- La redirección inversa, indicada con el metacaracter “**<**” permite también transferir el contenido de un archivo al stdin de un comando. Por ejemplo:

  ```
  > wc -l < archivo.txt
  ```

  **P:** Probar ingresar mediante un archivo la información solicitada por stdin del script interactivo desarrollado con anterioridad.

# Wildcards

Los wildcards (o comodines) son metacaracteres de Bash que se utilizan para hacer coincidir nombres de archivos o directorios en la CLI:

- **`*`**
  El asterisco se utiliza para hacer coincidir cualquier conjunto de caracteres.
  `> ls *.txt` Muestra todos los archivos que terminan con txt.

  P: Utilizar esto para contar la cantidad de archivos con extensiones .py, .sh, .
- **`?`**
  El signo de interrogación se utiliza para hacer coincidir un (1) carácter.
- **`[]`**
  Los corchetes especifican un conjunto de caracteres que pueden coincidir en esa posición. Por ejemplo, **`[aeiou]`** coincidiría con cualquier vocal, mientras que **`[0-9]`** coincidiría con cualquier dígito. Además, los corchetes pueden utilizarse para especificar rangos de caracteres, por ejemplo **`[a-z]`** coincidiría con cualquier letra minúscula.
  Los corchetes también son parte del conjunto de expresiones de búsqueda de patrones de texto conocidos como **expresiones regulares (regex)**.

# Condicionales

Los condicionales permiten controlar el flujo del código mediante decisiones basadas en variables y valores.
La sintaxis es:

```
if [ condicion ]; then
  # ejecutar si la condición es verdadera
else
  # ejecutar si la condición es falsa
fi
```

Donde la **condición** puede ser cualquier expresión que se pueda evaluar como **verdadera o falsa**, como una **comparación numérica**, una **comparación de strings** o una **evaluación booleana**. Por ejemplo:

**-gt**
Operador de comparación: **mayor que**

```
if [ $edad -gt 18 ]; then
  echo "Eres mayor de edad"
else
  echo "Eres menor de edad"
fi
```

# Condicionales

- También se pueden escribir en **una sola línea** utilizando el “**;**”. Por ejemplo:

```
> if [ $x == 1 ]; then echo "x es igual a 1"; fi
```

- Condicionales simples de una sola comparación se pueden escribir también con el &&:

```
> [ $x == 1 ] && echo "x es igual a 1"
```

- También es posible crear condicionales anidados con **elif**. Por ejemplo:

```
if [ $edad -lt 18 ]; then
  echo "Eres menor de edad"
elif [ $edad -lt 25 ]; then
  echo "Eres joven"
else
  echo "Eres mayor"
fi
```

**-lt**
Operador de comparación: **menor que**

# Condicionales

Los condicionales de Bash aceptan varios operadores de comparación y operaciones lógicas para evaluar expresiones.

**Operadores de comparación numérica (enteros):**

- **-eq**: igual que (también "**==**")
- **-ne**: no igual que
- **-lt**: menor que
- **-le**: menor o igual que
- **-gt**: mayor que
- **-ge**: mayor o igual que

**Operadores de comparación de cadenas:**

- **=**: igual que (también "**==**")
- **!=**: no igual que
- **<**: menor que (orden lexicográfico)
- **>**: mayor que (orden lexicográfico)
- **-z**: cadena vacía

Para **comparar cadenas** es recomendable usar **doble corchetes**, dado que esa sintaxis habilita el uso de wildcards y regex.

**Operadores lógicos:**

- **-a**: AND lógico
- **-o**: OR lógico
- **!**: NOT lógico

**P:** Escriba las tablas de verdad de las operaciones lógicas AND y OR, y sus NOT. Expanda con ejemplos dos operaciones de cada tabla.

# Condicionales

Las operaciones lógicas (y sus operadores) sirven para combinar condiciones. Por ejemplo:

```
if [ $a -gt 10 -a $b != "hello" ]; then
  # hacer algo si la expresión es verdadera
fi
```

El condicional sólo se cumple si a > 10 **y además** b es distinto “hello”.

# Condicionales

- En realidad, los corchetes son una sintaxis alternativa del comando **`test`**.

  Por ejemplo, las siguientes expresiones son equivalentes:

  `[ $var -gt 5 ]` y `test $var -gt 5`

  `[ $nom == "cris" ]` y `test $nom == "cris"`

- Además, test tiene opciones para evaluar existencia y propiedades de archivos. Por ejemplo:

  `test -f "datos.txt"` o también `[ -f "datos.txt" ]`

**P:** ¿Cómo se puede verificar que un archivo tenga permisos de escritura?

# Condicionales

**Algunos problemas de práctica:**

1. Cree un script que solicite al usuario un número entero y determine si es par o impar. (**Hint**: ver operación módulo `%`).
2. Cree un script que solicite al usuario una cadena y determine si contiene la letra "a".
3. Cree un script que solicite al usuario un número entero y determine si es positivo, negativo o cero.
4. Cree un script que solicite al usuario una cadena y determine si es una palabra palíndroma. Es decir, se lee igual al derecho y al revés (**Hint**: ver comando `rev`).
5. Cree un script que solicite al usuario dos números enteros y determine si el primero es divisible por el segundo.

# Variables: strings (cadenas)

Bash ofrece herramientas para trabajar con variables de strings que contienen múltiples subcadenas, incluyendo:

- **Extracción** de subcadena: Para extraer una subcadena de una variable de cadena, se puede utilizar la sintaxis ${variable:inicio:longitud}

```
> var="hola mundo"
> echo ${var:0:4}
```

- **Búsqueda y reemplazo**: Para buscar y reemplazar una subcadena en una variable de cadena, se puede utilizar la sintaxis ${variable/busqueda/reemplazo}

```
> echo ${var/mundo/profe}
```

- **Concatenación**: Para concatenar dos o más variables de cadena, se puede utilizar la sintaxis ${variable1}${variable2}
- **Longitud**: Para obtener la longitud de una variable de cadena, se puede utilizar la sintaxis ${#variable}

```
> echo ${#var}
```

# Variables: arrays

Un array es una variable que contiene una **lista de elementos**. Estos pueden ser de cualquier tipo, como strings, números o incluso otros arrays. La sintaxis es:

```
> nombre_array=(elemento1 elemento2 elemento3 ...)
```

Por ejemplo:
**> frutas=(manzana naranja plátano frutilla)**

- Para acceder a los elementos del array, se utilizan los **corchetes** con el **índice** de los elementos (**desde cero 0**). También se puede usar * o @ para indicar todos los elementos:
  ```
  > echo ${frutas[1]}
  > echo ${frutas[*]}
  ```
- Con el carácter **#** se puede acceder al número de elementos del array:
  ```
  > echo ${#frutas[@]}
  ```

  **P:** Probar que sucede si no se utilizan las llaves en estos casos.

# Variables: arrays

- También se pueden modificar o añadir elementos a un array existente. Por ejemplo, para añadir una fruta al final del array **`frutas`**:

  ```
  > frutas+=(mango)
  ```

- Para modificar un elemento particular simplemente se utiliza el índice correspondiente:

  ```
  > frutas[4]=papaya
  ```

- Para eliminar un elemento del array, se utiliza unset.

  ```
  > unset frutas[3]
  ```

  **P:** Probar acceder, modificar y eliminar elementos **fuera de la lista**.

# Expansión de llaves (brace expansion)

- **{ }**

  Se utilizan para crear un conjunto de elementos de manera rápida, generando una **lista de cadenas** a partir de una expresión que contiene llaves y comas.

  La expansión de llaves es una forma útil de generar listas de nombres de archivo o de argumentos para un comando. Por ejemplo, la expresión {a,b,c} se expande en tres cadenas distintas: "a", "b" y "c". De manera similar, {1..5} se expande en una lista de números del 1 al 5.

  Se utilizan a menudo junto con wildcards como el asterisco (*) para seleccionar archivos. Por ejemplo:

  `> ls *.{"txt","sh"}` Lista todos los archivos “txt” y “sh” de un directorio.

  También se pueden combinar. Por ejemplo:

  ```
  > echo {1..3}{a..b}
  ```

# Secuencia numérica

- **seq**

  Se utiliza para generar una secuencia de números con un incremento fijo. Por ejemplo:

  ```
  > seq 1 6
  ```
  Crea secuencia de 1 hasta 6 (inclusive)

  ```
  > seq 1 2 6
  ```
  Crea secuencia de 1 a 6 con un incremento de 2

  La opción -w fuerza a que los números generados tengan el mismo número de caracteres (agregando ceros si es necesario). Por ejemplo:

  ```
  > seq -w 8 11
  ```

  Esta opción es muy útil para trabajar con colecciones de archivos o directorios: genera nombres de tamaño homogéneo.

# Bucles: **while**

Se utiliza para **repetir** un conjunto de comandos **mientras se cumpla** una determinada condición. La sintaxis general es:

```
while [condición]
do
     # comandos a ejecutar mientras se cumpla la condición
done
```

Por ejemplo, se puede realizar una iteración condicionada a una variable numérica:

```
i=1
while [ $i -le 5 ]
do
    echo $i
    i=$(($i+1))
done
```

# Bucles: **while**

- También se puede utilizar para **iterar** sobre los **elementos de un array**.

```
nombres=("Juan" "María" "Pedro" "Laura")
i=0
while [ $i -lt ${#nombres[@]} ]
do
    echo ${nombres[$i]}
    i=$(($i+1))
done
```

En general, **`while`** es muy útil cuando no sabemos de antemano cuántas veces se repetirá el conjunto de comandos.

Tener precaución con la generación de **bucles infinitos**

# Bucles: **for**

Se utiliza para iterar sobre una **lista de elementos** y ejecutar un conjunto de comandos para **cada elemento** de la lista. Su sintaxis es:

```
for VARIABLE in LISTA
do
     # comandos a ejecutar en cada iteración
done
```

**VARIABLE**: es la variable que se utiliza para almacenar cada elemento de la lista en cada iteración.

**LISTA**: es la lista de elementos sobre la que se va a iterar

# Bucles: for

### Ejemplos

- Imprimir los números del 1 al 5:

```
for i in 1 2 3 4 5
do
  echo $i
done
```

- También se puede utilizar las expansiones de llaves:

```
for i in {1..5}
do
  echo $i
done
```

- También es posible iterar sobre los elementos de un array en Bash. Por ejemplo:

```
nombres=("Juan" "Ana" "Pedro")

for nombre in "${nombres[@]}"
do
  echo $nombre
done
```

# Bucles con iteraciones condicionadas

Agregando un condicional dentro de un bucle, es posible controlar el flujo de la iteración mediante los comandos **`break`** y **`continue`**.

- **`break`**
  Se utiliza para **terminar** el bucle **inmediatamente** si se cumple la condición (aunque queden elementos restantes).
  Por ejemplo:

```
for i in {1..10}
do
  if [ $i -eq 6 ]
  then
    break
  fi
  echo $i
done
```

- **continue**
  Se utiliza para **saltar** la iteración actual y **continuar** con la siguiente iteración del bucle.
  Por ejemplo:

```
for i in {1..10}
do
  if [ $(($i % 2)) -eq 0 ]
  then
    continue
  fi
  echo $i
done
```

**P:** Verificar qué sucede cuando la condición está precedida por otros comandos.

# Expansiones de comandos

- **`$(`**`comando`**`)`**
  Se utilizan para tomar la **salida** de un comando y utilizarla como **entrada** en otro comando o asignarla a una variable.

  Ejemplos:
  Obtener la lista de archivos en un directorio y guardarla en una variable:
  ```
  > files=$(ls /ruta/al/directorio)
  ```

  Ejecución de comandos dentro de los argumentos de otros:
  ```
  > echo "La fecha actual es: $(date)"
  ```

  Definir elementos de iteración de un bucle:
  ```
  > for i in $(seq 1 10) ; do echo $i ; done
  ```

# Problemas de práctica

1. Escriba un script Bash que, dado un nombre y un número entero N como argumentos de entrada, cree un conjunto de directorios /NOMBRE/datos_K, donde NOMBRE es igual al nombre ingresado, y K es el número del directorio creado desde cero hasta (N-1).
2. Cree un script que reciba como argumento un directorio y muestre en pantalla los nombres de todos los archivos con extensión ".txt".
3. Cree un script Bash que, dada una cadena como argumento de entrada, liste todos los directorios del lugar (no archivos) cuyo nombre contenga dicha cadena.
4. Cree un script Bash que, dada una cadena como argumento de entrada, liste todos los archivos del lugar (no directorios) cuyo nombre contenga dicha cadena y tengan permiso de escritura.
5. Crea un script que solicite al usuario una cadena y muestre en pantalla cada letra en una línea separada.
6. Cree una nueva versión del script interactivo de datos personales pero agregando la opción “categoría”. Esta categoría deberá almacenar los datos en archivos dentro de directorios datos/CAT/, siendo CAT la categoría ingresada. El subdirectorio debe ser creado automáticamente si no existe.

# Funciones

Sirven para definir **conjuntos de instrucciones** de forma modular, que se pueden llamar nuevamente luego, dentro del mismo ambiente de ejecución.

- La sintaxis para **declarar** una función es la siguiente:

```
mi_funcion(){
    # instrucciones, por ejemplo:
    echo "Hola desde la función"
}
```

  Lo cual también se puede hacer en una línea:

```
mi_funcion () { echo "Hola" ; }
```

- Para ejecutar (llamar) la función desde otro lugar del script, solo se debe utilizar el nombre de la misma.
- También se puede utilizar la palabra clave `function` al inicio de la declaración.

# Funciones: visibilidad de variables

- Las variables definidas en **el ambiente principal** (main) del script y **dentro de sus funciones** son **globales para el programa**, es decir son accesibles por todas las funciones definidas en el mismo archivo.
- Para que una variable solo sea visible dentro de la función en la que se declara, se debe anteponer la instrucción `local` en su declaración. Así, no será conocida ni accesible por el resto del script:

```
mi_funcion(){
    varg=0 # variable global (script)
    local varl=1  # variable local (función)
}
```

**P:** Cree un programa en Bash que contenga una función. Declare, muestre e intente modificar variables tanto dentro como fuera de la función definida para comprobar la visibilidad y acceso de cada una desde los diferentes lugares del script.
¿Una modificación de una variable local en una función es recordada en sus siguientes llamados?

# Funciones: parámetros de entrada

- Las funciones pueden recibir **parámetros de entrada**, los cuales son accesibles mediante las variables **$1**, **$2**, **$3**,..., **$9,** y también **$@** (lista de todos los parámetros) y **$#** (números de parámetros).

  El llamado de una función con parámetros se realiza separando estos últimos con un espacio. Por ejemplo:

```
mi_funcion param1 param2 param3 ...
```

**P:** Cree un programa con una función que cuente el número de parámetros recibidos, indique el resultado y liste los mismos junto a su índice correspondiente. Su script debe solicitar estos parámetros uno a uno al usuario.

¿Cómo se puede acceder, desde una función, a los parámetros utilizados en la ejecución del script correspondiente?

# Funciones: retorno

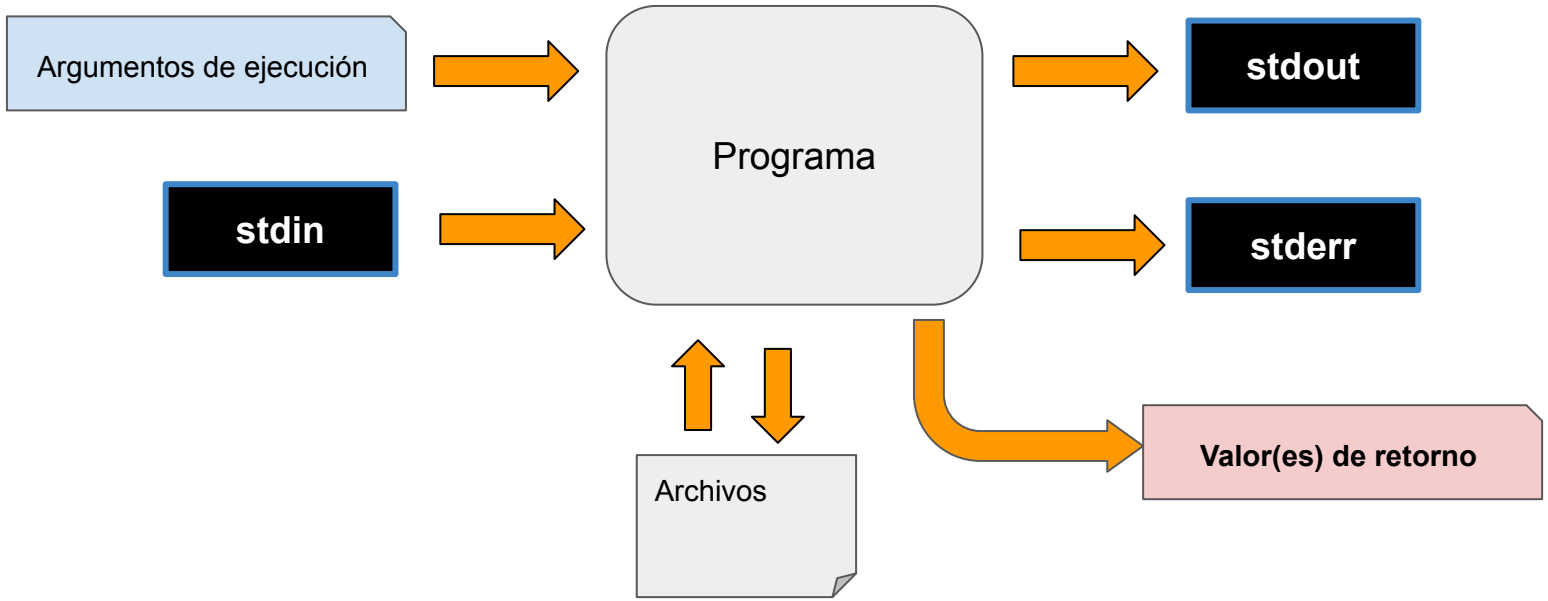
El **retorno** es una vía de comunicación común en los programas, tanto para sus funciones internas como en sus ejecuciones globales.

- El **retorno** de una función es una **vía de comunicación de información** de salida de la función correspondiente **hacia el entorno que la ejecutó**.
- Se realiza una única vez y está asociado al **término de la ejecución**.
- Puede ser almacenado y utilizado por el ambiente que recibe el valor de retorno.
- Se define utilizando la instrucción `return` seguida del valor que se desea retornar. Por ejemplo:

```
mi_funcion () { echo "Hola" ; return 0; }
```

- **Bash solo permite valores de retorno enteros entre 0 y 255,** dado que se suele usar para transmitir el estado de la ejecución (0: término correcto, >0 indicación de errores). Estos son almacenados automáticamente en la variable **$?**.

# Posibles input/output (I/O) de un software.



# Problemas de práctica

- Escriba un programa que solo acepte dos argumentos: un número y una cadena, y que se interrumpa si lo anterior no se cumple (arrojando un mensaje de error). Dentro debe tener una función que imprima la cadena tantas veces como el número indicado.
- Cree un script que solicite al usuario palabras (cadenas) por stdin hasta que se ingrese la letra “q” o “Q”. Luego, debe mostrar una lista con las palabras ingresadas que no contengan mayúsculas.
- Cree un script que solicite al usuario un número entero positivo N y luego imprima los primeros N números de la secuencia de Fibonacci.
- Cree un script con una función que tome como parámetro un número entero y devuelva el factorial de ese número. El script le debe solicitar el número al usuario por stdin si este no lo ingresa como parámetro.