Shell scripting

Explicación de práctica 3

Introducción a los Sistemas Operativos

Facultad de Informática
Universidad Nacional de La Plata

2023

Agenda

Agenda

¿Qué es una shell?

- Intérprete de comandos
- Interactivo
- En sistemas operativos *nix es configurable
- Proveen estructuras de control que permiten programar *shell scripts*

¿Qué puedo hacer con *shell scripts*?

- Automatización de tareas
- Aplicaciones interactivas
- Aplicaciones con interfaz gráfica (con el comando *zenity*, por ejemplo)

¿Qué es una shell?

- Intérprete de comandos
- Interactivo
- En sistemas operativos *nix es configurable
- Proveen estructuras de control que permiten programar *shell scripts*

¿Qué puedo hacer con *shell scripts*?

- Automatización de tareas
- Aplicaciones interactivas
- Aplicaciones con interfaz gráfica (con el comando *zenity*, por ejemplo)

Existen muchas *shells*. Sus diferencias consisten principalmente en sintaxis. A continuación se listan las más utilizadas:

- `sh`: *Shell* por defecto en Unix.
- `bash`: Cómoda, instalada por defecto en la mayoría de las distribuciones.
- `dash`: Eficiente, parcialmente compatible con `bash`.
- `csh`: Sintaxis incompatible con `bash`/`dash`.
- Otros...

Tip

En la materia utilizaremos `bash`.

Existen muchas *shells*. Sus diferencias consisten principalmente en sintaxis. A continuación se listan las más utilizadas:

- `sh`: *Shell* por defecto en Unix.
- `bash`: Cómoda, instalada por defecto en la mayoría de las distribuciones.
- `dash`: Eficiente, parcialmente compatible con `bash`.
- `csh`: Sintaxis incompatible con `bash`/`dash`.
- Otros...

Tip

En la materia utilizaremos `bash`.

¿Por qué *shell script* y no C, o Java, o Python?

- Práctico para manejar archivos
- Extremadamente simple para crear procesos y manipular sus salidas
- Independiente de la plataforma (a diferencia de C)
- Funciona en cualquier sistema operativo de tipo *nix (distribución GNU/Linux, Mac OS X, etc.)
- Se puede probar en el intérprete interactivo (a diferencia de C y Java)

Elementos del lenguaje

- Instrucciones: comandos
  - Internos o *built-in* (`help` para verlos)
  - Externos (archivos separados `man comando`)
- Redirecciones y *pipes*
- Comentarios que empiezan con #
- Estructuras de control
  - `if`
  - `while`
  - `for` (2 tipos)
  - `case`
- Variables
  - *Strings*
  - Arreglos ()
- Funciones

Agenda

Repaso de algunos comandos útiles

- Imprimir el contenido de un archivo

```
cat archivo
```

- Imprimir texto

```
echo "Hola mundo"
```

- Leer una línea desde entrada estándar en la variable `var`

```
read var
```

- Quedarme con la primer columna de un texto separado por `:` desde entrada estándar

```
cut -d: -f1
```

- Contar la cantidad de líneas que se leen desde entrada estándar

```
wc -l
```

Repaso de algunos comandos útiles

- Buscar todos los archivos que contengan la cadena **pepe** en el directorio `/tmp`

```
grep pepe /tmp/*
```

- Buscar todos los archivos dentro del *home* del usuario, cuyo nombre termine en `.doc`

```
find $HOME -name "*.doc"
```

- Buscar todos los archivos dentro del directorio actual que sean enlaces simbólicos

```
find -type l
```

- **Empaquetado:** Se unen varíos archivos en uno solo (`tar`)

```
tar -cvf archivo.tar archivo1 archivo2 archivo 3
tar -xvf archivo.tar
```

- **Compresión:** Se reduce el tamaño de un archivo (`gzip`/`bzip2`/etc.)

```
gzip archivo.tar # Genera archivo.tar.gz comprimido
gzip -d archivo.tar.gz # Descomprime archivo.tar
```

- El comando `tar` puede invocar a `gzip` por nosotros (argumento "z"):

```
tar -cvzf archivo.tar.gz arch1 arch2 arch3
tar -xvzf archivo.tar.gz
```

Redirecciones y pipes: stdin, stdout, stderr

Los procesos (programas en ejecución) normalmente cuentan con 3 "archivos" abiertos.

- **stdin:** Entrada estándar, normalmente el teclado.
- **stdout:** Salida estándar, normalmente el monitor.
- **stderr:** Error estándar, normalmente la salida estándar.

Se identifican en el S.O. con un número, el *file descriptor* (descriptor de archivo):

0. Entrada estándar
1. Salida estándar
2. Error estándar

Sintaxis básica:

```
comando > archivo
comando >> archivo
```

- > Redirección destructiva:
  - Si *archivo* no existe, se crea.
  - Si *archivo* existe, sobreescribe.
- >> Redirección no destructiva:
  - Si *archivo* no existe, se crea.
  - Si *archivo* existe, agrega al final.

¿Qué hace?:

```
cd
ls >> /tmp/lista.txt
cd /tmp
ls >> /tmp/lista.txt
```

¿Y si cambiamos >> por >?

Sintaxis básica:

```
comando > archivo
comando >> archivo
```

- > Redirección destructiva:
  - Si *archivo* no existe, se crea.
  - Si *archivo* existe, sobreescribe.
- >> Redirección no destructiva:
  - Si *archivo* no existe, se crea.
  - Si *archivo* existe, agrega al final.

¿Qué hace?:

```
cd
ls >> /tmp/lista.txt
cd /tmp
ls >> /tmp/lista.txt
```

¿Y si cambiamos >> por >?

Sintaxis básica:

```
comando 2> archivo
comando 2>> archivo
comando < archivo
```

- **2>** y **2>>** Redirigen el error estándar
- < Hace que *archivo* sea la entrada de *comando*.
  En otras palabras cuando *comando* intente leer entrada del teclado, en realidad, va a leer el contenido de *archivo*.

Sintaxis básica:

```
comando | comando2 | comando3
```

- Conectan la salida de un comando con la entrada de otro.
- Indispensables para hacer programas potentes en *shell script*.

Ejemplos:

```
cat archivo | tr a-z A-Z
cat archivo | grep hola | cut -d, -f1
cat /etc/passwd | cut -d: -f1 | grep a | wc -l
cat /etc/passwd | cut -d: -f7 | sort | uniq > res.txt
```

Variables

- `bash` soporta *strings* y *arrays*
- Los nombres son *case sensitive*
- Para crear una variable:

```
NOMBRE="pepe" # SIN espacios alrededor del =
```

- Para accederla se usa $:

```
echo $NOMBRE
```

- Para evitar ambiguedades se pueden usar llaves:

```
# Esto no accede a $NOMBRE
echo $NOMBREesto_no_es_parte_de_la_variable
# Esto sí
echo ${NOMBRE}esto_no_es_parte_de_la_variable
```

Variables

- `bash` soporta *strings* y *arrays*
- Los nombres son *case sensitive*
- Para crear una variable:

```
NOMBRE="pepe" # SIN espacios alrededor del =
```

- Para accederla se usa $:

```
echo $NOMBRE
```

- Para evitar ambiguedades se pueden usar llaves:

```
# Esto no accede a $NOMBRE
echo $NOMBREesto_no_es_parte_de_la_variable
# Esto sí
echo ${NOMBRE}esto_no_es_parte_de_la_variable
```

Variables

- `bash` soporta *strings* y *arrays*
- Los nombres son *case sensitive*
- Para crear una variable:

```
NOMBRE="pepe" # SIN espacios alrededor del =
```

- Para accederla se usa $:

```
echo $NOMBRE
```

- Para evitar ambiguedades se pueden usar llaves:

```
# Esto no accede a $NOMBRE
echo $NOMBREesto_no_es_parte_de_la_variable
# Esto sí
echo ${NOMBRE}esto_no_es_parte_de_la_variable
```

Variables

- `bash` soporta *strings* y *arrays*
- Los nombres son *case sensitive*
- Para crear una variable:

```
NOMBRE="pepe" # SIN espacios alrededor del =
```

- Para accederla se usa $:

```
echo $NOMBRE
```

- Para evitar ambiguedades se pueden usar llaves:

```
# Esto no accede a $NOMBRE
echo $NOMBREesto_no_es_parte_de_la_variable
# Esto sí
echo ${NOMBRE}esto_no_es_parte_de_la_variable
```

- `bash` soporta *strings* y *arrays*
- Los nombres son *case sensitive*
- Para crear una variable:

```
NOMBRE="pepe" # SIN espacios alrededor del =
```

- Para accederla se usa $:

```
echo $NOMBRE
```

- Para evitar ambiguedades se pueden usar llaves:

```
# Esto no accede a $NOMBRE
echo $NOMBREesto_no_es_parte_de_la_variable
# Esto sí
echo ${NOMBRE}esto_no_es_parte_de_la_variable
```

Los nombres de las variables pueden contener mayúsculas, minúsculas, números y el símbolo _ (underscore), pero no pueden empezar con un número.

```
NOMBRE="Fulano De Tal"
facultad=Informatica
carrera_1="Licenciatura en Sistemas"
carrera_2="Licenciatura en Informatica"
echo El alumno $NOMBRE de la Facultad de $facultad cursa
     $carrera_1 y $carrera_2
# imprime:
#  El alumno Fulano De Tal de la Facultad  de
#  Informática cursa Licenciatura en Sistemas
#  y Licenciatura en Informatica
```

Variables: Ejemplo 2

```
nombre=Carlos
echo "Hola $nombre" # Hola Carlos
echo Hola ${nombre} # Hola Carlos
nombre=5
echo "Hola $nombre" # Hola 5
```

- *"Bashismo"*
- Creación:

```
arreglo_a=() # Se crea vacío
arreglo_b=(1 2 3 5 8 13 21) # Inicializado
```

- Asignación de un valor en una posición concreta:

```
arreglo_b[2]=spam
```

- Acceso a un valor del arreglo (En este caso las llaves no son opcionales):

```
echo ${arreglo_b[2]}
copia=${arreglo_b[2]}
```

- Acceso a todos los valores del arreglo:

```
echo ${arreglo[@]} # o bien ${arreglo[*]}
```

- Tamaño del arreglo:

```
${#arreglo[@]} # o bien ${#arreglo[*]}
```

- Borrado de un elemento (reduce el tamaño del arreglo pero no elimina la posición, solamente la deja vacía):

```
unset arreglo[2]
```

- Los índices en los arreglos comienzan en 0

Variables: Ejemplo de arreglos

```
#!/bin/bash

arreglo=(1 2 3 5 8 13 21)
arreglo[2]=spam
echo "El primer elemento es ${arreglo[0]}"
echo "El tercer elemento es ${arreglo[2]}"
echo "La longitud: ${#arreglo[*]}"
echo "Todos sus elementos: ${arreglo[*]}"
```

- No hacen falta, a menos que:
  - el *string* tenga espacios.
  - que sea una variable cuyo contenido pueda tener espacios.
  - son importantes en las condiciones de los `if`, `while`, etc...
- Tipos de comillas
  - **“Comillas dobles”:**

```
var='variables'
echo "Permiten usar $var"
echo "Y resultados de comandos $(ls)"
```

  - **‘Comillas simples’:**

```
echo 'No permiten usar $var'
echo 'Tampoco resultados de comandos $(ls)'
```

Comillas: Ejemplos

```
Un ejemplo:
variable="un texto de varias palabras"
variable_2=UnaSolaPalabra
echo "Podemos leer $variable"
echo 'No podemos leer $variable'
variable_3="Asi concateno $variable_2 a otro string"
```

- ¿Qué se imprime en cada caso?
- ¿Cuál es el valor de variable_3?

- Permite utilizar la salida de un comando como si fuese una cadena de texto normal.
- Permite guardarlo en variables o utilizarlos directamente.
- Se la puede utilizar de dos formas, cada una con distintas reglas:

```
$(comando_valido)
`comando_valido`
```

Nota: La primer forma resulta más clara y posee reglas de anidamiento de comandos más sencillas.

- Ejemplo:

```
arch="$(ls)" # == arch="`ls`" == arch=`ls`
mis_archivos="$(ls /home/$(whoami))"
```

Agenda

¿Cómo hacer un script?

- Crear un archivo con cualquier editor de texto.
- Indicar el intérprete (opcional) en la primer línea con:

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

  Está línea, denominada *shebang*, especifica el intérprete que se utilizará para ejecutar *script*. Si no se especifica, se utiliza el intérprete por defecto del usuario.
    - ¿Qué problemas puede traer esto?
- Escribir una serie de comandos en el archivo, de la misma manera que los escribiriamos en la terminal.
- ¿Guardar el script con terminación *.sh*?
- ¿Permisos de ejecución?

¿Cómo hacer un script?

- Crear un archivo con cualquier editor de texto.
- Indicar el intérprete (opcional) en la primer línea con:

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

  Está línea, denominada *shebang*, especifica el intérprete que se utilizará para ejecutar *script*. Si no se especifica, se utiliza el intérprete por defecto del usuario.
    - ¿Qué problemas puede traer esto?
- Escribir una serie de comandos en el archivo, de la misma manera que los escribiriamos en la terminal.
- ¿Guardar el script con terminación *.sh*?
- ¿Permisos de ejecución?

¿Cómo hacer un script?

- Crear un archivo con cualquier editor de texto.
- Indicar el intérprete (opcional) en la primer línea con:

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

  Está línea, denominada *shebang*, especifica el intérprete que se utilizará para ejecutar *script*. Si no se especifica, se utiliza el intérprete por defecto del usuario.
  - ¿Qué problemas puede traer esto?
- Escribir una serie de comandos en el archivo, de la misma manera que los escribiriamos en la terminal.
- ¿Guardar el script con terminación *.sh*?
- ¿Permisos de ejecución?

¿Cómo hacer un script?

- Crear un archivo con cualquier editor de texto.
- Indicar el intérprete (opcional) en la primer línea con:

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

  Está línea, denominada *shebang*, especifica el intérprete que se utilizará para ejecutar *script*. Si no se especifica, se utiliza el intérprete por defecto del usuario.
  - ¿Qué problemas puede traer esto?
- Escribir una serie de comandos en el archivo, de la misma manera que los escribiriamos en la terminal.
- ¿Guardar el script con terminación *.sh*?
- ¿Permisos de ejecución?

¿Cómo hacer un script?

- Crear un archivo con cualquier editor de texto.
- Indicar el intérprete (opcional) en la primer línea con:

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

  Está línea, denominada *shebang*, especifica el intérprete que se utilizará para ejecutar *script*. Si no se especifica, se utiliza el intérprete por defecto del usuario.
  - ¿Qué problemas puede traer esto?
- Escribir una serie de comandos en el archivo, de la misma manera que los escribiriamos en la terminal.
- ¿Guardar el script con terminación *.sh*?
- ¿Permisos de ejecución?

¿Cómo hacer un script?

- Crear un archivo con cualquier editor de texto.
- Indicar el intérprete (opcional) en la primer línea con:

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

  Está línea, denominada *shebang*, especifica el intérprete que se utilizará para ejecutar *script*. Si no se especifica, se utiliza el intérprete por defecto del usuario.
  - ¿Qué problemas puede traer esto?
- Escribir una serie de comandos en el archivo, de la misma manera que los escribiriamos en la terminal.
- ¿Guardar el script con terminación *.sh*?
- ¿Permisos de ejecución?

Ejemplos

```
#!/bin/bash
# Si la primer línea de mi script comienza
#  con la cadena #! se interpretará como el
# path al intérprete a utilizar (podría ser
# python, perl, php, etc...)
# Ahora el script en sí
echo "Hola mundo"
```

Repaso de conceptos:

- **Path absoluto:** Empieza desde el directorio raíz /
- **Path relativo:** Empieza desde donde estamos posicionados
- **. y ..** Directorio actual y directorio padre respectivamente
- **Variable de entorno PATH**.

¿Cómo ejecutar un *script*?

- Lo guardamos
- ¿Le damos permisos de ejecución?
- Lo ejecutamos
  - `./mi_script.sh`
  - `bash mi_script.sh`
  - O podemos también ejecutarlo en modo *debug* (depuración) `bash -x mi_script.sh`

Ejecución del script

Repaso de conceptos:

- **Path absoluto:** Empieza desde el directorio raíz /
- **Path relativo:** Empieza desde donde estamos posicionados
- **. y ..** Directorio actual y directorio padre respectivamente
- **Variable de entorno PATH**.

¿Cómo ejecutar un *script*?

- Lo guardamos
- ¿Le damos permisos de ejecución?
- Lo ejecutamos
  - `./mi_script.sh`
  - `bash mi_script.sh`
  - O podemos también ejecutarlo en modo *debug* (depuración) `bash -x mi_script.sh`

Ejecución del script

Repaso de conceptos:

- **Path absoluto:** Empieza desde el directorio raíz /
- **Path relativo:** Empieza desde donde estamos posicionados
- **. y ..** Directorio actual y directorio padre respectivamente
- **Variable de entorno PATH**.

¿Cómo ejecutar un *script*?

- Lo guardamos
- ¿Le damos permisos de ejecución?
- Lo ejecutamos
  - `./mi_script.sh`
  - `bash mi_script.sh`
  - O podemos también ejecutarlo en modo *debug* (depuración) `bash -x mi_script.sh`

Repaso de conceptos:

- **Path absoluto:** Empieza desde el directorio raíz /
- **Path relativo:** Empieza desde donde estamos posicionados
- **. y ..** Directorio actual y directorio padre respectivamente
- **Variable de entorno PATH**.

¿Cómo ejecutar un *script*?

- Lo guardamos
- ¿Le damos permisos de ejecución?
- Lo ejecutamos
  - `./mi_script.sh`
  - `bash mi_script.sh`
  - O podemos también ejecutarlo en modo *debug* (depuración)
    `bash -x mi_script.sh`

Ejecución del script

Repaso de conceptos:

- **Path absoluto:** Empieza desde el directorio raíz /
- **Path relativo:** Empieza desde donde estamos posicionados
- **. y ..** Directorio actual y directorio padre respectivamente
- **Variable de entorno PATH**.

¿Cómo ejecutar un *script*?

- Lo guardamos
- ¿Le damos permisos de ejecución?
- Lo ejecutamos
  - `./mi_script.sh`
  - `bash mi_script.sh`
  - O podemos también ejecutarlo en modo *debug* (depuración)
    `bash -x mi_script.sh`

Repaso de conceptos:

- **Path absoluto:** Empieza desde el directorio raíz /
- **Path relativo:** Empieza desde donde estamos posicionados
- **. y ..** Directorio actual y directorio padre respectivamente
- **Variable de entorno PATH**.

¿Cómo ejecutar un *script*?

- Lo guardamos
- ¿Le damos permisos de ejecución?
- Lo ejecutamos
  - `./mi_script.sh`
  - `bash mi_script.sh`
  - O podemos también ejecutarlo en modo *debug* (depuración)
    `bash -x mi_script.sh`

Repaso de conceptos:

- **Path absoluto:** Empieza desde el directorio raíz /
- **Path relativo:** Empieza desde donde estamos posicionados
- **. y ..** Directorio actual y directorio padre respectivamente
- **Variable de entorno PATH**.

¿Cómo ejecutar un *script*?

- Lo guardamos
- ¿Le damos permisos de ejecución?
- Lo ejecutamos
  - `./mi_script.sh`
  - `bash mi_script.sh`
  - O podemos también ejecutarlo en modo *debug* (depuración)
    `bash -x mi_script.sh`

Selección de alternativas:

Decisión:

```
if [ condition ]
then
block
fi
```

Selección:

```
case $variable in
"valor 1")
block
;;
"valor 2")
block
;;
*)
block
;;
esac
```

Menú de opciones:

```
select variable in opcion1 opcion2 opcion3
do
# en $variable está el valor elegido
 block
done
```

Menú de opciones:

Ejemplo:

```
select action in New Exit
do
case $action in
"New")
echo "Selected option is NEW
    "
;;
"Exit")
exit 0
;;
esac
done
```

Imprime:

```
1) new
2) exit
#?
```

y espera el número de opción por teclado

- C-*style*:

```
for ((i=0; i < 10; i++))
do
block
done
```

- Con lista de valores (`foreach`):

```
for i in value1 value2 value3 valueN;
do
block
done
```

Iteración - Bloques WHILE y UNTIL:

while

```
while [ condition ] #Mientras se cumpla la condición
do
block
done
```

until

```
until [ condition ] #Mientras NO se cumpla la condición
do
block
done
```

Evaluación de condiciones lógicas:

Las condiciones lógicas normalmente se evalúan mediante:

```
[ condition ]
test condition
```

Operadores para condition:

Operador	**Con strings**	**Con números**
Igualdad	`"$nombre" = "Maria"`	`$edad -eq 20`
Desigualdad	`"$nombre" != "Maria"`	`$edad -ne 20`
Mayor	`A > Z`	`5 -gt 20`
Mayor o igual	`A >= Z`	`5 -ge 20`
Menor	`A < Z`	`5 -lt 20`
Menor o igual	`A <= Z`	`5 -le 20`

Ejemplos:

```
if [ "$USER" == root ]
then
echo "super user"
else
echo "Ud. es $USER"
fi
# ---------------------
n=0
while [ $n -ne 5 ]; do
echo $n
let n++
done
```

```
for archivo in $(ls)
do
echo "- $archivo"
done
# ---------------------
for ((i=0; i<5; i++))
do
echo $i
done
```

Adicionalmente:

- `break [n]` corta la ejecución de n niveles de *loops*.
- `continue [n]` salta a la siguiente iteración del enésimo *loop* que contiene esta instrucción.

Ejemplos:

```
if [ "$USER" == root ]
then
echo "super user"
else
echo "Ud. es $USER"
fi
# ---------------------
n=0
while [ $n -ne 5 ]; do
echo $n
let n++
done
```

```
for archivo in $(ls)
do
echo "- $archivo"
done
# ---------------------
for ((i=0; i<5; i++))
do
echo $i
done
```

Adicionalmente:

- `break [n]` corta la ejecución de n niveles de *loops*.
- `continue [n]` salta a la siguiente iteración del enésimo *loop* que contiene esta instrucción.

Estructuras de control - break

```
#!/bin/bash
# Imprime los numeros del 1 al 5
# (no es un código para nada elegante)
i=0
# true es un comando que siempre retorna 0
while true
do
let i  ++ # Incrementa i en 1
if [ $i -eq 6 ]; then
break # Corta el loop (while)
fi
echo $i
done
```

¿Qué hace el siguiente *script*?

```
#!/bin/bash
i=0
while true
do
let i++
if [ $i -eq 6 ]; then
break # Corta el while
elif [ $i -eq 3 ]; then
continue # Salta una iteración
fi
echo  $i
done
```

```
# AND
if [ $a = $b ] && [ $a = $c ]
then
#...
# OR
if [ $a = $b ] || [ $a = $c ]
then
#...
```

Argumentos y valor de retorno

- Los *scripts* pueden recibir argumentos en su invocación.
- Para accederlos, se utilizan variables especiales:
  - `$0` contiene la invocación al script.
  - `$1`, `$2`, `$3`, ... contienen cada uno de los argumentos.
  - `$#` contiene la cantidad de argumentos recibidos.
  - `$*` contiene la lista de todos los argumentos.
  - `$?` contiene en todo momento el valor de retorno del último comando ejecutado.

Ejemplo:

```
if [ $# -ne 2 ]; then
exit 1 # Error
else
echo "Nombre: $1, Apellido: $2"
fi
exit      0 # Funcionó correctamente
```

Para terminar un *script* usualmente se utiliza la función `exit`:

- Causa la terminación de un *script*
- Puede devolver cualquier valor entre 0 y 255:
  - El valor 0 indica que el script se ejecutó de forma exitosa
  - Un valor distinto indica un código de error
  - Se puede consultar el *exit status* imprimiendo la variable `$?`

Las funciones permiten modularizar el comportamiento de los *scripts*.

- Se pueden declarar de 2 formas:
  - `function nombre { block }`
  - `nombre() { block }`
- Con la sentencia `return` se retorna un valor entre 0 y 255
- El valor de retorno se puede evaluar mediante la variable `$?`
- Reciben argumentos en las variables `$1`, `$2`, etc.

```
# Recibe 2 argumentos y devuelve:
#   1 si el primero es el mayor
#   0 en caso contrario
mayor()
{
echo "Se van a comparar los valores $*"
if [ $1 -gt $2 ]; then
echo "$1 es el mayor"
return 1
fi
echo "$2 es el mayor"
return 0
}
mayor 5 6 # Invocación
echo $?   # Imprime el exit Status de la funcion
```

Variables: alcance y visibilidad

- Las variables no inicializadas son reemplazadas por un valor nulo o 0, según el contexto de evaluación.
- Por defecto las variables son globales.
- Una variable local a una función se define con `local`

```
test() {
local variable
}
```

- Las variables de entorno son heredadas por los procesos hijos.
- Para exponer una variable global a los procesos hijos se usa el comando `export`:

```
export VARIABLE_GLOBAL="Mi var global"
comando
# comando verá entre sus variables de
# entorno a VARIABLE_GLOBAL
```

¿Preguntas?