

Computer Systems Scripting



Contents

1	What are shell Scripts	2
2	Creating shell Scripts	2
2.1	Running the script	3
2.2	Comments	3
2.3	Variables	4
2.4	Assign values to Variables	5
2.5	Display variable values	5
2.6	Positional Parameter	6
2.7	Getting value from the user through the keyboard	7
2.8	Arithmetic Operations	8
2.9	String Operations	9
2.9.1	Extracting string	9
2.9.2	Removing substring	9
2.9.3	Replacing substring	10
3	Conditionals	10
3.1	Comparing numbers	10
3.2	Comparing strings	10
3.3	Conditionals with files	10
3.4	Test command	11
3.5	Testing condition with [(square brackets)	12
3.6	If structures	12
3.7	Case structures	13
4	For loop	14
4.1	Seq command	16
4.2	While and until loop	17
5	Functions	18
5.1	Defining and using functions	18
5.2	Returning values	19
5.3	Getting values	20
5.4	Global and local variables	21
5.4.1	Global variables	21
5.4.2	Local variables	21

1 What are shell Scripts

Basically, a shell script is a file containing a series of commands. The shell reads a file and carries out the commands as though they have been entered directly on the command line. Shell scripts can be written in several languages. One of the most common is bash shell script. Bash is included in the most important Linux distributions. In this unit, bash shell scripts will be shown but it is possible to write shell script using Perl, Python or other languages.

2 Creating shell Scripts

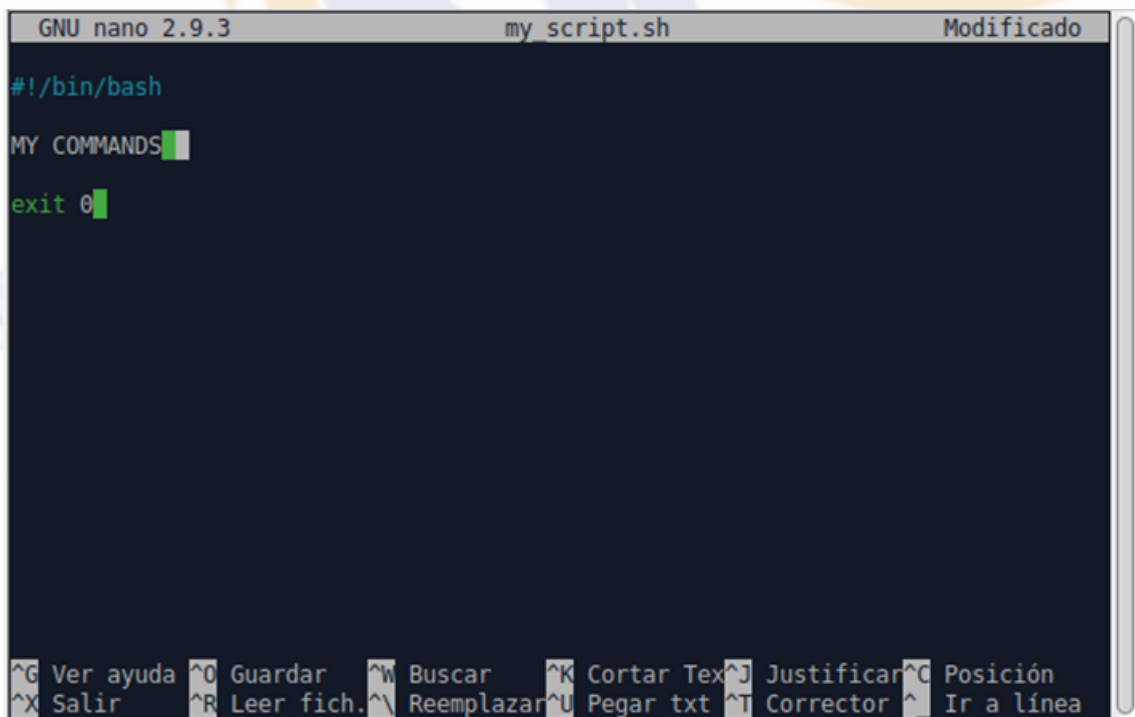
1. To create a bash shell script In Linux, it might be used by any of the text editors in the Operating System, either a graphic editor such as “gedit” or “nano”, “vim” in the command line or any other one.
2. The shell script extension should be “.sh”
3. Once the editor opens to create a bash shell script, it is necessary to write these 2 lines, one at the beginning of the scrip and the second one at the end of the script:

```
#!/bin/bash  
exit 0
```

- The first line is called shebang and it is at the beginning of the script. It says the shell, what program to interpret the script with, and when to be executed.
- The second line is at the end of the script and it is not necessary, but it is useful because it tells the operating system that the process has been running successfully.

So, to create a shell script, the process will be:

```
nano my_script.sh
```



```
GNU nano 2.9.3 my_script.sh Modificado  
#!/bin/bash  
MY COMMANDS  
exit 0  
  
^G Ver ayuda ^O Guardar ^W Buscar ^K Cortar Tex ^J Justificar ^C Posición  
^X Salir ^R Leer fich. ^_ Reemplazar ^U Pegar txt ^T Corrector ^_ Ir a línea
```

Remember to exit the nano editor, ctrl+X should be entered and answer Y in order to save the file with the name created.

2.1 Running the script

echo command

Echo is a command used for showing some sentence or word in the screen. For example:

```
echo Hello World
```

The sentence "Hello World" will be shown in the screen

For example, if the last file is modified in the order of the sentence "Hello world!!!" which is shown, the file will be:

```
GNU nano 2.9.3 my_script.sh Modificado
#!/bin/bash
echo "Hello World!!!"
exit 0
```

To execute the script, it will be necessary to write the name of the script at the command line.

```
usuario@usuario-VirtualBox:~$ ./my_script.sh
bash: ./my_script.sh: Permiso denegado
usuario@usuario-VirtualBox:~$
```

The problem is that the suitable permissions should be provided. In order to run the created script, it will be necessary to give permissions in order to be executed:

```
usuario@usuario-VirtualBox:~$ sudo chmod +x my_script.sh
[sudo] contraseña para usuario:
usuario@usuario-VirtualBox:~$ ./my_script.sh
Hello World!!!!
usuario@usuario-VirtualBox:~$
```

As seen in the last picture, it is possible for the sentence Hello World to be shown since suitable permission was given.

2.2 Comments

Comments are explanations which are written by programmers in order to help to understand the program easier.

To create comments, the # character is used; all the lines starting with that character will be ignored by the shell. For example:

```
GNU nano 2.9.3 my script.sh Modificado
#!/bin/bash
# This is my first bash script
echo "Hello World!!!!"
exit 0
```

2.3 Variables

In shell script as any other programming language, is possible to use variables. As it is well known, variables in programming are memory areas in order to save any value which is able to be changed. There are two kinds of variables:

- System Variables: These variables are those variables defined by the operating system. They are created and maintained by Linux bash shell itself. They are defined in capital letters with some exception.

System variables

It is possible to see all the system variables by writing in the console:

```
env
```

Some important variables:

- BASH_VERSION: Holds the version of this instance of bash.
- HOSTNAME: the name of your computer.
- CDPATH: The search path for the cd command.
- HISTFILE: The name of the file in which command history is saved.
- HOME: The home directory of the current user.
- PATH: The search path for commands. It is a colon-separated list of directories in which the shell looks for echo \$PATH commands.

- User defined variables: they are those variables created and maintained by the user. They can have any valid variable name, but it is better to avoid uppercase names.

2.4 Assign values to Variables

The variables do not have any type, and they exist when the value is assigned. To give some value to some variable, the syntax is:

```
varName=someValue
```

Take into account that there should not be any space between “=” symbol and varName neither “=” and the value which is given.

If any value is specified, the variable will be assigned with null string.

Example:

```
administrator@demeter:~$ myvariable=20
administrator@demeter:~$ _
```



date command

Command date show the date and the current hour

```
administrator@demeter:~$ date
Tue Mar 24 16:11:14 UTC 2020
administrator@demeter:~$ _
```

It is possible to add the output of some command to some variable. For example, if it is required that the output of the command date to the variable “now” is added, the parenthesis should be used like in the following example:

```
administrator@demeter:~$ now=$(date)
administrator@demeter:~$ echo $now
Tue Mar 24 16:12:50 UTC 2020
administrator@demeter:~$ _
```

2.5 Display variable values

To display the variable value, the command “echo” is used, as it is shown in the following example:

```
echo "$HOME"
```

It is necessary to write \$ before the name of the variable with quotes (“”). In the same way, it is possible to show the variable value with the following structure:

```
echo "${HOME}"
```

This way it is quite useful if it is required to add some information to some of the variable value:

```
administrator@demeter:~$ price=3
administrator@demeter:~$ echo ${price}€
3€
```

2.6 Positional Parameter

It is possible to access the script arguments through the positional variables. The script arguments are arguments which are written when the script is called. For instance, if the script name is script1.sh and the script is executed like:

```
administrator@demeter:~$ bash script1.sh file.txt
```

The parameter will be file.txt. The parameter will be used in the script with “\$1”, and if there are more than one, they will be used with \$2, \$3, \$4, and so on... until \$9. If it is required to use more than 9 parameters, it is possible to use the shift command in order to move the current parameter. The effect is that \$2 parameter will be \$1, and \$3 will be \$2 and so on. That is to say, all the parameters are moved to the left and the first one disappears.

```
#!/bin/bash
echo $1
shift
echo $1
exit 0
```

The first command in the last picture displays the first parameter. Later a shift command is used in order to move the value of the first parameter to the left. Eventually, another echo command will be displayed again in the first parameter; thus, the second original parameter will be displayed. The following picture shows the result.

```
administrator@demeter:~$ bash script1.sh Hi welcome
Hi
welcome
administrator@demeter:~$ _
```

Besides the last variables, there are some additional ones:

- \$0: in order to get the script name.
- \$*: to get all the arguments.
- \$#: to know how many arguments there are.
- \$? : to know what is the execution state of the last command.
- \$\$: in order to know the PID, the process which is running.

Through set command it is possible to set values to the positional's parameters. For instance, if the command is used like in the following picture, each parameter will be assigned with one of the words.

```
administrator@demeter:~$ set You are an first level DAM student
administrator@demeter:~$ echo $1
You
administrator@demeter:~$ echo $4
first
```


In the same way, it is possible to assign the output of some command to the positional parameters like in the following:

```
administrator@demeter:~$ date
Thu Apr  9 10:59:10 UTC 2020
administrator@demeter:~$ set `date`
administrator@demeter:~$ echo $1 $2 $3 $4 $5 $6
Thu Apr 9 10:59:17 UTC 2020
```

Date command output is shown in the first line and this output is later assigned to the positional parameters through the command `set`. The result is displayed in the last command since every parameter assigned with each value of the date command output is shown with the command `echo`.

Example:

```
#!/bin/bash

echo Parameters of the command $*
echo Arguments numbers $#
echo command $0
echo Parameter 1 $1
echo Parameter 2 $2
echo Parameter 3 $3
echo Parameter 4 $4
echo Parameter 5 $5
echo Parameter 6 $6
echo state $?
echo pid $$

exit 0
```

Output:

```
administrator@demeter:~$ bash ex_pos_par.sh one two three four five six
Parameters of the command one two three four five six
Arguments numbers 6
command ex_pos_par.sh
Parameter 1 one
Parameter 2 two
Parameter 3 three
Parameter 4 four
Parameter 5 five
Parameter 6 six
state 0
pid 4157
administrator@demeter:~$ _
```

2.7 Getting value from the user through the keyboard

Shell script also allows to ask for some input value to the user. Command “`read`” can accept input value through the keyboard. The `read` command syntax is:

```
read -p “Prompt” variable1 variable2 variableN
```

where:

- `-p “Prompt”`: Prompt refers to the message that is required for the user to display.

- Variable1, variable2, variableN: they are the variables which will be assigned with the first value, second, and n value. It depends on the number of the words that the user introduces via keyboard.

Example:

```

GNU nano 2.9.3 my_script.sh Modificado

#!/bin/bash

# This is my first bash script
read -p "Please, give me your name and last name: " namevar lastnamevar
echo "Your name is $namevar"
echo "Your surname is $lastnamevar"

exit 0

```

As it is displayed in the picture above, the script is asking for the first and last name of the user, who is running the script. When the user writes his or her first name, this value will be stored in namevar and the last name will be stored in lastnamevar. It is possible that the values will finally be displayed in the last two sentences of the green box.

2.8 Arithmetic Operations

There are three different ways of writing arithmetic operations:

- `$((expression))`

For example:

`$((n1+n2))` \Rightarrow Pay attention in the spaces between the variables and the parentheses, the spaces should be respected. If it is necessary to assign the operation to another variable, do the following:

$$\text{Res}=\$((\text{ n1}+\text{n2} \text{)})$$

- `$(expr expression)`

For example:

`$(expr n1 + n2)` \Rightarrow Again, the spaces should be respected between variable and operator. If it is necessary to assign the operation to another variable:

$$\text{Res}=\$(\text{expr } \$\text{n1} + \$\text{n2})$$

- That one is the easiest: let “expression”

For example:

```
let 'a=(4+3)*7'
```

```
let res=$n1+$n2
```

In this case, the entire expression is inside of the quotes.



Try it

You are asked to make a Shell script which asks 2 numbers. Both numbers will be added and finally the result will be displayed.

2.9 String Operations

2.9.1 Extracting string

It is used in order to manipulate string, that operation is able to extract a substring of a given string.

The operation is:

```
${string:position:size}
```

Example:

Taking into account that: String=abcABC123ABCabc

- echo \${string:0} ⇒ the result will be abcABC123ABCabc
- echo \${string:0:1} ⇒ a the first character
- echo \${string:7}: ⇒ 23ABCabc
- echo \${string:7:3} ⇒ 23A (3 characters from seventh position)
- echo \${string: -4} ⇒ Cabc the last 4 characters, pay attention: there is a space between “:” and “-4”
- echo \${string: -4:2} ⇒ Ca, from the last 4 characters, just 2, pay attention: there is a space between “:” and “-4”.

2.9.2 Removing substring

There are different ways to erase substring:

- \${string#substring} ⇒ erase the shorter coincidence in a substring from the beginning of the string
- \${string##substring} ⇒ erase the longer coincidence in a substring from the beginning of the string

Example: with the String=abcABC123ABCabc

- echo \${string#a*C}: result will be: 123ABCabc
- echo \${string##a*C}: abc

2.9.3 Replacing substring

It is possible to replace substring of a string:

- `${string/search/replace}`: It will be searched coincidences of the second field in the first one, replacing with the “replace” field.
- `${string//search/replace}`: All the coincidences of the string in the field search will be replaced in the string field with the string in the “replace” field.

Examples: With the `String=abcABC123ABCabc`

- `echo ${string/abc/xyz}`: the result will be: `xyzABC123ABCabc`
- `echo ${string//abc/xyz}`: the result will be: `xyzABC123ABCxyz`

3 Conditionals

As it is well known, conditional structures compare variables with others or with some other value. Depending on the result of the comparison, the program flow will follow different paths.

In Shell scripts it is possible to do comparisons with strings, numbers or even with files. So, depending on this kind of comparison, the operators will be different.

3.1 Comparing numbers

With that kind of data, it could be carried out with the following operations:

Operator	Meaning
-lt	Less than (<)
-gt	Greater than (>)
-le	Less than or equal than (<=)
-ge	Greater than or equal to(>=)
-eq	equal to (==)
-ne	not equal (i=)

3.2 Comparing strings

In order to compare string, the operators are:

Operator	Meaning
=	The strings are exactly equal
!=	The strings are not exactly equal
<	Less than (alphabetic order ASCII)
>	Greater than (alphabetic order ASCII)
-n	String is not empty
-z	String is empty

3.3 Conditionals with files

The operators used in conditional structures return true or false depending on the result of the operation. The operators are:

Operator	True if it will be returned
-e name	File called with the name exist
-f name	If name is a regular file (it is not a directory)
-s name	If the file exists, and the size is not 0
-d name	If the name is a directory
-r name	If the user who has run the script has read permission in the name file
-w name	If the user who has run the script has written permission in the name file
-x name	If the user who has run the script has execution permission in the name file

3.4 Test command

This structure is used in order to compare numbers, strings or check types of files.

There are different ways of using the structure:

- test condition
- test condition && command, executed when true
- test condition || command executed when false
- test condition && true-command || false command

Example:

```
#!/bin/bash
read -p "Give me 2 numbers to compare: " n1 n2
test $n1 -gt $n2 && echo "$n1 is greater than $n2" || echo "$n1 is equal or lower than $n2"
exit 0
```

```
#!/bin/bash
read -p "Give the name of the file which you want to check" n1
test -f $n1 && echo "File exists" || echo "File doesn't exist"
exit 0
```

The first example shows the comparison of two numbers in order to compare the two numbers. The second example checks whether a given file exists or not.



Try it

| Do a script in order to check if the string is empty.

3.5 Testing condition with [(square brackets)

In the same way, for testing conditions it is possible to use square brackets in order to test numerical expressions, strings or file parameters. Syntax:

```
[ condition ]  
OR  
[ ! condition ]  
OR  
[ condition ] && true-command  
OR  
[ condition ] || false-command  
OR  
[ condition ] && true-command || false-command
```

3.6 If structures

Like in others programming languages, in shell scripts there are control flow structures. IF statements are one of them. The syntax for the basic if statement is:

```
if condition  
then  
command 1  
command 2  
...  
command n  
fi
```

In the same way, there is an if... else statement with this structure:

```
if condition  
then  
command 1  
...  
command n  
else  
command 1  
...  
command n  
fi
```

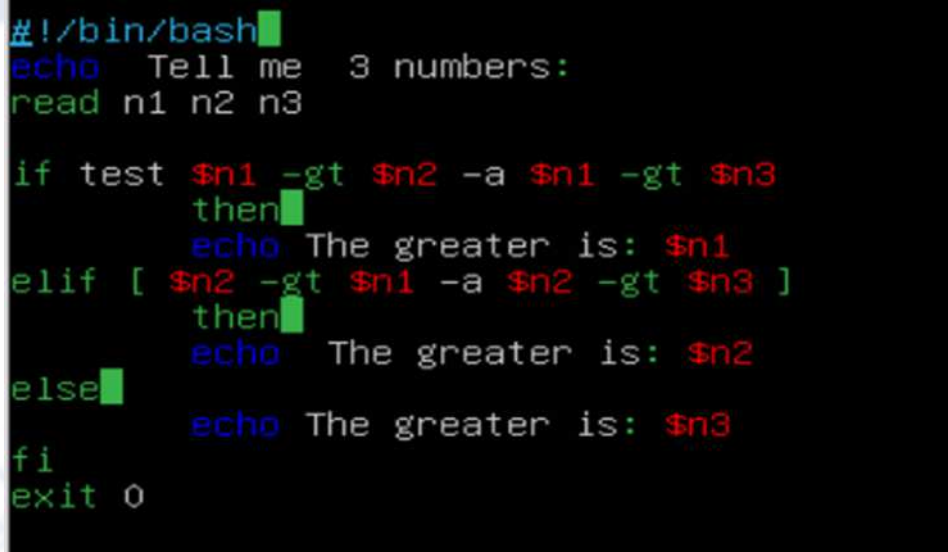
Example:

```
#!/bin/bash  
if cp $1 $2 2>/dev/null  
then  
    error=0  
    echo Copy from $1 to $2 succesfully  
else  
    error=1  
    echo It has been impossible to copy from $1 to $2  
fi  
exit $error
```

As it can be seen from the last picture, it is possible to use the state of the command in order to get the condition in the if statement. In this example, it shows how the mv command is executed, and depending on the result of the command execution, it will display one statement or another. However, at the end of the script, the relevant error (0 or 1) will be returned to the system. Another structure can be used in the if statement:

```
if condition
then
command 1
...
elif condition
then
command1
...
elif condition
...
...
else
...
fi
```

Example:



```
#!/bin/bash
echo Tell me 3 numbers:
read n1 n2 n3

if test $n1 -gt $n2 -a $n1 -gt $n3
then
echo The greater is: $n1
elif [ $n2 -gt $n1 -a $n2 -gt $n3 ]
then
echo The greater is: $n2
else
echo The greater is: $n3
fi
exit 0
```

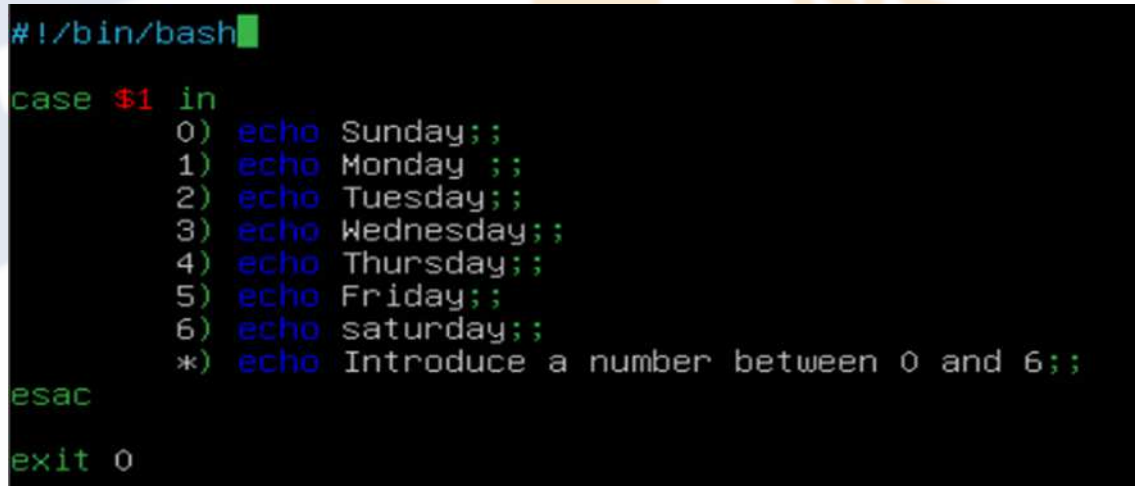
3.7 Case structures

What usually happens is that when it becomes necessary to use several if(s), it could be used to case structures:

```
case var in
case 1)
command 1
...
command N
...
case N)
commands
*)
commands
Esac
```

Example:

As shown from the given example, it introduces a number as a positional parameter, and the script displays what day of the week it is.



```
#!/bin/bash
case $1 in
0) echo Sunday;;
1) echo Monday ;;
2) echo Tuesday;;
3) echo Wednesday;;
4) echo Thursday;;
5) echo Friday;;
6) echo saturday;;
*) echo Introduce a number between 0 and 6;;
esac
exit 0
```

4 For loop

Loop is used for repeating instructions several times. The syntax of this structure is:

```
for var in item1 item2 ... itemN
do
command1
command2
....
...
commandN
done
```

Where item 1, Item2, item N is a variable list. Example:


```
#!/bin/bash

for i in a e i o u
do
    echo Files which are started with /usr/bin/$i
    echo -----
    ls -l /usr/bin/$i*
done

exit 0
```

The last script displays those files which are started by a in /usr/bin folder. After that are those files which are started by e in the same folder and the same with I o and u.

Loop can be expressed with the following syntax as well:

```
for((i=0;i<N;i++))
do
    command 1
    ...
    command N
done
```

Example:

```
#!/bin/bash

for ((i=0;i<5;i++))
do
    echo Number $i
done

exit 0
```

Another way for using loop is executing several commands. For the loop command:

```
for var in $(Linux-command-name)
do
    command1
    command2
    ....
    ...
    commandN
done
```

Example:

```
#!/bin/bash

for command in date pwd df
do
    echo The output of $command
    $command
done

exit 0
```

The date commands, pwd and df will be executed.
But loop can be used in the following way:

```
#!/bin/bash
num=0
for i in `ls`
do
    echo file $num: $i
    let num=num+1
done

exit 0
```

The i variable takes the value of each file which is returned from the ls execution command.
So, the result will be:

```
file 0: carpeta0
file 1: carpeta1
file 2: carpetanueva
file 3: exemple2.sh
file 4: exemple.sh
file 5: exer_script.sh
file 6: ex_if_else2.sh
file 7: ex_if_else.sh
file 8: ex_pos_par.sh
file 9: iptables_script.sh
file 10: script1.sh
file 11: script.sh
file 12: Thu#Apr#9#15:54:01#UTC#2020.tgz
```

4.1 Seq command

The seq command is a sequence of values which takes every value from the first until the last. The way in which it will be taken is defined by the increment of the values. The default value for the increment and the first value is 1. In addition, the increment should be positive if the first value is less than the last otherwise the increment should be negative.

```
#!/bin/bash
num=0
for i in `seq 1 2 20`
do
    echo $i
done
exit 0
```

4.2 While and until loop

The syntax for the while loop is:

```
while [ condition ]
do
command1
command2
..
....
commandN
done
```

While the command 1, command 2, command N are executed when the condition is true.

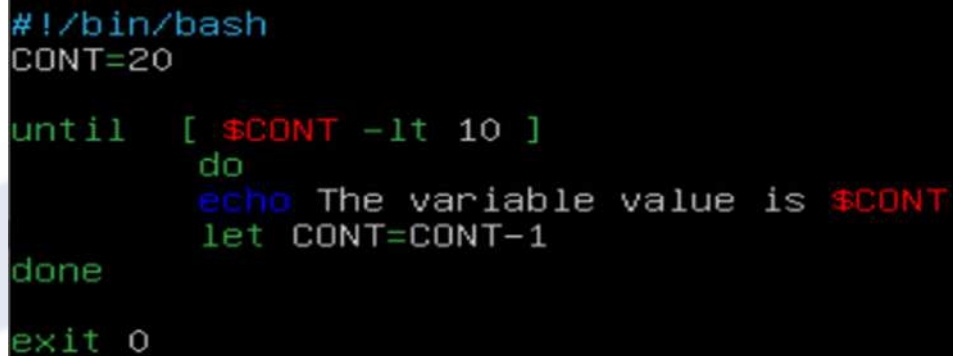
Example:

```
#!/bin/bash
CONT=0
while [ $CONT -lt 10 ]
do
    echo The variable value is $CONT
    let CONT=CONT+1
    #CONT=${CONT+1}
    #CONT=$((CONT+1))
    #CONT=`expr $CONT+1`
done
exit 0
```

The syntax for the until structure is:

```
until condition
do
command1
...
command N
done
```

Example:



```
#!/bin/bash
CONT=20

until [ $CONT -lt 10 ]
do
    echo The variable value is $CONT
    let CONT=CONT-1
done

exit 0
```

5 Functions

Functions are a commands set that are used in order to avoid repeating again and again the same commands set. The only thing to do, is write those lines with a name and call them when needed to use.

5.1 Defining and using functions

In order to create functions, the required syntax is:

```
functionName () {
commands
}

Or

function functionName {
commands
}
```

The parenthesis are not required in the second way. Instead, the reserved word “function” is used before the name of the function.

Functions must be defined before being used, but they are normally defined at the beginning of the script. Besides, the name should be unique otherwise, the last function with the name will be executed and the first one will be cancelled.

To use them, the name should be written in the place you want to use them.

Example:

```
#!/bin/bash

function myfunct {
    echo "Hello world!!!"
}

myfunct

exit 0
```

5.2 Returning values

In order to use function for obtain some value after carrying out some mathematical operation, it is possible to use the return command, this option is just for returning numerical values, after calling the function, the returning value will be got with \$? variable. Bear in mind that just after using the function it will be able to get the value, if some command is executed before getting the variable value with \$?, the value won't be the right value.

```
#!/bin/bash

function myfunct {
    read -p "Tell me a number" val
    return $((val + 10))
}

myfunct

echo "the value is $?"

exit 0
```



Try it

Modify the script so that after using the function some command is executed before getting the \$? Value. Pay attention to the result.

A different way to return values from a function exists. This way will be able to get string values from the function. In the example you can see that the value will be returned with the command echo. The value from the function will be got and stored in the "result" variable.

```
#!/bin/bash

function myfunct {
    read -p "Tell me your name" name
    echo $name
}

result=$(myfunct)
echo "Hello $result"

exit 0
```

5.3 Getting values

It is possible to send values to the function so that the function becomes reusable and more useful. In order to send value to the function, the positional parameters are used as it can be seen in the example:

```
#!/bin/bash

function myfunct {
    echo $(( $1 + $2 ))
}

myfunct 5 6
exit 0
```

The function gets the sent values with the positional parameters \$1 and \$2.

It can be a different scenario, the script receives values as arguments and these values are sent to the function. The following example shows how these values are sent:

```
#!/bin/bash

function myfunct {
    echo $(( $1 + $2 ))
}

value=$(myfunct $1 $2)
echo "The value of the addition is: $value"

exit 0
```

```
administrator@demeter:~$ bash scriptfunctions.sh 1 3
The value of the addition is: 4
administrator@demeter:~$
```

In the example, numbers 1 and 3 are sent to the script as positional parameters. In the script, the variables of those values are \$1 and \$2, in order to be sent to the function, it should be used this name and again inside of the function, these variables are used. It shouldn't be thought that the values \$1 and \$2 inside of the function are the values sent as argument to the script. Those values are different and should be sent again.

5.4 Global and local variables

As it is well known, the scope of the variables depends on whether the variable is local or global. If the variable is global, the value of that variable will be able to change in any place of the program. Otherwise, the scope of the local variable will be a function or a program.

5.4.1 Global variables

These variables are visible in any place of the script:

```
#!/bin/bash

function myfunct {
    value=$((value + 20))
}

read -p "tell me a number: " value
myfunct
echo "The value is: $value"
exit 0
```

The example shows that the value of the variable value is changed several times and the last value is the value given by the function because it is the last modification.

5.4.2 Local variables

The local variable is used just inside of the function in order to avoid changing the value of the variable out of the function. In the following example you can see that the variable value is declared inside of the function with the reserved word "local".

```
#!/bin/bash

function myfunct {
    local value=$(( $1 + 20 ))
    echo "The value variable inside function is $value"
}

read -p "tell me a number: " value
myfunct $value
echo "The value variable out of the function is: $value"
exit 0
```


If the script is executed, the value inside of the function is displayed as the variable sent to the function plus 20. The value variable outside of the function is displayed as well. In that case, the value is different.

```
administrator@demeter:~$ bash scriptfunctions.sh  
tell me a number: 5  
The value variable inside function is 25  
The value variable out of the function is: 5  
administrator@demeter:~$ _
```