Chapter 3d. completing scripting

3d.1. eval

eval reads arguments as input to the shell (the resulting commands are executed). This allows using the value of a variable as a variable.

```
clim@sop:~/test42$ answer=42
clim@sop:~/test42$ word=answer
clim@sop:~/test42$ eval x=\$$word ; echo $x 42
```

Both in **bash** and **Korn** the arguments can be quoted.

```
stud1@srvU$ answer=42
stud1@srvU$ word=answer

stud1@srvU$ eval "y=\$$word" ; echo $y 42
```

Sometimes the **eval** is needed to have correct parsing of arguments. Consider this example where the **date** command receives one parameter **1 week ago**.

```
clim@sop~$ date --date="1 week ago"

Thu Mar  8 21:36:25 CET 2012
```

When we set this command in a variable, then executing that variable fails unless we use **eval**.

```
clim@sop~$ lastweek='date --date="1 week ago"'
clim@sop~$ $lastweek

date: extra operand `ago"'

Try `date --help' for more information.
clim@sop~$ eval $lastweek

Thu Mar  8 21:36:39 CET 2012
```

3d.2. (( ))

The **(( ))** allows for evaluation of numerical expressions.

```
clim@sop:~/test42$ (( 42 > 33 )) && echo true || echo false
true

clim@sop:~/test42$ (( 42 > 1201 )) && echo true || echo false
false

clim@sop:~/test42$ var42=42

clim@sop:~/test42$ (( 42 == var42 )) && echo true || echo false
true

clim@sop:~/test42$ (( 42 == $var42 )) && echo true || echo false
true

clim@sop:~/test42$ var42=33

clim@sop:~/test42$ (( 42 == var42 )) && echo true || echo false
false
```

3d.3. let

The **let** built-in shell function instructs the shell to perform an evaluation of arithmetic expressions. It will return 0 unless the last arithmetic expression evaluates to 0.

```
[clim@sop ~]$ let x="3 + 4" ; echo $x 7
[clim@sop ~]$ let x="10 + 100/10" ; echo $x 20
[clim@sop ~]$ let x="10-2+100/10" ; echo $x 18
[clim@sop ~]$ let x="10*2+100/10" ; echo $x 30
```

The **shell** can also convert between different bases.

```
[clim@sop ~]$ let x="0xFF" ; echo $x
255
[clim@sop ~]$ let x="0xC0" ; echo $x
192
[clim@sop ~]$ let x="0xA8" ; echo $x
168
[clim@sop ~]$ let x="8#70" ; echo $x 56
[clim@sop ~]$ let x="8#77" ; echo $x 63
[clim@sop ~]$ let x="16#c0" ; echo $x
192
```

There is a difference between assigning a variable directly, or using **let** to evaluate the arithmetic expressions (even if it is just assigning a value).

```
stud1@srvU$ dec=15 ; oct=017 ; hex=0x0f
stud1@srvU$ echo $dec $oct $hex

15 017 0x0f

stud1@srvU$ let dec=15 ; let oct=017 ; let hex=0x0f
stud1@srvU$ echo $dec $oct $hex

15 15 15
```

3d.4. case

You can sometimes simplify nested if statements with a **case** construct.

```
[clim@sop ~]$ ./help

What animal did you see ? lion
You better start running fast!
[clim@sop ~]$ ./help

What animal did you see ? dog
Don't worry, give it a cookie.
[clim@sop ~]$ cat help

#!/bin/bash

#

# Wild Animals Helpdesk Advice

#

echo -n "What animal did you see ? "
read animal

case $animal in

        "lion" | "tiger")

                echo "You better start running fast!"

        ;;

        "cat")

                echo "Let that mouse go..."

        ;;

        "dog")

                echo "Don't worry, give it a cookie."

        ;;

        "chicken" | "goose" | "duck" )
                echo "Eggs for breakfast!"

        ;;

        "liger")

                echo "Approach and say 'Ah you big fluffy kitty...'."

        ;;

        "babelfish")

                echo "Did it fall out your ear ?"

        ;;

        *)

                echo "You discovered an unknown animal, name it!"

        ;;

esac

[clim@sop ~]$
```

3d.5. shell functions

Shell **functions** can be used to group commands in a logical way.

```
stud1@srvU$ cat funcs.ksh

#!/bin/ksh



function greetings {
echo Hello World!

echo and hello to $USER to!

}



echo We will now call a function
greetings

echo The end
```

This is sample output from this script with a **function**.

```
stud1@srvU$ ./funcs.ksh We
will now call a function
Hello World!

and hello to clim to!
The end
```

A shell function can also receive parameters.

```
stud1@srvU$ cat addfunc.ksh
#!/bin/ksh

function plus {
let result="$1 + $2"
echo  $1 + $2 = $result
}

plus 3 10
plus 20 13
plus 20 22
```

This script produces the following output.

```
kahlan@solexp11$ ./addfunc.ksh
3 + 10 = 13

20 + 13 = 33

20 + 22 = 42
```

3d.6. practice : more scripting

3d.1.1. Write a script that asks for two numbers, and outputs the sum and product (as shown here).

```
Enter a number: 5

Enter another number: 2


Sum:       5 + 2 = 7

Product:   5 x 2 = 10
```

3d.2.1. Improve the previous script to test that the numbers are between 1 and 100, exit with an error if necessary.

3d.3.1. Improve the previous script to congratulate the user if the sum equals the product.

3d.4.1. Write a script with a case insensitive case statement, using the shopt nocasematch option. The nocasematch option is reset to the value it had before the scripts started.

3d.5.1. If time permits (or if you are waiting for other students to finish this practice), take a look at Linux system scripts in /etc/init.d and /etc/rc.d and try to understand them. Where does execution of a script start in /etc/init.d/samba ? There are also some hidden scripts in ~, we will discuss them later.