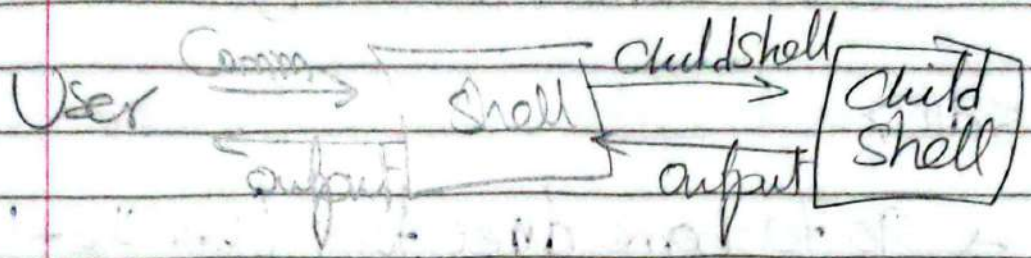
4 TCL Shell scripting

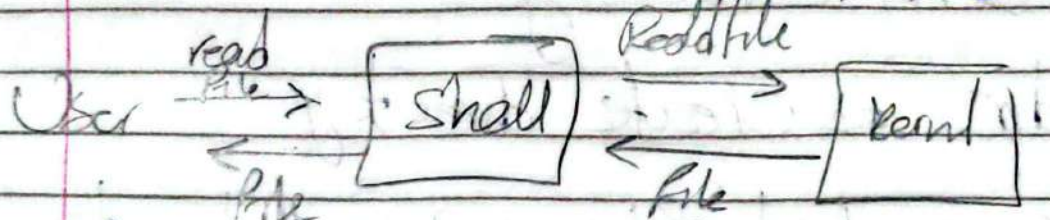
Chapter 1 : Common Commands

* How the Shell works:

→ Shell creates a child Shell for execution of utility



→ Shell Request the kernel for any Hw interaction



Types of Shell:

1. Bourne Shell (Sh/Bash) → Sh extension
2. C Shell (csh) → linux
3. korn shell → ksh
4. Restricted shell → Guest login → window
5. Windows shell → cmd / powershell
6. Most Common Shell used by Hw engineer → TCl → Hw

echo $SHELL → know type of shell
in linux

* Common Shell Commands:
① Navigation Commands

① PWd : show current directory

② cd <dir>

cd ⇒ go to home directory (user → ~)

③ ~ ⇒ home directory

④ mkdir ⇒ create directory

⑤ ls → list files in current directory

⑥ rmdir ⇒ remove directory
(can't remove non-empty folder)

⑦ rm -r <dir-name> : to remove
non-empty directory

3/11

(2) File Commands:

(1) touch → create a file.

(2) Code filename → open visual studio code

(3) Vi filename

i → insert mode dd → delete whole line

escape → get out from insert mode

write :wq → save & quit

(4) Cat filename → Display contents (can have multiple files)

(5) cp readme file modified newfile → Copy file to another

(6) head filename → display first 10 line
tail filename → last 10 lines

→ head -n5 filename

(7) more filename → View (paging file)

(8) less filename → Allow movement in forward & backward directions

(9) mv file1 file2 : Move file1 to file2

(10) Renaming using mv.

mv oldname newname

(11) rm filename → delete a file

(12) grep <pattern> path

grep "is" ./work1/*

(13) file <filename> → Get type of file

(3) General Commands:-

(1) man <command name> : enter manual

(2) Chmod o-r <path>

User Group Others

chmod o+r <path>

u+rw

a+rw

all

5

③ chown testUser <filename>
↓ ↓
change new User
ownership

④ Sudo → Super User do
↳ change big permission

⑤ uname -a → show System & kernel

⑥ Whoami → Show your username

⑦ date → Show system date

⑧ clear → cls

Process Commands :-

① ps → Show Snapshot of processes

② top → Show real time processes (task Manager)

③ kill -9 <pid> → kill process with pid

④ pkill -9 <name> → kill process with name

⑤ killall -g<name> → kill all processing with name at begin of username

⑥ ps -ef → to show all processes including system processes

kill ⇒ Send signal (not killing)

-9 → termination, we can pause it or send to background

⑦ piped

ps -ef | grep "gedit"

⑧ command ⑨ → run command in background

⑨ cmd < filename: Input of cmd from file

⑩ cmd > file : write output of cmd in file

⑪ cmd < /dev/null : discard output

⑫ cmd >> file → Append to file

⑬ cmd1 | cmd2 ⇒ output of cmd 1 is input to cmd 2

* Three types of Commands:

① Builtin . ② User defined ③ Application specific

Helloworld TCL :-

First line => #!/bin/tclsh (path of Tcl shell)

↓ (!) علشان نميزه عن ال comment

↓ (path) To get it you can run

$ which tclsh

* Commenting => #

* TCL Basics :-

It like any programming language if, for —

→ Commands are separated by SemiColon or new lines

Examples: Don't declare with type.

```
Set a 22   #Set variable a to 22
Puts "Helloworld" #print helloworld
```

*Basic operators:

→ to access the value of variable put dollar sign before name of variable

```
set a 1
Set b 3
Set c [expr $a + $b]
Puts " c value of $c "
```

→ if Condition:

```
if { $a > $b } {
Puts "a is bigger than b"
} else {
Puts "a is less than b"
}
```

→ incr c → src&dest
↑ without $

→ incr c <step>

→ we can unset variable
unset a

→ in nested commands to have command be executed first put in []

```
if {[info exists a]} {
    #print
} else {
    #print
}
```

For multiline comments
if 0 { } → Comment

→ # of comment must be not start of line

→ ; # in line comment

→ \ → will continue in the next line

operators

1. Bitwise ~, <<, >>, &, ^, |
2. logical !, &&, ||

[3] Boolean : >, <, <=, >=, ==, !=

[4] String operators → (eq, ne, <, >, >=, <=

[5] list Containment → in, ni

[6] Ternary → x ? y : z

[7] Math functions (log, sin, cos)

Control Flow

[1] if then else

```
Set income 32000
if { $income > 30000 } {
    puts "income -- high"
} elseif { $income > 20000 } {
    puts "income -- middle"
} else {
    puts "$income -- low"
}
```

[2] While loops

```
Set i 0
While {$i <100} {
  Puts "I am at Count $i"
  incr i
}
```

[3] For loops

```
for {set i 0} {$i <100} {incr i} {
  Puts "I am at Count $i and going up"
}
for {set i 100} {$i >0} {set i [expr $i-1]} {
  puts "I am at Count $i and going down"
}
```

4) Foreach

list

```
set lstColors {red orange yellow}
foreach c $lstColors {
    puts $c
}
```

Example 2:

```
foreach {a b c} $lstColors {
    puts "$c -- $b -- $a"
} ;# with step
set lstFoods {apple orange banana}
foreach f $lstFoods c lstColors {
    puts "a $f is usually $c"
}
```

foreach {a b} $lstfoods c $lstColors {
puts " "
}

⇒ Print until the longest finish

Chapter3: Procedures

* Creating a function

proc functionname {p1 p2 ...} {
return ...
}

Example:

proc foo {a b c} {
return [expr $a + $b - $c]
}

Call ⇒ puts [expr [foo 2 3 4] + 5]

Ex2:

```
proc bar {} {
    puts "I am in barfunction"
}
bar
```

* Procedure Calls with Variable Number of Arguments

```
proc sum {:args} {
    set s 0
    foreach arg $args {
        incr s $arg
    }
    return $s
}
puts [sum 1 2 3 4]
puts [sum 1 2]
```

* Procedure Calls (Implicit Variable)

```
proc power {a {b 2}} {
    if {$b == 2} {
        return [expr $a * $a]
    }
}
```

Note => Default Parameters at the end

Variable Scopes

- Local
  - → Defined inside Proc
- Global
  - → Defined globally
  - → Can be accessed anywhere

Global

- global
  - global a
  - ↳ inside Procedure
- ::
  - incr ::a
  - $::a
- Up var
  - upvar #0 a x
  - #0 → Global Scope
  - a → اصلي
  - x → لو هاستخدمه

* Parameters passed to proc

Global

(By Reference)

Value

Name

Passed by value

Foo $V $n 1 }}

upvar $V a

Foo $a

upvar # 0 $V a

↓

Start Counting from Global to inner

upvar 1 $V a

↑

level above me directly

upvar $V a → go level up by default

Data Structures (Chapter 4)

① lists ② Dictionaries / Associated Arrays

③ strings

Chapter 9: Data Structures in TCL

1) List:

list a b c → [a, b, c]

set mylist [list a b c]

set mylist {a b c}

set mylist [list $a $b $c]

set mylist [split "a-b-c" -];

set mylist "a b c"

set mylist "$a $b $c"

But you can't make ~~{$a $b $c}~~ → it will not have values, it will have $a, $b literally

* To know length of list

→ llength

[llength $l]

→ [string length $l]

As it counts the space as character

→ lindex $lst index

Puts " [lindex $lst 0] "

→ we have to put $lst not lst

→ end → express the index of last element (-1 in python)

* Adding element ⇐

- lappend list "Michael"
  ↑ new element

⇒ Apeend at the end

Format

- linsert value of list ~~elem~~ index element $list

linsert $list 2 "Radaw"

⇒ Return new list with modification

Format

- lreplace $list start end ele1 ele2

Ex:

lreplace $list 3 5 "Nour" "Samy"

- lreplace $list 3 5 "Nour Samy"
  ↳ Put list inside list

19

Note We Can Use replace to Delete an element

lreplace $list end end ; # delete the last element

- lrange : make a subset of list

lrange $list start end => format

lrange $list 1 end

- lsort : sort elements in list

lsort -option $list => format

lsort -ascii $list

- lset : replace element at specific index with another

lset list index newelement => format

lset list 1 "Ayman"

- lsearch : Search in a list & return its index (takes options

lsearch $list pattern => format

puts "[lsearch $list "Ayman"]"

20/

* List of lists

set a [list [list x y z]]

~~set~~ [lindex [lindex $a 0] 1]

b

outer list

Inner list

Ex:-

set ~~list~~ arg1 [list a]

Set arg2 [list b]

Set list [list $arg1 $arg2]

↳ list of lists.

→ Flattening the list

[ ] ~~string~~ join : Concatenate list elements into string

puts " [join $arg1] "

space

will Flatten the second level of list

[a [b [c]] [d [E]]]

↳ [a b [c] d [E]]

21

Format

join $list separator

2) ConCat : join multiple list into a single list

Puts " [ConCat $l1 $l2] "

Format

ConCat $list1 $list2

3) Lappend : Append elements to an existing list

lappend L1 $l2

→ append The list itself as list to l1

* To append the elements of l2 to l1

~~set~~ lappend l1 {*} $l2

* Accessing list of variables

```
Set a 1 ; Set c 3
Set b 2
Set mylist {a b c} ;
```

[1] First Method → Using upvar

```
foreach i $mylist {
    upvar 0 $i y
    Puts "$i : $y"
}
```

$i → a, b, c ; $y → 1, 2, 3

[2] Set Command

→ The dolar Sign is the same as set a

```
Puts "value of a is $a Same as [set a]
foreach i $mylist {
    Puts "$i is [set $i]"
} // as I make double dolar sign
```

3 eval command

```
foreach i $my_list {
    eval set x $$i
    puts "$i is $x"
}
```

4 Using $ in list

```
set mylist {$a $b $c}
Foreach i $mylist {
    eval set x $i
    puts "$i : $x"
}
```

* lassign : Unpacks a list into variables

```
set arr {a b c}
lassign $arr x y z
```

x → a
y → b
z → c

→ gives more variables → The rest is empty str

→ " less variables → return the rest

Note: We can use lassign to shift list

29/11

Set arr [lassign $arr x]

like we have popped the first element

Format

lassign $list var1 var2 ...

2. Associative Arrays

- HashMaps O(1) search
- All keys are strings
- Create Entries on the fly
- Except for reading array name, array name is used not its dereferenced name.

Create Entries:

1. set Color(rose) red
   Set Color(sky) blue

2. array set : Sets values in an Array

Format

array set <arrayname> "key1 value1 key2 value2" $list

array set engineering "Civil Buildings Computer Chips Communication wifi"

Accessing Associative Array Elements :-

puts "Sky Color is $Color(sky)"
puts "Civil Engineers Create $engineering(Civil)".

- array names : Retrieve the names of key in an Array

Format

array names <arrayname>.

puts [array names Color]

Iterating Over Associative Arrays :-

```
foreach item [array names Color] {
    puts "$item is $Color($item)"
}
```

- **array get** : Returns a list containing pairs of Elements

Format

```
array get <array name>
```

```
set lstColor [array get Color]
# key1 value1 key2 value2
```

- **array exists** : Check if variable is array

Format

```
array exists <array name>
```

puts [array exists Color]

1 → printed

- array size : get size of array
  array size <arr-name>

Note:

We can use info exists on checking if there a specific key

```
if {[info exists newColor(sky)]} {
    # Do Something
    }
```

- Multidimensional Arrays

```
set arr(1,1) "item1"
set arr(1,2) "item2"
set arr(1,3) "item3"
```

=> it's as we have key that's called "1,1" literally so it's not multidimensional in common sense

* Make two iterators i,j to loop over Array:

Set a 1

Puts $arr($a,$a)

i j

we can't have two levels of Associative array -> it's limited to just one level

3) Dictionary:

- dict set : Set a dictionary Entry

Format

dict set <dictname> key value

- dict create : Create a dictionary

Format

dict create key1 value1 key2 value2

return the dictionary

Note: We can print the dictionary directly without converting it to list first like array

- using lindex with dict will access dict like list → will flatten the key & values

Suppose we have

dict → {c1: black, c2: white}

puts [lindex $dict 1]

will print black

- dict keys : Get dictionary keys

Format

dict keys $<dictname>

dict keys $colors

- dict get : Retrieve a dictionary Entry

Format

dict get $<dictname> $<dictkey>

dict get $dict $item

Looping over Dictionary :-

```
foreach item [dict keys $dict] {
set value [dict get $dict $item]
puts "The value of $item is $value"
}
```

- **dict values** : Get Dictionary values

Format

```
dict values $<dict name>
```

```
dict values $dict
```

- **dict exists** : Check if key exists in dictionary

Format

```
dict exists $<dictname> key
```

```
dict exists $dict color1
```

3)

Multilevel of Dictionaries

```
dict set ldifvalues level0 {
    level1 {key3 er key4 sdf}
    key1 value1
    key2 value2
}

puts [dict get $ldifvalues level0
                level1 key4]
#prints -> sdf
```

Chapter 6 : Files & Error Handling

Eval Command

Evaluate the variable comes after it

```
set script {
set a 1
}
eval $script;# -> it will exec script
```

→ we used it in double dolar sign befor

2) Exec Command:

execute Command

exec Python3 hello.py

Set Cmd "python3 hello.py"

eval exec $Cmd

3) Error Handling:

- error : Throw error

error Msg ~~Code~~ Info Code

- Catch : to execute things if in case of failure

Catch { Code } errMsg

Chapter 5: Strings

1. String operations:

- trim → trimleft, trimright
  → Don't change original string

Format
string trim $statement character that's trimmed

string trim $stat -

- length
- index

Format
string index $statement index

puts "[string index $stat 9]"

- first ⇒ get the first occurance of substring

Format
string first "str" $statement

return the index of first occurance

- range : extracts a range of characters from a string

Format
string range $stat start end

- replace : Replaces a range of characters in a string

format

String replace $stat start end "str"

=> Take Care it doesn't change original string

- Compare : Compares two strings

format

String compare $stat1 $stat2

→ return 0 if They are equal

→ return 1 if stat1 > stat2

→ return -1 if stat2 > stat1

* Comparison is on Ascii

- tolower
- toupper

String tolower $stat
toupper

- **wordend / wordstart** : Finds the End or start of a word in a String

format

string wordend $stat index
wordstart

if wordend → return the end index of the word that starts with index given in command

if wordstart → return the start index of the word that ends with index

- **String match** : Checks if a String matches a pattern

format

String match ~~**~~pattern $stat

puts [String match "*student" $statement]

* Tcl treats everything as strings, it can do the following on strings:

→ Slice → Join → Substitute

→ Format → Compare → Search

1. Compare & Search

① → == | eq : Exact, return 0 or 1 → $str1 == eq $str2

② → String equal : Exact, ignore case, Prefix
return 0, 1

③ → String compare : Exact, ignore case
return -1, 0, 1

④ → string match : Exact, ignore case,
simple patterns
return 0, 1

Note: When we want to write pattern & concatenate something to variable

"${str2}*" Not "$str2*"

as it will see that there is variable called str2*

37

Note:

Anything inside { } is treated as it's like: set a {2\3}

escape character

5 glob : Files; IgnoreCase (always); Simple Patterns
Examples: -return list of strings

glob * ; # → return all Filenames in Current directory

glob -type d * ; # → return all directories only

glob *.tcl

glob *{.tcl, .txt} ; # glob introduce to separate by comma patterns
— or —

glob -d Search {pattern}

↓

directory in The current directory.

glob - tail -d Search &pattern}

To get just the filename without Search

Note: -noComplain

when glob doesn't find the pattern, it gives error & stop script. So we use the flag -noComplain to prevent this from happening

Exercise: Make script to print all tcl files recursively starting from current directory.

⑥ → regexp: Exact, IgnoreCase, Complex patterns
return 0, 1 + string
return indices

Format

regexp &pattern} $stat <matched string>
<Groups if any>

Group → any pattern between (.)

- alpha : represents any unicode Alphabet character

regexp {[[:alpha:]]} $Statement

- alphanum : alphabet + numbers

- line : Flag in regexp to Match Across lines

- -nocase : flag in regexp for case insensitive matching

- Start : Flag in regexp to start from Specific position

regexp -start d {pattern} $string1

- all : flag in regexp to Match all occurance return # of occurances

- inline : Flag in regexp to return Matches inline

if we used -all -inline together → return list of all Matches

- They will return list that has Full Match Group -->

- Indicies : Flag, to Get Match positions
  - → return First match
  - → with_all → return all Matches positions

Learn more on: https://www.tcl-lang.org/man/tcl/TclCmd/re_syntax.htm
(capital)

⑦ → reg sub : Exact, ignore case, Complex patterns
  - → return String
  - return indicies

Format

```
regsub <pattern> $list <newSub> newlist
```

regsub "Hoda" $lst Students "Nour" newlist

- & (Ampersand) → Matched Substring in regsub
  - ↳ Full Match

regex regsub -all {(\w+)} $list "-&-" newlst

9)

regsub {([^\.]*)\.c$} fil.c

{cc -c $1.o} cCmd

↓

First Group

⑧ → l.search: Exact, Ignore case, Simple patterns,

complex patterns;

→ return string

→ return indicies.

lsearch -all -regexp $lst {(M|W+)}

return indicies of Matches in $lst

lsearch -all -inline -regexp $lst {(M|W+)}

return the Matching substring

lsearch -all -inline -glob $lst {M*}

Use simple patterns used with glob & Meta

Simple patterns For glob/Match

* → Zero or more character of any type
Can be alone without any preceeding character unlike Regex

? → Match only character without any preceding character

[] → like in Regex

~~Chapter 7 : Creating Unique Names~~

الحمد لله الذي هدانا لهذا وما كنا لنهتدي لولا أن هدانا الله