SHELL VARIABLES

A shell variable is a <u>Keyword</u> that is set by the shell for a specific use.

— Typically in <u>UPPERCASE</u>; different from most other commands that are entered in <u>lowercase</u>.

— Display the contents of an individual variable by using <u>echo and placing a $ prior to the variable</u>.

PWD →	Most recent <u>current</u> working directory set with the cd command
OLDPWD →	Previous working directory set by the cd command
BASH →	Full path name used to invoke the bash shell (/bin/bash)
RANDOM →	Random integer between 0 and 32,767
HOSTNAME →	Current hostname of the system running Linux
PATH →	Contains a list of directories that are used to search for commands within the Linux tree hierarchy

②

HOME → Home directory of the current user. Each user has a home directory when their account is created.

TMOUT → Represents the amount of time the shell waits, without user input, before exiting current shell. "Timeout" the user's session.

export TMOUT=120
↑ seconds

PATH and Shell Environment:

- when you enter a command using a partial path such as ~~[illegible]~~ more, how does the shell know how to execute it?

- When a partial path is used while executing a command, the shell looks at the contents of PATH until it finds directory in which command is located.

- Each of the contents (directories) separated by a delimiter (:) colon

- Shell searches each directory from LEFT to RIGHT

- If directory where the command is located is found, command executed. Otherwise, ERROR.

- When executing a command using its full path, such as /bin/more the shell does not refer to PATH. It goes directly to command using the specified path.

Example /bin/more ... shell moves directly to the /bin directory, which is one of the system directories.

SHELL BUILTIN COMMANDS

- These are commands that are part of the shell program.

- They are compiled into the shell (not availa in any system directory)
- Cannot be modified or deleted

Examples

1. cd - change current directory
2. declare - declare a variable; -r makes read only

   declare var2

   declare -r ← readonly var

3. echo

4. exit - exit shell with a status

   exit 1

④

- **history** (history of commands)

- **kill** (To kill a process)

  kill $\overset{\text{PID}}{\text{(process number)}}$

  To Look at process lists

  ps U (login)

- **let** evaluates an expression let x=4+5.

- **local** creates a local variable local x=5

- **read** reads character from keyboard

  readonly → $\boxed{\begin{array}{l}\text{read } x \\ \text{readonly } x\end{array}}$

- return causes a function to exit with a certain value.

  $\boxed{\begin{array}{l}1 - \text{failure} \\ 0 - \text{success}\end{array}}$

SHELL GRAMMAR

- Rules to be followed for <u>proper operation of</u> the shell.

- 4 Building blocks of a shell grammar

(1) <u>Blank</u> — $\boxed{\text{Space}}$ OR $\boxed{\text{TAB}}$ used to separate items in the shell

(2) <u>Word or Token</u> — Sequence of characters considered a single unit

RESERVED WORDS - cannot be used e.g. if, then, else

(3) <u>Name</u> — word that consists of letters, numbers and underscore

(4) <u>Metacharacter</u> — A character used for a <u>specific purpose</u>

⇓

SHELL META CHARACTERS SYMBOLS

|	pipe	Pass command output to another command
&	ampersand	Run job in background
;	Semicolon	Put multiple commands on the same line
( )	L and R paranthesis	Allows you to run a command in a <u>sub shell</u>

GOOD FOR PARALLEL PROCESSING {
- another shell is spawned
- Helps in running multiple shells in background

Less than
< — Redirect input

> — Redirect output
Greater than

Control Operators is a token that performs specific control function

→ || — Two pipes — Command executes upon failure of another

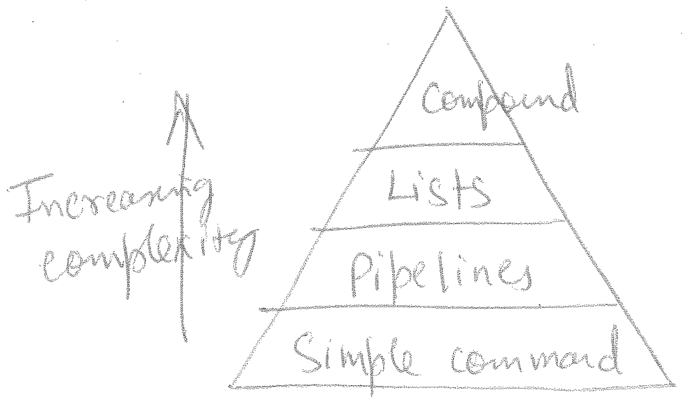
→ && — Command executes upon successful completion of another

---

Examples

1. echo "Hello"; who; pwd; date
2. (ls; pwd; who; date; cal 12 2005;)
3. pwd && date { date only executes on successful completion of pwd)
4. ~~cd~~ ls disk.dat || date { date only executes on failure of ls command }

Understanding Command Types



for Properly structuring your commands in scripts.

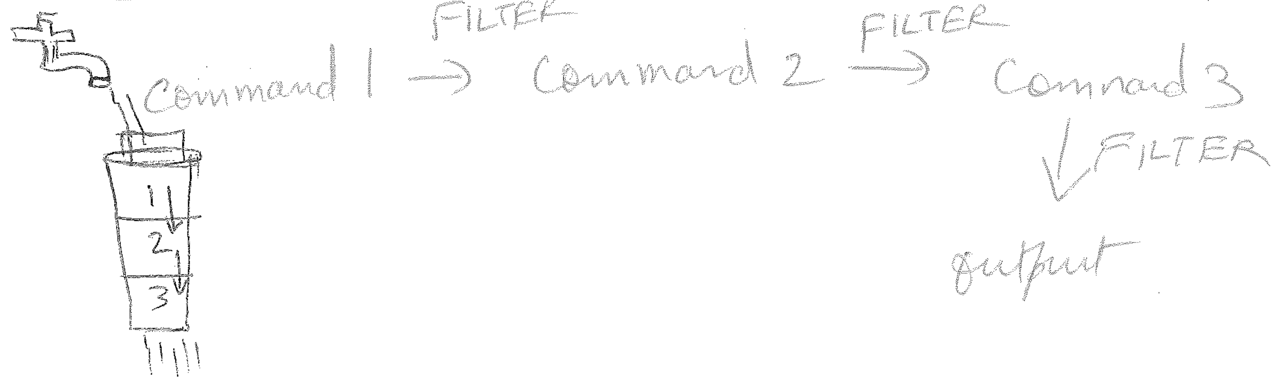
1. Simple command → Most basic type of operation you can do with a shell

2. Pipelines → Allows data to be parsed between processes.

```
Command 1 | Command 2 | Command 3
```

e.g.
```
ls | more
ls | sort
who | sort | more
```

OUTPUT of COMMAND ON LEFT IS USED AS INPUT FOR COMMAND ON THE RIGHT



3. List :

Sequence of 1 or more pipelines separated by one of these operators

$\boxed{;\ ,\ \&\ ,\ \&\&\ ,\ ||}$

$\boxed{\text{List terminated by } ;\ , \&\text{, or new line character}}$

OPERATORS

(a) ; : comamand1 ; command2

(b) & : Background process vs foreground process

$\downarrow$ Does not lock shell. Stop using kill

$\downarrow$ Locks shell. Stop using Ctlr + C

e.g, ls & pwd

(c) && : Causes shell to execute a command only if the preceding command completed succesfully (exit status of 0).

e.g. , lsxxx && pwd [Does not work]

(d) || : opposite of &&
look for exit status ≠ 0.

e.g.; lsxxx || pwd [works]

COMBINING COMMANDS & OPERATORS

(C) Combining commands and Operators. from (a)–(d) to get a list

E.g. ~~(1)~~ date ; pwdx && who | more ← not executed

~~(2)~~ date works, but others do not.

date ; pwd || who | more
not executed

lists

COMPOUND COMMANDS → Create Loops → Perform Calculation → Assign Variables → Decision Tests

~~(1)~~ + Group of commands executing in a Subshell ← (list)
Or the current shell ← {list; }

(1) $\boxed{\text{(list) group command}}$

```
y=5
echo $y
→5
(y=50)
echo $y
→5
```

variables do not keep changes!

(y=50) ← executed in a spawned shell; thus y does not change.

Similar to &, but you can only place one command befor & (e.g. who &) with group command list ( ) you can run multiple commands in background

e.g. (pwd ; who ; ls)

(10)

(2) $\boxed{\{\ \text{list;}\ \}\quad \text{Group commands}}$ Current Shell

- group of command executed in current shell
- list must be terminated with a <u>semicolon</u>

e.g. * $\underbrace{\{\ \text{who ; ls -l ;}\ \}\ |}_{\text{Collective output is piped to more}}$ more

*
```
y=5
echo $y
→ 5
{ y=50; }
echo $y
→ 50
```
Executed in same shell, hence variable maintain changes

EXPRESSIONS (another type of compound commands)

— Used when you want to assign a value to a variable, perform arithmetic calculations, etc.

Two formats:

+ ((expression)) $\longrightarrow$ Two forms
  - ((var_name = Value1 operator Value2))
  - ((Value1 operator Value2))
+ [[expression]]

Operators used with ((expression))

1. ((t= x++) — Increment
2. ((t= x--) — Decrement
3. ((x=2**3)) — Exponentiation 2^3
4. *, /, +, —.
5. % ((x=100%4)) — Remainder
6. (($x = =2)) equal to
7. (($x != 2)) not equal to
8. (($y=5 && $t ==5)) AND operation
9. OR operation using ||

e.g.

```
((x=5*6))
((y=$x+4))
echo $x $y
```

Precedence of arithmetic operations

1. ++, --
2. **, *, /, %
3. +, -
4. <=, >=, <, >
5. ==, !=
6. &&
7. ||

~~Precende~~

Precedence changed by using parantheses ( )

e.g.

```
((x= 100-3**2))
((y =(100-3)**2))
echo $x $y
```

[[expression]]

- Used to test attributes of a file
- String comparisons
- Numeric comparisons
- Used with the if command to make decision.