Lecture 29: Command Substitution

$var=' command' (back quote)
Evaluation of shell variables: ${}
computation on shell variables : expr command
Local and Global shell variables : export command

Another way of placing values into variables is by command substitution, where the output of a command is placed into a variable as its content.

```
$ echo date
date

$ echo 'date'
Wed Aug 23 12:43:34 IST 1995
$
```


Here the argument to the first echo is a string 'date'. So it is displayed as it is.

To the second echo command, date is provided within back quotes *C),* the character left to the digit one (1) on the standard key pad of the key board.

In the second echo, the date command is evaluated first and the result is supplied to echo as an argument.

Examples:

$ echo there are 'who | wc -1' users working on the system
will produce the output

there are 6 users working on the system

However, the number of the users depends upon the actual number of users, who are working on the system at the time of issuing the command.

In a similar manner, the output of a command can be placed into a variable for further reference.

```
$ cur_dir='pwd'
$ echo current directory is $cur_dir
```

Evaluation of shell variables

$var	: value of the variable 'var'
${var-value}	: value of the variable, if defined, otherwise 'value'
${var=value}	: value of the variable, if defined, otherwise 'value' is assigned to the variable
${var+string}	: value of string, if var is defined, otherwise nothing
${var?messg}	: value of the variable, if defined, otherwise the shell exits after printing the 'messg'

- Braces are used to define the values for the shell variables depending upon whether a variable is defined or not.

The Bourne shell tests the values of the variables and takes a decision accordingly.

Example: Conditional substitution of values for the variables.

```
$ name=mano
$ echo ${name-ajay}
mano            as name has been defined

$ echo ${year-1995}
1995            will take the value 1995 as year has not been defined

$ echo ${class=zoom} zoom
$ echo $class

zoom            as class has been assigned a value 'zoom'

$ echo ${class+excel}
excel           as the variable class has already been defined

$ echo ${drinks+pepsi}
                a null string as drinks has not been defined
```


If we execute the following commands as a shell script:

```
echo $ {drinks? "not available"}
echo done
```

the output will be "not available" only and it terminates the further execution of the script and comes to the prompt.

Computation on shell variables

- expr
- expr val1 op val2
- expr $var1 op $var2
- var3=' expr $var1 op $var2'

Arithmetic computations on the shell variables are done through the expr command. The expr command takes an arithmetic expression, evaluates it and gives the result.

```
$ expr 5 + 7
12
$ expr 6 - 2
4
$ expr 3 \* 4
12
$ expr 24 / 3
8
```

Note that the arguments of the expr command must be separated by a blank space from one another. The special symbols must be preceded by a back slash (\) so that the shell does not expand them. The division of expr is an integer division, Le. it displays only the integer portion of the result.

The output of the expr command can be stored in a variable. Arithmetic computation of variables are also possible through the expr command.

Examples:

```
$ sum='expr 5 + 6'
```

```
$ echo $sum              will display 11

$ a=12
$ b=90
$ echo sum is $a + $b
sum is 12 + 90           will display the arguments.
$ echo sum is 'expr $a + $b'
sum is 102               will store 102 in the variable sum
$ sum='expr $a + $b'
```

Local and Global variables

Execution of a shell program leads to the creation of a **child process** with a new environment. The shell programs are executed in this new child process.

Once the execution of the child process is complete, this new **child process** is destroyed.

By default the shell variables are **local** to the shell that creates them. The values of these shell variable are not available to the newly created child process.

In order to make these variables available to the child process, we should declare them as gl_bal. **UNIX** provides the **export** command, which is used to declare a variable as global.

```
$ TERM=vtlOO
$ export TERM
```

will assign the value **vt100** to the variable **TERM** and the **export** command makes it global.

The exported variable will only be accessible by the child shell. A variable can not be accessed by any parent shell, if it is declared and exported by a child shell.

Classroom Exercise:

Write the method to add the ***'/usr/mano/src'*** directory to the search path.

Conditional Execution Operators

&& :$ command1 && command2

|| : $ command1 II command2

if…then…elif…else...fi construct

```
if <condition>
then
        <command1>
elif <condition>
then
        <command2>
 else
        <command3>
 fi
```

Conditional execution of commands are useful when we want to execute the command based on the status of the previous command, i.e. whether the previous command has succeeded or failed.

This is examined by the **exit status** of each command.

The exit status of a command indicates whether the command was successful or not. It can either be 0 if successful or 1 if unsuccessful. Conditional operators check this exit status and behave

accordingly.

.The operator '&&' executes the command(s) following it, if and only if the preceding command was successfully completed.

.The 'II' (double pipe) operator executes the command(s) following it, if the preceding command failed.

Examples:

$ Is I grep "mydoc.doc" && rm mydoc.doc

The above command will remove mydoc.doc if it exists, otherwise, it will do nothing.

$ cat mydoc.doc || echo "file not found"

The above command will display the contents of mydoc.doc (cat if used) if it exists. In case, the file is not present, message followed by the echo command gets displayed.

In case, more than one command is to be executed or more than one condition need to be checked simultaneously, then this type of conditional execution is not helpful.

In such cases the if...then ....elif...else...fi construct is used. It gives the user the flexibility of conditional execution and provides a structured -programming approach. This construct is used maihTy" in shell programming.

.if... then...elif ...else...fi construct

The condition to be evaluated by the if command is based on the exit status of command(s) that follow it.

The following steps are followed in the execution of the if ..... then .... else fi construct:

1. The command following the **if** is executed.
2. If the command is successful, Le., exit status is '0', then the command(s) between the then and else get executed.
3. If the command returns an exit status '1', Le., the command failed, the else part of the construct is executed.

Example: We can modify the earlier kopy script file such that it check whether the target file is existing or not. If it exists, it will not overwrite the target and display a message to the user.

```
# kopy: make a duplicate copy of a file
echo "Enter source filename:\c"
read source
echo "Enter target file name:\c"
read target
if ( Is | grep "$target")
then
        echo target exists
else
        cp"$source $target
fi
# end of script
```

The else part of the if statement will be executed only when the command following the if failed.

Grouping of commands is done using the parenthesis '( )'. In this case the exit status will be the exist status of the last command in the pipe.

The elif clause is a combination of else...if clause.

Example: Modify the script in the above example, to check the presence of the source also.

```
#kopy: make duplicate copies of files
echo "Enter source filename:\c"
read source
echo "Enter target file name:\c"
read target
if ( ls |grep "$target")
then
        echo target exists
elif (ls | grep "$source")
then
        cp $source $target
else
        echo Source does not exist
fi
# end of script
```


The if...then ....else .....fi construct provides nested if construct to a Shell scripl.

. test

Operators on numeric variables : eq, ne, gt, It, Ie, ge

Operators on string variables : =, !=, z, n

Operators on files : s, f, d, w, r, x

Logical comparison operators : a, o!

The working of the if construct depends upon whether or not a command has a successful exit status. But the status of a command is not the only matter of interest.

Other factors such as comparing values of shell variables, evaluating a file type or existence of a file or directory also influence shell programming.

All commands return the exit status to a Pre-defined shell variable '?' (question mark), which can be displayed using the echo command.

$ cat mydoc.doc	If the cat command is successful there will be some output on the std output since there is no redirection
$ echo $?	a dollar sign is required as it is a variable
0	as the previous command was successful

The UNIX system provides test commands which investigate the exist status of the previous command and translate the result in the form of success or failure, i.e. either a '0' or '1 '.

The test command does not produce any output, but its exit status can be passed to the if statement to check whether the test failed or succeeded.

The test command has specific operators to operate on files, numeric values and strings.

.Operators on numeric variables

-eq equals to

-ne	not equals to
-gt	greater than
-It	less than
-ge	greater than or equal to
-Ie	less than or equal to

Example:

$a=12; b=23	
$test $a -eq: $b	testing for equality of the variables 'a' and 'b'
$echo $?	checking the exit status
1	false, as value of 'a' is not equal to the value of 'b'
$	

.Operators on string variables

=	equality of strings
!=	not equal
-z	zero length (string contains zero characters: null string)
-n	string length is non zero

Example:

$ name="ajay"	
$ test.-z $name	will return the exit status 1 as the string name is not null
$ test -n $name	will return 0 as the string is not null
$ test -z "$address"	will return 0 as the variable has not been defined
$ test $name = "Ajay"	will return 1 as the value of name is not equal to "Ajay"

.Operators on files

-f	the file exists
-s	the file exists and the file size is non zero
-d	directory exists
-r	file exists and has read permission
-w	file exists and has write permission
-x	file exists and has execute permission

Example:

$ test -f "mydoc.doc" will check for the file mydoc.doc, if exist, returns 0 else

1

$ test -r "mydoc.doc" will check for the read permission for mydoc.doc

$ test -d "$HOME" will check for the existence of the user's home directory

.Logical Operators

Combining more than one condition is done through the logical AND, logical OR and logical NOT operators.

-a	logical AND
-o	logical OR
!	logical NOT

Example:

$ test -r "mydoc.doc" –a -w "mydoc.doc"

will check both the read and write permission for the file mydoc.doc and returns either 0 or 1 depending upon the result.

$ echo $? <Enter> will show the exit status of the above command.

Classroom Exercise

Write the command to check whether a file is a directory and has read, write and execute permissions.