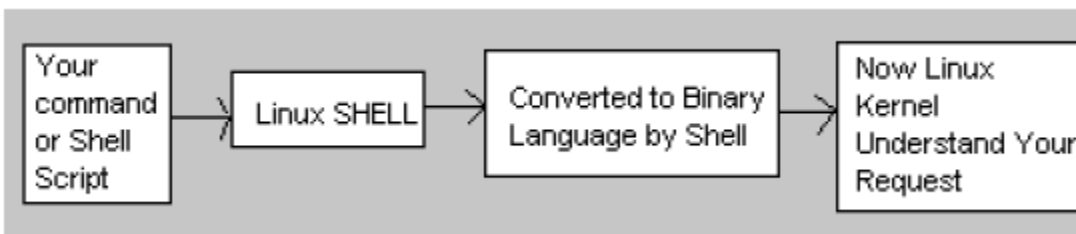


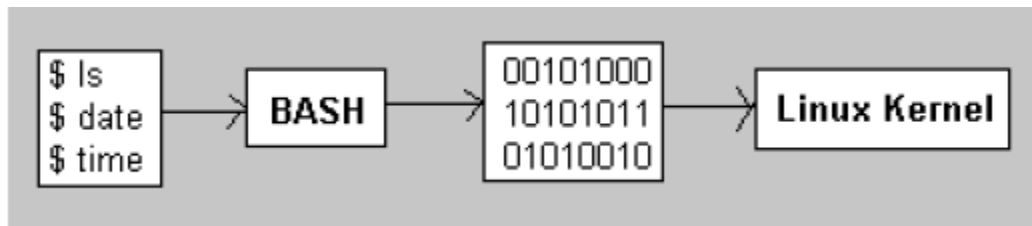
PSG COLLEGE OF TECHNOLOGY, COIMBATORE-641 004
DEPARTMENT OF APPLIED MATHEMATICS AND COMPUTATIONAL SCIENCES
Unix Shell and System Programming Lab – Shell Scripting

What's Linux Shell

Computer understand the language of 0's and 1's called binary language, In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in O/s there is special program called Shell. Shell accepts your instruction or commands in English and translates it into computers native binary language.



You type Your command and shell convert it as



Its environment provided for user interaction. Shell is command language interpreter that executes commands read from the standard input device (keyboard) or from a file. Linux may use one of the following most popular shells.

Why shell scripting?

- Shell scripts can take input from a user or file and output them to the screen.
- Whenever you find yourself doing the same task over and over again you should use shell scripting, i.e., repetitive task automation.
 - Creating your own power tools/utilities.
 - Automating command input or entry.
 - Customizing administrative tasks.
 - Creating simple applications.

- Since scripts are well tested, the chances of errors are reduced while configuring services or system administration tasks such as adding new users.

Practical examples where shell scripting actively used

- Monitoring your Linux system.
- Data backup and creating snapshots.
- Dumping Oracle or MySQL database for backup.
- Creating email based alert system.
- Find out what processes are eating up your system resources.
- Find out available and free memory.
- Find out all logged in users and what they are doing.
- Find out if all necessary network services are running or not. For example if web server failed then send an alert to system administrator via a pager or an email.
- Find out all failed login attempt, if login attempt are continue repeatedly from same network IP automatically block all those IPs accessing your network/service via firewall.
- User administration as per your own security policies.
- Find out information about local or remote servers.
- Configure server such as BIND (DNS server) to add zone entries.

Advantages

- Easy to use.
- Quick start, and interactive debugging.
- Time Saving.
- Sys Admin task automation.
- Shell scripts can execute without any additional effort on nearly any modern UNIX / Linux / BSD / Mac OS X operating system as they are written an interpreted language.

Disadvantages

- Slow execution speed.
- A new process launched for almost every shell command executed.

Bash

Bash is the shell, or command language interpreter, for the Linux operating system. The name is an acronym for the Bourne-Again SHell, a pun on Stephen Bourne, the author of the direct ancestor of the current Unix shell sh, which appeared in the Seventh Edition Bell Labs Research version of Unix Bash Reference Manual.

Introduction to BASH

- Developed by GNU project.
- The default Linux shell.
- Backward-compatible with the original sh UNIX shell.
- Bash is largely compatible with sh and incorporates useful features from the Korn shell ksh and the C shell csh.
- Bash is the default shell for Linux. However, it does run on every version of Unix and a few other operating systems such as ms-dos, os/2, and Windows platforms.

Quoting from the official Bash home page:

Bash is the shell, or command language interpreter, that will appear in the GNU operating system. It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard. It offers functional improvements over sh for both programming and interactive use. In addition, most sh scripts can be run by Bash without modification.

The improvements offered by BASH include:

The Bash syntax is an improved version of the Bourne shell syntax. In most cases Bourne shell scripts can be executed by Bash without any problems.

- Command line editing.
- Command line completion.
- Unlimited size command history.
- Prompt control.
- Indexed arrays of unlimited size (Arrays).
- Integer arithmetic in any base from two to sixty-four.
- Bash startup files - You can run bash as an interactive login shell, or interactive non-login shell. See Bash startup files for more information.

- Bash conditional expressions: Used in composing various expressions for the test builtin or [[or [commands.
- The Directory Stack - History of visited directories.
- The Restricted Shell: A more controlled mode of shell execution.
- Bash POSIX Mode: Making Bash behave more closely to what the POSIX standard specifies.

Bash v4.0 Features

- Usual run time environment: POSIX
- Command and file name completion - Bash can automatically fill in partially typed commands or arguments to the commands such as file name, hostname and much more.
- Arithmetic support:
 - Integer arithmetic supported.
 - Floating point arithmetic is not supported.
 - Exponential notation is limited via printf builtin.
 - Date and time arithmetic is not supported.
- Hash table: Bash uses a hash table to remember the full pathnames of executable files.
- Pattern Matching and regular expressions are supported.
- Globbing - For example, you can use *.conf to match all those conf files in /etc directory.
- Directory stack is supported via pushd and popd builtins.
- Command history and History completion fully supported by Bash.
- Custom command prompt - Allows you to change the default prompt.

The bash shell understands the following types of commands:

- **Aliases** such as ll
- **Keywords** such as if
- **Functions** (user defined functions such as genpasswd)
- **Built in** such as pwd
- **Files** such as /bin/date

The role of shells in the Linux environment

Shell is used for various purposes under Linux. Linux user environment is made of the following components:

- Kernel - The core of Linux operating system.
- Shell - Provides an interface between the user and the kernel.
- Terminal emulator - The xterm program is a terminal emulator for the X Window System. It allows user to enter commands and display back their results on screen.
- Linux Desktop and Windows Manager - Linux desktop is collection of various software apps. It includes the file manger, the windows manager, the Terminal emulator and much more. KDE and Gnome are two examples of the complete desktop environment in Linux.

You need to provide username and password to login locally into the console. Bash uses the following initialization and start-up files: **(Browse for further explanation for these following files information- Self Study)**

1. /etc/profile - The systemwide initialization file, executed for login shells.
2. /etc/bash.bashrc - The systemwide per-interactive-shell startup file. This is a non-standard file which may not exist on your distribution. Even if it exists, it will not be sourced unless it is done explicitly in another start-up file.
3. /etc/bash.logout - The systemwide login shell cleanup file, executed when a login shell exits.
4. \$HOME/.bash_profile - The personal initialization file, executed for login shells.
5. \$HOME/.bashrc - The individual per-interactive-shell startup file.
6. \$HOME/.bash_logout - The individual login shell cleanup file, executed when a login shell exits.
7. \$HOME/.inputrc - Individual readline initialization file.

Bash Startup Scripts

Script of commands executed at login to set up environment. For example, setup JAVA_HOME path.

Login Shell

Login shells are first shell started when you log in to the system. Login shells set environment which is exported to non-login shells. Login shell calls the following when a user logs in:

- /etc/profile runs first when a user logs in runlevel # 3 (the level numbers may differ depending on the distribution).
- \$HOME/.bash_profile, \$HOME/.bash_login, and \$HOME/.profile, runs second when a user logs in in that order. \$HOME/.bash_profile calls \$HOME/.bashrc, which calls /etc/bashrc (/etc/bash.bashrc).

Non-Login Shell

- When an interactive shell that is not a login shell is started, bash reads and executes commands from /etc/bash.bashrc or /etc/bashrc and \$HOME/.bashrc, if these files exist. First, it calls \$HOME/.bashrc. This calls /etc/bash.bashrc, which calls /etc/profile.d.

Bash Logout Scripts

- When a login shell exits, bash reads and executes commands from the file \$HOME/.bash_logout, if it exists.

Other standard shells

In Linux, a lot of work is done using a command line shell. Linux comes preinstalled with Bash. Many other shells are available under Linux:

- tcsh - An enhanced version of csh, the C shell.
- ksh - The real, AT&T version of the Korn shell.
- csh - Shell with C-like syntax, standard login shell on BSD systems.
- zsh - A powerful interactive shell.
- scsh- An open-source Unix shell embedded within Scheme programming language.

Pathnames of valid login shells

/etc/shells is a text file which contains the full pathnames of valid login shells. This file is consulted by chsh (to change shell like from bash to sh) and available to be queried by other programs such as ftp servers.

```
cat /etc/shells
```

Sample outputs:

```
/bin/sh
```

```
/bin/bash
```

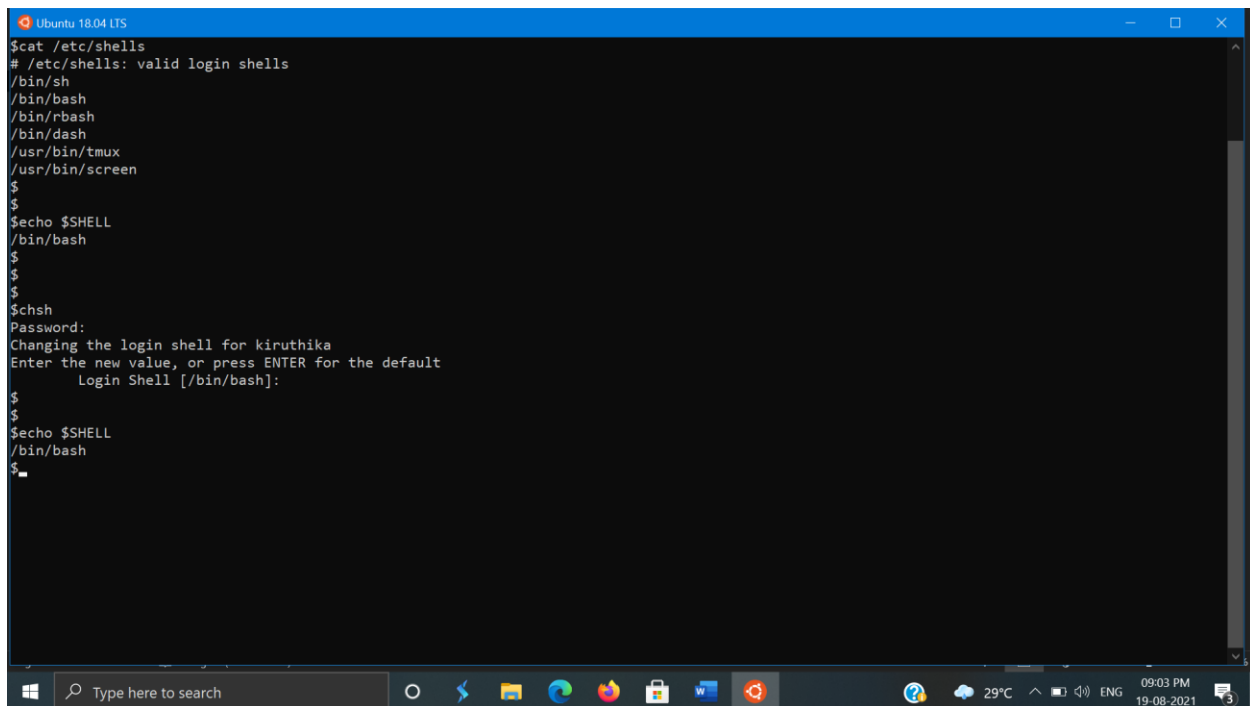
```
/sbin/nologin
```

```
/bin/tcsh
```

```
/bin/csh
```

```
/bin/zsh
```

```
/bin/ksh
```



```
Ubuntu 18.04 LTS
$cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/bin/rbash
/bin/dash
/usr/bin/tmux
/usr/bin/screen
$
$
$echo $SHELL
/bin/bash
$
$
$
$chsh
Password:
Changing the login shell for kiruthika
Enter the new value, or press ENTER for the default
  Login Shell [/bin/bash]:
$
$
$echo $SHELL
/bin/bash
$
```

Any of the above shell reads command from user (via Keyboard or Mouse) and tells Linux O/s what users want. If we are giving commands from keyboard it is called command line interface (Usually in-front of \$ prompt, This prompt is depend upon your shell and Environment that you set or by your System Administrator, therefore you may get different prompt).

NOTE: To find your shell type following command **\$ echo \$SHELL**

- In Operating System, there is special program called Shell. Shell accepts your instruction or commands in English (mostly) and if it's a valid command, it is passed to kernel.
- Shell is a user program or it's an environment provided for user interaction.
- Shell is a command language interpreter that executes commands read from the standard input device (keyboard) or from a file.
- Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.
- To find all available shells in your system type following command:
 - **\$ cat /etc/shells**
- Note that each shell does the same job, but each understand different command syntax and provides different built-in functions.
- To find your current shell type following command
 - **\$ echo \$SHELL**
- To use shell (You start to use your shell as soon as you log into your system) you have to simply type commands.

Getting started with Shell Programming

How to write shell script

- Use any editor to write shell script.
- After writing shell script set execute permission for your script as follows:


```
$ chmod +x your-script-name
```

(or)

```
$ chmod 755 your-script-name
```
- Execute your script as;


```
$ bash your-script-name
```

```
$ sh your-script-name
```

```
$ ./your-script-name
```

Note: In the last syntax ./ means current directory.

Now you are ready to write first shell script that will print "Knowledge is Power" on screen.


```
# My first shell script
```

```
clear
```

```
echo "Knowledge is Power"
```

After saving the above script, you can run the script as follows:

```
$ ./first
```

This will not run script since we have not set execute permission for our script first; to do this type command

```
$ chmod 755 first
```

```
$ ./first
```

Note: Statements preceded by # are comments

Ex:2 Script to print user information who currently login , current date and time

```
#Script to print user information who currently login, current date# & time
```

```
echo "Hello $USER"
```

```
echo -e "Today is \c ";date
```

```
echo -e "Number of users logged in : \c" ; who | wc -l
```

```
echo "Calendar"
```

```
cal
```

```
exit 0
```

Commands Related with Shell Programming

echo [options] [string, variables...] - Displays text or variables value on screen.

Options

-n Do not output the trailing new line.

-e Enable interpretation of the following backslash escaped characters in the strings:

\a alert (bell)

\b backspace

\c suppress trailing new line

\n new line

\r carriage return

\t horizontal tab

\\ backslash

For eg. \$ **echo -e "An apple a day keeps away \a\t\tdoctor\n"**

Variables in Shell

In Linux (Shell), there are two types of variable:

System variables- Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.

Note: Use the command **printenv** to see environment variables. You can see system variables by giving commands like **set** or **env** some of the important System variables are:

Table 1: Some of the important System variables

System	Variable Meaning
BASH	Our shell name
BASH_VERSION	Our shell version name
COLUMNS	No. of columns for our screen
LINES	No. of columns for our screen
PS1	Our prompt settings
HOME	Our home directory
OSTYPE	Our Os type
PATH	Our path settings
PWD	Our current working directory
SHELL	Our shell name
LOGNAME	Our logging name
USERNAME	User name who is currently login to this PC

User defined variables - Created and maintained by user. This type of variable defined in lower letters. The following script defines three variables;

```
# Script to test MY knowledge about variables!
#
myname=Vivek
myos = TroubleOS
myno=5
echo "My name is $myname"
echo "My os is $myos"
echo "My number is myno, can you see this number ?"
```

Shell Arithmetic - Used to perform arithmetic operations.

Syntax:

expr op1 math-operator op2

Examples:

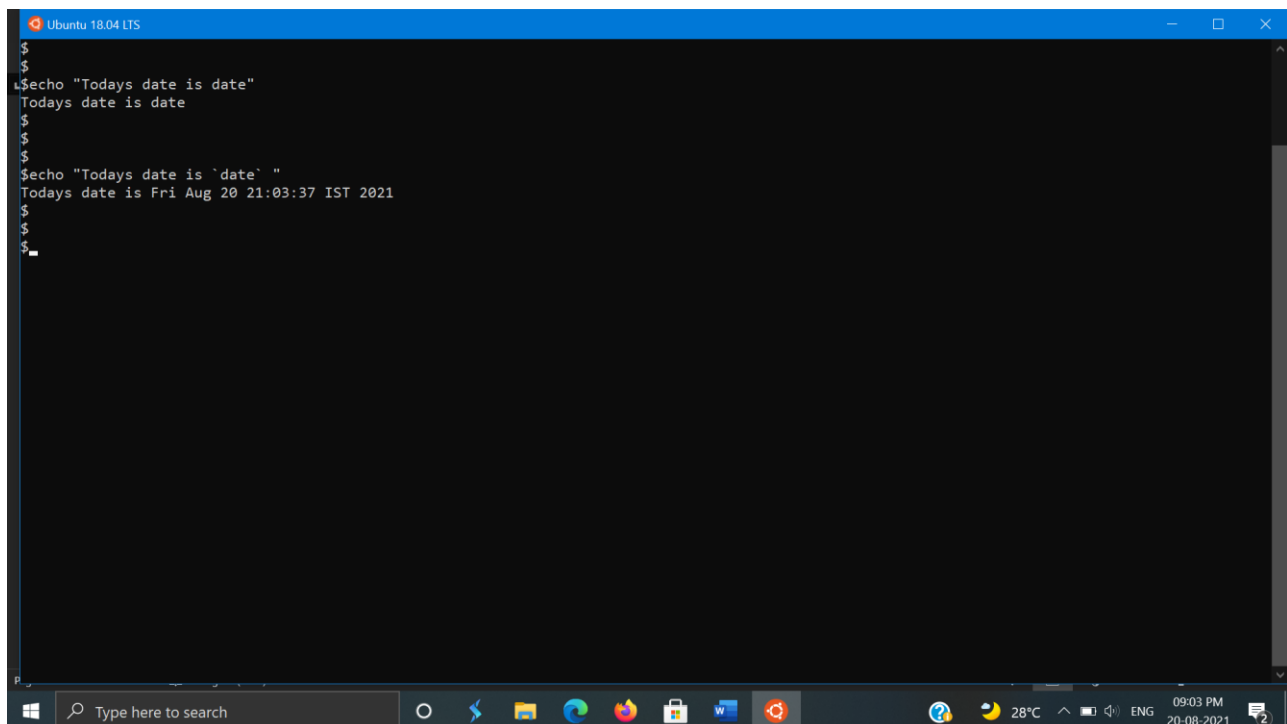
```
$ expr 1 + 3
$ expr 2 - 1
$ expr 10 / 2
$ expr 20 % 3
$ expr 10 \* 3
$ echo `expr 6 + 3`
```

Note : Before expr keyword we used ` (back quote) sign not the (single quote i.e. ') sign. expr also end with ` i.e. back quote. If you use double quote or single quote, it will NOT work.

There are three types of quotes

1. Double quotes (") - Anything enclosed in double quotes removed meaning of that characters (except n and \$).
2. Single quotes (') - Enclosed in single quotes remains unchanged.
3. Back quote (`) - To execute command

```
$ echo "Today is date"
$ echo "Today is `date`".
```



The screenshot shows a terminal window titled 'Ubuntu 18.04 LTS'. The terminal output is as follows:

```
$
$
$echo "Todays date is date"
Todays date is date
$
$
$
$echo "Todays date is `date` "
Todays date is Fri Aug 20 21:03:37 IST 2021
$
$
$
$
```

The terminal window is running on a Windows desktop, as evidenced by the taskbar at the bottom showing various application icons and system information like '28°C' and '09:03 PM 20-08-2021'.

Exit Status

By default in Linux if particular command/shell script is executed, it return two type of values which is used to see whether command or shell script executed is successful or not.

1. If return value is zero (0), command is successful.
2. If return value is nonzero, command is not successful or some sort of error executing command/shell script. This value is known as Exit Status.

To find out exit status of a command or shell script; use `$?` special variable of shell.

```
$ rm unknownfile
```

```
$ echo $?
```

```
$ ls
```

```
$ echo $?
```

```
$ expr 1 + 3
```

```
$ echo $?
```

```
$ echo Welcome
```

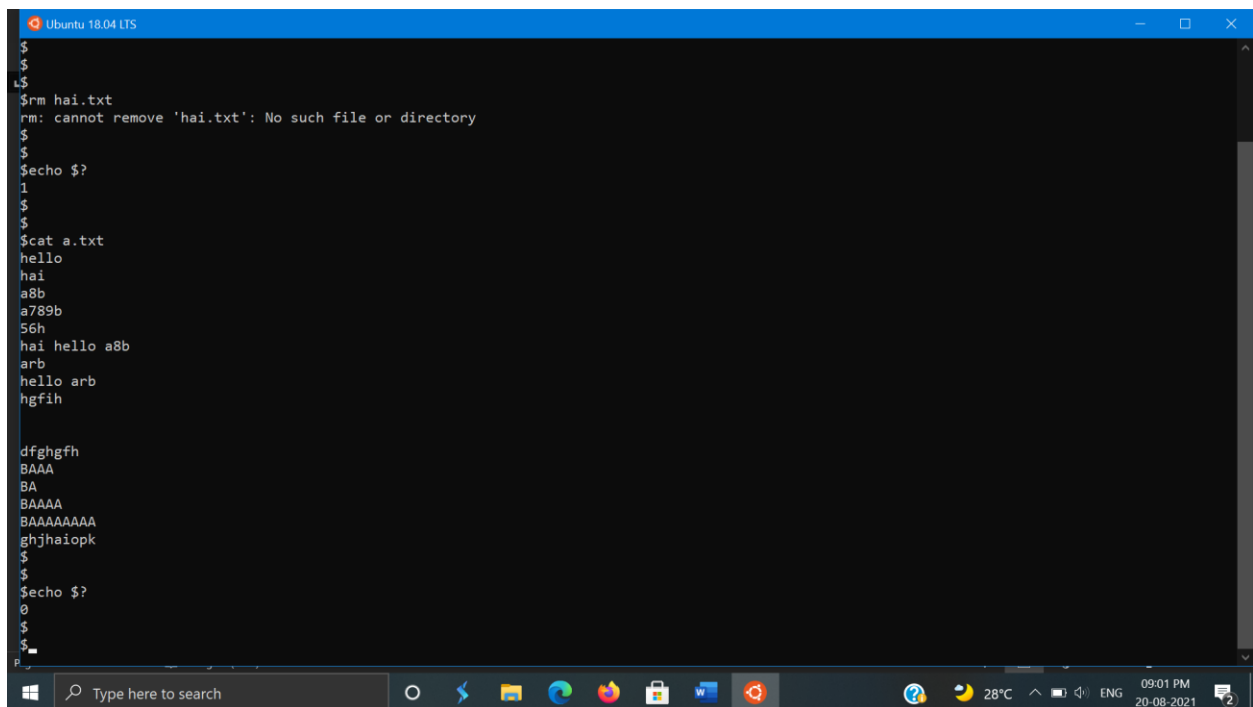
```
$ echo $?
```

```
$ somethingnonse
```

```
$ echo $?
```

```
$ date
```

```
$ echo $?
```



```
Ubuntu 18.04 LTS
$
$
$
$
$rm hai.txt
rm: cannot remove 'hai.txt': No such file or directory
$
$
$echo $?
1
$
$
$cat a.txt
hello
hai
a8b
a789b
56h
hai hello a8b
arb
hello arb
hgfih

dfghgfh
BAAA
BA
BAAAA
BAAAAAAAA
ghjhaioPk
$
$
$echo $?
0
$
$
$
```

The read Statement

Use to get input (data from user) from keyboard and store (data) to variable.

Following script first ask user, name and then waits to enter name from the user via keyboard. Then user enters name from keyboard (after giving name you have to press ENTER key) and entered name through keyboard is stored (assigned) to variable fname.

-p option with read can also be used as shown below.

#Script to read your name from key-board

echo "Your first name please:"

read fname

echo "Hello \$fname, Lets be friend!"

Example:

```
read -p "Enter Filename" file1
```

```
echo "Enter pattern"
```

```
read pattern1
```

```
grep $pattern1 $file1
```

Why Command Line arguments required?

- To tell the command/utility which option to use.
- Informing the utility/command which file or group of files to process (reading/writing of files).
- \$0, \$1, \$2 \$9 are called as **positional parameters**, used to access the command-line arguments in a shell script
- \$1, \$2 \$9 arguments can be passed to shell script which are actual arguments
- \$0 – contains the shell script name itself
- \$# - contains the number of arguments passed.
- @\$ or \$* - can be used to fetch all the arguments from command line starting from \$1
- Following script is used to print command line argument and will show you how to access them

Example:

```
Ubuntu 18.04 LTS
$cat thirscript.sh
if [ $# -eq 2 ]
then
    echo -e "All the cmd-line arguments \n"
    echo $$
    echo "Number of arguments"
    echo $#
    echo "First Argument"
    echo $1
    echo "Script File Name"
    echo $0
    echo "Searching for pattern $1 in $2"
    grep $1 $2
else
    echo "Invalid number of arguments"
fi
$
```

Output:

```
Ubuntu 18.04 LTS
$bash thirscript.sh hai a.txt
All the cmd-line arguments
hai a.txt
Number of arguments
2
First Argument
hai
Script File Name
thirscript.sh
Searching for pattern hai in a.txt
hai
hai hello a8b
ghjhaiopk
$
$
$
$
```

Redirection of Standard output/input

Mostly all commands give output on screen or take input from keyboard ,but it's also possible to send output to file or to read input from ie., There are three main redirection symbols >; >>;<

1. >Redirector Symbol.

Syntax: Linux-command >filename

To output Linux-commands result (output of command or shell lscript) to file. Note that if file already exist, it will be overwritten else new file is created.

\$ ls>myfiles

2. >>Redirector Symbol.

Syntax: Linux-command >>Filename

To output Linux-commands result (output of command or shell script) to END of file. Note that if file exist, it will be opened and new information/data will be written to END of file, without losing previous information/data, and if file is not exist, then new file is created.

\$ date>>myfiles

3. <Redirector Symbol.

Syntax: Linux-command<Filename

To take input to Linux-command from file instead of keyboard.

\$ cat<myfiles

Shells (bash) structured Language Constructs

This section introduces to the bash's structured language constructs such as:

1. Decision making
2. Loops

Decision making

i) if :If is used for decision making in shell script, If given condition is true then command1 is executed.

Syntax:**if condition****then**

...

...

fi

Condition is nothing but comparison between two values. { For comparison you can use test or [expr] statements or even exist status can be also used. An expression is nothing but combination of values, relational operator (such as > < <= >= etc) and mathematical operators (such as + * - / = etc).

#!/bin/sh**#Script to print file****if cat \$1****then****echo -e "\n\nFile \$1, found and successfully echoed"****fi**

test command or [expr]- test command or [expr] is used to see if an expression is true, and if it is true it return zero (0), otherwise returns nonzero for false.

Syntax:**test expression OR [expression]**

Following script determine whether given argument number is positive.

#!/bin/sh**# Script to see whether argument is positive****if test \$1 -gt 0****then****echo "\$1 number is positive"****fi**

Note : **test or [expr]** works with

1. Integer (Number without decimal point)
2. File types

3.Character strings

For Mathematics use following operators in Shell Script

Math- ematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [expr] statement with if command
-eq	is equal to	$5 == 6$	if test 5 -eq 6	if expr [5 -eq 6]
-ne	is not equal to	$5 != 6$	if test 5 -ne 6	if expr [5 -ne 6]
-lt	is less than	$5 < 6$	if test 5 -lt 6	if expr [5 -lt 6]
-le	is less than or equal to	$5 <= 6$	if test 5 -le 6	if expr [5 -le 6]
-gt	is greater than	$5 > 6$	if test 5 -gt 6	if expr [5 -gt 6]
-ge	is greater than or equal to	$5 >= 6$	if test 5 -ge 6	if expr [5 -ge 6]

```
Ubuntu 18.04 LTS
$cat fourthscript.sh
h=$(date +%H")
if [ $h -gt 6 -a $h -le 12 ]
then
echo good morning
elif [ $h -gt 12 -a $h -le 16 ]
then
echo good afternoon
elif [ $h -gt 16 -a $h -le 20 ]
then
echo good evening
else
echo good night
fi
$
```

NOTE: == is equal, != is not equal.
For string Comparisons use

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL or not defined
-n string1	string1 is NOT NULL and does exist
-z string1	string1 is NULL and does exist

Shell also test for file and directory types

Test	Meaning
-s file	Non empty file
-f file	Is File exist or normal file and not a directory
-d dir	Is Directory exist and not a file
-w file	Is writeable file
-r file	Is read-only file
-x file	Is file is executable

Logical Operators

Logical operators are used to combine two or more condition at a time

Operator	Meaning
! expression	Logical NOT
expression1 -a expression2	Logical AND
expression1 -o expression2	Logical OR

Example:

```
VAR1="goinuxcloud"
```

```
VAR2="goinuxcloud"
```

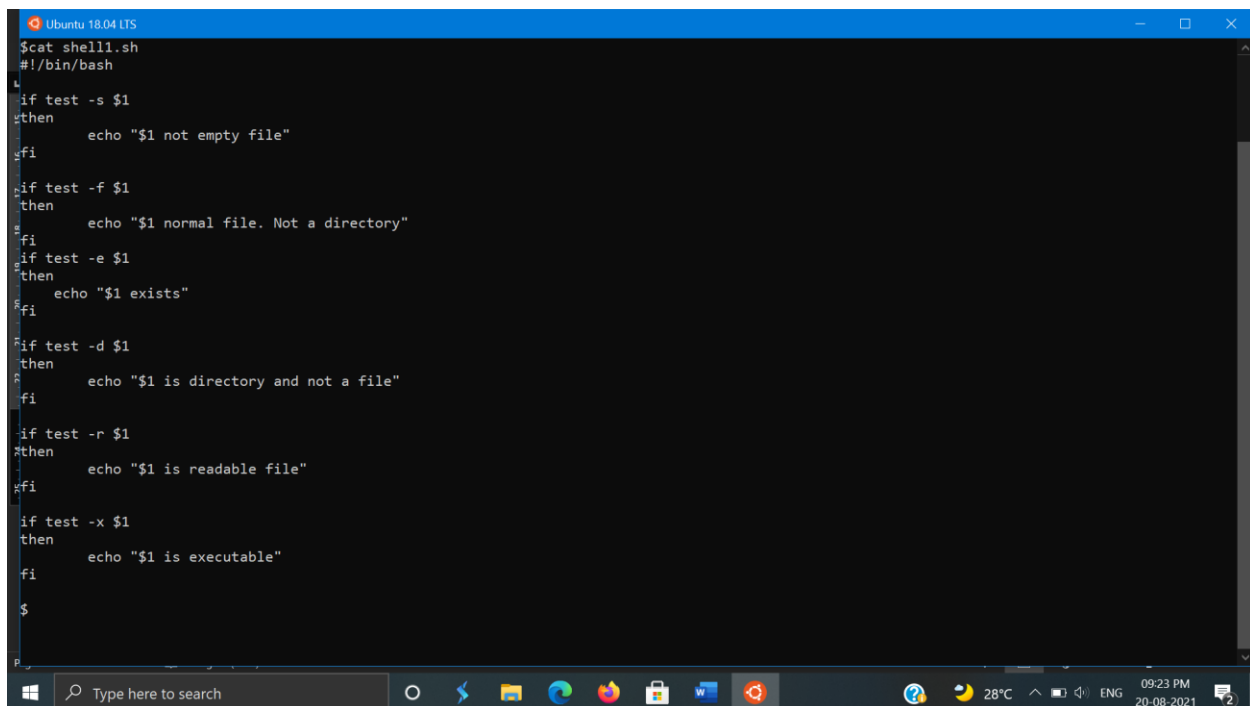
```
if [[ "$VAR1" == "$VAR2" ]];then
    echo "exit status: $?"
    echo "$VAR1 is equal to $VAR2"
else
    echo "exit status: $?"
    echo "$VAR1 is NOT equal to $VAR2"
fi
```

Example:

VAR1="A"

VAR2="B"

```
if [[ "$VAR1" > "$VAR2" ]];then
    echo "exit status: $?"
    echo "$VAR1 is greater than $VAR2"
elif [[ "$VAR1" == "$VAR2" ]];then
    echo "exit status: $?"
    echo "$VAR1 is equal to $VAR2"
else
    echo "exit status: $?"
    echo "$VAR1 is lesser than $VAR2"
fi
```

Example:A screenshot of a terminal window titled "Ubuntu 18.04 LTS". The terminal shows a shell script named "shell1.sh" being executed. The script uses various test operators to check file properties and prints messages accordingly. The terminal output shows the script's execution flow. The window has a blue title bar and standard Ubuntu window controls. The system tray at the bottom shows the date and time as 09:23 PM on 20-08-2021.

```
Ubuntu 18.04 LTS
$cat shell1.sh
#!/bin/bash

if test -s $1
then
    echo "$1 not empty file"
fi

if test -f $1
then
    echo "$1 normal file. Not a directory"
fi

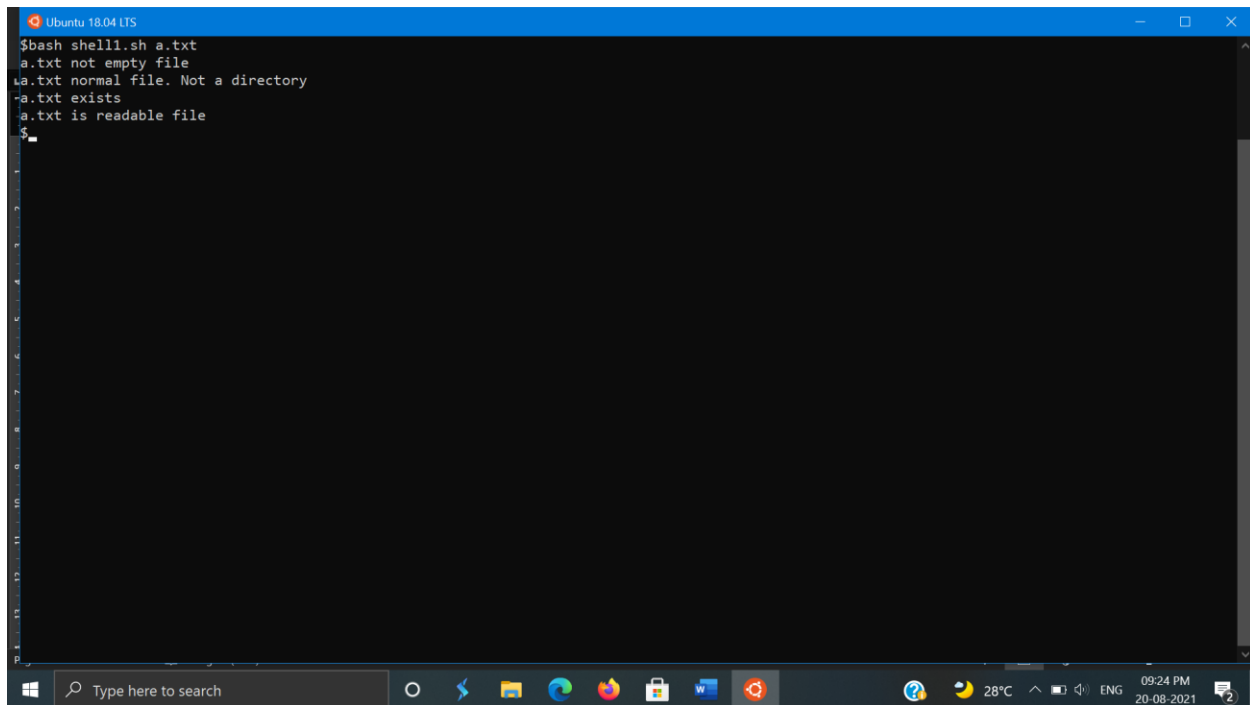
if test -e $1
then
    echo "$1 exists"
fi

if test -d $1
then
    echo "$1 is directory and not a file"
fi

if test -r $1
then
    echo "$1 is readable file"
fi

if test -x $1
then
    echo "$1 is executable"
fi

$
```



```
Ubuntu 18.04 LTS
$bash shell1.sh a.txt
a.txt not empty file
a.txt normal file. Not a directory
a.txt exists
a.txt is readable file
$
```

ii) if...else...

If given condition is true then command1 is executed otherwise command2 is executed. Syntax:

if condition

then

.... # condition is zero (true - 0) execute all commands up to else statement

else

..... # if condition is not true then execute all commands up to fi

fi

Example: Script to see whether argument is positive or negative;

```
#!/bin/sh
```

```
# Script to see whether argument is positive or negative
```

```
if [ $# -eq 0 ]
```

```
then
```

```
    echo "$0 : You must give/supply one integer"
```

```
    exit 1
```

```
fi
```

```

if test $1 -gt 0
then
    echo"$1 number is positive"
else
    echo "$1 number is negative"
fi

```

Nested if-else-fi

- You can write the entire if-else construct within either the body of the ifstatement or the body of an else statement. This is called the nesting of if.

Syntax:

```

if condition
then
    if condition
    then
        do this
    else
        do this
    fi
else
    do this
fi

```

Example:

```

#!/bin/sh
# Script to see nesting of ifs
osch=0
echo "1. Unix (Sun Os)"
echo "2. Linux (Red Hat)"
echo -n "Select your os choice [1 or 2]? "
read osch
if [ $osch -eq 1 ]
then
    echo "You Pick up Unix (Sun Os)"

```

```

else # nested if i.e. if within if #
    if [ $osch -eq 2 ]
    then
        echo "You Pick up Linux (Red Hat)"
    else
        echo "What you don't like Unix/Linux OS."
    fi
fi

```

Multilevel if-then-else Syntax:

```

if condition
then
    ..... #condition is zero (true - 0)execute all commands up to elif statement
elif condition1
then
    ..... #condition1 is zero (true - 0)execute all commands up to elif statement
elif condition2
then
    ..... #condition2 is zero (true - 0)execute all commands up to elif statement
else
    ..... #None of the above condition,condition1,condition2 are true (i.e.all of the
    above nonzero or false)execute all commands up to fi
fi

```

Example: For multilevel if-then-else statement try the following script

```

#!/bin/sh
# Script to test if..elif...else

if [ $1 -gt 0 ]
then
    echo "$1 is positive"
elif [ $1 -lt 0 ]
then
    echo "$1 is negative"

```

```

elif [ $1 -eq 0 ]
then
    echo "$1 is zero"
else
    echo "Oops! $1 is not number, give number"
fi

```

Note: Integer comparison is expected.

Loops : Loops in Shell Scripts

Loop defined as: Computer can repeat particular instruction again and again, until particular condition satisfies. A group of instruction that is executed repeatedly is called a loop.

Bash supports:

1. for loop

2. while loop

Note that in each and every loop,

- First, the variable used in loop condition must be initialized, then execution of the loop begins.
- A test (condition) is made at the beginning of each iteration.
- The body of loop ends with a statement that modifies the value of the test (condition) variable.

for Loop

Syntax:

```
for { variable name } in { list }
```

```
do
```

```

    .... # execute one for each item in the list until the list is not finished (And
    repeat all statement between do and done)

```

```
done
```

Example 1:

```

for i in 1 3 5
do
    echo "Welcome $i times"
done

```

Example 2:

```
#!/bin/sh
#Script to test for loop
if [ $# -eq 0 ]
then
    echo "Error - Number missing form command line argument"
    echo "Syntax : $0 number"
    echo "Use to print multiplication table for given number"
    exit 1
fi
n=$1
for i in 1 2 3 4 5 6 7 8 9 10
do
    echo "$n * $i = `expr $i \* $n`"
done
```

Syntax:

```
for (( expr1; expr2; expr3 ))
do
    .... #repeat all statements between do anddone until expr2 is TRUE
done
```

Example :

```
#!/bin/sh
#Script to test for loop 2
for (( i = 0 ; i <= 5; i++ ))
do
    echo "Welcome $i times"
done
```


Nesting of for Loop

As you see the ifstatement can nested, similarly loop statement can be nested. To understand the nesting of for loop see the following shell script.

Example:

```
#!/bin/sh
# Nesting of for loop
for (( i = 1; i <= 5; i++ ))          ### Outer for loop ###
do
    for (( j = 1 ; j <= 5; j++ )) ### Inner for loop ###
    do
        echo -n "$i "
    done
echo "" #### print the new line ###
done
```

Here, for each value of i the inner loop is cycled through 5 times, with the variable j taking values from 1 to 5. The inner for loop terminates when the value of j exceeds 5, and the outer loop terminates when the value of i exceeds 5.

while loop

Syntax:

```
while [ condition ]
do
    command1
    command2
    command3
    ..
done
```

Note: Loop is executed as long as given condition is true. Above for loop program (shown in last section of for loop) can be written using while loop as

Example:

```
#!/bin/sh
#Script to test while statement
```

```

if [ $# -eq 0 ]
then
    echo "Error - Number missing form command line argument"
    echo "Syntax : $0 number"
    echo " Use to print multiplication table for given number"
    exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
do
    echo "$n * $i = `expr $i \* $n`"
    i=`expr $i + 1`
done

```

The case Statement

The case statement is good alternative to multilevel if-then-else statement. It enables you to match several values against one variable. It's easier to read and write.

Syntax:

```

case $variable-name in
    pattern1) command
    ..
    command;;
    pattern 2) command
    ..
    command;;
    patternN) command
    ..
    command;;
    *) command
    ..
    command;;
esac

```

- The \$variable-name is compared against the patterns until a match is found.
- The shell then executes all the statements up to the two semicolons that are next to each other. The default is *) and its executed if no match is found.

Example;

```
#!/bin/sh
# if no vehicle name is given
# i.e. -z $1 is defined and it is NULL
# if no command line arg
if [ -z $1 ]
then
    rental="*** Unknown vehicle ***"
elif [ -n $1 ]
then
    # otherwise make first arg as rental
    rental=$1
fi

case $rental in
    "car") echo "For $rental Rs.20 per k/m";;
    "van") echo "For $rental Rs.10 per k/m";;
    "jeep") echo "For $rental Rs.5 per k/m";;
    "bicycle") echo "For $rental 20 paisa per k/m";;
    *) echo "Sorry, I cannot get a $rental for you";;
esac
```

Example:

```
Ubuntu 18.04 LTS
$
$cat shellex1.sh
while true
do
    echo "1.Number of directories"
    echo "2.Number of process"
    echo "3.EXIT"
    read -p "Enter Choice" choice
    case $choice in
        1) ls -l|grep ^d|wc -l ;;
        2) ps|wc -l;;
        3) exit ;;
        *) echo "Invalid Choice"
    esac
done
$
$
$
$
```

How to de-bug the shell script?

While programming shell sometimes you need to find the errors (bugs) in shell script and correct the errors (remove errors - debug). For this purpose you can use -v and -x option with sh or bash command to debug the shell script.

Syntax:

sh option { shell-script-name }

OR

bash option { shell-script-name }

Option can be

- v Print shell input lines as they are read.

- x After expanding each simple-command, bash displays the expanded value of system variable, followed by the command and its expanded arguments.

Example;

```
#!/bin/sh
```

```
# Script to show debug of shell
```

```
tot=`expr $1 + $2`
```

```
echo $tot
```

execute as

```
$ ./dsh1 4 5
```

9

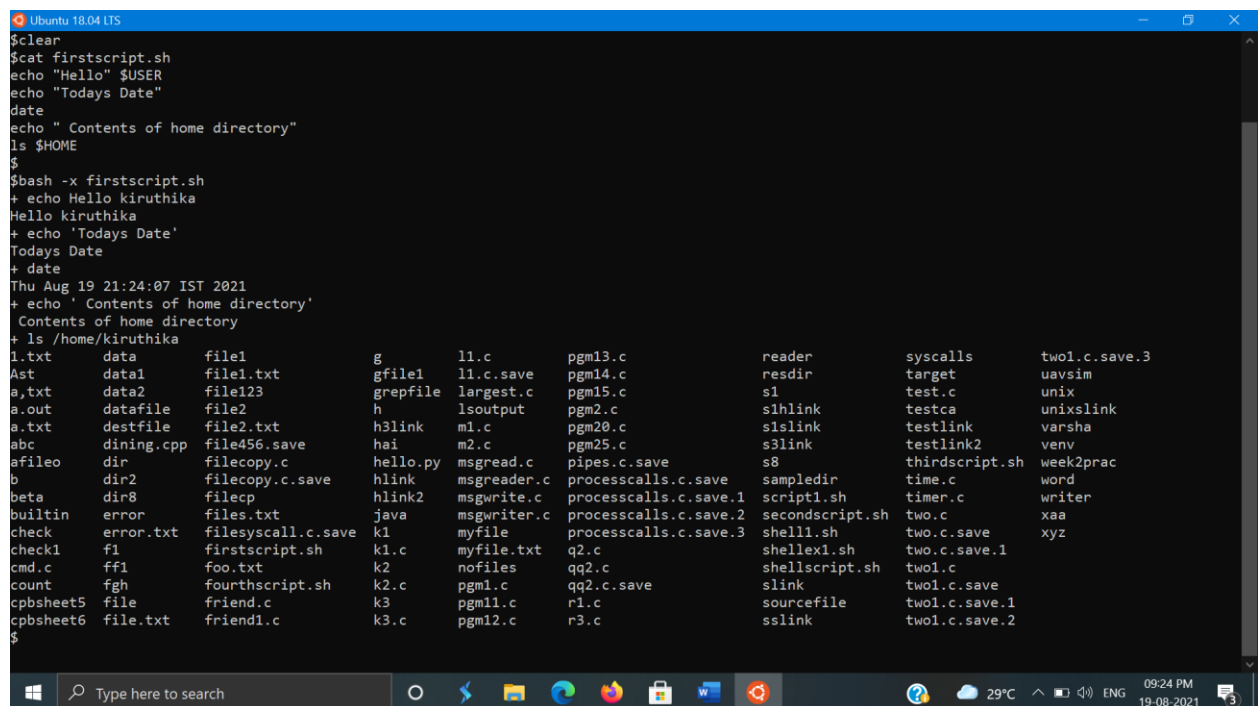
```
$ sh -x dsh1 4 5
```

```
++ expr 4 + 5
```

```
+ tot=9
```

```
+ echo 9
```

9



```
Ubuntu 18.04 LTS
$clear
$cat firstscript.sh
echo "Hello" $USER
echo "Today's Date"
date
echo "Contents of home directory"
ls $HOME
$
$bash -x firstscript.sh
+ echo Hello kiruthika
Hello kiruthika
+ echo 'Today's Date'
Today's Date
+ date
Thu Aug 19 21:24:07 IST 2021
+ echo 'Contents of home directory'
Contents of home directory
+ ls /home/kiruthika
1.txt      data      file1      g          l1.c      pgm13.c    reader     syscalls   two1.c.save.3
Ast        data1     file1.txt  gfile1    l1.c.save pgm14.c    resdir     target     uavsim
a.txt      data2     file123    grepfile  largest.c  pgm15.c    s1         test.c     unix
a.out      datafile  file2      h          lsoutput  pgm2.c     s1hlink    testca     unixslink
a.txt      destfile  file2.txt  h3link    m1.c      pgm20.c    s1slink    testlink   varsha
abc        dining.cpp file456.save hai        m2.c      pgm25.c    s3link     testlink2  venv
afileo     dir       filecopy.c hlink      hello.py   msgread.c  pipes.c.save s8         thirdsript.sh week2prac
b          dir2      filecopy.c.hlink  msgreader.c processcalls.c.save.1 sampledirtimer.cword
beta       dir8      filecp     hlink2     msgwrite.c processcalls.c.save.2 script1.shtimer.cwriter
builtin    error     files.txt  java       msgwriter.c processcalls.c.save.3 secondscript.shtwo.cxaa
check      error.txt filesyscall.c.save k1         myfile     q2.c       shell11.shtwo.c.save.1
check1     f1        firstscript.sh k2         nofiles    qq2.c      shellescript.shtwo1.c
cmd.c      ff1       foo.txt    k2.c       pgm1.c     qq2.c.save slink       two1.c.save
count      fgh       fourthscript.sh k3         pgm11.c    r1.c       sourcefile  two1.c.save.1
cpbsheet5  file      friend.c   k3.c       pgm12.c    r3.c       sslink      two1.c.save.2
cpbsheet6  file.txt  friend1.c
$
```

```
$ sh -v dsh1 4 5
```

```
#!/bin/sh
```

```
# Script to show debug of shell
```

```
tot=`expr $1 + $2`
```

```
echo $tot
```

9

```
Ubuntu 18.04 LTS
$cat firstscript.sh
echo "Hello" $USER
echo "Todays Date"
date
echo " Contents of home directory"
ls $HOME
$
$
$
$
$
$bash -v firstscript.sh
echo "Hello" $USER
Hello kiruthika
echo "Todays Date"
Todays Date
date
Thu Aug 19 21:25:13 IST 2021
echo " Contents of home directory"
Contents of home directory
ls $HOME
l.txt      data      file1      g          ll.c       pgm13.c    reader     syscalls   two1.c.save.3
Ast        data1     file1.txt  gfile1    ll.c.save  pgm14.c    resdir     target     uavsim
a.txt      data2     file123   grepfile  largest.c  pgm15.c    s1         test.c     unix
a.out      datafile  file2     h          lsoutput   pgm2.c     s1hlink    testca     unixslink
a.txt      destfile  file2.txt h3link    m1.c       pgm20.c    s1slink    testlink   varsha
abc        dining.cpp file456.save hai        m2.c       pgm25.c    s3link     testlink2  venv
afileo    dir       filecopy.c hello.py   msgread.c  pipes.c.save s8         thirdsript.sh week2prac
b          dir2     filecopy.c.save hlink      msgreader.c processcalls.c.save sampledir   time.c     word
beta       dir8     filecp     hlink2     msgwrite.c processcalls.c.save.1 script1.sh timer.c    writer
builtin    error    files.txt  java       msgwriter.c processcalls.c.save.2 secondsript.sh two.c      xaa
check      error.txt filesyscall.c.save k1         myfile     processcalls.c.save.3 shell11.sh two.c.save xyz
check1     f1       firstscript.sh k1.c       myfile.txt q2.c       shlex1.sh two.c.save.1
cmd.c      ff1      foo.txt    k2         nofiles    qq2.c      shellsript.sh two1.c
count      fgh      fourthscript.sh k2.c       pgm1.c     qq2.c.save slink       two1.c.save
cpsheet5   file     friend.c   k3         pgm11.c    r1.c       sourcefile two1.c.save.1
```

Local and Global Shell variable (export command)

- Normally all our variables are local.
- Local variable can be used in same shell, if you load another copy of shell (by typing the `/bin/bash` at the `$` prompt) then new shell ignored all old shell's variable.

```
Ubuntu 18.04 LTS
$y=10
$echo $y
10
$/bin/bash
$echo $y
$
```

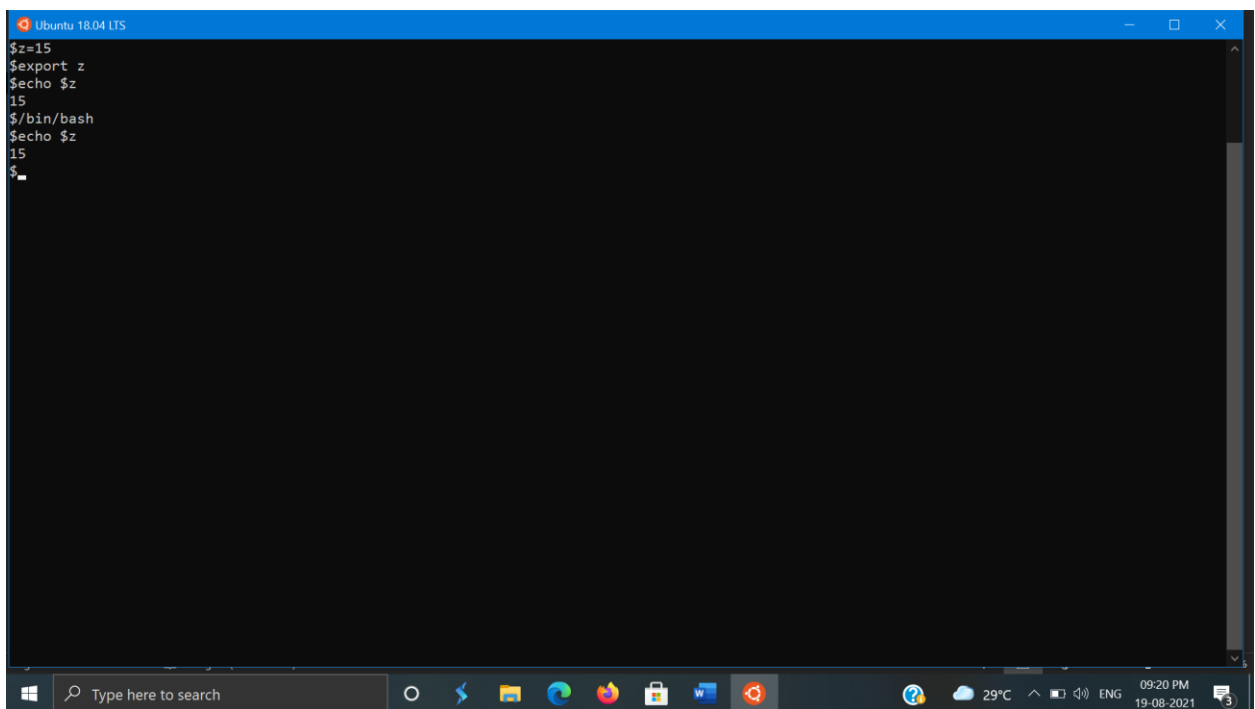
- Global shell defined as: "You can copy old shell's variable to new shell (i.e. first shell's variable to second shell), such variable is known as Global Shell variable."
- To set global variable you have to use export command.

Syntax:

export variable1, variable2,.....variableN

Examples:

```
$ vech=Bus
$ echo $vech
Bus
$ export vech
$ /bin/bash
$ echo $vech
Bus
$ exit
$ echo $vech
Bus
```



The screenshot shows a Windows terminal window titled "Ubuntu 18.04 LTS". The terminal displays the following commands and their outputs:

```
$z=15
$export z
$echo $z
15
$/bin/bash
$echo $z
15
$
```

The Windows taskbar is visible at the bottom, showing the search bar, task view, and various application icons. The system tray on the right indicates the temperature is 29°C, the time is 09:20 PM, and the date is 19-08-2021.

Conditional execution i.e. && and ||

The control operators are && (read as AND) and || (read as OR).

- The syntax for AND list is as follows;

command1 && command2

command2 is executed if, and only if, command1 returns an exit status of zero.

- The syntax for OR list as follows

command1 || command2

command2 is executed if and only if command1 returns a non-zero exit status.

- You can use both as follows

command1 && command2 (if exist status is zero) || command3

(if exit status is non-zero) if command1 is executed successfully then shell will run command2 and if command1 is not successful then command3 is executed.

Example:

```
$ rm myfile && echo "File is removed successfully" || echo "File is not removed"
```

Example

```
#!/bin/bash
# or example
if [ $USER == 'bob' ] || [ $USER == 'andy' ]
then
    ls -alh
else
    ls
fi
```

Write Shell Scripts for the following:

1. Write shell script to perform integer arithmetic operations.
2. Write an interactive file handling shell program. Let it offer the user the choice of copying removing, renaming, or linking files. Once the user has made a choice, have the same program ask the user for the necessary information, such as the file name, new name and so on.
3. Write a shell script that computes the gross salary of an employee according to the following rules:

- i. If basic salary is <1500 then HRA=10% of the basic and DA=90% of the basic
 - ii. If the basic salary is >=1500 then HRA=500/- and DA=98% of the basic
- The basic salary is entered interactively through the key board.
4. Write a shell script that accepts two integers as its arguments and computes the value of first number raised to the power of the second number.
5. Write shell script that takes a login name as command-line argument and reports when that person logs in.
6. Write a shell script which receives two file names as arguments. It should check whether the two file contents are same or not. If they are same then second file should be deleted.
7. Write a shell script that displays a list of all the files in the current directory to which the user has read, write and execute permissions.
8. Develop an interactive script that asks for a word and a file name and then tells how many times that word occurred in the file.
9. Write a shell script to perform the following string operations:
 - i. To extract a sub-string from a given string.
 - ii. To find the length of a given string.
10. Write a shell program to find out factorial of the given number.
11. Write a shell script to find out whether the given number is prime number or not.
12. Write a shell script to accept two file names and check if both exist. If the second filename exists, then the contents of the first filename should be appended to it. If the second file name does not exist then create a new file with the contents of the first file.
13. Write a shell script that accepts a file name, starting and ending line numbers as arguments and displays all the lines between the given line numbers.
14. Write a shell script that deletes all lines containing a specified word in one or more files supplied as arguments to it.
15. Write a shell script that displays a list of all the files in the current directory to which the user has read, write and execute permissions.
16. Write a shell script that receives any number of file names as arguments checks if every argument supplied is a file or a directory and reports accordingly. Whenever the argument is a file, the number of lines on it is also reported.
17. Write a shell script to perform floating point arithmetic operations using command line arguments.