CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test 2 – Dec. 2019

Sub:	Unix and Shell Programming							Sub Code:	18MCA12
Date:	16/12//2019	Duration:	90 min's	Max Marks:	50	Sem:	I	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

		MARKS	OBE	
			CO	RBT
	PART I			
1	Explain the below mentioned commands with its usage and examples. i) Read ii) pr iii) cut iv)tr v)sort	[10]	CO5	L2
	OR			
2	Explain the grep family commands with its options.	[10]	CO1	L2
	PART II			
3	Explain line addressing using "sed" command with examples.	[10]	CO1	L2
	OR			
4	Explain context addressing using "sed" command with examples.	[10]	CO1	L2
	PART III			
5	What are regular expressions? Explain the structure of regular expression.	[10]	CO1	L2
	OR			
6	Write a short note on appending, inserting and changing text with respect to text.	[10]	CO1	L2
	PART IV			
7	Explain awk built-in variables with suitable examples.	[10]	CO1	L2
	OR			
8	Create a script file called file properties that reads a filename entered and outputs it properties.	[10]	CO5	L3
	PART V			
9	Write a shell script that accepts file name as argument and display its creation time if file exists and if it does not send output error message.	[10]	CO5	L3
	OR			
10	How do you achieve substitution using sed command? Give examples.	[10]	CO1	L2

CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test 2– Nov. 2019

Sub:	Unix and Shell Programming					Sub Code:	17MCA12	Branch:	MCA
Date:	16/12/2019	Duration:	90 min's	Max Marks:	50	Sem	V		OBE

1. Explain the below mentioned commands with its usage and examples.
i) Read ii) pr iii) cut iv)tr v)sort

i)Read:

The ***read*** statement is the internal tool of the shell for taking input from the user. This will help the scripts to become interactive.

```
#!/bin/sh
#Illustration of read statement
echo "Enter your name:"
read fname
echo "Hello $fname"
```

ii)pr

The pr command prepares a file for printing by adding suitable headers, footers, and formatted text. When used with a filename as argument, pr doesn't behave like a filter:

$ pr group1
May 06 10:38 1999 group1 Page 1
root:x:0:root These seven lines are the original
bin:x:1:root,bin,daemon contents of group1
users:x:200:henry,image,enquiry
adm:x:25:adm,daemon,listen
dialout:x:18:root,henry
lp:x:19:lp
ftp:x:50:
... blank lines ...

pr adds five lines of margin at the top (simplified here) and five at the bottom. The header
shows the date and time of last modification of the file, along with the filename and page number. We generally don't use pr like this. Rather, we use it as a "preprocessor"
to impart cosmetic touches to text files before they are sent to the printer:

$ pr group1 | lp
Request id is 334

Since pr output often lands up in the hard copy, pr and lp form a common pipeline sequence.

Sometimes, lp itself uses pr to format the output, in which case this piping is not required.

iii)Cut

Cutting Columns (-c) To extract specific columns, you need to follow the -c option with a list of column numbers, delimited by a comma. Ranges can also be specified using
the hyphen. Here's how we extract the first four columns of the group file:

```
$ cut -c1-4 group1 -c or -f option always required
root
bin:
user
adm:
dial
lp:x
ftp:
```

Note that there should be no whitespace in the column list. Moreover, cut uses a special
form for selecting a column from the beginning and up to the end of a line:

```
cut -c -3,6-22,28-34,55- foo Must be an ascending list
```

The expression 55- indicates column number 55 to the end of the line. Similarly, -3 is
the same as 1-3.

Cutting Fields (-f) The -c option is useful for fixed-length lines. Most UNIX files (like /etc/passwd and /etc/group) don't contain fixed-length lines. To extract useful data from these files you'll need to cut fields rather than columns.

cut uses the tab as the default field delimiter, but it can also work with a different delimiter. Two options need to be used here, -d for the field delimiter and -f for the field list. This is how you cut the first and third fields:

```
$ cut -d: -f1,3 group1
root:0
bin:1
users:200
adm:25
dialout:18
lp:19
ftp:50
```

iv) tr

The tr (translate) filter manipulates individual characters in a line. More specifically, it translates characters using one or two compact expressions:

```
tr options expression1 expression2 standard input
```

Note that tr takes input only from standard input; it doesn't take a filename as argument.

By default, it translates each character in expression1 to its mapped counterpart in expression2. The first character in the first expression is replaced with the first character

in the second expression, and similarly for the other characters.
Let's use tr to replace the : with a ~ (tilde) and the / with a -. Simply specify two expressions containing these characters in the proper sequence:

```
$ tr ':/' '~-' < shortlist | head -n 3
2233~charles harris ~g.m. ~sales ~12-12-52~ 90000
9876~bill johnson ~director ~production ~03-12-50~ 130000
5678~robert dylan ~d.g.m. ~marketing ~04-19-43~ 85000
```

v) sort:

Sorting is arranging data in ascending or descending order. By default, the ***sort*** command reorders the lines in ASCII collating sequence (white space first, then numerals, uppercase, lowercase). For example,

```
$sort filename
$ sort emp.lst
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000
1006|chanchal sanghvi|director|sales|09/03/38|6700
1265|s.n. dasgupta|manager|sales|09/12/63|5600
```

Table 3.2 Options of ***sort*** command

Option	Description
-t*char*	Uses delimiter *char* to identify fields
-k *n*	Sorts on n^{th} field
-k *m,n*	Starts sort on m^{th} field and ends sort on n^{th} field
-k *m.n*	Starts sort on n^{th} column of m^{th} field
-u	Removes repeated lines
-n	Sorts numerically
-r	Reverses sort order
-f	Folds lowercase to equivalent uppercase (case-insensitive sort)
-m *list*	Merges sorted files in *list*
-c	Checks if file is sorted
-o *fname*	Places output in file *fname*

2. Explain the grep family commands with its options.

The ***grep*** command scans its input for a pattern and displays lines contain the pattern, the line numbers or filenames where the pattern occurs. The syntax is –

```
grep options pattern filename(s)
```

Various options for ***grep*** command are given in Table 3.3. They are discussed with suitable examples below.

Table 3.3 Options for *grep* command

Table 3.3 Options for *grep* command

Option	Description
-i	Ignores case for matching
-v	Doesn't display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Displays list of filenames only
-e *exp*	Specifies expression with this option. Can use multiple times. Also used for matching expression beginning with a hyphen
-x	Matches pattern with entire line (doesn't match embedded patterns)
-f *file*	Takes patterns from *file*, one per line
-E	Treats pattern as an extended regular expression (ERE)
-F	Matches multiple fixed strings (in **fgrep** – style)

· **Ignoring Case (-i):** When we are searching for a pattern, but not sure about the case, -i option is used. It ignores the case of the text and displays the result. For example,

```
$ grep -i 'agarwal' emp.lst
```

```
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
```

· **Deleting Lines (-v):** The –v (inverse) option selects all lines except those containing the pattern. The following example selects all lines in the file *emp.lst* except for those containing the term *director*.

```
$ grep -v 'director' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000
5423|n.k. gupta|chairman|admin|08/30/56|5400
6213|karuna ganguly|g.m.|accounts|06/05/62|6300
```

Displaying Line Numbers (-n): This option displays the line numbers containing the pattern along with the actual lines. For example,

```
$ grep -n 'marketing' emp.lst
3:5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000
11:6521|lalit chowdury|director|marketing|09/26/45|8200
14:2345|j.b. saxena|g.m.|marketing|03/12/45|8000
15:0110|v.k. agrawal |g.m.|marketing|12/31/40|9000
```

· **Counting Lines Containing Pattern (-c):** A pattern may be present in a file multiple times. If we would like to know how many times it has appeared, -c option can be used. The following example shows how many times the pattern *director* has appeared in the file *emp.lst*.

```
$ grep -c 'director' emp.lst
4
```

· **Displaying Filenames (-l):** The –l (el) option is used to display the names of files containing the pattern. Assume there are there are two more files *test.lst* and *testfile.lst* along with *emp.lst*. Now, let us check in which file(s) the pattern *manager* is present.

```
$grep –l 'manager' *.lst
emp.lst
test.lst
```

· **Matching Multiple Patterns (-e):** When we would like to search for multiple patterns in a file, we can use –e option. For example,

```
$ grep -e "Agarwal" -e "aggarwal" -e "agrawal" emp.lst
2476|anil aggarwal|manager|sales|05/01/59|5000
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000
```

· **Taking Patterns from a File (-f):** If various patterns are stored in a file each in different line, then –f option can be used by giving that filename as one of the arguments. For example, assume there is a file *pattern.lst* as –

```
$cat >pattern.lst
manager
executive
```

Then, give the command as –

```
$grep –f pattern.lst emp.lst
1265|s.n. dasgupta|manager|sales|09/12/63|5600
4290|jayant Chodhury|executive|production|09/07/50|6000
2476|anil aggarwal|manager|sales|05/01/59|5000
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
```

3. Explain line addressing using "sed" command with examples.

The ***sed*** command is a multipurpose tool which combines the work of several filters. It performs non-interactive operations on a data stream. It allows selecting lines and running instructions on them.

An instruction combines an ***address*** for selecting lines, with an ***action*** to be taken on them. The ***sed*** command uses such instructions. The syntax is –

sed *options 'address action' file(s)*

Line Addressing: Here, *address* specifies either one line number to select a single line or a set of two numbers to select a group of contiguous lines.

Option	Description

d	Deletes line(s)
10q	Quits after reading the first 10 lines
P	Prints line(s) on standard output
3,$p	Prints lines 3 to end (-n option required)
$!p	Prints all lines except last line (-n option required)
/begin/, /end/p	Prints lines enclosed between *begin* and *end* (-n option required)
q	Quits after reading up to addressed line
r *flname*	Places contents of file *flname* after line
w *flname*	Writes addressed lines to file *flname*
=	Prints line number addressed.

In line addressing, the instruction *3q* can be broken into the address 3 and the action q (quit). So, to display only first 3 lines, (similar to *head –n 3*) use the following statement –

```
$ sed '3q' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000
```

In the above example, 3 lines will be displayed and then quits.

Generally, the ***p*** (print) command is used to display lines. But, this command behaves strange – it prints selected lines as well as *all* lines. Hence, the selected lines will appear twice. To suppress this feature of *p*, the –n option has to be used. The following example selects the lines 5 through 7.

```
$ sed -n '5,7p' emp.lst
5423|n.k. gupta|chairman|admin|08/30/56|5400
1006|chanchal sanghvi|director|sales|09/03/38|6700
6213|karuna ganguly|g.m.|accounts|06/05/62|6300
```

The $ symbol can be used to print only the last line as below –

```
$ sed -n '$p' emp.lst
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000
```

The ***sed*** command can be used to select multiple groups of lines. In that case, each address has to be given in a different line, but enclosed within a single pair of quotes as shown below –

```
$ sed -n '1,2p
> 7,9p
> $p' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
6213|karuna ganguly|g.m.|accounts|06/05/62|6300
```

The ***sed*** command uses ! (exclamatory mark) as a negation operator. Assume, we would like to select first 2 lines of the file. Note that, selecting first two lines means – not selecting 3rd line to end. So, the command can be used as below –

```
$ sed -n '3,$!p' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
```

Here, ! is for ***p*** indicating not to print lines from 3 to end.

Using Multiple Instructions (-e and –f) : In the previous section, we have seen that when multiple groups of lines have to be selected, the pattern should be given in different lines with a line-break in-between. To avoid that, ***sed*** uses –e option. This option allows to enter as many instructions as you wish, in a single line, where each instruction is preceded by the option –e. For example, the following command selects multiple lines (1 to 2, 7 to 9 and last line) –

```
$ sed -n -e '1,2p' -e '7,9p' -e '$p' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
```

When we have too many instructions to use or when we have a set of a common instructions that are executed often, better to store them in a file. And, then use –f option with ***sed*** command to read from that file and to apply the instructions on input file. Consider

the example given below. Here, we have created a file *instr.dat* containing required instructions. Then use the ***sed*** command.

```
$ cat >instr.dat
1,2p
7,9p
$p
$ sed –n –f instr.dat emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
```

4. Explain context addressing using "sed" command with examples.

The ***sed*** command is a multipurpose tool which combines the work of several filters. It performs non-interactive operations on a data stream. It allows selecting lines and running instructions on them.

An instruction combines an ***address*** for selecting lines, with an ***action*** to be taken on them. The ***sed*** command uses such instructions. The syntax is –

sed *options 'address action' file(s)*

Context Addressing:

Context addressing allows to specify one or two patterns to locate the lines. The patterns must be bounded by a / on both the sides. When a single pattern is specified, all lines containing the pattern are selected. The following example is for selecting all the lines containing the pattern *director*.

```
$ sed -n '/director/p' emp.lst
9876|jai sharma|director|production|03/12/50|7000
2365|barun sengupta|director|personnel|05/11/47|7800
1006|chanchal sanghvi|director|sales|09/03/38|6700
6521|lalit chowdury|director|marketing|09/26/45|8200
```

One can give a comma-separated list of context addresses to select a group of lines. For example, to select all the lines between *dasgupta* and *saxena* use the following statement –

```
$ sed -n '/dasgupta/,/saksena/p' emp.lst
1265|s.n. dasgupta|manager|sales|09/12/63|5600
4290|jayant Chodhury|executive|production|09/07/50|6000
2476|anil aggarwal|manager|sales|05/01/59|5000
```

One can mix line addressing and context addressing. If we want to select all lines from 1st line till *dasgupta*, use the command as below –

```
$ sed -n '1,/dasgupta/p' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000
```

Regular expressions can be used as a part of context address. For example, the following command selects different spellings of *agarwal.*

```
$ sed -n '/[aA]gg*[ar][ar]wal/p' emp.lst
2476|anil aggarwal|manager|sales|05/01/59|5000
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000
```

One more example of ***sed*** command including regular expression is given below. It selects all lines containing *saksena, saxena,* and *gupta*. Note that, here also two different patterns should be given on different lines.

```
$ sed -n '/sa[kx]s*ena/p
> /gupta/p' emp.lst
2365|barun sengupta|director|personnel|05/11/47|7800
5423|n.k. gupta|chairman|admin|08/30/56|5400
1265|s.n. dasgupta|manager|sales|09/12/63|5600
```

The characters ^ and $ also can be used as a part of regular expression with ***sed*** command. Following example shows the people born in 1950. Note that, the five dots after 50 in the expressions indicate 5 characters (a delimiter | and 4 characters indicating salary) present before the end of line ($).

```
$ sed -n '/50.....$/p' emp.lst
9876|jai sharma|director|production|03/12/50|7000
```

```
4290|jayant Chodhury|executive|production|09/07/50|6000
```

5. What are regular expressions? Explain the structure of regular expression.

A regular expression (regex) is defined as a pattern that defines a class of strings. Given a string, we can then test if the string belongs to this class of patterns. Regular expressions are used by many of the UNIX utilities like *grep*, *sed*, *awk*, *vi* etc. A regular expression is a set of characters that specify a pattern. Regular expressions are used when we want to search for specify lines of text containing a particular pattern. Regular expressions search for patterns on a single line, and not for patterns that start on one line and end on another.

The regular expression uses meta-character set as given in Table 3.4.

Table 3.4 Character subset for BRE

Symbol/ Expression	Matches
*	Zero or more occurrences of the previous character
g*	Nothing or g, gg, ggg etc
.	A single character
.*	Nothing or any number of characters
[pqr]	A single character *p, q* or r
[c1-c2]	A single character within the ASCII range represented by c1 and c2
[1-3]	A digit between 1 and 3
[^pqr]	A single character which is not *p, q* or *r*
[^a-zA-Z]	A non-alphabetic character
^pat	Pattern *pat* at the beginning of line
pat$	Pattern *pat* at end of line
^pat$	*pat* as only word in line
^$	Lines containing nothing

The Character Class

A regular expression lets to specify a group of characters enclosed within a pair of rectangular brackets [ ]. The match is performed for a single character in the group. For example, the expression `[ra]` matches either `r` or `a`.

In the previous section, we have seen that ***grep*** with –e option is used to compare multiple patterns. Now, let us write the regular expression for searching different spellings of *agarwal* in *emp.lst*.

```
$ grep "[aA]g[ar][ar]wal" emp.lst
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000
```

The *

The * refers to the *immediately preceding* character. It matches *zero or more occurrences of the previous character*. Hence, the pattern `g*` matches *null string* or following strings – `g, gg, ggg, gggg` .....

As the * can match even a null string, if you want to search a string beginning with `g`, do not give pattern as `g*` , instead give as `gg*`.

Now check the following example, where all three types of spellings of *agarwal* can be searched.

```
$ grep "[aA]gg*[ar][ar]wal" emp.lst
2476|anil aggarwal|manager|sales|05/01/59|5000
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
0110|v.k. agrawal |g.m.|marketing|12/31/40|9000
```

The Dot (.)

The dot (`.`) matches a single character. For example, the pattern `2…` matches a fourcharacter

pattern beginning with a 2. The combination of * and dot (`.*`) constitutes a very useful regular expression. It signifies any number of characters or none. For example, when you are not sure about the initial of *saxena,* you can give the expression as -

```
$ grep ".*saxena" emp.lst
2345|j.b. saxena|g.m.|marketing|03/12/45|8000
```

Specifying Pattern Locations (^ and $)

When we need to search for a pattern either at the beginning or at the end of a line, we can use ^ and $ respectively. For example, following command searches all the employees whose employee ID starts with 2.

```
$ grep "^2" emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
2365|barun sengupta|director|personnel|05/11/47|7800
2476|anil aggarwal|manager|sales|05/01/59|5000
2345|j.b. saxena|g.m.|marketing|03/12/45|8000
```

6. Write a short note on appending, inserting and changing text with respect to text.

The ***sed*** command provides some text editing commands as a part of its action component. One can use ***i*** (insert), ***a*** (append), ***c*** (change) and ***d*** (delete) for doing appropriate action on the file. Since ***sed*** is a stream editor, the effect of these commands is **on every line of the input file by default.** If we want the command to be applied on a specific line, then the line number (termed as address) should be specified. The input file will be usually opened for reading. But, the actions like insert, append etc. are writing jobs. It is obvious that a file cannot be opened for reading as well as writing at a time. Hence, the output of these actions must be redirected to a temporary file first. The contents of temporary file must be moved to the input file to modify it using ***mv*** command.

For understanding ***i, a, d*** and ***c*** commands, let us use a file *test.lst* as below –

```
$cat test.lst
Manager
Director
Executive
```

Insertion: Use the command ***i*** to insert any number of lines into a file at a required position. We will consider different examples to understand the working of ***i*** command.

Ex1: Using ***i*** without any address inserts the given line(s) ***before every line*** of the file. For example,

```
$sed 'i Engineer' test.lst
Engineer
Manager
Engineer
Director
Engineer
Executive
```

One can observe that *Engineer* has been included before every line of the file. But, if you check the contents of the file *test.lst*, it will be unmodified –

```
$cat test.lst
Manager
Director
Executive
```

One can insert more than one string at a required position using single command. Following example inserts 3 strings into 2nd position of the file.

```
$sed '2i\ #Give \ before pressing enter key
>Software Engineer\ #except for the last line
>Test Engineer\
>CEO' test.lst > temp #Copy result into temporary file
$ mv temp test.lst #Move temp file to test.lst
$ cat test.lst #Check the contents of test.lst
Manager
Software Engineer
Test Engineer
CEO
Director
Executive
```

Append: The command ***a*** is used to append any number of lines at specified position. We will consider various situations of using ***a.***

Ex1. By default, the command ***a*** appends the given string ***after every line*** of the input file. For example,

```
$sed 'a Engineer' test.lst > temp
$ mv temp test.lst
$ cat test.lst
Manager
Engineer
Director
Engineer
Executive
Engineer
```

Ex2. To append the string at required position, use the address (line number) as shown below. Here, the string will be appended after 2nd line of *test.lst*.

```
$sed '2a Engineer' test.lst > temp
$ mv temp test.lst
$ cat test.lst
Manager
Director
Engineer
Executive
```

Ex3. The append command can be used to add line-spacing between the lines of given file–

```
$sed 'a\ #press enter key
> ' test.lst > temp #close single-quote without any text
$ mv temp test.lst
$ cat test.lst
Manager
Director
Executive
```

Change: Use the command ***c*** to change a particular line by required string. By default, the command ***c*** will change all the lines when address is not given. For example,

Ex1.

```
$sed 'c Engineer' test.lst > temp
$ mv temp test.lst
$ cat test.lst
Engineer
Engineer
Engineer
```

Here, we can observe that all the lines of the file *test.lst* got changed to *Engineer*.

Ex2. Change only the required line by specifying the address as below –

```
$sed '3c Deputy Managaer' test.lst > temp
$ mv temp test.lst
$ cat test.lst
Manager
Director
Deputy Manager
```

7. Explain awk built-in variables with suitable examples.

There are several built-in variables in *awk* as shown in Table 4.1. They all have their own default values, but it is possible for a user to assign different values to them.

Table 4.1 Built-in Variables of *awk*

Variable Significance

NR Cumulative number of lines read
FS Input Field Separator
OFS Output Field Separator
NF Number of fields in current line
FILENAME Current input file
ARGC Number of arguments in command line
ARGV List of arguments

Some of the built-in variables are explained here –

· **NR:** It is used to count number of lines read from the input file. Its usage has been shown in some of the previous examples.

· **FS:** As we have discussed earlier, the ***awk*** treats a contiguous array of spaces as the default delimiter between the fields. When some other character is a delimiter in out input file (for ex, emp.lst has | as the delimiter), we need to specify it using –F. An alternative way is to use ***FS*** variable and setting it within BEGIN section as –

```
BEGIN { FS= "|"}
```

Now, while running *awk*, -F is not necessary.

· **OFS:** When we use ***print*** statement with comma-separated arguments, each argument will be separated by a space. It is the default field separator in ***awk***. If we want some other character to be a field separator, ***OFS*** is used in BEGIN section as–

```
BEGIN { OFS= "~"}
```

NF: This variable is useful in checking whether all the lines in the input file have required number of fields or not. For example, assume few lines in *emp.lst* file do not contain all the 6 fields (empno, name, designation, department, date of birth and salary). Then, ***NF*** variable is used to check the lines which are not containing all 6 lines. Assume we have an input file *errEmp.lst* in which few lines do not contain all 6 fields of *emp.lst*. Then verify it using the below given command –

```
$ awk 'BEGIN { FS="|"}
> NF !=6 {
> print "Record No ", NR, "has ", NF, " fields"}' errEmp.lst
Record No 6 has 4 fields
Record No 10 has 5 fields
Record No 14 has 3 fields
```

· **FILENAME:** It stores the name of the current file being processed. By default, *awk* doesn't print the filename. One can print it using the statement like –

```
'$6<4000 {print FILENAME, $0}'
```

Here, $0 indicates entire line.

8. Create a script file called file properties that reads a filename entered and outputs it properties.

```
#!/bin/bash/
echo "enter file name"
read file
if [ -f $file ]
then
set - `ls -l $file`
echo "File permission : $1"
echo "File link : $2"
echo "File user name : $3"
echo "File group name : $4"
echo "File block size : $5"
echo "Date of modifiation : $6:$7"
echo "Time of modification : $8"
echo "File name : $9"
else
echo "File not found"
fi
```

9. Write a shell script that accepts file name as argument and display its creation time if file exists and if it does not send output error message.

```
#!/bin/bash/
if [ $# -eq 0 ]
then
echo "No arguments"
else
for i in $*
do
```

```
if [ ! -e $i ]
then
echo "File does not exist"
else
ls -l $1|tr -s " "|cut -d " " -f6,7,8,9
fi
done
fi
```

10.How do you achieve substitution using sed command? Give examples.

The command **s** (substitution) allows to replace a pattern in the input file with something else. The syntax is –

[address] **s** */ expr1 / expr2 / flags*

Here, the *expr1* is replaced by *expr2* in all the lines specified by *address*. When *address* is not specified, the substitution is performed for all matching lines in the file. Consider an example –

```
$ sed 's/|/:/' emp.lst
2233:a.k. shukla|g.m.|sales|12/12/52|6000
9876:jai sharma|director|production|03/12/50|7000
5678:sumit chakrobarty|d.g.m|marketing|04/19/43|6000
```

……

Here, our *expr1* is pipe symbol and *expr2* is colon. We are instructing to replace all pipes by colon in the file *emp.lst*. But, when we observe the output, only the first (left-most) occurrence of pipe in every line is replaced by colon. To replace all the pipes in a line, we need to use the flag ***g (global)***. For example,

```
$ sed 's/|/:/g' emp.lst
2233:a.k. shukla:g.m.:sales:12/12/52:6000
9876:jai sharma:director:production:03/12/50:7000
```

……

We can choose the number of lines on which the replacement should happen. In the below example, the pipe is replaced by colon only for first two lines –

```
$ sed '1,2s/|/:/g' emp.lst
2233:a.k. shukla:g.m.:sales:12/12/52:6000
9876:jai sharma:director:production:03/12/50:7000
5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000
2365|barun sengupta|director|personnel|05/11/47|7800
```

……

One can replace a string with another string. In the following example, the string *director* is replaced by *member* only in first 5 lines.

```
$ sed '1,5s/director/member /' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|member |production|03/12/50|7000
```

Regular expressions can be used for patterns while doing substitution. For example, all different spellings like *agarwal, aggarwal* and *agrawal* can all be replaced by one simple string *Agarwal* as shown below –

```
$ sed 's/[Aa]gg*[ar][ar]wal/Agarwal/g' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
```

……

```
2476|anil Agarwal|manager|sales|05/01/59|5000
```

……

```
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
```

…….

```
0110|v.k. Agarwal |g.m.|marketing|12/31/40|9000
```

The anchoring characters ^ and $ can also be used to indicate beginning and ending of the line in a file. For example, the following statement adds 2 as a prefix to every employee id in the file –

```
$ sed 's/^/2/' emp.lst
22233|a.k. shukla|g.m.|sales|12/12/52|6000
29876|jai sharma|director|production|03/12/50|7000
25678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000
```

……
The salary of every employee can be suffixed with **.00** using $ symbol as below –

```
$ sed 's/$/.00/' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000.00
9876|jai sharma|director|production|03/12/50|7000.00
5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000.00
2365|barun sengupta|director|personnel|05/11/47|7800.00
```

……

Using a single command, multiple strings can be substituted. For example, we would like to replace *director* by *member*, *executive* by *Execom* and *d.g.m* by *DGM*. Then use the command as –

```
$ sed 's/director/member/g #press enter key
> s/executive/Execom/g #press enter key
> s/d.g.m/DGM/g' emp.lst
```

…….

```
9876|jai sharma|member|production|03/12/50|7000
5678|sumit chakrobarty|DGM|marketing|04/19/43|6000
```