MODULE 4. AWK AND ADVANCED SHELL PROGRAMMING

4.1 INTRODUCTION TO awk – AN ADVANCED FILTER

AWK is one of the most prominent text-processing utility. It made its entry into the UNIX system in 1977. AWK derives its name from its authors – Aho, Weinberger and Kernighan. It combines the features of several filters. The *awk* command is a powerful method for processing or analyzing text files, which are organizes as rows and columns. Unlike other filters, *awk* operates at the *field* level and can easily access, transform and format individual fields in a line. It can accept extended regular expressions (ERE) for pattern matching. It has C-type programming constructs, variables and several built-in functions. Simple awk commands can be run on command line, whereas more complex tasks should be written as *awk programs* or *awk scripts* in files. In Linux, *awk* is used as *gawk* (GNU awk).

4.2 SIMPLE awk FILTERING

The syntax of *awk* command is –

```
awk options 'selection_criteria {action}' file(s)
```

Here, the *selection_criteria* is a form of addressing and it filters input and selects lines. The *action* indicates the action to be taken upon selected lines, which must be enclosed within the flower brackets. The *selection_criteria* and *action* together constitute an ***awk*** program enclosed within single quotes. The *selection_criteria* in *awk* can be a pattern as used in context addressing. It can also be a line addressing, which uses ***awk'***s built-in variable **NR.** The *selection_criteria* can be even a conditional expressions using && and || operators.

Consider the following example of ***awk*** command to select all the *directors* in the file *emp.lst.*

```
$ awk '/director/{print}' 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
```

Here, the selection criteria is `/director/`which selects the lines containing *director*. The action is `{print}`. If the selection criteria is missing, then *action* applies to all the lines. If *action* is missing, then entire line is printed.

The ***print*** statement prints the entire line, when it is used without any field specifier. It is a default action of ***awk*.**

4.2.1 Splitting a line into fields

The *awk* command uses special parameter $0 to indicate the entire line. It also uses $1, $2, $3 etc. to indentify various fields in the line. But, they should be enclosed within single quotes to avoid shell treating them as positional parameters.

The ***awk*** uses contiguous sequence of spaces and tabs as a single delimiter. But, when the input file has any other delimiter, we can specify it using ***–F*** option.

Ex1. The following command prints 2nd, 3rd, 4th and 6th fields from the lines containing *sales* from the file *emp.lst.*

```
$ awk -F "|" '/sales/{print $2,$3,$4,$6}' emp.lst
a.k. shukla g.m. sales 6000
chanchal sanghvi director sales 6700
s.n. dasgupta manager sales 5600
anil aggarwal manager sales 5000
```

Ex2. Regular expressions can be used in *awk* command to match the pattern as shown below –

```
$ awk -F"|" '/sa[kx]s*ena/' emp.lst
3212|shyam saksena|d.g.m|accounts|12/12/55|6000
2345|j.b. saxena|g.m.|marketing|03/12/45|8000
```

The above two examples have used context addressing format to match the pattern. One can use line-addressing format with the help of built-in awk variable NR.

Ex3. The following command prints 2nd, 3rd and 6th fields of the lines 3 to 6, along with the line numbers from the file *emp.lst*.

```
$ awk -F "|" 'NR==3, NR==6 {print NR, $2, $3,$6}' emp.lst
3 sumit chakrobarty d.g.m 6000
4 barun sengupta director 7800
5 n.k. gupta chairman 5400
6 chanchal sanghvi director 6700
```

The statement *NR==3* is a condition being tested, but not the assignment.

4.2.2 printf: Formatting Output

One can use formatted output with *awk* just like in C language. For that, ***printf*** statement is available and it uses format specifiers like %d, %s etc. as in C language. For example,

```
$ awk -F "|" '/director/ {
> printf "%3d %-20s %-12s %d\n", NR, $2, $3, $6}' emp.lst
  2 jai sharma           director     7000
  4 barun sengupta       director     7800
  6 chanchal sanghvi     director     6700
 11 lalit chowdury       director     8200
```

Here, the line number is printed with the width of 3. Name and designations are printed in 20 and 12 spaces respectively. These two fields are left justified using – symbol attached with %s. Note that *printf* requires \n to print a newline after each line.

Both ***print*** and ***printf*** statements can be redirected using > and | symbols. But, the file name or command after > or | must be enclosed within double quotes.

Ex1. In the following example, the lines containing *director* are selected. The result of ***printf*** is redirected using | to *sort* them.

```
$ awk -F "|" '/director/ {
>printf "%-20s %-12s\n", $2, $3 | "sort"}' emp.lst
barun sengupta      director
chanchal sanghvi    director
jai sharma          director
lalit chowdury      director
```

Ex2. In the below given example, we are trying to store the result of ***printf*** in a file *nlist* by redirecting the output using > symbol.

```
$ awk -F "|" '/director/ {
printf "%-20s %-12s\n", $2, $3 > "nlist"}' emp.lst

$ cat nlist          #use cat command to see contents of nlist
jai sharma          director
barun sengupta      director
chanchal sanghvi    director
lalit chowdury      director
```

4.3 VARIABLES AND EXPRESSIONS

The ***awk*** uses variables and expressions in its programs. Expressions consist of strings, numbers, variables and the operators. Unlike normal programming languages, ***awk*** doesn't contain data types like *int, char, float* etc. Every expression can be interpreted as either a string or a number. The *awk* makes the necessary conversion according to the context.

User-defined variables are used in ***awk***, but without declaration. Few points about ***awk*** variables –

- Variables are case-sensitive
- Unlike shell variables, *awk* variables don't require $ sign either in assignment or in evaluation.
- They do not need initialization. They are implicitly initialized to zero or null string, depending on the context.
- Strings in *awk* are always double-quoted and contain any character.
- Strings can include escape sequences, octal values, hexadecimal values etc. When octal and hexadecimal values are used, they are preceded by \ and \x respectively.
- String concatenation does not require any operator. Two strings can be placed side-by-side to concatenate them. For example,

```
x= “hello”;   y= “world”
print x y           #prints helloworld
print x “,” y       #prints hello,world
```

4.3.1 The Comparison Operator

The relational operators like greater than (>), less than (<), comparison (==), not equal to (!=) etc. can be used with *awk*.

Ex1. The command given below is to select the lines containing *director* OR (||) *chairman*. As the 3rd field of the file *emp.lst* has the designation, we can compare it directly.

```
$ awk -F "|" '$3=="director" || $3=="chairman" {
> printf "%-20s %-12s \n", $2, $3}' emp.lst
jai sharma          director
barun sengupta      director
n.k. gupta          chairman
chanchal sanghvi    director
lalit chowdury      director
```

Ex2. Using not equal to operator (!=) and AND (&&) operator, we can achieve the negation of list obtained in Ex1. That is, following command displays all the lines not containing *director* and *chairman*.

```
$ awk -F "|" '$3 != "director" && $3 != "chairman" {
>printf "%-20s %-12s \n", $2, $3}' emp.lst
a.k. shukla         g.m.
sumit chakrobarty   d.g.m
karuna ganguly      g.m.
s.n. dasgupta       manager
jayant Chodhury     executive
anil aggarwal       manager
shyam saksena       d.g.m
sudhir Agarwal      executive
j.b. saxena         g.m.
v.k. agrawal        g.m.
```

4.3.2 The Regular Expression Operators: ~ and !~

The *awk* provides two operators for handling regular expressions. The ~ operator is used to match a regular expression and !~ is used to negate the match. These operators must be used only with field specifiers like $1, $2 etc.

Ex1. In the below example, the 2nd field ($2) is matched with regular expressions that may result in any of *Chodhury*, *chowdury, saksena, saxena*. Observe the OR (||) operator used.

```
$ awk -F "|" '$2 ~ /[cC]how*dh*ury/ ||
> $2 ~ /sa[kx]s?ena/ {print}' emp.lst
4290|jayant Chodhury|executive|production|09/07/50|6000
6521|lalit chowdury|director|marketing|09/26/45|8200
3212|shyam saksena|d.g.m|accounts|12/12/55|6000
2345|j.b. saxena|g.m.|marketing|03/12/45|8000
```

Ex2. Following example uses `!~` operator to not to match *director* and *chairman.*

```
$ awk -F "|" '$3!~ /director|chairman/{print $2, $3}' emp.lst
a.k. shukla g.m.
sumit chakrobarty d.g.m
karuna ganguly g.m.
s.n. dasgupta manager
jayant Chodhury executive
anil aggarwal manager
shyam saksena d.g.m
sudhir Agarwal executive
j.b. saxena g.m.
v.k. agrawal  g.m.
```

4.3.3 Number Comparison

awk can handle both integer and floating type numbers. Relational operators also can be applied on them. Consider following examples:

Ex1. Listing of all the employees whose salary is more than 7500 can be done as below –

```
$ awk -F"|" '$6>7500 {printf "%-20s %d\n",$2,$6}' emp.lst
barun sengupta       7800
lalit chowdury       8200
sudhir Agarwal       8000
j.b. saxena          8000
v.k. agrawal         9000
```

Here, $6 is the 6th field (salary), is being compared with 7500.

Ex2. One can combine regular expression matching and numeric comparison. Following example lists out the people who have either born in 1945 OR getting the salary more than 8000.

```
$ awk -F"|" '$6>8000 || $5~/45$/{print $2, $5, $6}' emp.lst
lalit chowdury 09/26/45 8200
j.b. saxena 03/12/45 8000
v.k. agrawal  12/31/40 9000
```

In the file *emp.lst,* 5th field is date of birth. In this field, at the end we have year of birth. Hence, in the above example, $5 is matched with /45$/ indicated 45 is at the end (recollect meaning of $ in regular expressions).

4.3.4 Number Processing

Arithmetic operators like +, -, *, /, % etc can be used with *awk*. As an illustration, consider an example of calculating total salary of employees in the file *emp.lst.*

Let us assume that 6th field in the file *emp.lst* is a basic salary. And, every person get DA as 40% of basic and HRA as 15% of basic. The total salary of an individual is basic+DA+HRA. Now consider the command as shown –

```
$ awk -F "|" '$3=="director" {printf
> "%-20s %-12s %d %d %d\n", $2, $3,$6, $6*0.4, $6*0.15}' emp.lst
jai sharma           director     7000 2800 1050
barun sengupta       director     7800 3120 1170
chanchal sanghvi     director     6700 2680 1005
lalit chowdury        director     8200 3280 1230
```

Here, the arithmetic operator * indicating multiplication has been applied on field specifier $6.

4.3.5 Variables

User defined variables can be used with *awk*. The following example counts the number of *directors* whose salary is more than 7000.

```
     $ awk -F "|" '$3=="director" && $6>=7000 {
     > count++
     > printf "%d %-20s %-12s %d\n", count, $2, $3, $6}' emp.lst
     1 jai sharma           director     7000
     2 barun sengupta       director     7800
     3 lalit chowdury       director     8200
```

Observe that the variable *count* is directly used without any initialization. Note that, in *awk* variables do not need declaration. And they will be initialized to zero or null (depending on whether it is numeric or string in the context used). Hence, in the above example, *count* is incremented each time when one *director* is encountered whose salary is >=7000.

Note that, in *awk*, the arithmetic operators are applied on variables just similar to C language. Hence, the shorthand operator ++, -- etc. indicates increment/decrement by one.

4.4 THE –f OPTION

As we have discussed in the beginning of this chapter, an *awk* program can be of single line or can constitute multiple lines. When there are multiple lines are there in the program, it is better to put them into a file. One can create an *awk* program with the help of *cat* command. The filename usually will have the extension as ***.awk***. Then, to run this file, ***-f*** option is used.

Consider the program discussed in Section 4.3.5, which will select all the *directors* with salary more than 7000. Initially, create the file containing required code as below –

```
     $ cat >payroll.awk
     $3=="director" && $6>=7000 {
     printf "%d %-20s %-12s %d\n", ++count, $2, $3, $6}
     [Press Ctrl+c]
```

Now, run the file *payroll.awk* using the following statement –

```
$ awk -F "|" -f payroll.awk emp.lst
1 jai sharma           director     7000
2 barun sengupta       director     7800
3 lalit chowdury       director     8200
```

Observe that the code inside *payroll.awk* is not enclosed with single quotes.

4.5 THE BEGIN AND END SECTIONS

The *awk* statements are usually applied on all lines selected by address (selection criteria). But, if we want to print something before the processing starts or after completing the process, then BEGIN and END sections are useful.

The BEGIN and END sections are optional and have syntax as –

BEGIN{ action}
END { action}

The usage of these sections are depicted in the below given example. Here, in the BEGIN section we will print the heading for the columns and in the END section, we will print average basic pay. Let us first create a file *newPayroll.awk* (either using vi editor or cat command) as shown below –

newPayroll.awk

```
BEGIN {
        printf "SlNo \t Name \t\t Salary\n"
}
$6>7500{
        count++
        total += $6
        printf "%3d %-20s %d\n", count, $2, $6
}
END{
        printf "\nThe average salary is: %d\n", total/count
}
```

Note that, after the BEGIN section, we will write the selection criteria ($6>7500) and then the action within a pair of curly braces (flower brackets) as per the syntax of *awk* commands.

Now, run the file using the command –

```
$awk –F “|” –f newPayroll.awk emp.lst
```

```
SlNo    Name            Salary
  1  barun sengupta       7800
  2  lalit chowdury       8200
  3  sudhir Agarwal       8000
  4  j.b. saxena          8000
  5  v.k. agrawal         9000

The average salary is: 8200
```

NOTE:

- The *awk* command can read from standard input, when filename is not given.
- Unlike *expr* command, *awk* can be used to perform floating point arithmetic. For example,

```
     $awk ‘BEGIN{printf “%f\n”, 22/7}’
     3.142857
```

4.6 BUILT – IN VARIABLES

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.

4.7 ARRAYS

Array is a variable that can store a set of elements. Each element is accessed by a subscript called *index.* Arrays in *awk* are different compared to other programming languages as given –

- Arrays are not formally defined. An array is considered as declared the moment it is used.
- Array elements are initialized to zero or an empty string unless it is initialized explicitly.
- Arrays expand automatically
- The index need not be just integer. It can be anything, even a string.

Let us consider the salary field (6th field) of *emp.lst* as basic salary. Assume every person has DA as 25% of basic, HRA as 50% of basic. The gross salary of an individual would be sum of basic, DA and HRA. The question is to select employees who are either working at *sales* department or at *marketing* department. And, then print the average of basic, DA, HRA and gross salary.

In the below given program, we are using array *tot*. In this array, first element *tot[1]* is used to store basic salary, second element *tot[2]* is used for *DA,* 3rd element *tot[3]* is for HRA, and 4th element *tot[4]* is used to store Gross salary. The variable *c* is used as a count of number of selected employees.

The BEGIN section is used to print the heading and to set the field separator FS. The END section is used to display the average values of basic, DA, HRA and Gross.

Store the following file as *avgSal.awk.*

```
BEGIN{
          FS= "|"
          printf "%50s\n", "Basic       DA      HRA     Gross"
}/sale|marketing/{
   da=0.25*$6
   hra = 0.5*$6
   gross = $6 + hra +da

   tot[1] += $6
   tot[2] += da
   tot[3] +=hra
   tot[4] += gross
   c++
}
END{
   printf "\t Average %5d %5d %5d %5d\n", tot[1]/c, tot[2]/c,
                                             tot[3]/c, tot[4]/c
}
```

Now, run the above file as –

```
         $ awk -f avgSal.awk emp.lst
                   Basic      DA      HRA     Gross
           Average 6812       1703    3406    11921
```

4.7.1 Associative (Hash) Arrays

Arrays in *awk* are associative – means the information is stored as a pair of *key* and *value*. Even if we use integers as subscripts, the *awk* doesn't treat array indexes as integers. The array index is the *key*, which is saved internally as a string. In the following example, when we assign `mon[1]="jan"` , the *awk* converts number 1 to a string. Therefore, the index "1" is different from the index "01". Hence, there is no specific order in which the array elements are stored.

```
BEGIN{
        Dir["N"]= "North"
        Dir["E"]= "East"
        printf "N is %s and E is %s\n", Dir["N"], Dir["E"]

        mon[1]="jan"
        mon["1"]="January"
        mon["01"]= "JAN"
        printf "mon[1] is %s\n", mon[1]
        printf "mon[01] is also %s\n", mon[01]
        printf "mon[\"1\"] is also %s\n", mon["1"]
        printf "But mon[\"01\"] is %s\n", mon["01"]
 }
```

Run the above program as –

```
$ awk -f hashArry.awk
N is North and E is East
mon[1] is January
mon[01] is also January
mon["1"] is also January
But mon["01"] is JAN
```

The points to be observed from the output of above program –

- Setting the index as "1" has over-written the setting of `1` done before. Hence, `mon[1]` is printed as `January.` This is because *awk* treats 1 also as string "1".
- Accessing elements `mon[1]`and `mon[01]`both indicates with the subscript "1".
- The elements `mon["1"]` and `mon["01"]` are different.
- In the program, as *printf* uses double quotes, we need to escape the array indices like `mon["1"] mon["01"]` and using back-slash.

4.7.2 The Environment Array: ENVIRON[ ]

The array ENVIRON[ ] stores all environment variables. That is, it is helpful in knowing the name of the user who is running the program, home directory etc. Consider the following example –

```
$ awk 'BEGIN {
> print "Home = " ENVIRON["HOME"]
> print "Path = " ENVIRON["PATH"]
> }'
Home = /home/john
Path = /usr/lib64/qt-3.3/bin:/usr/kerberos/bin:/usr/local/bin:/bin:
       /usr/bin:/hon
```

4.8 FUNCTIONS

There are several built-in functions in *awk* for performing arithmetic and string operations as shown in Table 4.2. Most of these functions behave similar to that in C programming. The arguments of the function are enclosed within parentheses and separated by comma. These functions are explained hereunder:

- **int(x):** This function calculates the integral portion of a number, without rounding it off. For example,
  ```
  $awk 'BEGIN{ print int(3.7)}'
  3
  ```
- **sqrt(x):** It is used to compute the square root of the number x.
  ```
  $awk 'BEGIN{ print sqrt(25)}'
  5
  ```
- **length:** It determines the length of its argument. If no argument is present, the entire line is assumed to be the argument. One can use it to locate the lines whose length exceeds a specific number of characters. The following example lists all the lines from the file *emp.lst* where number of characters is more than 50.

```
$awk -F"|" 'length>50' emp.lst
5678|sumit chakrobarty|d.g.m|marketing|04/19/43|6000
2365|barun sengupta|director|personnel|05/11/47|7800
4290|jayant Chodhury|executive|production|09/07/50|6000
6521|lalit chowdury|director|marketing|09/26/45|8200
3564|sudhir Agarwal|executive|personnel|07/06/47|8000
```

The length function can be used with argument also. Following example lists all the employees whose name (second field, $2) has less than 12 characters.

```
$awk -F"|" 'length($2)<12' emp.lst
2233|a.k. shukla|g.m.|sales|12/12/52|6000
9876|jai sharma|director|production|03/12/50|7000
5423|n.k. gupta|chairman|admin|08/30/56|5400
2345|j.b. saxena|g.m.|marketing|03/12/45|8000
```

Table 4.2 Built-in Functions in *awk*

Function	Description
int(x)	Returns integer value of x
sqrt (x)	Returns square root of x
length	Returns length of complete line
length(x)	Returns length of x
substr(stg, m, n)	Returns portion of string of length n, starting from position m in string *stg*
index(s1, s2)	Returns position of string s2 in string s1
split(stg, arr, ch)	Splits string *stg* into array *arr* using *ch* as delimiter and returns number of fields
system("cmd")	Runs UNIX command *cmd* and returns its exit status

- **index(s1, s2):** It determines the position of a string *s2* within a larger string *s1*. This function is usually helpful in validating single character fields. Consider the following examples:

  Ex1. Checking for the character *b* in text *abcde*. The result displayed is 2.

  ```
  $awk 'BEGIN{
  > x=index("abcde", "b")
  > print x}'
  2
  ```

  Ex2. Checking for substring *cd*, whose position is 3.

  ```
  $ awk 'BEGIN{
  > x=index("abcde", "cd")
  > print x}'
  3
  ```

Ex3. Checking for substring *p*, which is not present. Result would be 0.

```
$ awk 'BEGIN{
> x=index("abcde", "p")
> print x}'
0
```

- **substr(stg, m, n):** This function extracts a substring of size *n* characters from a string *stg* starting from the position *m*. For example, following code extracts 3 characters from the string `hello how are you?` starting from 7th character.

  ```
  $ awk 'BEGIN{
  > x=substr("hello how are you?", 7,3)
  > print x}'
  how
  ```

  This function can be used to select those people who have born between 1945 and 1951 using the following code –

  ```
  $awk -F "|" 'substr($5, 7,2)>45 && substr($5,7,2)<52' emp.lst
  9876|jai sharma|director|production|03/12/50|7000
  2365|barun sengupta|director|personnel|05/11/47|7800
  4290|jayant Chodhury|executive|production|09/07/50|6000
  3564|sudhir Agarwal|executive|personnel|07/06/47|8000
  ```

- **split(stg, arr, ch):** This function breaks up a string *stg* on the delimiter *ch* and stores the fields in an array *arr[ ]*. The following example uses space (given within double quotes as a 3rd argument to *split* function) as a delimiter and stores each word of the string `hello how are you?` in the array `arr`.

  ```
  $ awk 'BEGIN{
  split("hello how are you?", arr, " ")
  printf "%s\n%s\n%s\n%s\n",arr[1],arr[2],arr[3],arr[4]}'
  hello
  how
  are
  you?
  ```

- **system:** This function can be used for running UNIX commands within awk. For example,

  ```
  $ awk 'BEGIN{
  > system("date")
  > system("pwd")}'
  Mon Jan  1 08:54:14 IST 2007
  /home/john
  ```

4.9 CONTROL FLOW

Just like any higher programming languages, the *awk* supports conditional structures like *if* statement and looping structures *while* and *for*. All these will execute a set of statements depending on the success or failure of the control command (or condition).

4.9.1 The if Statement

The syntax of *if* statement is as below –

```
if(condition is true) {
     statements
} else {
     Statements
}
```

When there is a single statement, curly brackets are not necessary. The following example selects people whose salary is more than 8000. Note that, *selection criteria* is not given in the below command. Instead, the *action* part itself contains the condition to select the required records.

```
$ awk -F "|" '{
> if ($6>8000)
> printf "%s\t%d\n", $2, $6}' emp.lst
lalit chowdury  8200
v.k. agrawal    9000
```

The *if* statement can take the condition involving regular expressions, relational operators like &&, || etc.

Ex: The code given below extracts the record only if the salary is greater than 8000 and the designation is director.

```
$ awk -F"|" '{
> if($6>8000 && $3=="director")
> printf "%s\t %s\n", $2, $6}' emp.lst
lalit chowdury   8200
```

Ex: Regular expressions are used below to match the name as different spellings of *choudhury* and *saxksena*.

```
$ awk -F"|" '{
> if($2~/[cC]ho[wu]dh?ury|sa[xk]s?ena/)
> printf "%s\t%s\n", $2, $3}' emp.lst
lalit chowdury  director
shyam saksena   d.g.m
j.b. saxena     g.m.
```

Ex: Showing usage of *else* part –

```
$ awk 'BEGIN{
> a=10
> b=20
> if(a>b)
>   print "a is greater"
> else
>   print "b is greater" }'
b is greater
```

4.9.2 Looping with *for*

The *for* loop in *awk* has two formats. First one is similar to that in C programming language. Consider the example given below to display numbers from 1 to 5.

```
$ awk 'BEGIN{
> for(i=1;i<=5;i++)
>   printf "%d\t", i }'
1       2       3       4       5
```

The second form of *for* loop has the following syntax –

```
for(k in arr)
     statements
```

Here, *k* is the subscript of the array. As the subscript of an array can be a string, one can use this form of array for various purposes. Following example is used to print various values of ENVIRON array (which is built-in array in awk):

```
$ awk 'BEGIN{
> for(k in ENVIRON)
> print k "=" ENVIRON[k]}'
```

Following example counts number of people belonging to each of distinct designations in the file *emp.lst*.

```
$ awk -F"|" '{count[$3]++}
> END {for (desig in count)
> print desig, count[desig]}' emp.lst
d.g.m 2
director 4
executive 2
manager 2
g.m. 4
chairman 1
```

In this example, the array `count` takes the designations like *director, manager* etc. as index. So, when each line of the file *emp.lst* is processed, the respective designation count is incremented.

4.9.3 Looping with *while*

The *while* loop is iterated till the condition remains true. The following example prints the values from 0 to 4.

```
$ awk 'BEGIN{
>  while(k<5)
>  {  printf "%d\t", k
>     k++
>  }
> }'
0       1       2       3       4
```

Note that, the variable *k* is initialized to zero automatically.

4.10 ADVANCED SHELL PROGRAMMING

It has been discussed earlier (in Module 1, Section 1.7) that the shell is an interpreter and also a scripting language. Shell is a process too, through which it makes itself available to programs. So, it is necessary to understand the environmental changes that take place when the shell executes a program (a shell script). One should know how to change these environmental parameters. Most of such advanced shell programming is needed for a system administrator.

4.10.1 The *sh* Command

When a shell script is executed, the shell spawns (a child process created by parent process is known as spawning) a sub-shell and this sub-shell actually executes the script (Read Section 1.7.2 from the Module 1 for more explanation). When the script execution is completed the child shell is terminated and the control is returned to the parent shell.

One can explicitly invoke a sub-shell using ***sh*** command to execute a shell script. The syntax would be –

```
$sh filename.sh
```

A shell script that runs with the help of ***sh*** (or ***ksh, bash***) need not have execute permission.

Note that, usage of above line creates a sub-shell (child process), which is of same type as that of login shell. And, the interpreter line written inside the script will be ignored. Hence, if the script is written in ksh and the login shell is bash, then there might be unexpected result.

4.10.2 Exporting Shell Variables

The values stored in the shell variables are local to the shell and they are not passed on to the child shell. But, one can use ***export*** command to pass the variables used in the parent shell into the child shell. To understand the concept, consider the following illustration.

Step1. Create a file *ex.sh* as below –

```
$ cat > ex.sh
echo “The value of x is $x”
x=20
echo “The value of x is $x”
```

Step2. Now, assign a value 10 to x in the prompt and then run the above script *ex.sh*. Observe the output.

```
$ x=10          #x is 10 on a parent shell
$ sh ex.sh      #child shell is spawned to execute ex.sh
The value of x is         #nothing got printed for x
The value of x is 20      #x is 20 inside the script
```

Here, on the command line we are assigning a variable x with a value 10. Hence, now the parent shell has a variable x. Then, we are executing the file *ex.sh*, using *sh* command. This will create a child shell. The first line in the file is to print variable x. But, we can observe through output that it displays nothing. This assures that the variable x in the parent shell did not pass to child shell. Later, in the script file, we are creating x again with a value 20, which will be printed by the script without any issues.

Now, to pass the parent-shell-variable into the child, use the ***export*** command.

Step3. Assign 10 to x in the prompt, then use ***export*** command and then run the script.

```
$ x=10          #x is 10 on a parent shell
$ export x      #parent x is exported to child
$ sh ex.sh      #child shell is spawned to execute ex.sh
The value of x is 10      #value 10 got printed for x
The value of x is 20      #x is 20 after new assignment
```

The above exercise clearly indicates that the variable declared in the login shell is local. To make it globally available to all child shells, the ***export*** command is essential.

Note that, the variable declared inside child shell cannot be made available to parent shell. To ensure this, we can just print value of x again (after doing above exercise) in the prompt as –

```
$ echo $x
10              #value available with parent shell is printed
```

4.11 CONDITIONAL PARAMETER SUBSTITUTION

We know that to evaluate a variable in shell scripts, we will use $ symbol. We have also discussed earlier (Section 1.7.11 of Module1) that the ***expr*** command is used for evaluation of expression. But, Shell provides an interesting format to evaluate a variable depending on whether the variable has some defined value or a null value. This is known as ***parameter substitution***. The syntax is as given –

```
${<var>: <opt> <stg>}
```

Here, `var`	is a variable to be evaluated
`opt`	can be any one of the symbols +, -, = or ?
`stg`	is a string

The behavior of the parameter substitution can be better understood based on the symbols with which it is used as discussed hereunder.

The + Option: If the `var` contains a defined value and not null, then it evaluates to `stg`. This can be understood in the following examples:

- **Ex1:** Assign a variable x with 10 and test for parameter substitution as shown –
  ```
  $x=10
  $echo ${x: + “x is defined”}
  x is defined
  ```
  As the value of x is not null, the string (stg) part in the expression got evaluated.

- **Ex2:** Check for a non-existing variable, or the one with null value.
  ```
  $echo ${y: + “y is defined”}
  ```
  Observe that as the variable y is not assigned any value before (or it contains a null value), the string (stg) is not evaluated.

- **Ex3:** Checking whether directory is empty or not.
  ```
  $found=`ls`     #ls within back-quotes
  $echo ${found: + “This directory is not empty”}
  This directory is not empty
  ```
  Here, we have initialized the variable `found` with the command `ls` using back-quotes. Hence, if there are some files/sub-directories in the current working directory, the variable `found` will not be null.

The – Option: This is quite opposite to + option. Here if the `var` do not contains a defined value and it is null, then it evaluates to `stg`. This is useful in setting default value for a variable/filename when user forgets to provide the value. But, the default value provided with – option is only for that moment, and `var` will not be assigned with that value. Consider following examples:

- **Ex1:** Use an un-assigned variable x and give it a default value.
  ```
  $echo ${x: - 10}
  10
  $echo $x
                 #prints nothing
  $
  ```
  Here, the variable x was unassigned before. So, as it is null, value 10 will be assigned. But, in the prompt if we check its value now, it is still null only.

- **Ex2:** Consider a shell script *test.sh* –

  ```
  $ cat > test.sh
  echo “Enter a file name:”
  read fname
  fname=${fname: - emp.lst}
  echo fname
  [Ctrl+C]
  ```

  Run the above script as –

  ```
  $ sh test.sh
  Enter a file name:
                           #do not provide any filename
  fname= emp.lst
  ```

  We are running the script and not providing filename when it is asked. Hence, `fname` will be null. Now, `fname` takes the value `emp.lst` from parameter substitution. As we have used assignment operator with parameter substitution, the value of `fname` would be `emp.lst` now onwards.

The = Option: This option goes one step ahead of – option, where it assigns the value to the variable. If we use = option inside the parameter substitution, then explicit assignment is not required.

- **Ex:**

  ```
  $echo ${x:=10}      #10 is assigned to x permanently
  10
  $echo $x            #verify value of x
  10
  ```

The = option is useful in loops. For example, the while-loop used as below

```
x=1
while [ $x –le 10 ]
```

can be re-written as –

```
while [ ${x:=1} –le 10 ]
```

The ? Option: It evaluates the parameter if the variable is assigned and not null. Otherwise, it echoes a string and kills the shell. Consider the following script –

```
$ cat > test.sh
echo “Enter a file name:”
read fname
${fname: ? “No file name entered”}
echo fname
[Ctrl+C]
```

Run the above script as –

```
     $ sh test.sh
     Enter a file name:
                                #do not provide any filename
     No file name entered
     $                          #shell aborted
```

4.12 SHELL FUNCTIONS

One can create user-defined functions in shell. A function is a collection of statements to perform a particular task. The syntax would be –

```
function_name(){
     statements
     return value
}
```

The *return* statement is optional. When *return* is used, it returns a value representing the success or failure of the function.

The functions in shell support command line arguments and usual notations like $1, $2, $*, $# etc are supported. While defining a function, we use parentheses (), but while invoking the function, it is not used. A function with arguments is also invoked without parentheses.

Consider a function definition on the prompt as –

```
     $ fun(){
     > ls -l $* | more
     > }
```

Now call a function as –

```
$ fun
total 352
-rw-r--r-- 1 john STAFF    296 Jan 25  2007 avgSal.awk
-rw-rw-r-- 1 john john      280 Jan 30  2007 caseEx.sh
-rw-rw-r-- 1 john john     104 Feb  3  2007 cmdArg.sh
-rw-r--r-- 1 john STAFF    326 Jan  5  2007 cutlist1
-rw-r--r-- 1 john STAFF    415 Jan  5  2007 cutlist2
-rw-r--r-- 1 john STAFF    207 Jan 10  2007 dlist
-rw-r--r-- 1 john STAFF    534 Jan 18  2007 dlist.lst
-rw-r--r-- 1 john STAFF    342 Jan 10  2007 e1.lst
-rw-r--r-- 1 john STAFF    399 Jan 10  2007 e2.lst
-rw-r--r-- 1 john STAFF    110 Jan 10  2007 elist
```

Observe that the `fun` is called without arguments. Hence, $* inside the body of the function is not effective. Thus, `ls –l` is executed on a current directory. We can give argument to function as –

```
$ fun test.*
-rw-r--r-- 1 chetana STAFF   216 Jan 25  2007 test.awk
-rw-rw-r-- 1 chetana chetana  15 Jan 19  2007 test.bak
-rw-r--r-- 1 chetana STAFF    60 Jan 18  2007 test.lst
-rwxrwxr-x 2 chetana chetana  22 Jan 21  2007 test.sh
-rw-rw-r-- 1 chetana chetana  33 Jan 19  2007 test.txt
-rw-rw-r-- 1 chetana chetana  30 Jan 19  2007 test.txt.bak
```

The argument to `fun` is `test.*` . Hence, inside the function, the statement is treated as –

```
ls –l test.*
```

Hence, all the files whose name is `test` (with all possible extensions) will be displayed.

4.13 EVALUATING TWICE: eval COMMAND

The ***eval*** command will take an argument and construct a command out of it, which will be executed by the shell. The ***eval*** statement tells the shell to take eval's arguments as command and run them through the command-line.

To understand the concept, let us first define few strings and then try to display the value of string with a numbered variable.

```
$ text1= “Emp ID:”
$ text1= “Name:”
$ text1= “Designation:”
$ x=1
$ echo $text$x
1                    #output
```

Here, we expect the output to be `Emp ID:` because, `$x` is `1` and `$text$x` would be `text1`. But, the shell evaluates the command line from left to right. So, it first encounters `$text` which is not defined at all. Then it evaluates `$x`. Hence we will get the output as 1.

The ***eval*** statement evaluates a command line twice. In the first pass, it suppresses some evaluation and performs it only in the second pass. This is what we want in our previous example. So, if we escape the first $ symbol in `$text$x`, then the first pass evaluates only $x. So, we will get only `text1`. In the second pass, we have to evaluate it using ***eval*** as shown below –

```
$eval echo \$text$x
Emp ID:                  #displayed value of text1
```

Consider one more example in which we have a variable x assigned as 10 and another variable y which is assigned as x. Here, we would like to retrieve the value of x through y as shown below –

```
$ x=10
$ y=x
$ echo $y           #it prints just x, but not 10
x
$ eval echo \$$y    #escape first $ using \
10                  #to get 10
```

In the above example, when we use the statement,

```
$ eval echo \$$y
```

the first $ symbol is escaped using slash. So, only $y is evaluated to get x. Then ***eval*** is used to evaluate value of x and result would be 10.

4.14 THE exec STATEMENT

Usually, when we run a command in UNIX, a new process is spawned (using ***fork*** system call). But, the forking just creates a child process and it is not enough to run a new program. To do that, the forked child needs to overwrite its own image with the code and data of the new program. This is done with ***exec*** internally. (Details are discussed in Module 5)

When ***exec*** command is used with another command externally on the command line, a new process is not spawned. Instead, the current process is overlaid with the new command. In other words, the exec command is executed in place of the current shell without creating a new process. This is useful for shell programmers when they need to overwrite the current shell itself with the code of another program. When the ***exec*** is preceded with any UNIX command, that command overwrites the current process – most of the times the shell. As the shell is overwritten by some other command, the user will be logged out immediately after the completion of that command.

Consider the following example –

```
$ exec date
Tue Jan 28 21:25:53 IST 2017
login:
```

Observe that, after displaying the date, login prompt is appeared. This indicates that the user is logged out because the *exec* has made the *date* command to overwrite the shell itself.

Sometimes, the system administrator what the user to run only one program automatically after logging in and the user should be logged out after completion of that program. That means, the user is denied to use the shell as per his/her wish. In such situations, the system admin can place the command in the file `.profile` preceded by ***exec***. The shell overlays itself the code of the program to be executed, and when command execution is complete, the user is logged out – as there is not shell waiting for it.

NOTE: The following topic is mentioned in Module 1 as per the syllabus. But, as it requires the knowledge of regular expressions, it is discussed here. Students are suggested to read this topic as a part ***expr*** command given in Section 1.7.11 in Module 1.

String Handling:

The ***expr*** command uses two expressions separated by colon for manipulating strings. The string to be manipulated is placed on the left side of the colon (:) and a regular expression is placed on its right. Depending on the composition of the expression, the ***expr*** command can perform three important string functions as explained hereunder:

- **Determining Length of the String:** The length of the string can be computed using ***expr*** command along with the help of regular expression (`.*`). This indicates that the number of character matching pattern must be extracted.
  ```
  $ expr “abcdefgh” : ‘.*’
  8
  ```
  Note that there must be space on both sides of colon symbol.

- **Extracting a Substring:** A substring can be extracted with the help of escaped characters \(and \). For example,
  ```
  $ var=2017
  $ expr “$var” : ‘..\(..\)’
  17
  ```
  The pattern group `\(..\)` here is actually the TRE (tagged regular expression) used by ***sed*** command. But, this is used with different meaning here. The two dots in the beginning indicate that two characters should be ignored in the beginning and two dots later indicate that two characters should be extracted.

  Consider one more example –
  ```
  $ var= “hello how are you?”
  $ expr “$var” : ‘.\(....\)’
  ello
  ```
  Here the first character (h) is ignored with the help of one dot. Later, 4 characters are extracted using 4 dots.

- **Locating Position of a Character:** To locate the first occurrence of a character in a given string, we need to count number of other characters which have appeared before the specific character. In the following example, we are trying to locate first occurrence of the character *d.* So, we need to count number of characters which are not *d*, and then there must be a *d*. Hence, the command should be –
  ```
  $ var= “abcdefgh”
  $ expr “$var” : ‘[^d]*d’
  4
  ```