Awk- An Advanced Filter

Introduction

awk is a programmable, pattern-matching, and processing tool available in UNIX. It works equally well with text and numbers. It derives its name from the first letter of the last name of its three authors namely Alfred V. Aho, Peter J. Weinberger and Brian W. Kernighan.

Simple awk Filtering

awk is not just a command, but a programming language too. In other words, awk utility is a pattern scanning and processing language. It searches one or more files to see if they contain lines that match specified patterns and then perform associated actions, such as writing the line to the standard output or incrementing a counter each time it finds a match.

Syntax:

```
awk option ‘selection_criteria {action}’ file(s)
```

Here, selection_criteria filters input and selects lines for the action component to act upon. The selection_criteria is enclosed within single quotes and the action within the curly braces. Both the selection_criteria and action forms an awk program.

Example: `$ awk ‘/manager/ { print }’ emp.lst`

Output:

```
9876   Jai      Manager   Productions
       Sharma
2356   Rohit    Manager   Sales
5683   Rakesh   Manager   Marketing
```

In the above example, /manager/ is the selection_criteria which selects lines that are processed in the action section i.e. {print}. Since the print statement is used without any field specifiers, it prints the whole line.

Note: If no selection_criteria is used, then action applies to all lines of the file.

Since printing is the default action of awk, any one of the following three forms can be used:

awk ‘/manager/ ’ emp.lst

awk ‘/manager/ { print }’ emp.lst

awk ‘/manager/ { print $0}’ emp.lst *$0 specifies complete line.*

Awk uses regular expression in sed style for pattern matching.

Example: awk –F “|” ‘ /r [ao]*/’ emp.lst

Output:

```
2356   Rohit       Manager   Sales
5683   Rakesh      Manager   Marketing
```

Splitting a line into fields

Awk uses special parameter, $0, to indicate entire line. It also uses $1, $2, $3 to identify fields. These special parameters have to be specified in single quotes so that they will not be interpreted by the shell.

awk uses contiguous sequence of spaces and tabs as a single delimiter.

Example: awk –F “|” ‘/production/ { print $2, $3, $4 }’ emp.lst

Output:

```
Jai Sharma |   Manager |      Productions
Rahul |        Accountant |   Productions
Rakesh |       Clerk      |   Productions
```

In the above example, comma (,) is used to delimit field specifications to ensure that each field is separated from the other by a space so that the program produces a readable output.

Note: We can also specify the number of lines we want using the built-in variable NR as illustrated in the following example:

Example: awk –F “|” ‘NR==2, NR==4 { print NR, $2, $3, $4 }’ emp.lst

Output:

```
2        Jai          Manager       Productions
         Sharma
3        Rahul        Accountant    Productions
4        Rakesh       Clerk         Productions
```

printf: Formatting Output

The printf statement can be used with the awk to format the output. Awk accepts most of the formats used by the printf function of C.

Example: awk –F “|” ‘/[kK]u?[ar]/ { printf “%3d %-20s %-12s \n”, NR, $2, $3}’
>emp.lst

Output:

```
4   R Kumar             Manager
8   Sunil kumaar        Accountant
4   Anil Kummar         Clerk
```

Here, the name and designation have been printed in spaces 20 and 12 characters wide respectively.

Note: The printf requires \n to print a newline after each line.

Redirecting Standard Output:

The print and printf statements can be separately redirected with the > and | symbols. Any command or a filename that follows these redirection symbols should be enclosed within double quotes.

Example1: use of |

printf “%3d %-20s %-12s \n”, NR, $2, $3 | “sort”

Example 2: use of >

printf “%3d %-20s %-12s \n”, NR, $2, $3 > “newlist”

Variables and Expressions

Variables and expressions can be used with awk as used with any programming language. Here, expression consists of strings, numbers and variables combined by operators.

Example: (x+2)*y, x-15, x/y, etc..,

Note: awk does not have any data types and every expression is interpreted either as a string or a number. However awk has the ability to make conversions whenever required.

A **variable** is an identifier that references a value. To define a variable, you only have to name it and assign it a value. The name can only contain letters, digits, and underscores, and may not start with a digit. Case distinctions in variable names are important: Salary and salary are two different variables. awk allows the use of user-defined variables without declaring them i.e. variables are deemed to be declared when they are used for the first time itself.

```
Example: X= “4”
         X= “3”
         Print X
         Print x
```

Note: 1. Variables are case sensitive.
2. If variables are not initialized by the user, then implicitly they are initialized to
zero.

Strings in awk are enclosed within double quotes and can contain any character. Awk strings can include escape sequence, octal values and even hex values.

Octal values are preceded by \ and hex values by \x. Strings that do not consist of numbers have a numeric value of 0.

Example 1: z = "Hello"
            print z   prints Hello

Example 2: y = “\t\t Hello \7”
            print y   prints two tabs followed by the string Hello and sounds a beep.

String concatenation can also be performed. Awk does not provide any operator for this, however strings can be concatenated by simply placing them side-by-side.

Example 1: z = "Hello" "World"
            print z   *prints* Hello World

Example 2 : p = “UNIX” ; q= “awk”
            print p q   *prints* UNIX awk

Example 3: x = “UNIX”
            y = “LINUX”
            print x “&” y   *prints* UNIX & LINUX

A numeric and string value can also be concatenated.

Example : l = “8” ; m = 2 ; n = “Hello”
            Print l m   *prints 82 by converting m to string.*
            Print l - m   *prints 6 by converting l as number.*
            Print m + n   *prints 2 by converting n to numeric 0.*

Expressions also have true and false values associated with them. A nonempty string or any positive number has true value.

Example: if(c)   *This is true if c is a nonempty string or positive number.*

The Comparison Operators

awk also provides the comparison operators like >, <, >=, <= ,==, !=, etc..,

Example 1 : $ awk –F “|” ‘$3 == “manager” || $3 == “chairman” {

```
> printf “%-20s %-12s %d\n”, $2, $3, $5}’ emp.lst
```

Output:

```
ganesh               chairman      15000
jai sharma           manager       9000
rohit                manager       8750
rakesh               manager       8500
```

The above command looks for two strings only in the third filed ($3). The second attempted only if (||) the first match fails.

Note: awk uses the || and && logical operators as in C and UNIX shell.

```
Example 2 : $ awk –F “|” ‘$3 != “manager” && $3 != “chairman” {
        > printf “%-20s %-12s %d\n”, $2, $3, $5}’ emp.lst
```

Output:

```
Sunil kumaar          Accountant     7000
Anil Kummar           Clerk          6000
Rahul                 Accountant     7000
Rakesh                Clerk          6000
```

The above example illustrates the use of != and && operators. Here all the employee records other than that of manager and chairman are displayed.

~ and !~ : The Regular Expression Operators:

In awk, special characters, called *regular expression operators* or *metacharacters*, can be used with regular expression to increase the power and versatility of regular expressions. To restrict a match to a specific field, two regular expression operators **~** (matches) and **!~** (does not match).

Example1: $2 ~ /[cC]ho[wu]dh?ury / || $2 ~ /sa[xk]s ?ena / *Matches second field*

Example2: $2 !~ /manager | chairman / *Neither manager nor chairman*

Note:

The operators ~ and !~ work only with field specifiers like $1, $2, etc.,.

For instance, to locate g.m s the following command does not display the expected output, because the word g.m. is embedded in d.g.m or c.g.m.

```
$ awk –F “|” ‘$3 ~ /g.m./ {printf “…..
```

prints fields including g.m like g.m, d.g.m and c.g.m

To avoid such unexpected output, awk provides two operators ^ and $ that indicates the beginning and end of the filed respectively. So the above command should be modified as follows:

```
$ awk –F “|” ‘$3 ~ /^g.m./ {printf “…..
```

prints fields including g.m only and not d.g.m or c.g.m

The following table depicts the comparison and regular expression matching operators.

Operator	Significance
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
>=	Greater than or equal to
>	Greater than
~	Matches a regular expression
!~	Doesn’t matches a regular expression

Table 1: Comparison and regular expression matching operators.

Number Comparison:

Awk has the ability to handle numbers (integer and floating type). Relational test or comparisons can also be performed on them.

Example:
```
$ awk –F “|” ‘$5 > 7500 {
        > printf “%-20s %-12s %d\n”, $2, $3, $5}’ emp.lst
```

Output:

```
ganesh              chairman      15000
```

```
jai sharma          manager      9000
rohit               manager      8750
rakesh              manager      8500
```

In the above example, the details of employees getting salary greater than 7500 are displayed.

Regular expressions can also be combined with numeric comparison.

Example: $ awk –F “|” ‘$5 > 7500 || $6 ~/1980$/’ {
              > printf “%-20s %-12s %d\n”, $2, $3, $5, $6}’ emp.lst

Output:

```
ganesh              chairman     15000   30/12/1950
jai sharma          manager      9000    01/01/1980
rohit               manager      8750    10/05/1975
rakesh              manager      8500    20/05/1975
Rahul               Accountant   6000    01/10/1980
Anil                Clerk        5000    20/05/1980
```

In the above example, the details of employees getting salary greater than 7500 or whose year of birth is 1980 are displayed.

Number Processing

Numeric computations can be performed in awk using the arithmetic operators like +, -, /, *, % (modulus). One of the main feature of awk w.r.t. number processing is that it can handle even decimal numbers, which is not possible in shell.

Example: $ awk –F “|” ‘$3’ == “manager” {
               > printf “%-20s %-12s %d\n”, $2, $3, $5, $5*0.4}’ emp.lst

Output:

```
jai sharma          manager     9000     3600
rohit               manager     8750     3500
rakesh              manager     8500     3250
```

In the above example, DA is calculated as 40% of basic pay.

Variables

Awk allows the user to use variables of there choice. You can now print a serial number, using the variable kount, and apply it those directors drawing a salary exceeding 6700:

```
$ awk –F”|”  ‘$3 == “director” && $6 > 6700 {
➢ kount =kount+1
➢ printf “ %3f  %20s %-12s %d\n”, kount,$2,$3,$6 }’ empn.lst
```

The initial value of kount was 0 (by default). That’s why the first line is correctly assigned the number 1. awk also accepts the C- style incrementing forms:

```
Kount ++
Kount +=2
Printf “%3d\n”, ++kount
```

THE –f OPTION: STORING awk PROGRAMS INA FILE

You should holds large awk programs in separate file and provide them with the .awk extension for easier identification. Let’s first store the previous program in the file empawk.awk:

```
$ cat empawk.awk
```

Observe that this time we haven’t used quotes to enclose the awk program. You can now use awk with the –f *filename* option to obtain the same output:

```
Awk –F”|” –f empawk.awk empn.lst
```

THE BEGIN AND END SECTIONS

Awk statements are usully applied to all lines selected by the address, and if there are no addresses, then they are applied to every line of input. But, if you have to print something before processing the first line, for example, a heading, then the BEGIN section can be used gainfully. Similarly, the end section useful in printing some totals after processing is over.

The BEGIN and END sections are optional and take the form

```
BEGIN {action}
END {action}
```

These two sections, when present, are delimited by the body of the awk program. You can use them to print a suitable heading at the beginning and the average salary at the end. Store this program, in a separate file **empawk2.awk**

Like the shell, awk also uses the # for providing comments. The BEGIN section prints a suitable heading , offset by two tabs (\t\t), while the END section prints the average pay (tot/kount) for the selected lines. To execute this program, use the –f option:

```
$awk –F”|” –f empawk2.awk empn.lst
```

Like all filters, awk reads standard input when the filename is omitted. We can make awk behave like a simple scripting language by doing all work in the BEGIN section. This is how you perform floating point arithmetic:

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

This is something that you can’t do with **expr.** Depending on the version of the awk the prompt may be or may not be returned, which means that awk may still be reading standard input. Use [*ctrl-d*] to return the prompt.

BUILT-IN VARIABLES

Awk has several built-in variables. They are all assigned automatically, though it is also possible for a user to reassign some of them. You have already used NR, which signifies the record number of the current line. We’ll now have a brief look at some of the other variable.

The FS Variable: as stated elsewhere, awk ues a contiguous string of spaces as the default field delimeter. FS redefines this field separator, which in the sample database happens to be the |. When used at all, it must occur in the BEGIN section so that the body of the program knows its value before it starts processing:

```
BEGIN {FS=”|”}
```

This is an alternative to the –F option which does the same thing.

The OFS Variable: when you used the print statement with comma-separated arguments, each argument was separated from the other by a space. This is awk’s default output field separator, and can reassigned using the variable OFS in the BEGIN section:

```
BEGIN { OFS=”~” }
```

When you reassign this variable with a ~ (tilde), awk will use this character for delimiting the print arguments. This is a useful variable for creating lines with delimited fields.

The NF variable: NF comes in quite handy for cleaning up a database of lines that don’t contain the right number of fields. By using it on a file, say emp.lst, you can locate those lines not having 6 fields, and which have crept in due to faulty data entry:

```
$awk ‘BEGIN { FS = “|” }
➢  NF !=6 {
```

- Print “Record No “, NR, “has ”, “fields”}’ empx.lst

The FILENAME Variable: FILENAME stores the name of the current file being processed. Like grep and sed, awk can also handle multiple filenames in the command line. By default, awk doesn’t print the filename, but you can instruct it to do so:

```
‘$6<4000 {print FILENAME, $0 }’
```

With FILENAME, you can device logic that does different things depending on the file that is processed.

ARRAYS

An array is also a variable except that this variable can store a set of values or elements. Each element is accessed by a subscript called the **index. Awk** arrays are different from the ones used in other programming languages in many respects:

- They are not formally defined. An array is considered declared the moment it is used.
- Array elements are initialized to zero or an empty string unless initialized explicitly.
- Arrays expand automatically.
- The index can be virtually any thing: it can even be a string.

In the program empawk3.awk, we use arrays to store the totals of the basic pay, da, hra and gross pay of the sales and marketing people. Assume that the da is 25%, and hra 50% of basic pay. Use the tot[] array to store the totals of each element of pay, and also the gross pay:
Note that this time we didn’t match the pattern sales and marketing specifically in a field. We could afford to do that because the patterns occur only in the fourth field, and there’s no scope here for ambiguity. When you run the program, it outputs the average of the two elements of pay:

```
$ awk –f empawk3.awk empn.lst
```

C-programmers will find the syntax quite comfortable to work with except that awk simplifies a number of things that require explicit specifications in C. there are no type declarations, no initialization and no statement terminators.

Associative arrays

Even though we used integers as subscripts in the tot [ ] array, awk doesn’t treat array indexes as integers. Awk arrays are associative, where information is held as *key-value* pairs. The index is the key that is saved internally as a string. When we set an array element using mon[1]=”mon”, awk converts the number 1 to a string. There’s no specified order in which the array elements are stored. As the following example suggests, the index “1” is different from “01”:

```
$ awk ‘BEGIN {
    ➢ direction [“N”] = “North” ; direction [“S”] ;
    ➢ direction [“E”] = “East” ; direction [“W”] = “West” ;]
    ➢ printf(“N is %s and W is %s \n”, direction[“N”], direction [“W”]);
    ➢
    ➢ Mon[1] = “Jan”; mon[“1”] = “january” ; mon[“01”] = “JAN” ;
    ➢ Printf(“mon 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”]);
    }
```

There are two important things to be learned from this output. First, the setting with index “1” overwrites the setng made with index 1. accessing an array element with subscript 1 and 01 actually locates the element with subscript “1”. Also note that mon[“1”] is different from mon[“01”].

ENVIRON[]: The Environment Array:

You may sometimes need to know the name of the user running the program or the home directing awk maintains the associative array, ENVIRON[], to store all environment variables. This POSIX requirement is met by recent version of awk including *nawk* and *gawk*. This is how we access the shell variable , HOME and PATH, from inside an awk program:

```
$nawk ‘BEGIN {
>print “HOME” “=” ENVIRON[“HOME”]
>print “PATH” “=” ENVIRON[“PATH”]
>}’
```

FUNCTIONS

Awk has several built in functions, performing both arithmetic and string operations. The arguments are passed to a function in C-style, delimited by commas and enclosed by a matched pair of parentheses. Even though awk allows use of functions with and without parentheses (like printf and printf()), POSIX discourages use of functions without parentheses.

Some of these functions take a variable number of arguments, and one (length) uses no arguments as a variant form. The functions are adequately explained here so u can confidently use them in perl which often uses identical syntaxes.

There are two arithmetic functions which a programmer will except **awk** to offer. **int** calculates the integral portion of a number (without rounding off),while sqrt calculates square root of a number. **awk** also has some of the common string handling function you can hope to find in any language. There are:

length: it determines the length of its arguments, and if no argument is present, the enire line is assumed to be the argument. You can use length (without any argument) to locate lines whose length exceeds 1024 characters:

```
awk –F”|” ‘length  > 1024’ empn.lst
```

you can use length with a field as well. The following program selects those people who have short names:

```
awk –F”|” ‘length ($2) < 11’ empn.lst
```

index(*s1, s2*): it determines the position of a string s2within a larger string s1. This function is especially useful in validating single character fields. If a field takes the values a, b, c, d or e you can use this function n to find out whether this single character field can be located within a string abcde:

```
x = index (“abcde”, “b”)
```

This returns the value 2.

substr (stg, m, n): it extracts a substring from a string *stg. m* represents the starting point of extraction and n indicates the number of characters to be extracted. Because string values can also be used for computation, the returned string from this function can be used to select those born between 1946 and 1951:

```
awk –F”|” ‘substr($5, 7, 2) > 45 && substr($5, 7, 2) < 52’ empn.lst
2365|barun sengupta|director|personel|11/05/47|7800|2365
3564|sudhir ararwal|executive|personnel|06/07/47|7500|2365
4290|jaynth Choudhury|executive|production|07/09/50|6000|9876
9876|jai sharma|director|production|12/03/50|7000|9876
```

you can never get this output with either **sed** *and **grep*** because regular expressions can never match the numbers between 46 and 51. Note that awk does indeed posses a mechanism of identifying the type of expression from its context. It identified the date field string for using substr and then converted it to a number for making a numeric comparison.

split(stg, arr, ch): it breaks up a string stg on the delimiter ch and stores the fields in an array arr[]. Here's how yo can convert the date field to the format YYYYMMDD:

```
$awk –F "|" '{split($5, ar, "/"); print "19"ar[3]ar[2]ar[1]}' empn.lst
19521212
19501203
19431904
………..
```

You can also do it with **sed,** but this method is superior because it explicitly picks up the fifth field, whereas **sed** would transorm the only date field that it finds.

system: you may want to print the system date at the beging of the report. For running a UNIX command within a awk, you'll have to use the system function. Here are two examples:

```
BEGIN {
        system("tput clear")            Clears the screen
        system("date")                 Executes the UNIX date command
        }
```

CONTROL FLOW- THE if STATEMENT:

Awk has practically all the features of a modern programming language. It has conditional structures (the if statement) and loops (while or for). They all execute a body of statements depending on the success or failure of the *control command.* This is simply a condition that is specified in the first line of the construct.

Function	***Description***
int(x)	returns the integer value of x
sqrt(x)	returns the square root of x
length	returns the complete length of 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(1s, s2)	returns position of string s2 in string s1
splicit(stg, arr, ch)	splicit string stg into array *arr* using ch as delimiter, returns number of fields.
System("cmd")	runs UNIX command cmd and returns its exit status

The **if** statement can be used when the && and || are found to be inadequate for certain tasks. Its behavior is well known to all programmers. The statement here takes the form:

```
If  (condition is true) {
        Statement
} else {
Statement
```

```
        }
```

Like in C, none of the control flow constructs need to use curly braces if there's only one *statement* to be executed. But when there are multiple actions take, the statement must be enclosed within a pair of curly braces. Moreover, the control command must be enclosed in parentheses.

Most of the addresses that have been used so far reflect the logic normally used in the **if** statement. In a previous example, you have selected lines where the basic pay exceeded 7500, by using the condition as the selection criteria:

```
$6 > 7500 {
```

An alternative form of this logic places the condition inside the action component rather than the selection criteria. But this form requires the if statement:

```
Awk –F "|" '{ if ($6 > 7500) printf  ……….
```

if can be used with the comparison operators and the special symbols ~ and !~ to match a regular expression. When used in combination with the logical operators || and &&, awk programming becomes quite easy and powerful. Some of the earlier pattern matching expressions are rephrased in the following, this time in the form used by **if:**

```
if ( NR > = 3 && NR <= 6 )
if ( $3 == "director" || $3 ==  "chairman" )
if ( $3 ~ /^g.m/ )
if ( $2 !~ / [aA]gg?[ar]+wal/ )
if ( $2 ~[cC]ho[wu]dh?ury|sa[xk]s?ena/ )
```

To illustrate the use of the optional **else** statement, let's assume that the dearness allowance is 25% of basic pay when the latter is less than 600, and 1000 otherwise. The **if-else** structure that implants this logic looks like this:

```
If ( $6 < 6000 )
        da = 0.25*$6
else
        da = 1000
```

You can even replace the above **if** construct with a compact conditional structure:

```
$6 <  6000 ? da = 0.25*$6 : da = 1000
```

This is the form that C and **perl** use to implement the logic of simple **if-else** construct. The ? and : act as separator of the two actions.

When you have more than one statement to be executed, they must be bounded by a pair of curly braces (as in C). For example, if the factors determining the hra and da are in turn dependent on the basic pay itself, then you need to use terminators:

```
If ( $6 < 6000 ) {
        hra = 0.50*$6
        da = 0.25*$6
}else {
        hra = 0.40*$6
        da = 1000
}
```

<u>LOOPING WITH for:</u>

a**wk** supports two loops – **for** and **while.** They both execute the loop body as long as the control command returns a true value. **For** has two forms. The easier one resembles its C counterpart. A simple example illustrates the first form:

```
for (k=0; k<=9; k+=2)
```

This form also consists of three components; the first component initializes the value of k, the second checks the condition with every iteration, while the third sets the increment used for every iteration. **for** is useful for centering text, and the following examples uses **awk** with **echo** in a pipeline to do that:

```
$echo "
>Income statement\nfor\nthe month of august, 2002\nDepartment : Sales" |
>awk ' { for (k=1 ; k < (55 –length($0)) /2 ; k++)
>printf "%s"," "
>printf $0}'

Income statement
        for
the month of August, 2002
   Department : Sales
```

The loop here uses the first **printf** statement to print the required number of spaces (page width assumed to be 55). The line is then printed with the second **printf** statement, which falls outside the loop. This is useful routine which can be used to center some titles that normally appear at the beginning of a report.

<u>Using for with an Associative Array:</u>

The second form of the **for** loop exploits the associative feature of **awk**'s arrays. This form is also seen in perl but not in the commonly used languages like C and java. The loop selects each indexof an array:

```
for ( k in array )
        commamds
```

Here, k is the subscript of the array *arr.* Because k can also be a string, we can use this loop to print all environment variables. We simply have to pick up each subscript of the ENVIRON array:

```
$ nawk ‘BIGIN {
>for ( key in ENVIRON )
>print key “=” ENVIRON [key]
>}’

LOGNAME=praveen
MAIL=/var/mail/Praveen
PATH=/usr/bin::/usr/local/bin::/usr/ccs/bin
TERM=xterm
HOME=/home/praveen
SHELL=/bin/bash
```

Because the index is actually a string, we can use any field as index. We can even use elements of the array counters. Using our sample databases, we can display the count of the employees, grouped according to the disgnation (the third field). You can use the string value of $3 as the subscript of the array kount[]:

```
$awk –F’|’ ‘{ kount[$3]++ }
>END { for ( desig in kount)
>print desig, kount[desig] }’ empn.lst

g.m          4
chairman     1
executive    2
director     4
manager      2
d.g.m        2
```

The program here analyzes the databases to break up of the employees, grouped on their designation. The array kount[] takes as its subscript non-numeric values g.m., chairman, executive, etc.. **for** is invoked in the END section to print the subscript (desig) and the number of occurrence of the subscript (kount[desig]). Note that you don’t need to sort the input file to print the report!

LOOPING WITH while

The **while** loop has a similar role to play; it repeatedly iterates the loop until the command succeeds. For example, the previous **for** loop used for centering text can be easily replaced with a **while** construct:

```
        k=0
        while (k < (55 – length($0))/2) {
                printf “%s”,“ ”
                k++
        }
        print $0
```

The loop here prints a space and increments the value of k with every iteration. The condition (k < (55 – length($0))/2) is tested at the beginning of every iteration, and the loop body only if the test succeeds. In this way, entire line is filled with a string spacesbefore the text is printed with **print $0.**

Not that the **legth** function has been used with an argument ($0). This **awk** understands to be the entire line. Since length, in the absence of arguments, uses the entire line anyway, $0 can be omitted. Similarly, **print $0** may also be replaced by simply **print.**

Programs

1)awk script to delete duplicate

lines in a file.

```
BEGIN { i=1;}
{
 flag=1;
   for(j=1; j<i && flag ; j++ )
   {
        if( x[j] == $0 )
        flag=0;
   }
if(flag)
   {
        x[i]=$0;
        printf "%s \n",x[i];
        i++;
   }
}
```

<u>Run1:</u>

```
[root@localhost shellprgms]$ cat >for7.txt
hello
world
world
hello
this
is
this
```

<u>Output:</u>

```
[root@localhost shellprgms]$ awk -F "|" -f 11.awk for7.txt
hello
world
this
is
```

2)awk script to print the transpose of a matrix.

```
 BEGIN{
            system("tput clear")
            count =0
                }
{
                           split($0,a);
   for(j=1;j<=NF;j++)

{  count = count+1
    arr[count] =a[j]
 }
     K=NF
}
END{
printf("Transpose\n");
for(j=1;j<=K;j++)
{
   for(i=j; i<=count; i=i+K)
{
                     printf("%s\t", arr[i]);
                 }
                   printf("\n");
        }
 }
```

Run1:

```
 [root@localhost shellprgms]$ qwk –f p8.awk
        2        3
        5        6

Transpose
        2        5
        3        6
```

3)Awk script that folds long line into 40 columns. Thus any line that exceeds 40 Characters must be broken after 40th and is to be continued with the residue. The inputs to be supplied through a text file created by the user.

```
BEGIN{
start=1; }
{ len=length;
for(i=$0; length(i)>40; len-=40)
{
print substr(i,1,40) "\\"
i=substr(i,41,len);
}
print i; }
```

Run1:

```
[root@localhost shellprgms]$ awk -F "|" -f 15.awk sample.txt
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\
aaaaaaaaaaaa
aaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\
aaaaaaaaa
```

<u>Output:</u>

```
[root@localhost shellprgms]$ cat sample.txt
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaa
aaaaaaaaaaaa
aaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

4)This is an awk program to provide extra spaces at the end of the line so that the line length is maintained as 127.

```
awk ‘ { y=127 – length($0)
        printf “%s”, $0
        if(y > 0)
        for(i=0;i<y;i++)
           printf “%s”, “ ”
        printf “\n”
       }’ foo
```

5)A file contains a fixed number of fields in the form of space-delimited numbers. This is an awk program to print the lines as well as total of its rows.

```
awk ‘{ split($0,a)
       for (i=1;i<=NF;i++) {
        row[NR]+=a[$i]
        }
```

```
    printf “%s”, $0
    printf “%d\n”, row[NR]
  } ’ foo
```

Essential Shell Programming

Definition:

Shell is an agency that sits between the user and the UNIX system.

Description:

Shell is the one which understands all user directives and carries them out. It processes the commands issued by the user. The content is based on a type of shell called Bourne shell.

Shell Scripts

When groups of command have to be executed regularly, they should be stored in a file, and the file itself executed as a shell script or a shell program by the user. A shell program runs in interpretive mode. It is not complied with a separate executable file as with a C program but each statement is loaded into memory when it is to be executed. Hence shell scripts run slower than the programs written in high-level language. .sh is used as an extension for shell scripts. However the use of extension is not mandatory.

Shell scripts are executed in a separate child shell process which may or may not be same as the login shell.

Example: script.sh

```
#! /bin/sh
# script.sh: Sample Shell Script
echo “Welcome to Shell Programming”
echo “Today’s date : `date`”
echo “This months calendar:”
cal `date “+%m 20%y”`                  #This month’s calendar.
echo “My Shell :$ SHELL”
```

The # character indicates the comments in the shell script and all the characters that follow the # symbol are ignored by the shell. However, this does not apply to the first line which beings with #. This because, it is an interpreter line which always begins with #! followed by the pathname of the shell to be used for running the script. In the above example the first line indicates that we are using a Bourne Shell.

To run the script we need to first make it executable. This is achieved by using the chmod command as shown below:

```
$ chmod +x script.sh
```

Then invoke the script name as:

```
$ script.sh
```

Once this is done, we can see the following output :

```
Welcome to Shell Programming
Today's date: Mon Oct 8 08:02:45 IST 2007
This month's calendar:
            October 2007
Su   Mo   Tu   We   Th   Fr   Sa
      1    2    3    4    5    6
 7    8    9   10   11   12   13
14   15   16   17   18   19   20
21   22   23   24   25   26   27
28   29   30   31
My Shell: /bin/Sh
```

As stated above the child shell reads and executes each statement in interpretive mode. We can also explicitly spawn a child of your choice with the script name as argument:

```
sh script.sh
```

Note: Here the script neither requires a executable permission nor an interpreter line.

Read: Making scripts interactive

The read statement is the shell's internal tool for making scripts interactive (i.e. taking input from the user). It is used with one or more variables. Inputs supplied with the standard input are read into these variables. For instance, the use of statement like

read name

causes the script to pause at that point to take input from the keyboard. Whatever is entered by you will be stored in the variable *name*.

Example: A shell script that uses read to take a search string and filename from the terminal.

```
#! /bin/sh
# emp1.sh: Interactive version, uses read to accept two inputs
#
echo "Enter the pattern to be searched: \c"                  # No newline
read pname
echo "Enter the file to be used: \c"                         # use echo
–e in bash
read fname
echo "Searching for pattern $pname from the file $fname"
grep $pname $fname
echo "Selected records shown above"
```

Running of the above script by specifying the inputs when the script pauses twice:

```
$ emp1.sh
Enter the pattern to be searched : director
Enter the file to be used: emp.lst
Searching for pattern director from the file emp.lst
9876   Jai        Director   Productions
       Sharma
2356   Rohit      Director   Sales
Selected records shown above
```

Using Command Line Arguments

Shell scripts also accept arguments from the command line. Therefore e they can be run non interactively and be used with redirection and pipelines. The arguments are assigned to special shell variables. Represented by $1, $2, etc; similar to C command arguments argv[0], argv[1], etc. The following table lists the different shell parameters.

Shell parameter	Significance
$1, $2…	Positional parameters representing command line arguments
$ #	No. of arguments specified in command line
$ 0	Name of the executed command
$ *	Complete set of positional parameters as a single string
"$ @"	Each quoted string treated as separate argument
$?	Exit status of last command
$$	Pid of the current shell
$!	PID of the last background job.

Table: shell parameters

exit and Exit Status of Command

To terminate a program exit is used. Nonzero value indicates an error condition.

Example 1:

```
$ cat foo

Cat: can't open foo
```

Returns nonzero exit status. The shell variable $? Stores this status.

Example 2:

```
grep director emp.lst > /dev/null:echo $?
0
```

Exit status is used to devise program logic that braches into different paths depending on success or failure of a command

The logical Operators && and ||

The shell provides two operators that aloe conditional execution, the && and ||.

Usage:

```
cmd1 && cmd2
cmd1 || cmd2
```

&& delimits two commands. cmd 2 executed only when cmd1 succeeds.

Example1:

```
$ grep  'director' emp.lst && echo "Pattern  found"
```

Output:

```
9876   Jai           Director   Productions
       Sharma
2356   Rohit         Director   Sales
Pattern  found
```

Example 2:

```
$ grep  'clerk' emp.lst || echo "Pattern not found"
```

Output:

```
Pattern not found
```

Example 3:

```
grep "$1" $2 || exit 2
echo "Pattern Found Job Over"
```

The if Conditional

The if statement makes two way decisions based on the result of a condition. The following forms of if are available in the shell:

Form 1	**Form 2**	**Form 3**
if *command is successful*	if *command is successful*	if *command is successful*
then	then	then
execute commands	*execute commands*	*execute commands*
fi	else	elif *command is successful*
	execute commands	then...
	fi	else...
		fi

If the command succeeds, the statements within if are executed or else statements in else block are executed (if else present).

Example:

```
#! /bin/sh
if grep “^$1” /etc/passwd 2>/dev/null
then
        echo “Pattern Found”
else
        echo “Pattern Not Found”
fi
```

Output1:

```
$ emp3.sh ftp
```

```
ftp: *.325:15:FTP User:/Users1/home/ftp:/bin/true
Pattern Found
```

Output2:

```
$ emp3.sh mail
Pattern Not Found
```

While: Looping

To carry out a set of instruction repeatedly shell offers three features namely while, until and for.

Syntax:

```
while condition is true
do
        Commands
done
```

The commands enclosed by do and done are executed repeatedly as long as condition is true.

Example:

```
  #! /bin/usr
 ans=y
while [“$ans”=”y”]
do
        echo “Enter the code and description : \c” > /dev/tty
        read code description
        echo “$code $description” >>newlist
        echo “Enter any more [Y/N]”
        read any
        case $any in
        Y* | y* ) answer =y;;
        N* | n*) answer = n;;
              *) answer=y;;
        esac
done
```

Input:

```
Enter the code and description : 03 analgestics
Enter any more [Y/N] :y
Enter the code and description : 04 antibiotics
Enter any more [Y/N] : [Enter]
Enter the code and description : 05 OTC drugs
Enter any more [Y/N] : n
```

Output:

```
$ cat newlist
03 | analgestics
04 | antibiotics
05 | OTC drugs
```

Using test and [ ] to Evaluate Expressions

Test statement is used to handle the true or false value returned by expressions, and it is not possible with if statement. Test uses certain operators to evaluate the condition on its right and returns either a true or false exit status, which is then used by if for making decisions. Test works in three ways:

- Compare two numbers
- Compares two strings or a single one for a null value
- Checks files attributes

Test doesn't display any output but simply returns a value that sets the parameters $?

Numeric Comparison

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to

-lt	Less than
-le	Less than or equal

Table: Operators

Operators always begin with a – (Hyphen) followed by a two word character word and enclosed on either side by whitespace.

Numeric comparison in the shell is confined to integer values only, decimal values are simply truncated.

Ex:
$x=5;y=7;z=7.2

```
1. $test $x –eq $y; echo $?
    1                               Not equal

2. $test $x –lt $y; echo $?
    0                               True

3. $test $z –gt $y; echo $?
    1                               7.2 is not greater than 7
    2
4. $test  $z –eq $y ; echo $y
    0                               7.2 is equal to 7
    1
```

Example 3 and 4 shows that test uses only integer comparison.

The script emp.sh uses test in an if-elif-else-fi construct (Form 3) to evaluate the shell parameter $#

```
#!/bin/sh
#emp.sh: using test, $0 and $# in an if-elif-else-fi construct
#
If test $# -eq 0; then
Echo “Usage : $0 pattern file” > /dev/tty
Elfi test $# -eq 2 ;then
Grep “$1” $2 || echo “$1 not found in $2”>/dev/tty
Else
echo “You didn’t enter two arguments” >/dev/tty
fi
```

It displays the usage when no arguments are input, runs grep if two arguments are entered and displays an error message otherwise.

Run the script four times and redirect the output every time

```
$emp31.sh>foo
Usage : emp.sh pattern file
```

```
$emp31.sh ftp>foo
You didn't enter two arguments
$emp31.sh henry  /etc/passwd>foo
Henry not found in /etc/passwd
$emp31.sh ftp /etc/passwd>foo
ftp:*:325:15:FTP User:/user1/home/ftp:/bin/true
```

Shorthand for test

[and] can be used instead of test. The following two forms are equivalent

```
Test $x –eq $y

        and

[ $x –eq $y ]
```

String Comparison

Test command is also used for testing strings. Test can be used to compare strings with the following set of comparison operators as listed below.

Test	True if
s1=s2	String s1=s2
s1!=s2	String s1 is not equal to s2
-n stg	String stg is not a null string
-z stg	String stg is a null string
stg	String stg is assigned and not null
s1= =s2	String s1=s2

Table: String test used by test

Example:

```
#!/bin/sh
#emp1.sh checks user input for null values finally turns emp.sh developed
previously
#

if [ $# -eq 0 ] ; then
echo "Enter the string to be searched :\c"
read pname
if [ -z "$pname" ] ; then
```

```
echo “You have not entered th e string”; exit 1
fi
echo “Enter the filename to be used :\c”
read flname
if [ ! –n “$flname” ] ; then
echo “ You have not entered the flname” ; exit 2
fi
emp.sh “$pname” “$flname”
else
emp.sh $*
fi
```

Output1:

```
$emp1.sh
Enter the string to be searched :[Enter]
You have not entered the string
```

Output2:

```
$emp1.sh
Enter the string to be  searched :root
Enter the filename to be searched :/etc/passwd
Root:x:0:1:Super-user:/:/usr/bin/bash
```

When we run the script with arguments emp1.sh bypasses all the above activities and calls emp.sh to perform all validation checks

```
$emp1.sh jai
You didn’t enter two arguments
```

```
$emp1.sh jai emp,lst
9878|jai sharma|director|sales|12/03/56|70000
```

```
$emp1.sh “jai sharma” emp.lst
You didn’t enter two arguments
```

Because $* treats jai and sharma are separate arguments. And $# makes a wrong argument count. Solution is replace $* with “$@” (with quote” and then run the script.

File Tests

Test can be used to test various file attributes like its type (file, directory or symbolic links) or its permission (read, write. Execute, SUID, etc).

Example:

```
$ ls –l emp.lst
-rw-rw-rw-    1 kumar group       870 jun 8 15:52 emp.lst
$ [-f emp.lst] ; echo $?                          → Ordinary file
0
$ [-x emp.lst] ; echo $?                          → Not an executable.
1
$ [! -w emp.lst]  || echo "False that file not writeable"
False that file is not writable.
```

Example: filetest.sh

```
#! /bin/usr
#
if [! –e $1] : then
        Echo "File doesnot exist"
elif [! –r S1]; then
        Echo "File not readable"
elif[! –w $1]; then
        Echo "File not writable"
else
        Echo "File is both readable and writable"\
fi
```

Output:

```
$ filetest.sh emp3.lst
```

```
File does not exist
$ filetest.sh emp.lst
File is both readable and writable
```

The following table depicts file-related Tests with test:

Test	**True if**
-f file	File exists and is a regular file
-r file	File exists and readable
-w file	File exists and is writable
-x file	File exists and is executable
-d file	File exists and is a directory
-s file	File exists and has a size greater than zero
-e file	File exists (Korn & Bash Only)
-u file	File exists and has SUID bit set
-k file	File exists and has sticky bit set
-L file	File exists and is a symbolic link (Korn & Bash Only)
f1 –nt f2	File f1 is newer than f2 (Korn & Bash Only)
f1 –ot f2	File f1 is older than f2 (Korn & Bash Only)
f1 –ef f2	File f1 is linked to f2 (Korn & Bash Only)

Table: file-related Tests with test

The case Conditional

The case statement is the second conditional offered by the shell. It doesn't have a parallel either in C (Switch is similar) or perl. The statement matches an expression for more than one alternative, and uses a compact construct to permit multiway branching. case also handles string tests, but in a more efficient manner than if.

Syntax:

```
case expression in
        Pattern1) commands1 ;;
        Pattern2) commands2 ;;
        Pattern3) commands3 ;;
        …
Esac
```

Case first matches expression with pattern1. if the match succeeds, then it executes commands1, which may be one or more commands. If the match fails, then pattern2 is matched and so forth. Each command list is terminated with a pair of semicolon and the entire construct is closed with esac (reverse of case).

Example:

```
#! /bin/sh
#
echo "          Menu\n
1. List of files\n2. Processes of user\n3. Today's Date
4. Users of system\n5.Quit\nEnter your option: \c"
read choice
case "$choice" in
            1)  ls –l;;
            2)  ps –f ;;
            3)  date ;;
            4)  who ;;
            5)  exit ;;
            *) echo "Invalid option"
esac
```

Output

```
$ menu.sh

        Menu

1. List of files
2. Processes of user
3. Today's Date
4. Users of system
5. Quit
Enter your option: 3
Mon Oct 8 08:02:45 IST 2007
```

Note:

- case can not handle relational and file test, but it matches strings with compact code. It is very effective when the string is fetched by command substitution.
- case can also handle numbers but treats them as strings.

Matching Multiple Patterns:

case can also specify the same action for more than one pattern . For instance to test a user response for both y and Y (or n and N).

Example:

```
Echo “Do you wish to continue? [y/n]: \c”
Read ans
Case “$ans” in
        Y | y );;
        N | n) exit ;;
esac
```

Wild-Cards: case uses them:

case has a superb string matching feature that uses wild-cards. It uses the filename matching metacharacters *, ? and character class (to match only strings and not files in the current directory).

Example:

```
Case “$ans” in
        [Yy] [eE]* );;                  Matches YES, yes, Yes, yEs, etc
        [Nn] [oO]) exit ;;              Matches no, NO, No, nO
                *)  echo “Invalid Response”
esac
```

expr: Computation and String Handling

The Broune shell uses expr command to perform computations. This command combines the following two functions:

- Performs arithmetic operations on integers
- Manipulates strings

Computation:

expr can perform the four basic arithmetic operations (+, -, *, /), as well as modulus (%) functions.

Examples:

```
$ x=3 y=5

$ expr  3+5
8

$ expr  $x-$y
-2

$ expr  3 \* 5
```

*Note:\ is used to prevent the shell from interpreting * as metacharacter*

```
15

$ expr  $y/$x
1

$ expr  13%5
3
```

expr is also used with command substitution to assign a variable.

Example1:

```
$ x=6 y=2 : z=`expr $x+$y`
$ echo $z
8
```

Example2:

```
$ x=5
$ x=`expr $x+1`
$ echo $x
```

6

String Handling:

expr is also used to handle strings. For manipulating strings, expr uses two expressions separated by a colon (:). The string to be worked upon is closed on the left of the colon and a regular expression is placed on its right. Depending on the composition of the expression expr can perform the following three functions:

1. Determine the length of the string.
2. Extract the substring.
3. Locate the position of a character in a string.

1. **Length of the string:**

   The regular expression .* is used to print the number of characters matching the pattern .

   Example1:

```
$ expr "abcdefg" : '.*'
7
```

   Example2:

```
while echo "Enter your name: \c" ;do
        read name
        if [`expe "$name" :'.*'`   -gt  20] ; then
                echo "Name is very long"
        else
                break
        fi
done
```

2. **Extracting a substring:**

   expr can extract a string enclosed by the escape characters \ (and \).

   Example:

```
$ st=2007
$ expr "$st" :'..\(..\)'
07                                  Extracts last two characters.
```

3. **Locating position of a character:**

   expr can return the location of the first occurrence of a character inside a string.

   Example:

   ```
   $ stg = abcdefgh ; expr “$stg” : ‘[^d]*d’
   4                                    Extracts the position of character d
   ```

$0: Calling a Script by Different Names

There are a number of UNIX commands that can be used to call a file by different names and doing different things depending on the name by which it is called. $0 can also be to call a script by different names.

Example:

```
#! /bin/sh
#
lastfile=`ls –t *.c |head -1`
command=$0
exe=`expr $lastfile: ‘\(.*\).c’`
case $command in
        *runc) $exe ;;
        *vic) vi $lastfile;;
        *comc) cc –o $exe $lastfile &&
                Echo “$lastfile compiled successfully”;;
esac
```

After this create the following three links:

```
ln comc.sh comc
ln comc.sh runc
ln comc.sh vic
```

Output:

```
$ comc
hello.c compiled successfully.
```

While: Looping

To carry out a set of instruction repeatedly shell offers three features namely while, until and for.

Synatx:

```
while condition is true
do
        Commands
done
```

The commands enclosed by do and done are executed repadetedly as long as condition is true.

Example:

```
  #! /bin/usr
 ans=y
while [“$ans”=”y”]
do
        echo “Enter the code and description : \c” > /dev/tty
        read code description
        echo “$code $description” >>newlist
        echo “Enter any more [Y/N]”
        read any
        case $any in
        Y* | y* ) answer =y;;
        N* | n*) answer = n;;
              *) answer=y;;
        esac
done
```

```
Input:
Enter the code and description : 03 analgestics
Enter any more [Y/N] :y
Enter the code and description : 04 antibiotics
Enter any more [Y/N] : [Enter]
Enter the code and description : 05 OTC drugs
Enter any more [Y/N] : n
```

```
Output:
$ cat newlist
03 | analgestics
04 | antibiotics
05 | OTC drugs
```

Other Examples: An infinite/semi-infinite loop

```
      (1)                                  (2)
while true ; do                    while [ ! -r $1 ] ; do
   [ -r $1 ] && break                 sleep $2
   sleep $2                         done
done
```

for: Looping with a List

for is also a repetitive structure.

Synatx:

```
for variable in list
do
        Commands
done
```

list here comprises a series of character strings. Each string is assigned to variable specified.

Example:

```
for file in ch1 ch2; do
> cp $file ${file}.bak
> echo $file copied to $file.bak
done
```

Output:

```
ch1 copied to ch1.bak
ch2 copied to ch2.bak
```

Sources of list:

- **List from variables**: Series of variables are evaluated by the shell before executing the loop

  Example:

  ```
  $ for var in $PATH $HOME; do echo “$var” ; done
  ```

  Output:
  ```
  /bin:/usr/bin;/home/local/bin;
  /home/user1
  ```

- **List from command substitution**: Command substitution is used for creating a list. This is used when list is large.

  Example:

  ```
  $ for var in `cat clist`
  ```

- **List from wildcards**: Here the shell interprets the wildcards as filenames.

  Example:

  ```
  for file in *.htm *.html ; do
          sed ‘s/strong/STRONG/g
          s/img src/IMG SRC/g’ $file > $$
          mv $$ $file
  done
  ```

- **List from positional parameters**:

  Example: emp.sh

```
#! /bin/sh
for pattern in “$@”; do
grep “$pattern” emp.lst || echo “Pattern $pattern not found”
done
```

  Output:

```
$emp.sh 9876 “Rohit”
```

9876	Jai Sharma	Director	Productions
2356	Rohit	Director	Sales

basename: Changing Filename Extensions

They are useful in chaining the extension of group of files. Basename extracts the base filename from an absolute pathname.

Example1:

```
$basename /home/user1/test.pl
```

Ouput:

```
test.pl
```

Example2:

```
$basename test2.doc doc
```

Ouput:

```
test2
```

Example3: Renaming filename extension from .txt to .doc

```
for file in *.txt ; do
    leftname=`basename $file .txt` Stores left part of filename
     mv $file ${leftname}.doc
done
```

set and shift: Manipulating the Positional Parameters

The set statement assigns positional parameters $1, $2 and so on, to its arguments. This is used for picking up individual fields from the output of a program.

Example 1:

```
$ set 9876 2345 6213
$
```

This assigns the value 9876 to the positional parameters $1, 2345 to $2 and 6213 to $3. It also sets the other parameters $# and $*.

Example 2:

```
$ set `date`
$ echo $*
Mon Oct 8 08:02:45 IST 2007
```

Example 3:

```
$ echo “The date today is $2 $3, $6”
The date today is Oct 8, 2007
```

Shift: Shifting Arguments Left

Shift transfers the contents of positional parameters to its immediate lower numbered one. This is done as many times as the statement is called. When called once, $2 becomes $1, $3 becomes S2 and so on.

Example 1:

```
$ echo “$@”                              $@ and $* are interchangeable
Mon Oct 8 08:02:45 IST 2007
$ echo $1 $2 $3
```

```
Mon Oct 8
$shift
$echo $1 $2 $3
Mon Oct 8 08:02:45
$shift 2                                        Shifts 2 places
$echo $1 $2 $3
08:02:45 IST 2007
```

Example 2: emp.sh

```
#! /bin/sh
Case $#  in
        0|1) echo "Usage: $0 file pattern(S)" ;exit ;;
          *) fname=$1
           shift
           for pattern in "$@" ; do
                   grep "$pattern" $fname || echo "Pattern $pattern not found"
        done;;
esac
```

Output:

```
$emp.sh emp.lst
Insufficient number of arguments
$emp.sh emp.lst Rakesh  1006 9877

9876   Jai        Director   Productions
       Sharma
2356   Rohit      Director   Sales
Pattern 9877 not found.
```

Set – : Helps Command Substitution

Inorder for the set to interpret - and null output produced by UNIX commands the – option is used . If not used – in the output is treated as an option and set will interpret it wrongly. In case of null, all variables are displayed instead of null.

Example:

```
$set `ls –l chp1`
```

Output:

```
-rwxr-xr-x: bad options
```

Example2:

```
$set `grep usr1 /etc/passwd`
```

Correction to be made to get correct output are:

```
$set -- `ls –l chp1`
$set -- `grep usr1 /etc/passwd`
```

<u>The Here Document (<<)</u>

The shell uses the << symbol to read data from the same file containing the script. This is referred to as a here document, signifying that the data is here rather than in aspirate file. Any command using standard input can slo take input from a here document.

Example:

```
mailx kumar << MARK
Your program for printing the invoices has been executed
on `date`.Check the print queue
The updated file is $flname
MARK
```

The string (MARK) is delimiter. The shell treats every line following the command and delimited by MARK as input to the command. Kumar at the other end will see three lines of message text with the date inserted by command. The word MARK itself doesn’t show up.

Using Here Document with Interactive Programs:

A shell script can be made to work non-interactively by supplying inputs through here document.

Example:

```
$ search.sh << END
> director
>emp.lst
>END
```

Output:

```
Enter the pattern to be searched: Enter the file to be used: Searching for director
from file emp.lst


9876   Jai          Director   Productions
       Sharma
2356   Rohit        Director   Sales
Selected records shown above.
```

The script search.sh will run non-interactively and display the lines containing "director" in the file emp.lst.

trap: interrupting a Program

Normally, the shell scripts terminate whenever the interrupt key is pressed. It is not a good programming practice because a lot of temporary files will be stored on disk. The trap statement lets you do the things you want to do when a script receives a signal. The trap statement is normally placed at the beginning of the shell script and uses two lists:

```
trap 'command_list' signal_list
```

When a script is sent any of the signals in signal_list, trap executes the commands in command_list. The signal list can contain the integer values or names (without SIG prefix) of one or more signals – the ones used with the kill command.

Example: To remove all temporary files named after the PID number of the shell:

```
trap 'rm $$* ; echo "Program Interrupted" ; exit' HUP  INT  TERM
```

trap is a signal handler. It first removes all files expanded from $$*, echoes a message and finally terminates the script when signals SIGHUP (1), SIGINT (2) or SIGTERM(15) are sent to the shell process running the script.

A script can also be made to ignore the signals by using a null command list.

Example:

```
trap ‘’ 1  2  15
```

Programs

1)

```
#!/bin/sh
IFS="|"
While echo "enter dept code:\c"; do
Read dcode
Set -- `grep "^$dcode"<<limit
01|ISE|22
02|CSE|45
03|ECE|25
04|TCE|58
limit`
Case $# in
3) echo "dept name :$2 \n  emp-id:$3\n"
*) echo "invalid code";continue
esac
done
```

Output:

```
$valcode.sh
Enter dept code:88
Invalid code
Enter dept code:02
Dept name  : CSE
Emp-id :45
Enter dept code:<ctrl-c>
```

2)
```
#!/bin/sh
x=1
While [$x –le 10];do
 echo “$x”
 x=`expr $x+1`
done
#!/bin/sh
sum=0
for I in “$@” do
 echo “$I”
sum=`expr $sum + $I`
done
Echo “sum is $sum”
```

3)
```
#!/bin/sh
sum=0
for I in `cat list`; do
 echo “string is $I”
x= `expr “$I”:’.*’`
Echo “length is $x”
Done
```

4)
This is a non-recursive shell script that accepts any number of arguments and prints them in a reverse order.

For example if A B C are entered then output is C B A.

```
#!/bin/sh
if [ $# -lt 2 ]; then
echo "please enter 2 or more arguments"
exit
fi
for x in $@
do
y=$x" "$y
done
echo "$y"
```

<u>Run1:</u>
```
[root@localhost shellprgms]# sh sh1a.sh 1 2 3 4 5 6 7
```

```
7 6 5 4 3 2 1
```

Run2:
```
[root@localhost shellprgms]# sh ps1a.sh this is an argument

argument an is this
```

5)

The following shell script to accept 2 file names checks if the permission for these files are identical and if they are not identical outputs each filename followed by permission.

```
#!/bin/sh
if [ $# -lt 2 ]
then
echo "invalid number of arguments"
exit
fi
str1=`ls -l $1|cut -c 2-10`
str2=`ls -l $2|cut -c 2-10`

if [ "$str1" = "$str2" ]
then
echo "the file permissions are the same: $str1"
else
echo " Different file permissions "
echo -e "file permission for $1 is $str1\nfile permission for $2 is  $str2"
fi
```

Run1:

```
[root@localhost shellprgms]# sh 2a.sh ab.c xy.c
file permission for ab.c is rw-r--r--
file permission for xy.c is  rwxr-xr-x
```

Run2:

```
[root@localhost shellprgms]# chmod +x ab.c
[root@localhost shellprgms]# sh 2a.sh ab.c xy.c
the file permissions are the same: rwxr-xr-x
```

6)This shell function that takes a valid directory name as an argument and recursively descends all the subdirectories, finds the maximum length of any file in that hierarchy and writes this maximum value to the standard output.

```
#!/bin/sh
if [ $# -gt 2 ]
then
echo "usage sh flname dir"
exit
fi
if [ -d $1 ]
then
ls -lR $1|grep -v ^d|cut -c 34-43,56-69|sort -n|tail -1>fn1
echo "file name is `cut -c 10- fn1`"
echo " the size is `cut -c -9 fn1`"
else
echo "invalid dir name"
fi
```

Run1:

```
[root@localhost shellprgms]# sh 3a.sh
file name is   a.out
the size is     12172
```

7)This shell script that accepts valid log-in names as arguments and prints their corresponding home directories. If no arguments are specified, print a suitable error message.

```
if [ $# -lt 1 ]
then
echo " Invlaid Arguments....... "
```

```
exit
fi
for x in "$@"
do
  grep -w "^$x" /etc/passwd | cut -d ":" -f 1,6
done
```

Run1:

```
[root@localhost shellprgms]# sh 4a.sh root
root:/root
```

Run2:

```
[root@localhost shellprgms]# sh 4a.sh
Invalid Arguments.......
```

8) This shell script finds and displays all the links of a file specified as the first argument to the script. The second argument, which is optional, can be used to specify the directory in which the search is to begin. If this second argument is not present .the search is to begin in current working directory.

```
#!/bin/bash
if [ $# -eq 0 ]
then
        echo "Usage:sh 8a.sh[file1] [dir1(optional)]"
        exit
fi
if [ -f $1 ]
then
        dir="."
if [ $# -eq 2 ]
then
        dir=$2
fi
```

```
inode=`ls -i $1|cut -d " " -f 2`
echo "Hard links of $1 are"
find $dir -inum $inode -print
echo "Soft links of $1 are"
find $dir -lname $1 -print
else
echo "The file $1 does not exist"
fi
```

<u>Run1:</u>

```
[root@localhost shellprgms]$ sh 5a.sh  hai.c
Hard links of hai.c are
./hai.c
Soft links of hai.c are
./hai_soft
```

9) This shell script displays the calendar for current month with current date replaced by * or ** depending on whether date has one digit or two digits.

```
#!/bin/bash
n=` date +%d`
echo " Today's date is : `date +%d%h%y` ";
cal > calfile
if [ $n -gt 9 ]
then
 sed "s/$n/\**/g" calfile
else
 sed "s/$n/\*/g" calfile
[root@localhost shellprgms]# sh 6a.sh
 Today's date is : 10 May 05
      May 2005
Su  Mo  Tu  We  Th  Fr  Sa
```

```
 1  2   3  4   5  6  7
 8  9  **  11  12 13 14
15 16  17 18  19 20 21
22 23  24 25  26 27 28
29 30  31
```

10) This shell script implements terminal locking. Prompt the user for a password after accepting, prompt for confirmation, if match occurs it must lock and ask for password, if it matches terminal must be unlocked

```
trap “ " 1 2 3 5 20
clear
echo -e “\nenter password to lock terminal:"
stty -echo
read keynew
stty echo
echo -e “\nconfirm password:"
stty -echo
read keyold
stty echo
if [ $keyold = $keynew ]
then
echo "terminal locked!"
while [ 1 ]
do
echo "retype the password to unlock:"
stty -echo
read key
if [ $key = $keynew ]
then
stty echo
echo "terminal unlocked!"
stty sane
exit
fi
```

```
echo "invalid password!"
done
else
echo " passwords do not match!"
fi
stty sane
```

<u>Run1:</u>

```
[root@localhost shellprgms]# sh 13.sh
enter password:
confirm password:
terminal locked!
retype the password to unlock:
invalid password!
retype the password to unlock:
terminal unlocked!
```

****