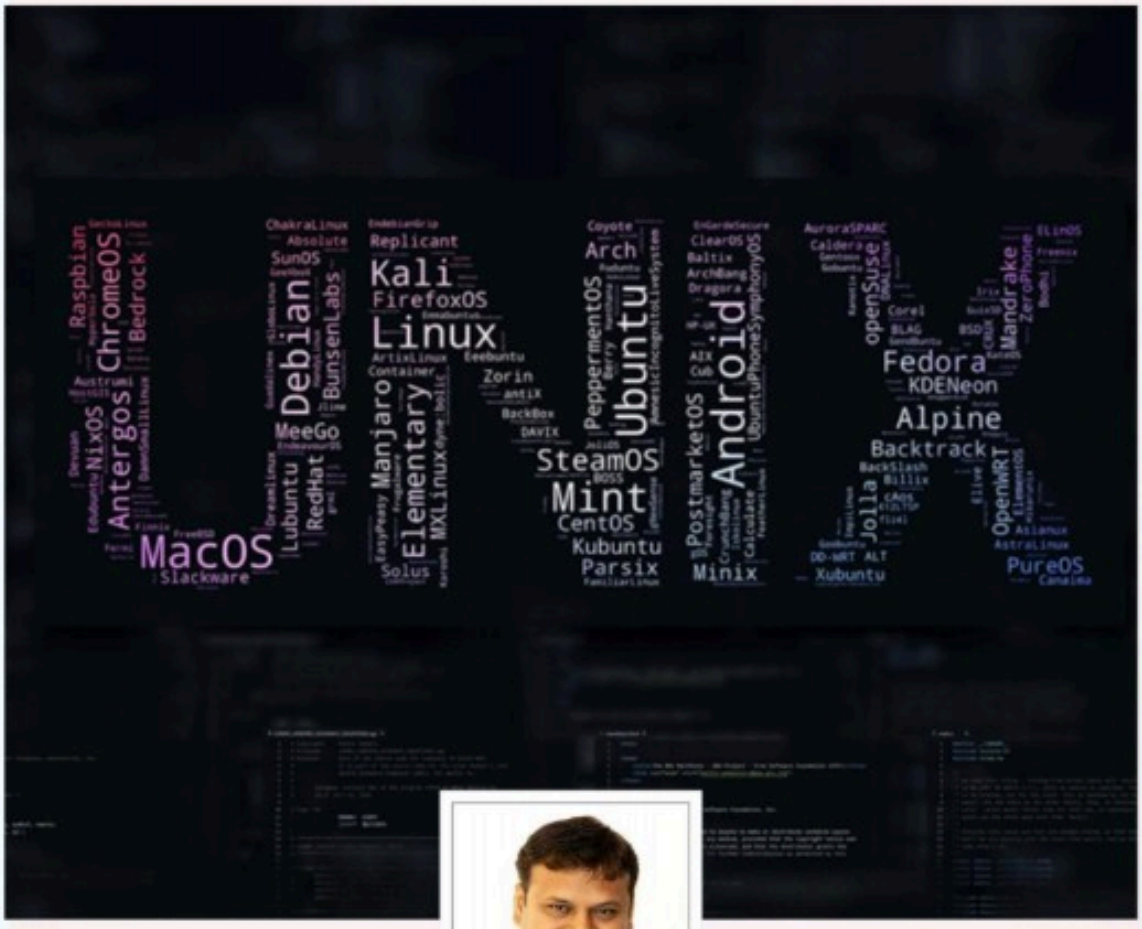
unix shell scripting

Jump2Learn
The Online Learning Place

UNIX
& SHELL PROGRAMMING

CHAPTER 4 - ADVANCED SHELL PROGRAMMING



AUTHOR

Mr. Manish Dolia

M.C.A.,Ph.D.(Pursuing)

DOLAT USHA INSTITUTE OF
APPLIED SCIENCES, VALSAD

4.1 FILTERING UTILITIES (GREP, SED) :

GREP UTILITY:

The grep filter is used to search a file for a particular pattern of characters, and display all records/lines that contain that pattern. The pattern that is searched in the file is referred to as the **regular expression**. GREP stands for globally search a regular expression and print it.

It scans a file for the occurrences of a pattern, and can display the selected patter, the line numbers in which they were found, or the filenames where the pattern occurs. grep can also select lines that does not containing the pattern.

The syntax for the grep command is as follows:

GREP [OPTIONS] PATTERN [FILENAME(S)]

In the above syntax square bracket indicates option part. The filename(s) is(are) optional in the grep command. Without a filename grep expects standard input. As a line is input grep searches for the regular expression in the line and displays the line if it contains that regular expression. Execution stops when the user indicates end of input by pressing ctrl d.

grep requires an expression to represent the pattern to be searched for, followed by one or more filenames. The first argument is always treated as the expression, and the ones remaining as filenames.

➢ **SPECIFYING REGULAR EXPRESSION:**

An expression formed with some special and ordinary characters, which is expanded by a command (and not by the shell) to match more than on string. A regular expression is always quoted to prevent its interpretation by the shell.

Regular expressions can be used to specify very simple patterns of characters to highly complex ones. Some very simple patterns are as follows:

REGULAR EXPRESSION	MEANING
A	It displays all lines that contain character A.
"new"	It displays all lines that contains pattern "new"

More complex regular expressions can be specified by using the following characters [in the double-quotes].

Character **Use**

[ ] → To specify a pattern which consists of any one of a set of characters.

e.g. grep "new[abc]" filename

Specifies the search patterns as 'newa' or 'newb' or 'newc'

[ ] with hyphen → i.e. grep "new[a-c]" filename

^pattern → To specify that the pattern following it must occur at the beginning of each line.

e.g. grep "^new" filename – searches all lines begins with 'new'

grep [^abc] filename – it specifies the pattern which does not contain either 'a' or 'b' or 'c'.

pattern$ → To specify that pattern preceding it must occur at the end of line.

e.g. grep "new[a-c]$" filename – specifies the search pattern as 'newa' or 'newb' or 'newc', but these patterns must occur at the end

of the line.

.(dot) → It matches a single character.

e.g. grep "new.[a-c]" filename – specifies the search pattern as 'new' followed by any character, followed by either 'a' or 'b' or 'c'.

\ (backslash) → This is used in conjuction with above characters. It indicates that grep should ignore the special meaning of the character following it in

regular expression.

e.g. grep "new\.\[abc\]" filename – specifies one search pattern, i.e. 'new.[abc]' in which the dot signifies a dot character itself and not any character.

- **grep** is a representative UNIX command that silently returns the prompt in case the pattern cannot be located.

e.g. $grep hello filename

$ # No hello found

When grep is used with a series of strings, it interprets the first argument as the pattern and the rest as filenames along with the output.

NOTE: Quote is compulsory when [you are looking for] pattern contains more than one words e.g. grep "hello world" filename

Or

contains special characters that can be interpreted otherwise by the shell. You can generally use either single or double quotes, but if command substitution or variable evaluation is involved, you must use double quotes.

- **GREP OPRIONS:**
  1. –c (count): Display frequency of matched pattern
  2. –l (list): Displays only the names of files where a pattern has been found.
  3. –n (number): It can be used to display the line numbers containing the pattern, along with
     the lines.
  4. -v (inverse): The –v option select all but the lines containing the pattern.
  5. –i (ignore): It ignores case for pattern matching.
  6. –h (hide): Omits filenames when handling multiple files.
  7. –e exp : Specifies expression with this option. Can use multiple times.
     e.g. gerp –e "hello" –e "unix" f1 → It displays all lines which contain either 'unix' or 'hello' or both.
  8. –A n : Displays line and N lines after matching lines (Linux only)
     e.g. gerp –A 5 "hello" f1 → It displays matched line and 5 lines after matching lines.
  9. –B n : Displays line and N lines before matching lines (Linux only)
     e.g. gerp –B 5 "hello" f1 → It displays matched line and 5 lines before matching lines.
  10. –n : Displays line and N line above and below (Linux only)
      e.g. gerp –5 "hello" f1 → It displays matched line and 5 lines above and below matching line.

Note : Negating a class Regular expressions use the ^(caret) to negate the character class, while the shell uses the !(bang).

The * (asterisk) refers to the immediately preceding character. It matches zero or more occurrences of previous character. The pattern g* matches a null string, single character 'g' and any number of gs. i.e. g gg ggg gggg .....

The * has significance in regular expression only if it is preceded by a character. If it is the first character in a regular expression, then it is treated literally (i.e. matches itself).

The ^(caret) symbol placed at the beginning of a character class (e.g. [^a-z]), it negates every character of the class. When placed outside it, and the beginning of expression (e.g. ^2...), the pattern is matched at the beginning of the line. At any other location (e.g. a^b), it matches itself literally.

The . (dot) and * lose their meaning when placed inside the character class. The * also matched literally (lose its meaning) if it is the first character of the expression.

Exercise:

(1) How do you locate lines containing "Agarwal" and "agrawal"?

- grep [Aa]g[ar][ar]wal filename

(2) How do you locate lines containing "Agarwal" , "agrawal" and "aggarwal"?

- grep [Aa]gg*[ar][ar]wal filename

(3) Write a command to display all lines that contains * character in a line.

- grep "*" filename OR grep [*] filename

(4) Write a command to display all lines that contains pattern g* in a line.

- grep g* filename

(5) Write a command to display all lines contains characters more then five characters.

- grep '.....' f1

EGREP(EXTENDED GREP) COMMAND:

egrep extends grep's pattern-matching capabilities. It was invented by Alfred Aho. It offers all the options of grep, but its most useful feature is the facility to specify more than one pattern for search. Each pattern is separated from the other by a | (pipe). While grep uses some more characters that are not recognized by egrep, egrep includes some additional characters not used by either grep or sed.

Expression	Meaning
ch+	→ Matches one or more occurrences of character ch
ch?	→ Matches zero or one occurrences of character ch
exp1| exp2	→ Matches expression exp1 or exp2
(x1|x2)x3	→ Matches expression x1x3 or x2x3.

e.g. b+ matches b, bb, bbb, etc. while b? matches either nothing or a single b

e.g. egrep "(soft|hard)ware" f1 → display all lines which contains either software or hardware or both.

NOTE: Pattern must be quoted in egrep.

Exercise:

(1) How do you locate lines containing "Agarwal" , "agrawal" and "aggarwal"?

egrep [Aa]gg?[ar]+wal filename

(2) How do you locate lines containing "sengupta" and "dasgupta"?

egrep 'sengupta|dasgupta' filename OR

egrep '(sen|das)gupta' filename

➢ **STORING PATTERN IN A FILE:**

If there are number of pattern, you have to match; egrep offers the –f (file) option to take such patterns from the file. e.g. a file contains following patterns:

$cat pat.lst

admin|accounts|sales

When you execute egrep with the –f option in this way:

egrep –f pat.lst emp.lst

the command takes the expression from pat.lst

FGREP (FIXED GREP OR FAST GREP) COMMAND:

fgrep accepts multiple patterns, both from the command line and a file, but unlike grep and egrep, does not accept regular expressions. So, if the pattern to search for is a simple string, or a group of them, fgrep is recommended. It is arguably faster than grep and egrep, and should be used when using fixed strings.

Alternative patterns in fgrep are specified one pattern from another by the new-line character. This is unlike in egrep, which uses the | to delimit two expressions. You may either specify these patterns in the command line itself, or store them in a file as follow:

$cat pat.lst

admin

account

sales

and the command is as follows:

$fgrep –f pat.lst emp.lst

- The disadvantage with grep family is that none of them has separate facilities to identify fields. This limitation is overcome by awk.

Exercise:

(1) How will you remove blank lines from a file?
- grep –v "^[<tab>][<tab>]*$" filename OR
- grep "[^ <tab>][<tab>]*$" filename

(2) What does grep "^*" do? Is the \ really necessary?
- It searches for an asterisk at the beginning of the line. \ is not necessary.

(3) Locate all lines longer than 15 characters.
- gerp ".\{16\}" filename OR
- grep "....." filename

(4) How will you list the ordinary files in your current directory that are not writable?
- ls –l | grep "^-.[^w]"

(5) What does ^[^$]$ pattern match?
- only a $

(6) What does $$$*$ pattern match?
- at least three $s at the end of a line

(7) What output this command sequence produce?

who | grep –c "$LOGNAME"

- count of users using the same username as the user executing the command.

(8) Locate lines beginning and ending with a dot (.) and containing anything between them.
- grep "^\..*\.$" f1

(9) Write a command to display all lines which contains two or more $ symbol at end of line.
- grep '\$\$\$*$' f1

(10) Write a command to display all lines which contains two or more ^ symbol at beginning

of line.

- grep '^\^\^\^*$' f1

Exercise:

(1) WASST accepts a string and check whether it is palindrome or not.

(2) WASST simulates LEFT command of basic.

(3) WASST simulates RIGHT command of basic.

(4) WASST simulates MID command of basic.

(5) WASST accepts a string and display character triangle as follow:

e.g. input string : surat

s

su

sur

sura

surat

SED (STREAM EDITOR) FILTER :

sed is a multi-purpose tool which combines the work of several filters. It was designed by Lee McMohan that is derived from the ed line editor. Everything in sed is an instruction. An instruction combines an address for searching lines with an action to be taken.

Syntax:

Sed [ptions] instruction filename(s)

Instruction consists of two components address and action that are enclosed within single quotes. Following table shows different commands used in action component.

Command	Meaning
i a, c	→ Inserts, appends and changes text
d	→ Delete line(s)
	e.g. 1,4d – delete lines 1 to 4
r fname	→ Places contents of file fname after line
w fname	→ Writes addressed lines to file fname.
p	→ Prints line(s) on standard output.
	e.g. 3,$p – prints lines 3 to end of line (-n option required)
/begin/,/end/p	→ prints lines enclosed between begin and end (-n option required)
q	→ quits after reading up to addressed line

e.g. 10q – quits after the first 10 lines.

= → prints line number addressed

s/s1/s2 → replaces first occurrences of string or regular expression s1 in all lines with s2.

e.g. 10,20s/-/:/ - Replaces first occurrences of – in lines 10 to 20 with a ':'

s/s1/s2/g → Replaces all occurrences of string or regular expression s1 in all lines with string s2

e.g. 10,20s/-/:/g - Replaces all occurrences of – in lines 10 to 20 with a ':'

LINE ADDRESSING:

Addressing in sed is done in two ways.

- By line number (like 1,3p)
- By specifying a pattern which occurs in a line (like /unix/p).

e.g. sed '3q' file1 → It quits after line number 3

unix

linux

shell programming

- **sed** also uses the p (print) command to print the addressed line.

e.g. sed '1,3p' file1

unix

unix

linux

linux

shell programming

shell programming

Here sed by default prints all lines on the standard output, in addition to the lines affected by the action. So, the addressed lines are printed twice. To overcome the problem of printing duplicate lines, you should use the –n option whenever you use the p command.

e.g. sed –n '1,3p' file1

unix

linux

shell programming

To select the last line of the file, use the $ symbol

i.e. sed –n '$p' file1

- **Reversing line selection criteria (!):** you can use sed's negation operator (!) with any action. So selecting first five lines is the same as not selecting lines six through end of file. Therefore the command is:

```
sed -n '6,$!p' file1
```

OR

```
Sed –n '1,5p' file1
```

- **To select non-contiguous groups of lines, the command is**

```
$sed –n '1,3p
> 7,9p                #It select lines 1 to 3, 7 to 9
> $p' file1           # and last line
```

OR

```
$sed –ne '1,3p' –e '7,9p' –e '$p' file1
```

- **Inserting text in file:**

Sed uses i, a and c commands to perform insert, append and change operation on file respectively.

```
e.g.  $sed '$a\              # It appends two lines at the end of file
      >Red hat Linux \
      > C++
      > ' file1
```

- **IF YOU WANT TO REDIRECT IT INTO ANOTHER FILE THEN THE COMMAND IS**

```
$sed '$a\
> sco unix
> ' file1 > newfile
```

- **IF YOU DO NOT USE LINE ADDRESS WITH I OR A COMMAND, THEN BLANK LINE IS INSERTED BEFORE (I) OR AFTER (A) EVERY LINE OF THE FILE.**

i.e. $sed 'i\ $sed 'i\ # It insert blank line before every line

> OR > # in a file

>' file1 >

.

>' file1

- If instruction is 1a – It insert new line after 1st line
- If instruction is 1i – It insert new line before 1st line
- c command is used to change text in file

e.g. $sed '1c\ # It replaces 1st line with 'unix'

> unix ' file1

$sed 'c\ # Changes made in all lines

> unix' file1

- **CONTEXT ADDRESSING:**

The second form of addressing specify a pattern as well. When you specify a single pattern, all lines containing the pattern are selected.

e.g.$sed –n '/unix/p' file1 → It prints all lines that contains pattern 'unix'

An addressing lines by specifying the context or pattern is known as context addressing.

- You can also specify comma-separated pair of context addresses to select a group of lines.

e.g.$sed –n '/unix/,/linux/p' file1 → It prints all lines that contains pattern 'unix' through 'linux'.

- You can mixed line and context addresses togather.

e.g.$sed '1,/unix/p' file1 → It prints lines 1 to the first lines that contains pattern 'unix'

e.g.$sed –ne '1,3p #It prints lines 1 to 3 and all lines that contains pattern

> /unix/p ' file1 # 'unix'

- Sed also accepts regular expressions similar to grep command. i.e. [], *, . etc as a part of the regular expression.

➢ BRACED REGULAR EXPRESSION:

If you have to locate a string (line) longer than 100 characters, it is meaning-less to use 100 dots. Sed and grep also accept a special form of a regular expression which let you specify the number of characters preceding a pattern.

e.g.$sed –n '/.\{51\}/p' file1 → it prints all lines longer than 50 characters. Here the expression \{51\} specifies that the previous character (.) has to occur 51 times. This method is also used in grep command such as

e.g.$grep '.\{51\}' file1

This expression derived from ed, and using the escaped pair of curly braces will be referred to as a brace regular expression (BRE). It takes the three forms:

Ch\{m\} ch\{m,n\} ch\{m,\}

All these forms have the single character regular expression ch as the first element. This character can be either a literal character, a . or a character class. It is followed by a pair of escaped curly braces, containing either a single character m, or a range of numbers lying between m and n to determine the number of times the character preceding it can occur. The value of m and n cannot exceed 255.

e.g. (1) WACT display all lines having length between 101 and 150

sed '/.\{101,150\}/p' file1

(2) WACT display all lines having length of atleast 101 characters.

Sed '/.\{101,\}/p' file1

➢ DELETING LINES:

Using the d (delete) command, sed can emulate grep's –v option to select lines not containing the pattern. –n option not to be used with d command.

e.g. $sed '/unix/d' file1 >ofile

OR

$sed '/unix/!p' file1 >ofile

It select all lines that does not contains pattern 'unix' and saves it into ofile.

e.g. $sed '/^[<tab>]*$/d' file1 → It deletes all lines from file1. (ctrl+I for tab key)

➢ WRITING SELECTED LINES TO A FILE:

The w (write) command makes it possible to write the selected lines in a separate file.

e.g. $sed –n '/unix/w ofile' file1 → it selects all lines that contains pattern 'unix' and write it to file ofile

e.g. $sed '/unix/w ofile' file1 → it display file contents (entire file) and write selected lines to file ofile.

e.g. $sed '1,5w fout' file1 → It writes lines 1 to 5 into file fout.

$sed '1,5d" file1 → it list out line 6 to end of line only.

- Sed accepts more than one address, you can perform a full context splitting of the file (original file into many more).

e.g. $sed –n '/linux/w lfile

> /unix/w ufile ' file1

It writes all lines that contains pattern 'linux' to lfile and pattern 'unix' to ufile

e.g. $sed '1,5w ofile' file1 → It displays as well as writes lines 1 through 5 on screen and ofile file respectively.

THE –F OPTION: INSTRUCTION FROM A FILE:

When there are numerous editing instructions to be performed, it will be better to use the –f option to accept instructions from a file.

e.g. create a file as follow:

$cat instr.txt

/unix/w ulist

/linux/w llist

and then sed used with the –f filename option:

$sed –n –f instr.txt file1

You can use the –f option with multiple files.

e.g. $sed –n –f instr1.txt –f instr2.txt file1

You can combine the –e and –f options as many times as you want.

e.g. $sed –ne '/linux/p' –f instr.txt –f instr2.txt file1

➢ SUBSTITUTION:

This is achieved with its s (substitution) command. It lets you replace a pattern in its input with something else. The syntax for such a command can be described as

Syntax:

[address]s/string1/string2/flag

Here, string1 will be replaced by string2 in all lines specified by the address. If the address is not specified, the substitution will be perform for all lines containing string1.

e.g. $sed 's/unix/linux/' file1 → It replaces all the word 'unix' by 'linux' in given file.

$sed '1,5s/unix/linux/' file1 → It replaces all word 'unix' by 'linux' in first five lines of file1.

Sed also uses regular expression for patterns to be substituted. To replace all occurrences of 'agarwal', 'aggarwal' and 'agrawal' by simply 'Agrawal', use the regular expression as follows:

$sed –n 's/[Aa]gg*[ar][ar]wal/agarwal/p' file1

Here only affected lines will be displayed (because of p command and –n option). If we omit –n option then it display affected lines two times and remaining will be display once.

➢ GLOBAL SUBSTITUTION:

If you give command as follow:

$sed –n 's/unix/linux/p' file1 → this replaces the first occurrences of the 'unix' pattern in all lines with linux. The other 'unix' pattern remain unaffected. To replace all occurrences, you need to use the g(global) flag at the end of the instruction. This is referred to as global substitution. Now, the command is:

$sed –n 's/unix/linux/pg' file1

13

➢ COMPRESSING MULTIPLE SPACES:

$sed –n 's/ *//gp' file1 → Replaces all spaces.

➢ **Reading in a file:**

The r filename lets you read in a file at a certain location of the file.

e.g. $sed '/unix/r file1.txt' file2

- Write a command to list files which have write permission for the group.

  ls –l| sed –n '/^.....w/p'

- WACT display all files which have read and write permission for the group.
- WACT display all files which have read, write and execute permission for the group.
- WACT count number of users who are currently logged in (Do not make use of wc command)

Who | grep –c .

- WA script to simulate wc command of UNIX.

➢ REMEMBERED PATTERN:

The three commands below do the same job:

Sed 's/unix/linux/' file1

Sed '/unix/s//linux/' file1

Sed '/unix/s/unix/linux/' file1

The 2nd pattern suggest that sed 'remembers' the scanned pattern, and stores it in // (2 frontslashes). The // representing an empty (or null) regular expression is interpreted to mean that the search and substituted patterns are the same. We will call it the remembered pattern.

However, when you use // in the target string, it means you use removing the pattern totally.

e.g. $sed 's/unix//g' file1 → it removes all 'unix' pattern in file1.

The address /unix/ in the third form appears to be redundant.

It is possible that you may like to replace a string in all lines containing a different string:

e.g. $sed –n '/The unix/s/unix/UNIX/p' file1

It searches pattern 'The unix' and replace each 'unix' with UNIX.

- If // in source string, it implies that the scanned pattern is stored there. If the target string is //, it means that the source pattern is to be removed.

➢ THE REPEAT PATTERN:

When a pattern in the source string also occurs in the replaced string, you can use the special characters '&' to represent it. Following three commands are equal:

Sed 's/director/executive director/' file1

Sed 's/director/executive &/' file1

Sed '/director/s//executive &/' file1

The '&' known as repeated pattern, expands to the entire source string.

- WAC which display the listing for those file that have write bit set either for group or others:

```
Ls –l|grep "^.\{5,8\}w"
```

- WACT display all lines having length between 101 and 150

```
Grep '^.\{101,150\}$' file1
```

- WACT display all lines having length at least 101

```
Sed –n '/.\{101,\}p' file1
```

THE TAGGED REGULAR EXPRESSION(TRE):

Sed uses the '&' for reproducing the entire source string. Sometimes, you require just a part of the source string to be present in the target string. This is also possible with the special sed feature of attempting a tag to a pattern.

The pattern to be tagged (or extracted) is identified by enclosing it with an escaped pair or parentheses. For instance, the pattern 'unix' in the source string is to be written as \(unix\). If this is the first such 'grouped' pattern in the line, then it automatically gets the tag \1. You can now refer to this pattern in the replacement string with \1. Tags are consecutively numbered from 1 to 9, and are referenced in the target string by the descriptors \1, \2 and so on.

e.g. we want to replace the word 'new line' by new-line'. Then the command is

(1) echo "new line" | sed 's/\(new\) \(line\)/\1-\2/'

(2) echo "production manager" | sed 's/\(production\) \(manager\)/\2\1/'

(3) write a command to convert date in the format mm/dd/yy in to dd-mm-yy.

```
Sed 's/\(..\)\/\(..\)\/\(..\)/\2-\1-\3/' file1
```

Here, we have two tagged patterns \(production\) ans \(manager\) in the source string. They are automatically reproduced in the target string with the values \1 and \2, respectively. Each escaped pattern is called a tagged regular expression (TRE).

4.2. AWK UTILITY

Awk is a scripting language used for manipulating data and generating reports.The awk command programming language requires no compiling, and allows the user to use variables, numeric functions, string functions, and logical operators.

Awk is a utility that enables a programmer to write tiny but effective programs in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line. Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then performs the associated actions.

Awk is abbreviated from the names of the developers – Aho, Weinberger, and Kernighan.

WHAT CAN WE DO WITH AWK ?

1. AWK OPERATIONS:

(a) Scans a file line by line

(b) Splits each input line into fields

(c) Compares input line/fields to pattern

(d) Performs action(s) on matched lines

2. USEFUL FOR:

(a) Transform data files

(b) Produce formatted reports

3. Programming Constructs:

(a) Format output lines

(b) Arithmetic and string operations

(c) Conditionals and loops

Syntax:

AWK OPTIONS 'SELECTION _CRITERIA {ACTION }' INPUT-FILE > OUTPUT-FILE

OPTIONS:

-f program-file : Reads the AWK program source from the file
program-file, instead of from the
first command line argument.

-F fs : Use fs for the input field separator

SAMPLE COMMANDS

Example:
Consider the following text file as the input file for all cases below.

$cat > employee.txt

ajay manager account 45000

sunil clerk account 25000

varun manager sales 50000

amit manager account 47000

tarun peon sales 15000

deepak clerk sales 23000

sunil peon sales 13000

satvik director purchase 80000

1. Default behavior of Awk : By default Awk prints every line of data from the specified file.

$ awk '{print}' employee.txt

OUTPUT:

ajay manager account 45000

sunil clerk account 25000

varun manager sales 50000

amit manager account 47000

tarun peon sales 15000

deepak clerk sales 23000

sunil peon sales 13000

satvik director purchase 80000

In the above example, no pattern is given. So the actions are applicable to all the lines. Action print without any argument prints the whole line by default, so it prints all the lines of the file without failure.

2. PRINT THE LINES WHICH MATCHES WITH THE GIVEN PATTERN.

$ awk '/manager/ {print}' employee.txt

Output:

ajay manager account 45000

varun manager sales 50000

amit manager account 47000

In the above example, the awk command prints all the line which matches with the 'manager'.

3. Splitting a Line Into Fields : For each record i.e line, the awk command splits the record delimited by whitespace character by default and stores it in the $n variables. If the line has 4 words, it will be stored in $1, $2, $3 and $4 respectively. Also, $0 represents the whole line.

$ awk '{print $1,$4}' employee.txt

Output:

ajay 45000

sunil 25000

varun 50000

amit 47000

tarun 15000

deepak 23000

sunil 13000

satvik 80000

In the above example, $1 and $4 represents Name and Salary fields respectively.

BUILT IN VARIABLES IN AWK

Awk's built-in variables include the field variables—$1, $2, $3, and so on ($0 is the entire line) — that break a line of text into individual words or pieces called fields.

NR: NR command keeps a current count of the number of input records. Remember that records are usually lines. Awk command performs the pattern/action statements once for each record in a file.

NF: NF command keeps a count of the number of fields within the current input record.

FS: FS command contains the field separator character which is used to divide fields on the input line. The default is "white space", meaning space and tab characters. FS can be reassigned to another character (typically in BEGIN) to change the field separator.

RS: RS command stores the current record separator character. Since, by default, an input line is the input record, the default record separator character is a newline.

OFS: OFS command stores the output field separator, which separates the fields when Awk prints them. The default is a blank space. Whenever print has several parameters separated with commas, it will print the value of OFS in between each parameter.

ORS: ORS command stores the output record separator, which separates the output lines when Awk prints them. The default is a newline character. print automatically outputs the contents of ORS at the end of whatever it is given to print.

Examples:

USE OF NR BUILT-IN VARIABLES (DISPLAY LINE NUMBER)

$ awk '{print NR,$0}' employee.txt

Output:

1 ajay manager account 45000

2 sunil clerk account 25000

3 varun manager sales 50000

4 amit manager account 47000

5 tarun peon sales 15000

6 deepak clerk sales 23000

7 sunil peon sales 13000

8 satvik director purchase 80000

In the above example, the awk command with NR prints all the lines along with the line number.

USE OF NF BUILT-IN VARIABLES (DISPLAY LAST FIELD)

$ awk '{print $1,$NF}' employee.txt

Output:

ajay 45000

sunil 25000

varun 50000

amit 47000

tarun 15000

deepak 23000

sunil 13000

satvik 80000

In the above example $1 represents Name and $NF represents Salary. We can get the Salary using $NF , where $NF represents last field.

ANOTHER USE OF NR BUILT-IN VARIABLES (DISPLAY LINE FROM 3 TO 6)

$ awk 'NR==3, NR==6 {print NR,$0}' employee.txt

Output:

3 varun manager sales 50000

4 amit manager account 47000

5 tarun peon sales 15000

6 deepak clerk sales 23000

More Examples

For the given text file:

$cat > geeksforgeeks.txt

A B C

Tarun A12 1

Man B6 2

Praveen M42 3

1) TO PRINT THE FIRST ITEM ALONG WITH THE ROW NUMBER(NR) SEPARATED WITH " – " FROM EACH LINE IN GEEKSFORGEEKS.TXT:

$ awk '{print NR "- " $1 }' geeksforgeeks.txt

1 - Tarun

2 – Manav

3 - Praveen

2) TO RETURN THE SECOND ROW/ITEM FROM GEEKSFORGEEKS.TXT:

$ awk '{print $2}' geeksforgeeks.txt

A12

B6

M42

3) TO PRINT ANY NON EMPTY LINE IF PRESENT

$ awk 'NF > 0' geeksforgeeks.txt

0

4) TO FIND THE LENGTH OF THE LONGEST LINE PRESENT IN THE FILE:

$ awk '{ if (length($0) > max) max = length($0) } END { print max }' geeksforgeeks.txt

13

5) TO COUNT THE LINES IN A FILE:

$ awk 'END { print NR }' geeksforgeeks.txt

3

6) PRINTING LINES WITH MORE THAN 10 CHARACTERS:

$ awk 'length($0) > 10' geeksforgeeks.txt

Tarun A12 1

Praveen M42 3

7) TO FIND/CHECK FOR ANY STRING IN ANY COLUMN:

$ awk '{ if($3 == "B6") print $0;}' geeksforgeeks.txt

8) TO PRINT THE SQUARES OF FIRST NUMBERS FROM 1 TO N SAY 6:

$ awk 'BEGIN { for(i=1;i<=6;i++) print "square of", i, "is",i*i; }'

square of 1 is 1

square of 2 is 4

square of 3 is 9

square of 4 is 16

square of 5 is 25

square of 6 is 36

4.3. BATCH PROCESS

The heart of a batch processing system is the queuing mechanism of the program that is to be run. The file queuedefs is used to describe the characteristics of the queues managed by cron daemon. The file uses a letter between "a" and "y" to uniquely define the name of a queue. Tasks that are started by the at command are placed in the default queue for a. Those started by the batch command are placed in the b queue and the c queue is for tasks from the crontab file. The other queue names may be defined by users.

The at command uses the facilities of the cron daemon to schedule the execution of a script at a specified time. at does not do true batch processing, because many jobs may be scheduled to execute a the same time instead of having a queue of jobs to be executed with each job running in succession. The standard usage for the at command is:

```
$ at 1440 May 31
at> date
at> ^D
warning: commands will be executed using /bin/csh
job 833571600.a at Fri May 31 14:40:00 1996
```

This causes the creation of a the file, /var/spool/cron/atjobs/833571600.a on this example host which is SunOS 5.x. Once the job has run this file is removed.

UNIX SCHEDULING PRIORITIES

Unix processes have an associated system nice value which is used by the kernel to determine when it should be scheduled to run. This value can be increased to facilitate processes executing quickly or decreased so that the processes execute slowly and thus do not interfere with other system activities.

The process scheduler, which is part of the Unix kernel, keeps the CPU busy by allocating it to the highest priority process. The nice value of a process is used to calculate the scheduling priority of a process. Other factors that are taken into account when calculating the scheduling priority for a process include the recent CPU usage and its process state, for example "waiting for I/O" or "ready to run".

Normally, processes inherit the system nice value of their parent process. At system initialization time, the system executes the init process with a system nice value of 20, this is the system default priority. All processes will inherit this priority unless this value is modified with the command nice. The nice value of 0 establishes an extremely high priority, whereas a value of 39 indicates a very low priority on SVR4 derived systems. On BSD derived

systems scheduling priorities range from 0 to 127. The higher the value, the lower the priority, and the lower the value, the higher the priority.

There are two versions of nice; one that is built in to the C shell and the standalone nice command which is available to other shell users. Only the nice command will be described here. Refer to the man page for more information on the nice available to C shell users.

On systems derived from BSD, the nice command uses the numbers -20 to 20 to indicate the priorities, where 20 is the lowest and -20 is the highest. Any user can lower the priority of their processes, however only superuser and increase the priority of a job. To decrease the priority:

nice -6 mybigjob

To increase the priority:

nice --6 mybigjob

The nice levels for SVR4 systems are from 0 to 39. The default is 20. To decrease the priority:

nice -6 mybigjob

In this example the level has been set to 20-6, or 14.
To increase the priority:

nice +6 mybigjob

Examples

COMMAND	EXPLANATION
at noon tar -cf /users/dvader dvader.tar Ctrl-d	The job will run at noon the same day if submitted in the morning, or noon the next day if submitted in the afternoon. When the task is performed, a tarball of the /users/dvader directory will be created.
batch -f /home/hsolo/script1	Rather than entering the commands into standard input, the user submits a batch command for a job that will execute the script /home/hsolo/script1.
at -m 0530 November 9, 2009 /users/chewie/hb28.script Ctrl-d	At 5:30am on November 9, 2009, the script hb28.script will run. A mail message indicating the script has executed will be sent to the user who submitted the job.
at -r skywalker.887664428.b	Delete the job skywalker.887664428.b.

4.4. SPLITTING (CAT, CUT, HEAD AND TAIL), COMPARING (CMP, COMM., DIFF), SORTING(SORT), MERGING & ORDERING FILES (PASTE, UNIQ)

1) CAT = The cat command displays the content of one or more text files on the screen without pausing. ,can be used to join multiple files together and print the result on screen (it will not show page by page)

EXAMPLE:

cat 01.txt

to displat the contents of file 01.txt

cat 01.txt 02.txt

to display the contents of both files

cat file1.txt file2.txt > file3.txt – Reads file1.txt and file2.txt and combines those files to make

cat note5 >> notes – attach note5 to notes

cat >> file1 – add additional data in file1

Do not use the cat command to read binary files. Using the cat command to read binary files can cause a terminal window to freeze. If your terminal window freezes, close the terminal window, and open a new terminal window.

Note: Before you attempt to open a file with the cat command, it is recommended that you first run the file command to determine the file type.

2) CUT COMMAND

The cut command extracts a given number of characters or columns from a file. For cutting a certain number of columns it is important to specify the delimiter. A delimiter specifies how the columns are separated in a text file

Example: Number of spaces, tabs or other special characters.

Syntax:

cut [options] [file]

The cut command supports a number of options for processing different record formats. For fixed width fields, the -c option is used.

$ cut -c 5-10 file1

This command will extract characters 5 to 10 from each line.

For delimiter separated fields, the -d option is used. The default delimiter is the tab character.

$ cut -d "," -f 2,6 file1

This command will extract the second and sixth field from each line, using the ',' character as the delimiter.

SYMBOL	DESCRIPTION
-l	Line count
-w	Word count
-c	Byte count
-m	Character count

Example:

Assume the contents of the data.txt file is:

Employee_id;Employee_name;Department_name;Salary
10001;Employee1;Electrical;20000
10002; Employee2; Mechanical;30000

10003;Employee3;Electrical;25000
10004; Employee4; Civil;40000

And the following command is run on this file:

$ cut -c 5 data.txt

The output will be:

o

1

2

3

4

If the following command is run on the original file:

$ cut -c 7-15 data.txt

The output will be:

ee_id; Emp

Employee1

Employee2

Employee3

Employee4

If the following command is run on the original file:

$ cut -d "," -f 1-3 data.txt

The output will be:

Employee_id;Employee_name;Department_name

10001;Employee1;Electrical

10002; Employee2; Mechanical

10003;Employee3;Electrical

10004; Employee4; Civil

3) HEAD = THE HEAD COMMAND DISPLAYS THE FIRST 10 LINES OF A FILE.

$ head -n filename

You can change the number of lines displayed by using the -n option. For example, to display the first five lines of the /var/log/messages file, enter the head command with the -n option set to 5.

$ head -5 /var/log/messages

head myfile.txt – Would display the first ten lines of myfile.txt.

head -15 myfile.txt – Would display the first fifteen lines of myfile.txt.

.

4) TAIL = DISPLAY THE LAST PART OF THE FILE

The tail command displays the last 10 lines of a file.

$ tail –n/+n filename

You can change the number of lines displayed by using the -n or +n options.

– The -n option displays n lines from the end of the file.

– The +n option displays the file from line n to the end of the file.

For example, to display the last four lines of the /var/log/messages file, enter the tail command with the -n option set to 4.

$ tail -4 /usr/dict/words

For example, to display line 10 through the end of the data.txt file, enter the tail command with the +n option set to 10.

$ tail +10 data.txt

usage : tail filename

tail -n filename : display the last n lines of the file

5) MORE = TO VIEW A TEXT FILE ONE PAGE AT A TIME, PRESS SPACEBAR TO GO TO THE NEXT PAGE

more filename : show the document one page at a time

more -num filename : show the document page few lines as specified bu (-num)

example : more -10 filename will show 10 lines for every page

.

6) LESS = IS MUCH THE SAME AS MORE COMMAND EXCEPT:

a) You can navigate the page up/down using the less command and not possible in more command.

b) You can search a string in less command. (use /keywordto search)

c) "more" was fairly limited, and additional development on "more" had stopped

d) it uses same functions as vi editor

the usage : less filename

7) WC COMMAND

The wc command displays the number of lines, words, and characters contained in a file.

$ wc -options filename

You can use the following options with the wc command.

SYMBOL	DESCRIPTION
-l	Line count
-w	Word count
-c	Byte count
-m	Character count

When you use the wc command without options, the output displays the number of lines, words, and characters contained in the file. For example, to display the number of lines, words, and characters in the dante file, use the wc command.

$ wc data.txt

32 223 1319 data.txt

For example, to display the number of lines in the dante file, enter the wc command with the -l option.

$ wc -l data.txt

32 data.txt

8) CMP: THIS COMMAND IS USED TO COMPARE TWO FILES CHARACTER BY CHARACTER.

Syntax: cmp [options] file1 file2

Example: Add write permission for user, group and others for file1.

$ cmp file1 file2

9) COMM: THIS COMMAND IS USED TO COMPARE TWO SORTED FILES.

Syntax: comm [options] file1 file2

One set of options allows selection of 'columns' to suppress.

-1: suppress lines unique to file1 (column 1)

-2: suppress lines unique to file2 (column 2)

-3: suppress lines common to file1 and file2 (column3)

Example: Only show column-3 that contains lines common between file1 and file2

$ comm -12 file1 file2

10) DIFF: THIS COMMAND IS USED TO COMPARE TWO FILES LINE BY LINE.

Description: The output indicates how the lines in each file are different, and the steps invoved to change file1 to file2. The 'patch' command can be used to make the suggested changes. The output is formatted as blocks of:

diff [options] from-file to-file

DESCRIPTION

In the simplest case, diff compares the contents of the two files from- file and to-file.

-b Ignore changes in amount of white space.

-B Ignore changes that just insert or delete blank lines.

--brief Report only whether the files differ, not the details of the differences.

-e, --ed Make output that is a valid ed script.

-I Ignore changes in case; consider upper- and lower-case letters equivalent.

-s, --report-identical-files Report when two files are the same.

-w Ignore white space when comparing lines.

-y Use the side by side output format.

11) CHANGE COMMANDS

< lines from file1

—

> lines from file2

The change commands are in the format [range][acd][range]. The range on the left may be a line number or a comma-separated range of line numbers referring to file1, and the range on the right similarly refers to file2. The character in the middle indicates the action i.e. add, change or delete.

'LaR' – Add lines in range 'R' from file2 after line 'L' in file1.

'FcT' – Change lines in range 'F' of file1 to lines in range 'T' of file2.

'RdL' – Delete lines in range 'R' from file1 that would have appeared at line 'L' in file2

Syntax: diff [options] file1 file2

Example: Add write permission for user, group and others for file1

$ diff file1 file2

12) DIRCMP: THIS COMMAND IS USED TO COMPARE THE CONTENTS OF DIRECTORIES.

Description: This command works on older versions of Unix. In order to compare the directories in the newer versions of Unix, we can use diff -r

Syntax: dircmp [options] dir1 dir2

Example: Compare contents of dir1 and dir2

$ dircmp dir1 dir2

13) UNIQ: THIS COMMAND IS USED TO FILTER THE REPEATED LINES IN A FILE WHICH ARE ADJACENT TO EACH OTHER

Syntax: uniq [options] [input [output]]

Example: Omit repeated lines which are adjacent to each other in file1 and print the repeated lines only once

$ uniq file1

14) UNIX SORT COMMAND

The Unix sort command is a simple command that can be used to rearrange the contents of text files line by line.

The command is a filter command that sorts the input text and prints the result to stdout. By default, sorting is done line by line, starting from the first character.

Numbers are sorted to be ahead of letters.

Lowercase letters are sorted to be ahead of uppercase letters.

SORT SYNTAX:

sort [options] [files]

SORT OPTIONS:

- sort -b: Ignore blanks at the start of the line.
- sort -r: Reverse the sorting order.
- sort -o: Specify the output file.
- sort -n: Use the numerical value to sort.
- sort -M: Sort as per the calendar month specified.
- sort -u: Suppress lines that repeat an earlier key.
- sort -k POS1, POS2: Specify a key to do the sorting. POS1 and POS2 are optional parameters and are used to indicate the starting field and the ending field indices. Without POS2, only the field specified by POS1 is used. Each POS is specified as "F.C" where F represents the field index, and C represents the character index from the start of the field.
- sort -t SEP: Use the provided separator to identify the fields.

With the "-k" option, the sort command can be used to sort flat file databases. Without the "-k" option, the sorting is performed using the entire line. The default separator for fields is the space character. The -t option can be used to change the separator.

Examples:

ASSUME THE BELOW INITIAL CONTENTS OF FILE1.TXT FOR THE FOLLOWING EXAMPLES

01 Priya

04 Shreya

03 Tuhina

02 Tushar

SORT WITH DEFAULT ORDERING:

$ sort file1.txt

01 Priya

02 Tushar

03Tuhina

04 Shreya

In this example, the sorting is first performed using the first character. Since this is the same for all lines, the sorting then proceeds to the second character. Since the second character is unique for each line, the sorting ends there.

SORT IN REVERSE ORDERING:

$ sort -r file1.txt

04 Shreya

03Tuhina

02 Tushar

01 Priya

In this example, the sorting is done similar to the above example, but the result is in the reverse order.

SORT BY THE SECOND FIELD:

$ sort -k 2 file1.txt

01 Priya

04Shreya

03Tuhina

02 Tushar

NOW ASSUME THE ORIGINAL FILE2.TXT IS AS BELOW

01 Priya

01 Pooja

01 Priya

01 Pari

SORT WITH DEFAULT ORDERING

$ sort file2.txt

01 Pari

01 Pooja

01Priya

01Priya

SORT SUPPRESSING REPEATED LINES

$ sort -u file2.txt

01 Pari

01 Pooja

01Priya