When a program takes its input from another program, it performs some operation on that input, and writes the result to the standard output. It is referred to as a ***filter***.

The grep Command

The grep command searches a file or files for lines that have a certain pattern. The syntax is −

```
$grep pattern file(s)
```

The name **"grep"** comes from the ed (a Unix line editor) command **g/re/p** which means "globally search for a regular expression and print all lines containing it".

A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. If you don't give grep a filename to read, it reads its standard input; that's the way all filter programs work −

```
$ls -l | grep "Aug"
-rw-rw-rw-   1 john  doc     11008 Aug  6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--   1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

There are various options which you can use along with the **grep** command −

Sr.No.	Option & Description
1	**-v** Prints all lines that do not match pattern.
2	**-n** Prints the matched line and its line number.
3	**-l** Prints only the names of files with matching lines (letter "l")
4	**-c** Prints only the count of matching lines.
5	**-i** Matches either upper or lowercase.

Let us now use a regular expression that tells grep to find lines with **"carol"**, followed by zero or other characters abbreviated in a regular expression as ".*"), then followed by "Aug".−

Here, we are using the ***-i*** option to have case insensitive search −

```
$ls -l | grep -i "carol.*aug"
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

The sort Command

The **sort** command arranges lines of text alphabetically or numerically. The following example sorts the lines in the food file −

```
$sort food
Afghani Cuisine
Bangkok Wok
Big Apple Deli
Isle of Java

Mandalay
Sushi and Sashimi
Sweet Tooth
Tio Pepe's Peppers
$
```

The **sort** command arranges lines of text alphabetically by default. There are many options that control the sorting −

Sr.No.	Description
1	**-n** Sorts numerically (example: 10 will sort after 2), ignores blanks and tabs.
2	**-r** Reverses the order of sort.
3	**-f** Sorts upper and lowercase together.
4	**+x** Ignores first **x** fields when sorting.

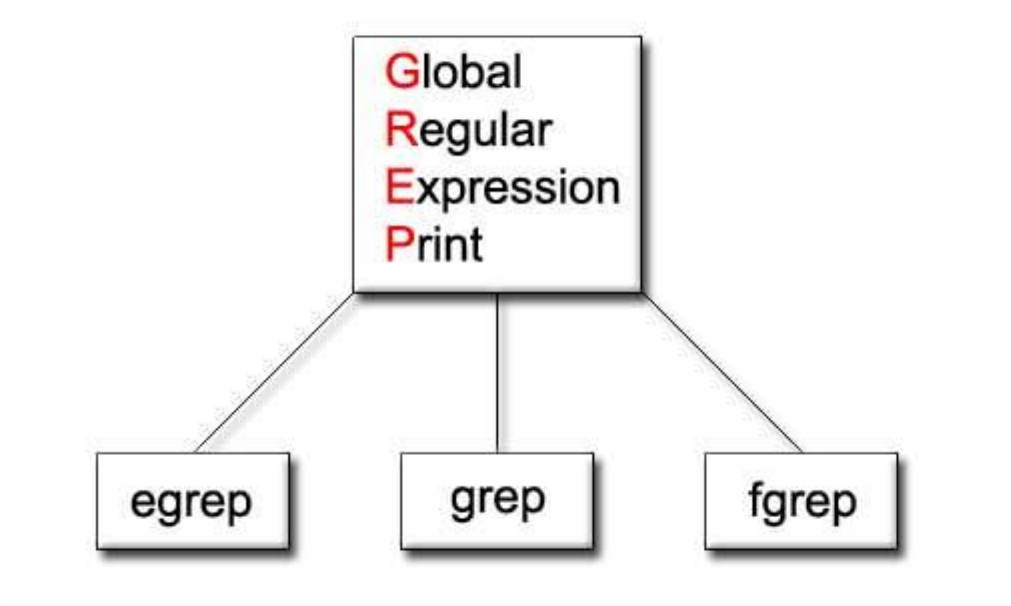
More than two commands may be linked up into a pipe. Taking a previous pipe example using **grep**, we can further sort the files modified in August by the order of size.

The following pipe consists of the commands **ls**, **grep**, and **sort** −

```
$ls -l | grep "Aug" | sort +4n
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-rw-  1 john  doc     11008 Aug  6 14:10 ch02
$
```

This pipe sorts all files in your directory modified in August by the order of size, and prints them on the terminal screen. The sort option +4n skips four fields (fields are separated by blanks) then sorts the lines in numeric order.

The Grep Family
In the simplest terms, grep (**g**lobal **r**egular **e**xpression **p**rint) is a small family of commands that search input files for a search string, and print the lines that match it.
Grep is made up of three separate, yet connected commands, grep, egrep, and fgrep, a sort of holy trinity of Unix commands.
All three of the grep commands work the same way. Beginning at the first line in the file, grep copies a line into a buffer, compares it against the search string, and if the comparison passes, prints the line to the screen.
Grep will repeat this process until the file runs out of lines. Notice that nowhere in this process does grep store lines, change lines, or search only a part of a line.



A Simple Example

he simplest possible example of grep is simply:

```
grep "boot" a_file
```

In this example, grep would loop through every line of the file "a_file" and print out every line that contains the text "boot." To see this command in action, you will need to provide a file for grep to process. You may either create your own, or if you wish to follow along with this tutorial, you can fetch my example file from this wesite by typing the following command at the command prompt

The file's contents are:

```
boot
book
booze
machine
boots
bungie
bark
aardvark
broken$tuff
robots
```

The file is not particulary interesting, but it gives us something to test our commands with. When you are ready to proceed, try the following command

```
grep "boo" a_file
```

Grep will list all of the lines that contain the word 'boo':

```
boot
book
booze
boots
```

Useful Options

This is nice, but if you were working with a large c file of something similar, it would probably be much more useful to you if the lines identified which line in the file they were, what way you could track down a particular string more easily, if you needed to open the file in an editor to make some changes. This can be accomplished by ading the -n parameter:

```
grep -n "boo" a_file
```

This yeilds a much more useful result, which explains which lines matched the search string:

```
1:boot
2:book
3:booze
5:boots
```

Another interesting switch is -v, which will print the negative result. In other words, grep will print all of the lines that do not match the search string, rather than printing the lines that match it. In the following case, grep will print every line that does not contain the string "boo," and will display the line numbers, as in the last example

```
grep -vn "boo" a_file
```

In this particular case, it will print

```
4:machine
6:bungie
7:bark
8:aaradvark
9:robots
```

The -c option tells grep to supress the printing of matching lines, and only display the number of lines that match the query. For instance, the following will print the number 4, because there are 4 occurences of "boo" in a_file.

```
grep -c "boo" a_file
4
```

The -l option prints only the filenames of files in the query that have lines that match the search string. This is useful if you are searching through multiple files for the same string. like so:

```
grep -l "boo" *
```

An option more useful for searching through non-code files is -i, ignore case. This option will treat upper and lower case as equivalent while matching the search string. In the following example, the lines containg "boo" will be printed out, even though the search string is uppercase.

```
grep -i "BOO" a_file
```

The -x option looks for eXact matches only. In other words, the following command will print nothing, because there are no lines that only contain the pattern "boo"

```
grep -x "boo" a_file
```

Finally, -f allows you to specify a file containing the search string, one instance where this could be useful is if one had a complex search string that one may not want to type over and over again.

```
echo "i want to search for this text" > search
grep -f search a_file
```

Regular Expressions

Since grep is named the "global regular expression print" it's not surprising that grep can also search for regular expressions in addition to normal strings. Regular expressions are searched for in the same way a normal string is. In fact, the strings we entered before were just very simple regular expressions. If you are unfamiliar with regular expressions, this page provides an excellent tutorial. The following command will search the file for lines ending with the letter e:

```
grep "e$" a_file
```

This will, of course, print

```
booze
machine
bungie
```

egrep

While grep supports a handful of regular expression commands, it does not support certain useful sequences such as the + and ? operators. If you would like to use these, you will have to use extended grep (egrep). Egrep is equivalent to grep -E, but as it is fairly common to want the extended functionality, egrep is also its own separate command.The Following command illustrates the ?, which matches 1 or 0 occurences of the previous character

```
grep "boots?" a_file
```

This query will return

```
boot
boots
```

One of the more powerful constructs that egrep supports that grep does not is the pipe (|), wich funcitons as an "or." another way I could get the same result as above with a different query is:

```
egrep "boot|boots"
```

fgrep

Fgrep is the third member of the grep family. It stands for "fast grep" and for good reason. Fgrep is faster than other grep commands because it does not interpret regluar expressions, it only searches for strings of literal characters. Fgrep is equivalent to grep -F. If one fgreped for boot|boots, rather than interpreting that as a search for either the word boot or the word boots, frep would simply search for the literal string "boot|boots" in the file. For instance, with normal grep the following command would search for lines ending with the word "broken"

```
fgrep "broken$" a_file
```

However, we can see that with fgrep, it will return the line "broken$tuff" because it is not interpreting the dollar sign, only the entire string as literal characters. It is a good practice to use fgrep instead of grep for situations like these.

Examples

Now that we have skimmed over the basic funtions of the commands in the grep family, we can look at a few examples of more advanced functionality. The following example is an example of grepping through the output of another program rather than a file. This particualar example will print out the files that find returns that contain the text "hello" (although this could be done without using grep at all)

```
find | grep "hello"
```

Normally, grep does not have a way to search through portions of files, but when the file is first processed by another program, this is possible. This example performs a grep on the last 8 lines of a_file

```
tail -n8 a_file | grep "boo"
```

By using the exec switch with the find command, we can find files that contain the search string. The following will search for the string "boo" in every directory below the current directory

```
find . -exec grep "boo" {} \;
```

grep is the only command of the three that supports backreferences and saving. The following uses backreferences to find lines that contain two of the same lowercase letter in succession.

```
grep "\([a-z]\)\1" a_file
```

Sed Command in Linux/Unix with examples

SED command in UNIX is stands for stream editor and it can perform lot’s of function on file like, searching, find and replace, insertion or deletion. Though most common use of SED command in UNIX is for substitution or for find and replace. By using SED you can edit files even without opening it, which is much quicker way to find and replace something in file, than first opening that file in VI Editor and then changing it.

- SED is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).

- SED command in unix supports regular expression which allows it perform complex pattern matching.

Syntax:

sed OPTIONS... [SCRIPT] [INPUTFILE...]

Example:
Consider the below text file as an input.

$cat > geekfile.txt
```
unix is great os. unix is opensource. unix is free os.
learn operating system.
unix linux which one you choose.
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

Sample Commands

1. **Replacing or substituting string :** Sed command is mostly used to replace the text in a file. The below simple sed command replaces the word "unix" with "linux" in the file.
2. **$sed 's/unix/linux/' geekfile.txt**

   Output :

   ```
   linux is great os. unix is opensource. unix is free os.
   learn operating system.
   linux linux which one you choose.
   linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
   ```

   Here the "s" specifies the substitution operation. The "/" are delimiters. The "unix" is the search pattern and the "linux" is the replacement string.

   By default, the sed command replaces the first occurrence of the pattern in each line and it won't replace the second, third…occurrence in the line.

3. **Replacing the nth occurrence of a pattern in a line :** Use the /1, /2 etc flags to replace the first, second occurrence of a pattern in a line. The below command replaces the second occurrence of the word "unix" with "linux" in a line.
4. **$sed 's/unix/linux/2' geekfile.txt**

   Output:

   ```
   unix is great os. linux is opensource. unix is free os.
   ```

```
learn operating system.
unix linux which one you choose.
unix is easy to learn.linux is a multiuser os.Learn unix .unix is a powerful.
```

5. **Replacing all the occurrence of the pattern in a line :** The substitute flag /g (global replacement) specifies the sed command to replace all the occurrences of the string in the line.
6. **$sed 's/unix/linux/g' geekfile.txt**

   Output :

   ```
   linux is great os. linux is opensource. linux is free os.
   learn operating system.
   linux linux which one you choose.
   linux is easy to learn.linux is a multiuser os.Learn linux .linux is a powerful.
   ```

7. **Replacing from nth occurrence to all occurrences in a line :** Use the combination of /1, /2 etc and /g to replace all the patterns from the nth occurrence of a pattern in a line. The following sed command replaces the third, fourth, fifth… “unix” word with “linux” word in a line.
8. **$sed 's/unix/linux/3g' geekfile.txt**

   Output:

   ```
   unix is great os. unix is opensource. linux is free os.
   learn operating system.
   unix linux which one you choose.
   unix is easy to learn.unix is a multiuser os.Learn linux .linux is a powerful.
   ```

9. **Parenthesize first character of each word :** This sed example prints the first character of every word in paranthesis.
10. **$ echo "Welcome To The Geek Stuff" | sed 's/\(\b[A-Z]\)/\(\1\)/g'**

    Output:

    ```
    (W)elcome (T)o (T)he (G)eek (S)tuff
    ```

11. **Replacing string on a specific line number :** You can restrict the sed command to replace the string on a specific line number. An example is
12. **$sed '3 s/unix/linux/' geekfile.txt**

    Output:

    ```
    unix is great os. unix is opensource. unix is free os.
    learn operating system.
    linux linux which one you choose.
    ```

```
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

The above sed command replaces the string only on the third line.

13. **Duplicating the replaced line with /p flag :** The /p print flag prints the replaced line twice on the terminal. If a line does not have the search pattern and is not replaced, then the /p prints that line only once.
14. **$sed 's/unix/linux/p' geekfile.txt**

    Output:

```
linux is great os. unix is opensource. unix is free os.
linux is great os. unix is opensource. unix is free os.
learn operating system.
linux linux which one you choose.
linux linux which one you choose.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

15. **Printing only the replaced lines :** Use the -n option along with the /p print flag to display only the replaced lines. Here the -n option suppresses the duplicate rows generated by the /p flag and prints the replaced lines only one time.
16. **$sed -n 's/unix/linux/p' geekfile.txt**

    Output:

```
linux is great os. unix is opensource. unix is free os.
linux linux which one you choose.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

    If you use -n alone without /p, then the sed does not print anything.

17. **Replacing string on a range of lines :** You can specify a range of line numbers to the sed command for replacing a string.
18. **$sed '1,3 s/unix/linux/' geekfile.txt**

    Output:

```
linux is great os. unix is opensource. unix is free os.
learn operating system.
linux linux which one you choose.
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

    Here the sed command replaces the lines with range from 1 to 3. Another example is

    $sed '2,$ s/unix/linux/' geekfile.txt

Output:

```
unix is great os. unix is opensource. unix is free os.
learn operating system.
linux linux which one you choose.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful
```

Here $ indicates the last line in the file. So the sed command replaces the text from second line to last line in the file.

19. **Deleting lines from a particular file :** SED command can also be used for deleting lines from a particular file. SED command is used for performing deletion operation without even opening the file
    Examples:
    1. To Delete a particular line say n in this example
20. Syntax:
21. $ sed 'nd' filename.txt
22. Example:
23. $ sed '5d' filename.txt

2. To Delete a last line

Syntax:
$ sed '$d' filename.txt

3. To Delete line from range x to y

Syntax:
$ sed 'x,yd' filename.txt
Example:
$ sed '3,6d' filename.txt

5. To Delete from nth to last line

Syntax:
$ sed 'nth,$d' filename.txt
Example:
$ sed '12,$d' filename.txt

6. To Delete pattern matching line

Syntax:
$ sed '/pattern/d' filename.txt
Example:
$ sed '/abc/d' filename.txt

AWK command in Unix/Linux with examples

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. Spliting 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
```