COMMUNICATIONS:

UNIX provides a rather rich set of communication tools. Even with all of its capabilities, if you see email extensively, you will most likely want to use a commercial product to handle your mail.

The communication utilities in UNIX are: ***talk, write, mail, Telnet, FTP***

USER COMMUNICATION:

The first two communication utilities are **talk** and **write,** deal with communications between two users at different terminals.

talk command:

The **talk** command allows two UNIX users to chart with each other just like you might do on the phone except that you type rather than talk. When one user wants to talk to another, he or she simply types the command **talk** and the others person's login id. The command format is shown below:

talk options user_id terminal

The conversation doesn't begin however until the called user answers. When you send a request to talk, the user you are calling gets a message saying that you want to talk. The message is shown here:

```
Message from Talk_Daemon@challenger at 18:07 . . .
talk: connection requested by gilberg@challenger.atc.fhda.edu.
talk: respond with: talk gilberg@challenger.atc.fhda.edu.
```

It your friend doesn't want to talk, he or she can ignore the message, much like you don't have to answer the phone when it rings. However UNIX is persistent, it will keep repeating the message so that the person you are calling has to respond.

There are two possible responses. The first which agrees to accept the call is a corresponding **talk** command to connect to you. This response is seen in the third line of the preceding example. Note that all that is needed is the user id. The address starting with the at sing (@) is not required if the person calling is on the same system.

To refuse to talk, the person being called must turn messages off. This is done with the message (**mesg**) command as shown below:

mesg n

The message command has one argument, either y, yes I want to receive messages, or n, no I don't want to receive messages. It is normally set to receive messages when you log in to the system. Once you turn it off, it remains off until you turn it back on or restart. To turn it on, set it to yes as follow:

mesg y

To determine the current message status key **mesg** with no parameters as shown in the example below (the response is either yes (y) or no (n)):

```
$ mesg
is n
```

when you try to talk with someone who has messages turned off, you get the following message:

[Your party is refusing messages]

After you enter the talk response the screen is split in to two portions. In the upper portion which represents your half of the conversation, you will see a message that the person you are calling is being notified that you want to talk. This message is:

[Waiting for your party to respond]

Once the connection has been made you both see a message saying that the connection has been established. You can then begin talking. What you type is shown on the top half of the screen. Whatever you type is immediately shown in the bottom half of your friend's screen.

write command:

The **write** command is used to send a message to another terminal. Like **talk**, it requires that the receiving party be logged on. The major difference between write and talk is that write is one way transmission. There is no split screen, and the person you are communicating with does not see the message as it is being typed. Rather it is sent one line at a time; that is the text is collected until you key Enter, and then it is all sent at once.

You can type as many lines as you need to complete your message. You terminate the message with either an end of file (ctrl+d) or a cancel command (ctrl+c). When you terminate the message, the recipient receives the last line and end of transmission (<EOT>) to indicate that the transmission is complete. The format for the **write** is shown below:

write options user_id terminal

When you write to another user, a message is sent telling him or her that you are about to send a message. A typical message follows. It shows the name of the sender, the system the message is coming from, the sender's terminal id, and the date and time.

```
Message from Joan on sys (ttyq1)     [Thu Apr 9 21:21:25]
```

Unless you are very quick, the user can quickly turn your message off by keying *mesg n*. when this happens, you get the following error when you try to send your message:

```
Can no longer write to /dev/ttyq2
```

If you try to write to a user who is not logged on, you get the following error message:

```
dick is not logged on.
```

Sometimes a user is logged on to more than one session. In this case UNIX warns you that he or she is logged on multiple times and shows you all of the alternate sessions. A sample of this message is:

```
Joan is logged on more than one place.
You are connected to "ttyq1".
Other locations are:
        ttyq2
```

ELECTRONIC MAIL:

Although talk and write are fastest ways to communicate with someone who's online, it doesn't work it they're not logged on. Similarly if you need to send a message to someone who's on a different operating system, they don't work.

There are many different email systems in use today. While UNIX does not have all of the capabilities of many of the shareware and commercial products, it does offer a good basic system.

A mail message is composed of two parts: the header and the body. The **body** contains the text of the message. The **header** consists of the subject, the addressees (To:), sender (From:), a list of open copy recipients (Cc:) and a list of blind copy recipients (Bcc:).

In addition to the four basic header entries there are seven extended fields:

1) Reply-To: if your do not specify a reply-to-address, your UNIX address will automatically be used.

2) Return-Receipt-To: if the addressee's mail system supports return recipiets, message will be sent to the address in return-receipt-to when the message is delivered to the addressee.

3) In-Reply-To: this is a text string that you may use to tell the user you are replying to a specific note. Only one line is allowed. It is displayed as header data when the addressee opens the mail.

4) References: when you want to put a reference, such as to a previous memo sent by the recipient you can include a references field.

5) Keywords: Provides a list of keywords to the context of the message.

6) Comments: allows you to place a comment at the beginning of the message.

7) Encrypted: Message is encrypted.

All of these data print as a part of the header information at the top of the message. A final word of caution on extended header information: Not all email systems support all of them.

Mail Addresses:

Just as with snail mail, to send email to someone, you must know his or her address. When you send mail to people on your own system, their address is their user id. So Joan can send mail to Tran using his id, *tran*. This is possible because *tran* is a local username or alias and UNIX knows the address for everyone on it.

To send mail to people on other systems, however you need to know not only their local address (user id) but also their domain address. The local address and domain address are separated by an at sign (@) as shown below:

Local Address@Domain Address

Local Address:

The local address identifies a specific user in a local mail system. It can be user id, a login name, or an alias. All of them refer to the user mailbox in the directory system. The local address is created by the system administrator or the postmaster.

Domain Address:

The domain address is a hierarchical path with the highest level on the right and the lowest level on the left. There must be at least two levels in the domain address.

The parts of the address are separated by dots (periods). The highest level is an internet label. When an organization or other entity joins the internet, it selects its internet label from one of the two label designations: generic domain labels or country domain labels. Below the domain label, the organization (system administrator), controls the hierarchical breakdown, with each level is referring to a physical or logical organization of computers or services under its control.

Generic Domains:

Most Internet domains are drawn from the generic domain levels, also known as top-level domains. These generic names represent the basic classification structure for the organizations in the grouping (table given below).

Label	Description
com	Commercial (business) profit organizations
edu	Educational institutions (college or university)
gov	Government organizations at any level.
int	International organizations.
mil	Military organizations
net	Network support organizations such as ISP's (internet service providers)
org	Nonprofit organizations

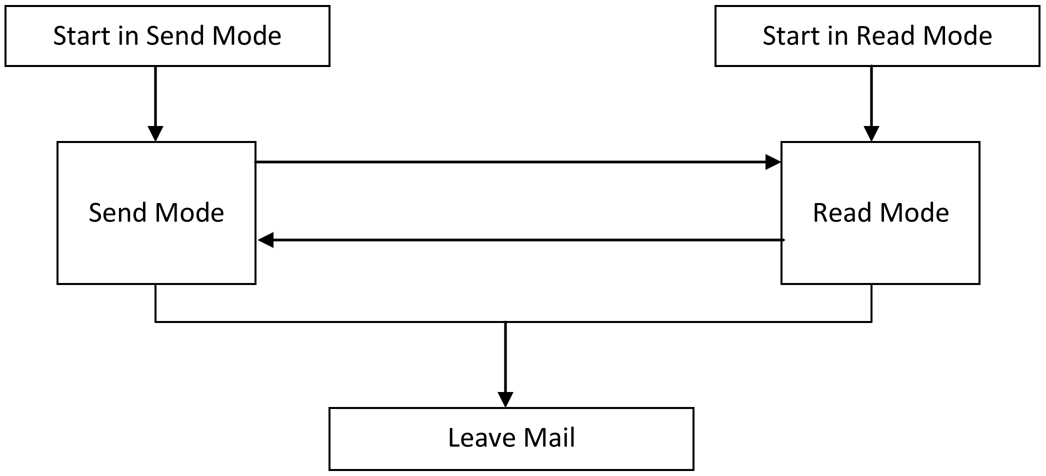
Country Domain:

In the country domain, the domain label is a two-character designation that identifies the country in which the organization resides. While the rest of the domain address varies, it often uses a governmental breakdown such as state and city. A typical country domain address might look like the following address for an internet provider in California:

yourInternet.ca.us

Mail Mode:

When you enter, the mail system, you are either in the send mode or the read mode. When you are in the send mode, you can switch to the read mode to process your incoming mail; likewise, when you are in the read mode, you can switch to the send mode to answer someone's mail. You can switch back and forward as often as necessary, but you can be in only one mode at a time. The basic mail system operation appears in the figure as follows:



mail Command:

The mail command is used to both read and send mail. It contains a limited text editor for composing mail notes.

Send Mail:

To send mail from the system prompt, you use the mail command with one or more addresses as shown below:

mail options address(es)

example: $ **mail tran, dilbert, harry, carolyn@atc.com**

When your message is complete, it is placed in the mail system spool where the email system can process it. If you are sending the message to more than one person, their names may be separated by commas or simply by spaces.

If the person you are sending mail to is not on your system, you need to include the full email address. The last addressee (carolyn@atc.com) is a person on a different system. *The mail system provides a very limited text editor for you to write you note. It does not text wrap, and once you have gone to the next line, you cannot correct previous lines without invoking a text editor.*

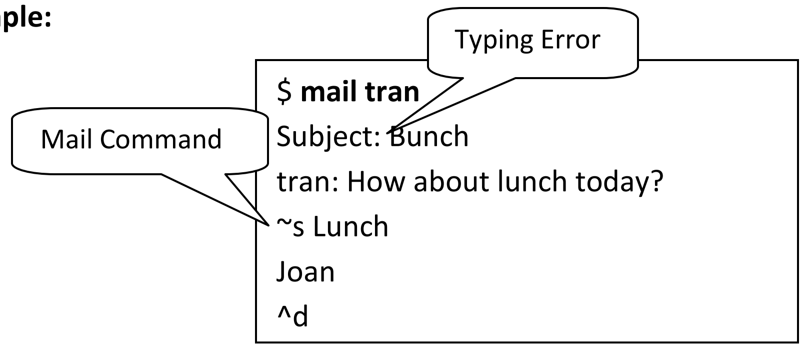
To invoke the editor use the edit message command (~e). This will place you in your standard editor such as vi.

Once in the editor you can use any of its features. When you are through composing or correcting your message, the standard save and exit command will take you back to your send mail session.

Send Mail Commands:

The mail utility program allows you to enter mail commands as well as text. It assumes that you are entering text. To enter a mail command, you must be at the beginning of the line; a mail command in the middle of a line is considered text. When the first character you enter on the line is a tilde (~), the mail utility interprets the line as a command.

Example:



```
$ mail tran
Subject: Bunch
tran: How about lunch today?
~s Lunch
Joan
^d
```

In the above figure, Joan makes a typing mistake and doesn't notice it until she is ready to send the message. Fortunately there is mail command to change the subject line. So she simply types ***~s Lunch*** at the beginning of a line, and when Tran receives the message, it has a correct subject line. The complete list of send mail commands is given the table below:

Command	**Action**
~~	Quotes a single tilde.
~b users	Appends users to bcc ("blind" cc) header field.
~cm text	Appends users to cc header field.
~d	Reads in dead.letter.
~E or ~eh	edits the entire message
~e	edits the message body
~en text	Puts text in encrypted header field.
~f messages	Reads in messages.
~H	Prompts for all possible header fields.
~h	Prompts for important header fields.
~irt text	Appends text to in-response-to header field.
~k text	Appends text to keywords header field.
~m messages	Reads in messages, right shifted by a tab.
~p	prints the message buffer
~q	quits: do not send
~r file	reads a file in to the message buffer
~rf text	Appends text to references header field.
~rr	Toggles Return-Receipt-To header field
~rt users	Appends users to Reply-To header field
~s text	Puts text in subject header field.
~t users	Appends users to To header field.

Quit mail and write command:

To quit the mail utility, you enter ctrl+d. If you are creating a message, it is automatically sent when you enter ctrl+d. If you are not ready to send the message and must still quit, you can save it in a file. To write the messages to a file, you use the **write** command (~w). The name of the file follows the command and once the file has been written, you exit the system using the **quit** command (~q) as shown in the example below:

```
~w noteToTran
q
```

To quit without saving file you use the quit command. In truth the quit command saves your mail in a special dead-letter file known as ***dead.letter***. You can retrieve this file and continue working on it at a later date just like any other piece of mail you may have saved.

Reloading Files: The Read File Command:

There are two basic reasons for reading a file in to a message. First if you have saved a file and want to continue working on it, you must reload it to the send mail buffer. The second reason is to include a standard message or a copy of a note you received from someone else.

To copy a file in to a note, open mail in the send mode as normal and then load the saved file using the read file command (~r). When you load the file, however the file is not immediately loaded. Rather the mail utility simply makes a note that the file is to be included.

Example:

```
$ mail tran
Subject: Lunch
~r note-to-tran
"note-to-tran" 3/52
```

To see the note and continue working on it, you use the print command (~p). This command reprints the mail buffer, expanding any files so that you can read their contents. If the first part of the message is okay, you can simply continue entering the text.

If you want to revise any of your previous work, then you need to use an editor. To start an editor use either the start editor (~e) or start vi command (~v). The start editor command starts the editor designated for mail.

If none is specified, vi is automatically used. When you close the editor you are automatically back in the mail utility and can then send the message or continue entering more text.

Distribution Lists:

Distribution Lists are special mail files that can contain aliases for one or more mail addresses. For example you can create a mail alias for a friend using only his or her first name. Then you don't need to key a long address that may contain a cryptic user id rather than a given name. Similarly if you are working on a project with 15 others, you can create a mail distribution list and use it rather than typing everyone's address separately.

Distribution lists are kept in a special mail file known as *.mailrc*. Each entry in the distribution list file begins with alias and designates one or more mail addresses. Because all entries start with the word alias, they are known as alias entries.

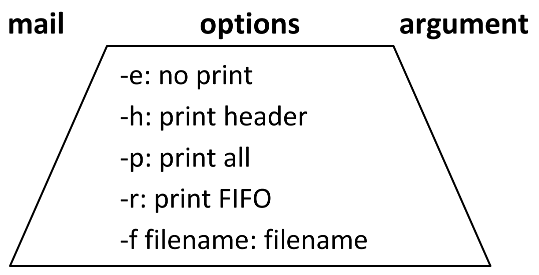
To add an alias entry to the distribution list you must edit it and insert the alias line in the following format:

alias project joan jxc5936 tran bill@testing.atc.com

In this example the alias name is "project". Within project we have included four mail addresses. The first three are for people on our system as indicated by the local names. The last is for a person on a different system.

Read Mode:

The read mode is used to read your mail. Its operation is presented below:



Although the mail system is a standard utility its implementation is anything but standard. You enter mail in the read mode by simply keying mail at the command line with-out any parameters (addresses). If you don't have any mail a no-mail message is printed and you stay at the command line. A typical no-mail response is as follows:

```
$ mail
No mail for tran
```

Reading New Mail: If you receive notice that mail has arrived while you are in the read mail mode, you will not be able to read it until you redisplay the mail list.

If you use the headers command to reprint the list, you will still not see the new mail. To reprint the list with the new header, you must use the folder command (folder %) at the mail prompt.

The folder command may be abbreviated fo %. Note however that this command has the same effect as quitting mail and reentering. For example, if you had deleted any mail, you will not be able to undelete it.

Replying to and Forwarding Mail

After you have read a piece of mail, you can reply to it. There are two reply commands. The first command (R) replies only to the sender. The second command (r) replies to the sender and all of the addressees. Since these two commands are so close, you want to get into the habit of reading the To list when you use reply.

When you use reply, you will see the header information for the To and Subject fields. If you want to add other header information, you must use the tilde commands described in send mail. **Note:** & is the read mail prompt.

Example:

```
& r
To: joan
Subject: re: Project Review
```

Quitting:

Quit terminates mail. There are three ways to quit. From the read mail prompt, you can enter the quit command (q), the quit and do not delete command (x), or the end of transmission command (ctrl+d). All undeleted messages are saved in your mailbox file.

Deleted messages, provided they weren't undeleted, are removed when you use quit (q) and end (ctrl+d); they are not deleted when you use do not delete (x). If new mail has arrived while you were in mail, an appropriate message is displayed.

Saving Messages:

To keep you mail organized, you can save messages in different files. For example all project messages can be saved in a project file, and all status reports can be kept in another file. To save a message, use the command (s or s#). The simple save command saves the current message in the designated file as in the following example:

```
s project
```

If the file project in this example doesn't exist, it will be created. If it does exist, the current message is appended to the end of the file. An alternative to saving the message is to write it to a file. This is done with the write command (w), which also requires a file. The only difference between a save and write is that the first header line in the mail is not written; the message body of each written message is appended to the specified file.

To read messages in a saved file, start mail with the file option on the command prompt as shown next. Note that there is no space between the option and the filename.

```
$ mail -fproject
```

Deleting and Undeleting Mail:

Mail remains in your mailbox until you delete it. There are three ways to delete mail. After you have read mail, you may simply key ***d*** at the mail prompt, and it is deleted. Actually anytime you key ***d*** at the mail prompt, the current mail entry is deleted. After you read mail, the mail you just read remains the current mail entry until you move to another message.

If you are sure of the message number you can use it with delete command. For example to delete message 5, simply key ***d5*** at the mail prompt. You can also delete a range of messages by keying a range after the delete command. All three of these delete formats are shown below:

```
& d          # Deletes current mail entry
& d5         # Deletes entry 5 only
& d5..17     # Deletes entries 5 through 17
```

You can undelete a message as long as you do it before you exit mail or use the folder command. To undelete a message you use the undelete command (u) and the message number. To undelete message 5, key u5. You cannot undelete multiple messages with one command.

Read Mail Commands:

The complete list of read mail commands are shown in the table below:

Mail Command	Explanation
t <message list>	Types messages.
n	Goes to and types next message.
e <message list>	Edits messages.
d <message list>	Deletes messages.

Mail Command	Explanation
folder % or fo %	Reprints mail list including any mail that arrived after start of read mail session. Has the effect of quitting read mail and reentering.
s <message list> file	Appends messages to file.
u <message list>	Undelete messages
R <message list>	Replies to message senders
r <message list>	Replies to message senders and all recipients
pre <message list>	Makes messages go back to incoming mail file.
m <user list>	Mails to specific users.
h <message list>	Prints out active message headers
q	Quits, saving unresolved messages in mailbox.
x	Quits, do not remove from incoming mail file.
w <number> filename	Appends body of specified message number to filename.
!	Shell escape.
cd [directory]	Changes to directory or home if none given.

Read Mail Options:

There are five read mail options of interest. They are shown in the table below:

Option	Usage
-e	Does not print messages when mail starts.
-h	Displays message header list and prompt for response on start.
-p	Prints all messages on start
-r	Prints messages in first in, first out (FIFO) order.
-f file_name	Opens alternate mail file (file_name).

Mail Files:

UNIX defines two sets of files for storing mail: arriving mail and read mail.

Arriving Mail Files:

The system stores mail in a designated file when it arrives. The absolute path to the user's mail file is stored in the MAIL variable. As it arrives incoming mail is appended to this file. a typical mail path is in the next example:

```
$ echo $MAIL
/usr/mail/forouzan
```

The mail utility checks the incoming mail file periodically. When it detects that new mail has arrived, it informs the user. The time between mail checks is determined by a system variable, MAILCHECK. The default period is 600 seconds (10 minutes). To change the time between the mail checks we assign a new value, in seconds, to MAILCHECK as follows:

$ **MAILCHECK = 300**

Read Mail File:

When mail has been read, but not deleted, it is stored in the mbox file. As mail is read, it is deleted from the incoming mail file and moved to the mbox file. This file is normally stored in the user's home directory.

REMOTE ACCESS:

telnet → termina**l net**work

The telnet Concept:

The telnet utility is a TCP/IP standard for the exchange of data between computer systems. The main task of telnet is to provide remote services for users. For example we need to be able to run different application programs at a remote site and create results that can be transferred to our local site.

One way to satisfy these demands is to create different client/server application programs for each desired service. Programs such as file transfer programs and email are already available. But it would be impossible to write a specific client/server program for each requirement.

The better solution is a general-purpose client/server program that lets a user access any application program on a remote computer; in other words, it allows the user to log into a remote computer. After logging on, a user can use the services available on the remote computer and transfer the results back to the local computer.

Time-Sharing Environment:

Designed at a time when most operating systems such as UNIX, were operating in a time-sharing environment, telnet is the standard under which internet systems interconnect. In a time sharing environment a large computer supports multiple users. The interaction between a user and the computer occurs through a terminal, which is usually a combination of keyboard, monitor and mouse. Our personal computer can simulate a terminal with a terminal emulator program.

Login:

In a time sharing environment, users are part of the system with some right to access resources. Each authorized user has identification and probably a password. The user identification defines the user as a part of the system. To access the system, the user logs into the system with a user id or login name. The system also facilitates password checking to prevent an unauthorized user from accessing the resources.

Local Login: When we log in to a local time sharing system, it is called local login. As we type at a terminal or a work station running a terminal emulator, the keystrokes are accepted by the terminal driver. The terminal driver passes the characters to the operating system. The operating system in turn interprets the combination of characters and invokes the desired application program or utility.

Remote Login: When we access an application program or utility located on a remote machine, we must still login only this time it is a remote login. Here the telnet client and server programs come into use. We send the keystrokes to the terminal driver where the local (client) operating system accepts the characters but does not interpret them. The characters are sent to the client telnet interface, which transforms the characters to a universal character set called **Network Virtual Terminal** (NVT) characters and then sends them to the server using the networking protocols software.

The commands or text, in NVT form, travel through the Internet and arrive at the remote system. Here the characters are delivered to the operating system and passed to the telnet server, which changes the characters to the corresponding characters understandable by the remote computer.

However the characters cannot be passed directly to the operating system because the remote operating system is not designed to receive characters from a telnet server: It is designed to receive characters from a terminal driver. The solution is to add a piece of software called a pseudo terminal driver, which pretends that the characters are coming from a terminal. The operating system then passes the characters to the appropriate application program.

Connecting to the Remote Host:

To connect to a remote system, we enter the telnet command at the command line. Once the command has been entered, we are in the telnet system as indicated by the telnet prompt. To connect to a remote system, we enter the domain address for the system. When the connection is made, the remote system presents its login message. After we log in, we can use the remote system as though it were in the same room. When we complete our processing, we log out and are returned to our local system.

There are several telnet subcommands available. The more common ones are listed in table given below:

Command	Meaning
open	Connects to a remote computer
close	Closes the connection
display	Shows the operating parameters
mode	Changes to line mode or character mode
Set	Sets the operating parameters
status	Displays the status information
send	Sends special characters
quit	Exits telnet
?	The help command. telnet displays its command list

FILE TRANSFER:

Whenever a file is transferred from a client to a server, a server to a client or between two servers, a transfer utility is used. In UNIX the ftp utility is used to transfer files.

The ftp Command:

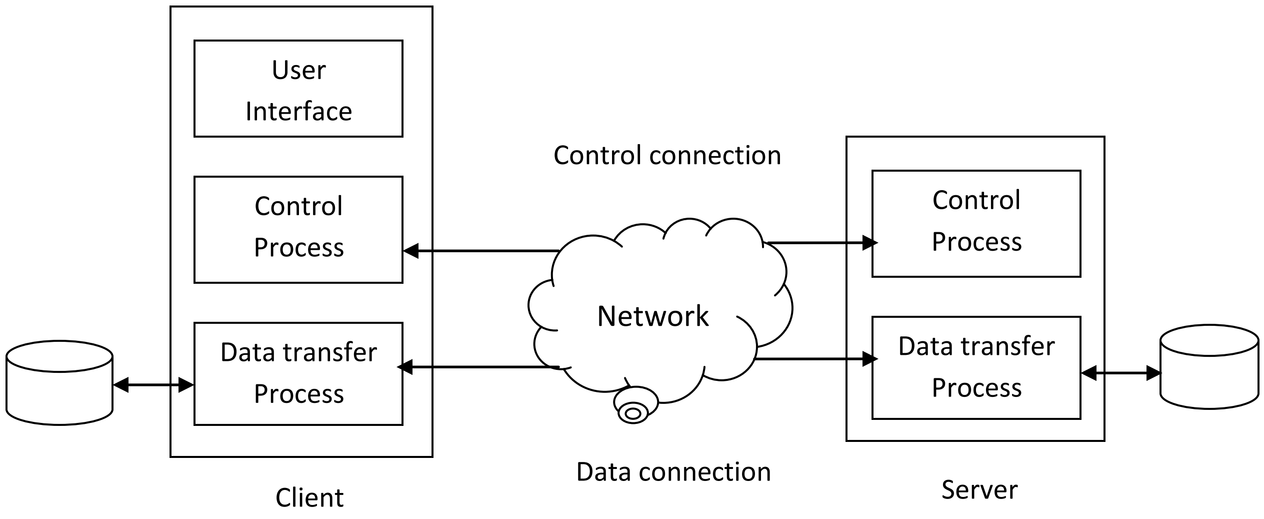
File Transfer Protocol (ftp) is a TCP/IP standard for copying a file from one computer to another. Transferring files from one computer to another is one of the most common tasks expected from a networking or internetworking environment.

The ftp protocol differs from other client-server applications in that it establishes two connections between the hosts. One connection is used for data transfer, the other for control information (commands and responses). Separation of commands and data transfer makes ftp more efficient.

The control connection uses very simple rules of communication. We need to transfer only a line of command or a line of response at a time. The data connection, on the other hand, needs more complex rules due to the variety of data types transferred.

The following figure shows the basic ftp mode. The client has three components: user interface, client control process, and the client data transfer process. The server has two components: the server control process and the server data transfer process. The control connection is made between the control processes. The data connection is made between the data transfer processes.

The control connection remains connected during the entire interactive ftp session. The data connection is opened and then closed for each file transferred. It opens each time a file transfer command is used, and it closes when the file has been transferred.



Establishing ftp Connection:

To establish an ftp connection, we enter the ftp command with the remote systems domain name on the prompt line. As the alternative, we can start the ftp session without naming the remote system. In this case we must open the remote system to establish a connection. Within ftp, the open command requires the remote system domain name to make the connection.

Closing an ftp connection:

At the end of the session, we must close the connection. The connection can be closed in two ways: to close the connection and terminate ftp, we use quit. After the connection is terminated, we are in the command line mode as indicated by the command prompt ($).

Example for Terminate the ftp session:

```
ftp> quit
221 Goodbye.
$
```

To close the connection and leave the ftp active so that we can connect to another system, we use close. We verify that we are still in an ftp session by the ftp prompt. At this point, we could open a new connection.

Example for close the ftp session:

```
ftp> quit
221 Goodbye.
ftp>
```

Transferring Files:

Typically files may be transferred from the local system to the remote system or from the remote system to the local system. Some systems only allow files to be copied from them; for security reasons they do not allow files to be written to them.

There are two commands to transfer files: **get** and **put**. Both of these commands are made in reference to the local system. Therefore **get** copies a file from the remote system to the local system, whereas **put** writes a file from the local system to the remote system.

When we ftp a file, we must either be in the correct directories or use a file path to locate and place the file. The directory can be changed on the remote system by using the change directory (**cd**) command within ftp.

There are several commands that let us change the remote file directory. For example we can change a directory, create a directory, and remove a directory. We can also list the remote directory. These commands work just like their counterparts in UNIX. A complete list of ftp commands is shown in table given below:

!	debug	mdir	pwd	size
$	dir	mget	quit	status
account	direct	mkdir	quote	struct
append	disconnect	mls	recv	sunique
ascii	form	mode	reget	system
bell	get	modtime	rename	tenex
binary	glob	mput	reset	trace
bye	hash	newer	restart	type
case	help	nlist	rhelp	umask
cd	idle	nmap	rmdir	user
cdup	image	ntrans	rstatus	verbose
chmod	lcd	open	ruinque	win
close	ls	prompt	send	?
cr	macdef	proxy	sendport	
Delete	mdelete	put	site	

vi EDITOR:

The **vi** editor is the interactive part of **vi/ex**. When initially entered, the text fills the buffer, and one screen is displayed. If the file is not large enough to fill the screen, the empty lines below text on the screen will be identified with a tilde (~) at the beginning of each line. The last line on the file is a **status line;** the status line is also used to enter **ex** commands.

Commands:

Commands are the basic editing tools in **vi.** As a general rule, commands are case sensitive. This means that a lowercase command and its corresponding uppercase command are different, although usually related.

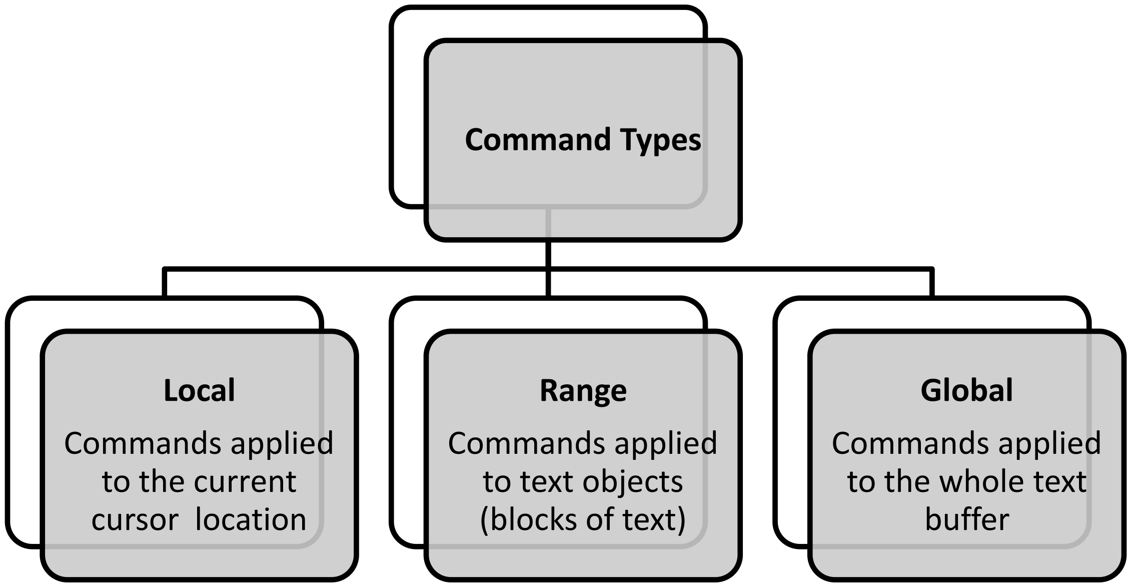
For example the lower case insert command (i) inserts before the cursor, whereas the uppercase insert command (I) inserts at the beginning of the line.

One of the things that most bothers new vi users is that the command is not seen. When we key the insert command, we are automatically in the insert mode, but there is no indication on the screen that anything has changed.

Another problem for new users is that commands do not require a Return key to operate. Generally command keys are what are known as hot keys, which means that they are effective as soon as they are pressed.

COMMAND CATEGORIES:

The commands in vi are divided in to three separate categories: local commands, range commands, and global commands as shown in figure below. **Local commands** are applied to the text at the current cursor. The **range commands** are applied to blocks of text are known as text objects. **Global commands** are applied to all of the text in the whole buffer, as contrasted to the current window.


Command Types
Local
Commands applied to the current cursor location
Range
Commands applied to text objects (blocks of text)
Global
Commands applied to the whole text buffer

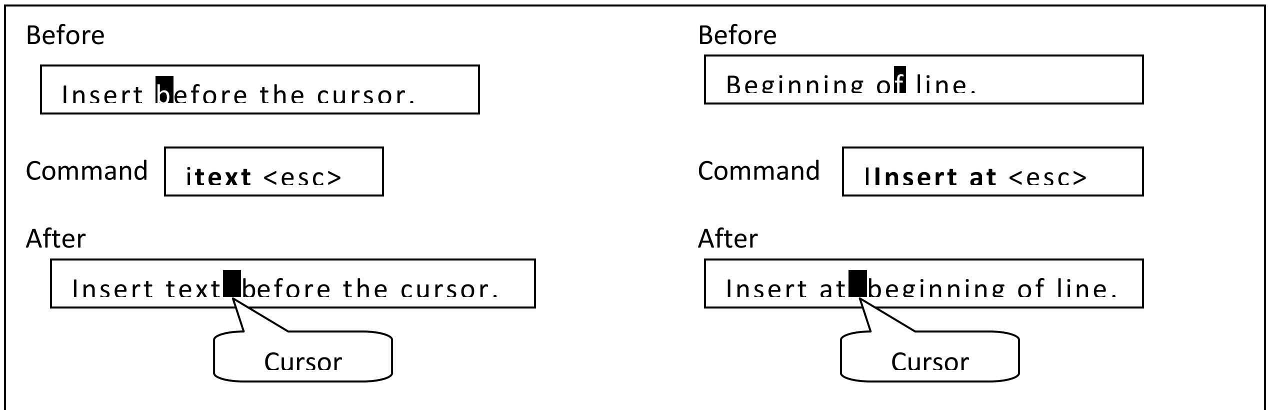
LOCAL COMMANDS IN vi

Local commands are commands are applied to the text relative to the cursors current position. We call the character at the cursor the current character, and the line that contains the cursor is the current line. The local commands are discussed below:

Insert Text Commands (i, I)

We can insert text before the current position or at the beginning of the current line. The lowercase insert command (i) changes to the text mode. We can then enter text. The character at the cursor is pushed down the line as the new text is inserted. When we are through entering text, we return to the command mode by keying *esc*.

The uppercase insert (I) opens the beginning of the current line for inserting text. The following figure shows the examples of character insertion.



(a) Insert Before Cursor (b) Insert at Beginning

Append Text Commands (a, A)

The basic concept behind all append commands is to add the text after a specified location. In vi, we can append after the current character (a) or after the current line (A). The following figure demonstrates the append command.

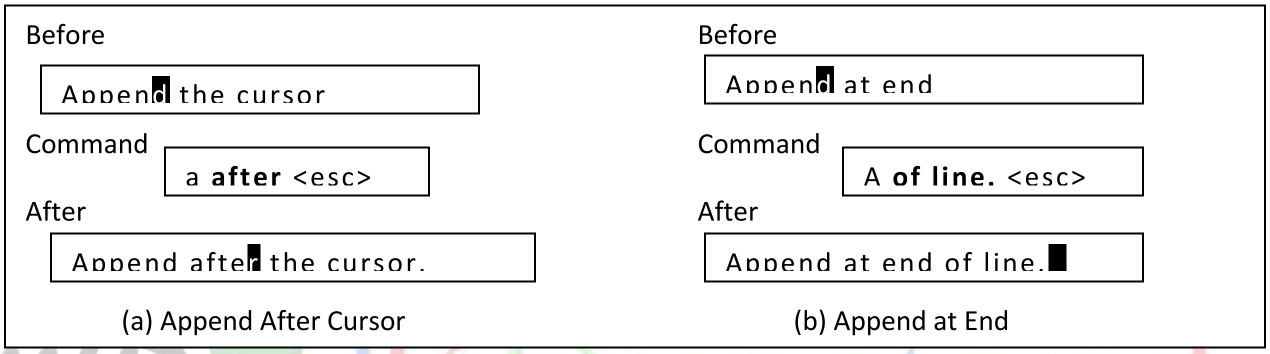
Newline Commands (o, O)

The newline command creates an open line in the text file and inserts the text provided with the command. Like insert and append, after entering the text, we must use the escape key to return to the command mode. To add the text in a new line below the current line, use the lowercase newline command (o). To add the new text in a line above the current line, use the uppercase newline command (O). The following figure demonstrates the newline command.

Replace Text Commands (r, R)

When we replace text, we can only change text in the document. We cannot insert the text nor can we delete old text. The lowercase replace command (r) replaces a single character with another character and immediately returns to command mode. It is not necessary to use the Escape key. With the uppercase replace command (R), on the other hand, we can replace as many characters as necessary. Every keystroke will replace one character in the file. To return to the command mode in this case, we need to use the escape key. The following figure demonstrates the replace command.

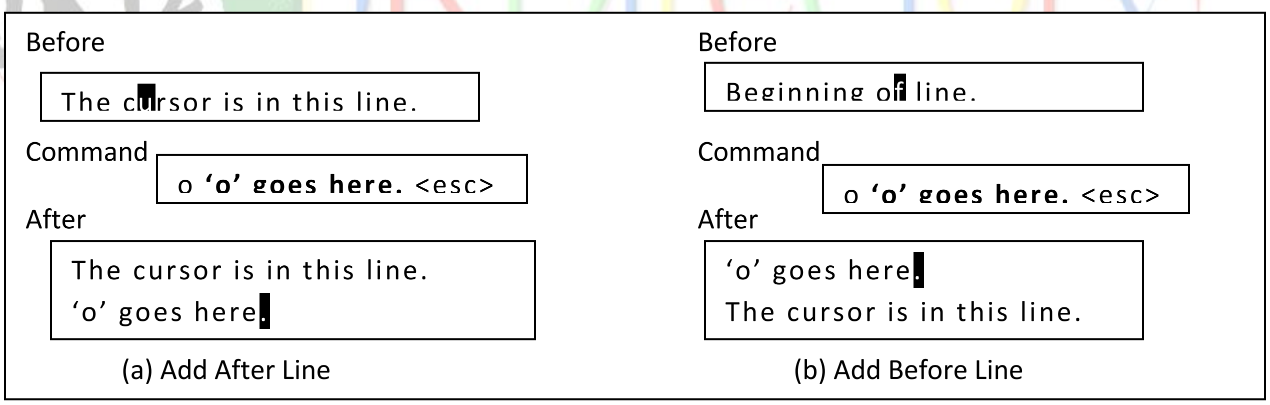
Note: At the end of both the commands the cursor is on the last character replaced.



Before
```
Append the cursor
```
Command
```
a after <esc>
```
After
```
Append after the cursor.
```
(a) Append After Cursor

Before
```
Append at end
```
Command
```
A of line. <esc>
```
After
```
Append at end of line.
```
(b) Append at End

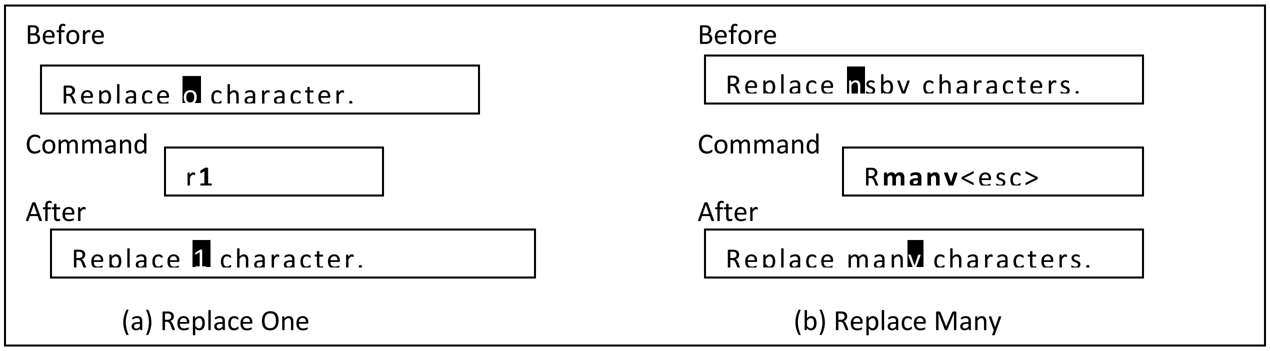
Figure: Append text after current character/line



Before
```
The cursor is in this line.
```
Command
```
o 'o' goes here. <esc>
```
After
```
The cursor is in this line.
'o' goes here.
```
(a) Add After Line

Before
```
Beginning of line.
```
Command
```
O 'O' goes here. <esc>
```
After
```
'O' goes here.
The cursor is in this line.
```
(b) Add Before Line

Figure: Newline Command



Before
```
Replace o character.
```
Command
```
r1
```
After
```
Replace 1 character.
```
(a) Replace One

Before
```
Replace nsby characters.
```
Command
```
Rmany<esc>
```
After
```
Replace many characters.
```
(b) Replace Many

Figure: Replace Commands

Substitute Text Commands (s, S)

Whereas the replace text command is limited to the current characters in the file, the lowercase substitute (s) replaces one character with one or more characters. It is especially useful to correct spelling errors. The uppercase substitute command (S) replaces the entire current line with new text. Both substitute commands require an Escape to return to the command mode.

Delete Character Commands (x, X)

The delete character command deletes the current character or the character before the current character. To delete the current character, use the lowercase delete command (x). To delete the character before the cursor, use the uppercase delete command (X). After the character has been deleted, we are still in the command mode, so no Escape is necessary.

Mark Text Command (m)

Marking text places an electronic "finger" in the text so that we can return to it whenever we need to. It is used primarily to mark a position to be used later with a range command. The mark command (m) requires a letter that becomes the name of the marked location.

The letter is invisible and can be seen only by vi. Marked locations in the file are erased when the file is closed; they are valid only for the current session and are not permanent locations.

Change Case Command (~)

The change case command is the tilde (~). It changes the current character from upper- to lowercase or from lower-to uppercase. Only one character can be changed at a time, but the cursor location is advanced so that multiple characters can easily be changed. It can also be used with the repeat modifier to change the case of multiple characters. Nonalphabetic characters are unaffected by the command.

Put Command (p, P)

In a word processor, the put command would be called "paste". As with many other commands, there are two versions, a lowercase p and an uppercase P. The lowercase put copies the contents of the temporary buffer after the cursor position. The uppercase put copies the buffer before the cursor. The exact placement of the text depends on the type of data in the buffer. If the buffer contains a character or a word, the buffer contents are placed before or after the cursor in the current line.

If the buffer contains a line, sentence, or paragraph, it is placed on the previous line or the next line.

Join Command (J)

Two lines can be combined using the join command (J). The command can be used anywhere in the first line. After the two lines have been joined, the cursor will be at the end of the first line.

RANGE COMMANDS IN vi

A range command operates on a text object. The vi editor consider the whole buffer as a long stream of characters of which only a portion may be visible in the screen window. Because range commands can have targets that are above or below the current window, they can affect unseen text and must be used with great care.

There are four range commands in vi: move cursor, delete, change, and yank. The object of each command is a text object and defines the scope of the command. Because the range commands operate on text objects, we begin our discussion of range commands with text objects.

Text Object:

A text object is a section of text between the two points: the cursor and a target. The object can extend from the cursor back toward the beginning of the document or forward toward the end of the document. The target is a designator that identifies the end of the object being defined.

The object definitions follow a general pattern. When the object is defined before the cursor, it starts with the character to the left of the cursor and extends to the beginning of the object. When the object is defined after the cursor, it begins with the cursor character and extends to the end of the object.

Object ranges:

To left:	Starts with character on left of cursor to the beginning of the range.
To right:	Starts with cursor character to the end of range.

There are seven general classes of objects: character, word, sentence, line, paragraph, block, and file. All text objects start with the cursor.

Character Object: A character object consists of only one character. This means that the beginning and end of the object are the same. To target the character immediately before the cursor, we use the designator h. To target the current character, we use the designator l.

Word Object: A word is defined as a series of non whitespace characters terminated by character. A word object may be a whole word or a part of a word depending on the current character location (cursor) and the object designator. There are three word designators.

The designator b means to go backward to the beginning of the current word; if the cursor is at the beginning of the current word, it moves to the beginning of the previous word.

The designator w means to go forward to the beginning of the next word, including the space between the words if any.

The designator e means to go forward to the end of the current word; it does not include the space.

Line Object: A line is all of the text beginning with the first character after a newline to the next newline. A line object may consist of one or more lines. There are six line designators:

To target the beginning of the current line, use 0 (zero)

To target the end of the line, use $

To target the beginning of the line above the current line, use – (minus)

To target the beginning of the next line, user +.

To target the character immediately above the current character, use k.

To target the character immediately below the current character, use j.

Sentence Object: A sentence is a range of text that ends in a period or a question mark followed by two spaces or a new line. This means that there can be many lines in one sentence. A sentence designator defines a sentence or a part of a sentence as an object. There are two sentence designators 1) **(** and 2) **)**. To remember the sentence targets, just think of a sentence enclosed in parenthesis.

The open parenthesis is found at the beginning of the sentence; it selects the text from the character immediately before the cursor to the beginning of the sentence. The close parenthesis selects the text beginning from and including the cursor to the end of the sentence, which includes the two spaces that delimit it. Both sentence objects work consistently with all operations.

Paragraph Object: A paragraph is a range of text starting with the first character in the file buffer or the first character after a blank line (a line consisting only of a new line) to the next blank line or end to the buffer. Paragraph may contain one or more sentences. There are two paragraph designators 1) **}** and 2) **{**. To remember the paragraph targets, think of them as enclosed in braces, which are bigger than parenthesis.

The left brace targets the beginning of the paragraph including the blank line above it. The right brace targets the end of the paragraph but does not include the blank line.

Block Object: A block is a range of text identified by a marker. It is the largest range object and can span multiple sentences and paragraphs. In fact, it can be used to span the entire file. The two block designators are `a and ‘a.

Screen Objects: We can define the part of the text or the whole screen as a screen object. There are three simple screen cursor moves:

H:	Moves the cursor to the beginning of the text line at the top of the screen.
L:	Moves the cursor to the beginning of the text line at the bottom of the screen.
M:	Moves the cursor to the beginning of the line at the middle of the screen.

File Object: There is one more target, the file. To move to the beginning of the last line in the file object, or to create a range using it, use the designator G.

Text Object Commands:

Four vi commands are used with the text objects: move cursor, delete, change and yank.

The range commands share a common syntactical format that includes the command and three modifiers. The format is shown below:

repeat command **command** repeat text object text object

none:	move cursor
d:	delete
c:	change
y:	yank

Yank command: The yank command copies the current text object and places it in a temporary buffer. It parallels the delete command except that the text is left in the current position. The typical use of the yank command is the first step in a copy and paste operation. After the text has been placed in the temporary buffer, we move to an insert location and then use the put command to copy the text from the buffer to the file buffer.

Global Commands in vi:

Global commands are applied to the edit buffer without reference to the current position.

Scroll Commands:

Scroll commands affect the part of the edit buffer (file) that is currently in the window. They are summarized in the table given below:

Command	Function
ctrl + y	Scrolls up one line.
ctrl + e	Scrolls down one line.
ctrl + u	Scrolls up half a screen.
ctrl + d	Scroll down half a screen.
ctrl + b	Scrolls up whole screen.
ctrl + f	Scrolls down whole screen.

Undo Commands:

The vi editor provides a limited undo facility consisting of two undo commands: undo the most recent change (u) and restore all changes to the current line (U).

Repeat Command:

We can use the dot command (.) to repeat the previous command. For example, instead of using the delete line (dd) several times, we can use it once and then use the dot command to delete the remaining lines. The dot command can be used with all three types of vi commands: local, range and global.

Screen Regeneration Commands:

There are times when the screen buffer can become cluttered and unreadable. For example, if in the middle of an edit session someone sends you a message; the message replaces some of the text on the screen but not in the file buffer. At other times, we simply want to reposition the text. Four vi commands are used to regenerate or refresh the screen. These four commands are summarized in the table given below:

Command	Function
z return	Regenerates screen and positions current line at top.
z.	Regenerates screen and positions current line at middle.
z-	Regenerates screen and positions current line at bottom.
ctrl + L	Regenerates screen without moving cursor.

Display Document Status Line:

To display the current document status in the status line at the bottom of the screen, we use the status command (ctrl + G). The status fields across the line are the filename, a flag indicating if the file has been modified or not, the position of the cursor in the file buffer, and the current line position as a percentage of the lines in the file buffer.

For example, if we are at line 36 in our file, *TheRaven*, we would see the following status when we enter the ctrl + G:

"TheRaven" [Modified] line 36 of 109 - - 33%- -

When we are done with the edit session, we quit vi. To quit and save the file, we use the vi zz command. If we don't want to save the file, we must use ex's quit and don't save command, q!

Rearrange Text in vi:

Most word processors have cut, copy, and paste commands. vi's equivalents use delete and yank commands to insert the text in to a buffer and then a put command to paste it back to the file buffer. In this section we will see how to use these commands to move and copy text.

Move Text:

When we move the text, it is deleted from its original location in the file buffer and then put in to its new location. This is the classic cut and paste operation in a word processor. There are three steps required to move text:

1. Delete text using the appropriate delete command, such as delete paragraph.
2. Move the cursor to the position where the text is to be displayed.
3. Use the appropriate put command (p or P) to copy the text from the buffer to the file buffer.

In the following example, we move three lines starting with the current line to the end of the file buffer. To delete the lines, we use the delete line command with a repeat object modifier. We then position the cursor at the end of the file buffer and copy the text from the buffer.

```
3dd     # Delete three lines to buffer

G       # Move to end of file buffer

p       # put after current line
```

Copy Text:

When we copy text, the original text is left in the file buffer and also put (pasted) in a different location in the buffer. There are two ways to copy text. The preferred method is to yank the text. Yank leaves the original text in place. We then move the cursor to the copy location and put the text. The steps are as follows:

1. Yank text using yank block, y.
2. Move cursor to copy location.
3. Put the text using p or P.

The second method is to delete the text to be copied, followed immediately with a put. Because the put does not empty the buffer, the text is still available. Then move to the copy location and put again. The following code yanks a text block and then puts it in a new location:

Move cursor to the beginning of the block

```
ma      # set mark 'a'
        # manually move cursor to the end of the block
y`a     # yank block 'a' leaving it in place
        # manually move cursor to copy location
p       # put text in new location
```

Named Buffers:

The vi editor uses one temporary buffer and 35 named buffers. The text in all buffers can be retrieved using the put command. However, the data in the temporary buffer are lost whenever a command other than a position command is used. This means that a delete followed by an insert empties the temporary buffer.

Then named buffers are available until their contents are replaced. The first time named buffers are known as the numeric buffers because they are identified by the digits 1 through 9. The remaining 26 named buffers are known as the alphabetic buffers; they are identified by the lower case letters a through z.

To retrieve the data in the temporary buffer, we use the basic put command. To retrieve the text in a named buffer, we preface the put command with a double quote, and the buffer name as shown in the following command syntax. ***Note that there is no space between the double quote, the buffer name, and the put command.***

“buffer-namep

Using this syntax, the following three examples would retrieve the text in the temporary buffer, buffer 5 and buffer k.

P “5p “kP

Numeric Named Buffers:

The numeric named buffers are used automatically whenever sentence, line, paragraph, screen, or file text is deleted. The deleted text is automatically copied to the first buffer (1) and is available for later reference. The deleted text for the previous delete can be retrieved by using either the put command or by retrieving numeric buffer 1.

Each delete is automatically copied to buffer 1. Before it is copied, buffer 8 is copied to buffer 9, buffer 7 is copied to buffer 8, and so forth until buffer 1 has been copied to buffer 2. The current delete is then placed in buffer 1. This means that at any time the last nine deletes are available for retrieval.

If a repeat modifier is used, then all of the text for the delete is placed in the buffer 1. In the following example, the next three lines are considered as one delete and are placed in buffer 1: 3dd

Alphabetic Named Buffers:

The alphabetic named buffers are to save up to 26 text entities. Whereas the numeric buffers can only store text objects that are at least a sentence long, we can store any size object in an alphabetic buffer. Any of the delete or yank commands can be used to copy data to an alphabetic buffer.

To use the alphabetic buffers, we must specify the buffer name in the delete or yank command. As previously stated, the buffer names are the alphabetic characters a through z. The buffer name is specified with a double quote followed immediately by the buffer name and the yank command as shown below:

“ad # Delete and store line in ‘a’ buffer
“my # Yank and store line in ‘m’ buffer

Once the text has been stored, it is retrieved in the same way we retrieved data from the numeric buffers. To retrieve the two text objects created in the previous example, we would enter:

“mp # Retrieve line in ‘m’ buffer after current cursor

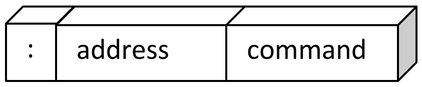
“ap # Retrieve line in ‘a’ buffer before current cursor

Overview of ex Editor:

The ex editor is a line editor. When we start ex, we are given its colon prompt at the bottom of the screen. Each ex instruction operates on one specific line. ex is the integral part of the vi editor. We can start in either of these editors, and from each, we can move to the other.

ex Instruction Format:

An ex instruction is identified by its token, the colon (:). Following the token is an address and a command. The ex instruction format is shown in figure below:



Addresses:

Every line in the file has a specific address starting with one for the first line in the file. An ex address can be either a single line or a range of lines.

Single Line:

A single line address defines the address for one line. The five single line address formats are discussed below:

Current line: The current line, represented by a period (.), is the last line processed by the previous operation. When the file is opened, the last line read into the buffer (i.e. the last line in the file) is the current line. In addition to the period, if no address is specified, the current line is used.

Top of Buffer: The top of the buffer, represented by zero (0), is a special line designation that sets the current line to the line before the first line in the buffer. It is used to insert lines before the first line in the buffer.

Last Line: The last line of the buffer is represented by the dollar sign ($).

Line Number: Any number within the line range of the buffer (1 to $) is the address of that line.

Set-of-Line Addresses:

There are two pattern formats that can be used to search for a single line. If the pattern search is to start with the current line and search forward in the buffer – that is, toward the end of the buffer – the pattern is enclosed in slashes (/ . . . /). If the search is to start with the current line and move backward – that is, toward the beginning of the buffer (1) – the pattern is enclosed in question marks (?).

Note however in both searches, if the pattern is not found, the search wraps around to completely search the buffer. If we start forward search at line 5, and the only matching line is in line 4, the search will proceed from line 5 through line $ and then restart at line 1 and search until it finds the match at line 4.

The following table summarizes these two addresses:

Address	Search Direction
/pattern/	Forward
?pattern?	Backward

Range Addresses:

A range address is used to define a block of consecutive lines. If has three formats as shown in the table given below:

Address	Range
%	Whole file
address1, address2	From address1 to address2 (inclusive)
address1;address2	From address1 to address2 relative to address1 (inclusive)

The last two addresses are similar; the only difference is one uses a comma separator and one uses a semicolon. To see the differences, therefore, we need to understand the syntactical meaning of the comma and the semicolon separators.

Comma Separator: When the comma is used, the current address is unchanged until the complete instruction has been executed. Any address relative to the current line is therefore relative to the current line when the instruction is given.

Semicolon Separator: When we use a semicolon, the first address is determined before the second address is calculated. It sets the current address to the address determined by the first address.

Commands: There are many ex commands. Some of the basic commands that are commonly used are given in the following table:

Command	Description
d	Delete
co	Copy
m	Move
r	Read
w	Write to file
y	Yank

Command	Description
p	Print (display)
pu	Put
vi	Move to vi
w filename	Write file and continue editing
s	Substitute
q	Quit
q!	Quit and do not save
wq	Write and quit
x	Write file and exit

Delete Command (d): The delete command (d) can be used to delete a line or range of lines. The following example deletes lines 13 through 15:

```
:13,15d
```

Copy Command (co): The copy command (co) can be used to copy a single line or a range of lines after a specified line in the file. The original text is left in place. The following examples copy lines 10 through 20 to the beginning of the file (0), the end of the file ($), and after line 50:

```
:10,20co 0
:10,20co $
:10,20c0 50
```

Move Command (m): The format of the move command (m) is the same as the copy command. In a move, however, the original text is deleted.

Read and Write Commands (r, w): The read command (r) transfers lines from a file to the editor's buffer. Lines are read after a single line or a set of lines identified by a pattern. The write command (w file_name), on the other hand, can write the whole buffer or a range within the buffer to a file. All address ranges (except nested range) are valid.

```
$ ex file1
"file1" 5 lines, 10 characters
:r file2
"file2" 6 lines, 330 characters
:w TempFile
"TempFile" [New file] 11 lines, 340 characters
:q
$ ls –l TempFile
-rw-r- -r- -     1     gilberg     staff   340   Oct   18    18:16  TempFile
```

Print Command (p): The print command (p) displays the current line or a range of lines on the monitor.

Move to vi Command (vi): The move to vi command (vi) switches the editor to vi. The cursor is placed at the current line, which will be at the top of the monitor. Once in vi, however, you will remain in it as though you had started it. You will be able to execute ex command, but after each command, you will automatically return to vi.

Substitute Command (s): The Substitute command (s) allows us to modify a part of line or a range of lines. Using substitute, we can add text to a line, delete text, or change text. The format of the substitute command is:

addresss/pattern/replacement-string/flag

Quit Command (q, q!, wq): The ex quit command (q) exits the editor and returns to the UNIX command line. If the file has been changed, however, it presents an error message and returns to the ex prompt. At this point we have two choices. We can write the file and quit (wq), or we can tell ex to discard the changes (q!).

Exit Command (x): The exit command (x) also terminates the editor. If the file has been modified, it is automatically written and the editor terminated. If for any reason, such as write permission is not set, the file cannot be written, the exit fails and the ex prompt is displayed.

ATOMS AND OPERATORS:

A regular expression is a pattern consisting of a sequence of characters i.e. matched against text. A regular expression is like a mathematical expression. A mathematical expression is made of operands (data) and operators. Similarly, a regular expression is made of atoms and operators.

The **atom** specifies what we are looking for and where in the text the match is to be made. The **operator,** which is not required in all expressions, combines atoms into complex expressions.

ATOMS: An atom in a regular expression can be one of the five types: a single character, a dot, a class, an anchor, or a back reference.

Single Character: The simplest atom is a single character. When a single character appears in a regular expression, it matches itself. In other words, if a regular expression is made of one single character, that character must be somewhere in the text to make the pattern match successful.

Dot: A dot matches any single character except the newline character (\n). This universal matching capability makes it a very powerful element in the operation of regular expressions. By itself, however, it can do nothing because if matches everything. Its power in regular expressions comes from its ability to work with other atoms to create an expression.

For example consider the following example:

a.

This expression combines the single-character atom, a, with the dot atom. It matches any pair of characters where the first character is a. Therefore, it matches aa, ah, ab, ax, and a5, but it does not match Aa.

Class: The class atom defines a set of ASCII characters, any one of which may match any of the characters in the text. The character set to be used in the matching process is enclosed in brackets. The class set is a very powerful expression component. Its power is extended with three additional tokens: ranges, exclusion, and escape characters. A range of text characters is indicated by a dash (-). Thus the expression [a-d] indicates that the characters a and b and c and d all included in the set.

Sometimes it is easier to specify which characters are to be excluded from the set – that is to specify its complement. This can be done using exclusion, which is the UNIX *not* operator (^). For example to specify any character other than a vowel, we would use [^aeiou].

The third additional token is the escape character (\). It is used when the matching character is one of the other two tokens. For example to match a vowel or a dash, we would use the escape character to indicate that the dash is a character and not a range token. This example is coded as [aeiou\-].

Anchors: Anchors are atoms that are used to line up the pattern with a particular part of a string. In other words anchors are not matched to the text, but define where the next character in the pattern must be located in the text. There are four types of anchors: beginning of line (^), end of line ($), beginning of word (\<), and end of word (\>).

Anchors are another atom that is often used in combinations. For example, to locate the string that begins with letter Q, we would use the expression ^Q. similarly to find a word that ends in g, we would code the expression as g\>.

Back references: We can temporarily save text in one of the nine save buffers. When we do we refer the text in a saved buffer using a back reference. A back reference is coded using the escape character and a digit in the range of 1 to 9 as shown below:

\1 \2 . . . \9

A back reference is used to match text in the current or designated buffer with text that has been saved in one of the systems nine buffers.

OPERATORS:

To make the regular expressions more powerful, we can combine atoms with operators. The regular expression operators play the same role as mathematical operators. Mathematical expression operators combine mathematical atoms (data); regular expression operators combine regular expression atoms.

We can group the regular expressions in to five different categories: *sequence operators, alternation operators, repetition operators, group operators,* and *save operators.*

The **sequence** operator is *nothing*. This means that if a series of atoms such as a series of characters are shown in a regular expression, it is implied that there is an invisible sequence operator between them. Examples of sequence operators are shown below:

Dog	➔	matches the pattern “Dog”
a . . b	➔	matches “a”, any two characters, and “b”
[2 – 4] [0 – 9]	➔	matches a number between 20 and 49
[0 – 9] [0 – 9]	➔	matches any two digits
`$	➔	matches a blank line
^.$	➔	matches a one-character line
[0 – 9] – [0 – 9]	➔	matches two digits separated by a “–“

The **Alternation** operator (|) is used to define one or more alternatives. For example if we want to select between A or B, we would code the regular expression as A | B. Alternation can be used with single atoms, but it is usually used for selecting between two or more sequences of characters or groups of characters. That is, the atoms are usually sequences.

For single alternation we suggest that you use the class operator. An example of alternation among sequences is presented in example below:

UNIX | unix	➔	Matches “UNIX” or “unix”
Ms|Miss|Mrs	➔	Matches “Ms” or “Miss” or “Mrs”

The **repetition** operator is a set of escaped braces (\ { … \ }) that contains two numbers separated by a comma. It specifies that the atom or expression immediately before the repetition may be repeated. The first number (m) indicates the minimum required times the previous atom must appear in the text.

The second number (n) indicates the maximum number of times it may appear. For example \ { 2, 5 \ } indicates that the previous atom may be repeated two to five times.

Example:

A\ {3, 5\}	➔	matches “AAA”, “AAAA”, or “AAAAA”
BA\ {3, 5\}	➔	matches “BAAA”, “BAAAA”, or “BAAAAA”

Basic Repetition Forms:

The m and n values are optional, although at least one must be present. That is either may appear without the other. If only one repetition value (m) is enclosed in the braces, the previous atom must be repeated exactly m times – no more, no less. E.g. \ {3 \}.

If the minimum value (m) is followed by a comma without a maximum value, the previous atom must be present at least m times, but it may appear more than m times. In the following example the previous atom may be repeated three or more times but no less than three times. E.g. \ {3, \}.

If the maximum value (n) is preceded by a comma without a minimum, the previous atom may appear up to n times and no more. In the following example the previous atom may appear zero to three times, but no more. E.g. \ {, 3\}

Short Form Operators:

Three forms of repetition are so common that UNIX has special shortcut operators for them. The asterisk (*) may be used to repeat an atom zero or more times. (It is same as \ {0 , \}). The plus (+) is used to specify that the atom must appear one or more times. (It is same as \ {1, \}). The question mark (?) is used to repeat the pattern zero or one time only. (It is same as \ {0, 1\}).

The **group** operator is a pair of opening and closing parentheses. When a group of characters is enclosed in parentheses, the next operator applies to the whole group, not only to the previous character. In the following example, the group (BC) must be repeated exactly three times.

A(BC)\ {3 \}	➔	matches ABCBCBC

The **save** operator which is a set of escaped parentheses, \ (… \), copies a matched text string to one of the nine buffers for later reference. Within an expression, the first saved text is copied to buffer 1, the second saved text is copied to buffer 2 and so forth for up to nine buffers. Once text has been saved, it can be referred to by using a back reference.

GREP FAMILY AND OPERATIONS:

The command **grep** stands for **g**lobal **r**egular **e**xpression **p**rint. It is the family of programs that is used to search the input file for all lines that match a specified regular expression and write them to the standard output file (monitor). The format of grep is shown below:

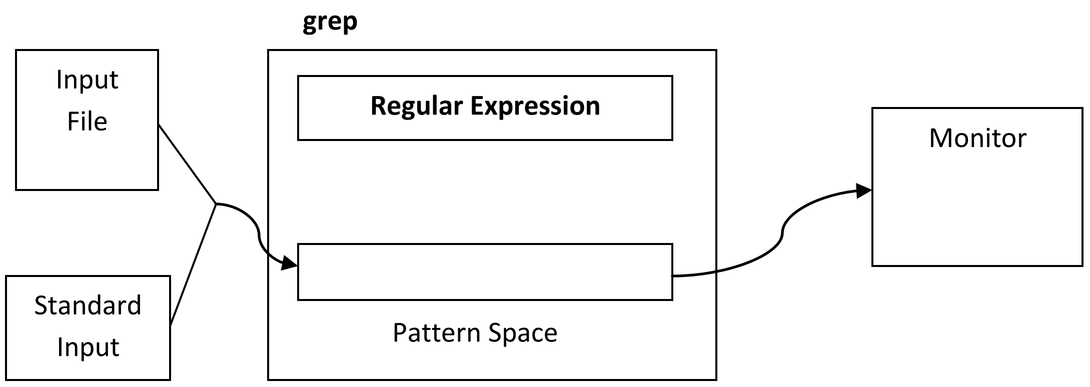
grep *options* **regexp** **filelist**

Options:

-b:	print block numbers
-c:	print only match count
-i:	ignore upper-/lowercase
-l:	print files with at least one match
-n:	print line numbers
-s:	silent mode; no output
-v:	print lines that do not match
-x:	print only lines that match
-f file:	expressions are in file

OPERATION:

To write scripts that operate correctly, you must understand how the grep utilities work. We begin, therefore with a short explanation of how they work.



For each line in the standard input (input file or keyboard), grep performs the following operations:

1. Copies the next input line into the pattern space. The pattern space is a buffer that can hold only one text line.

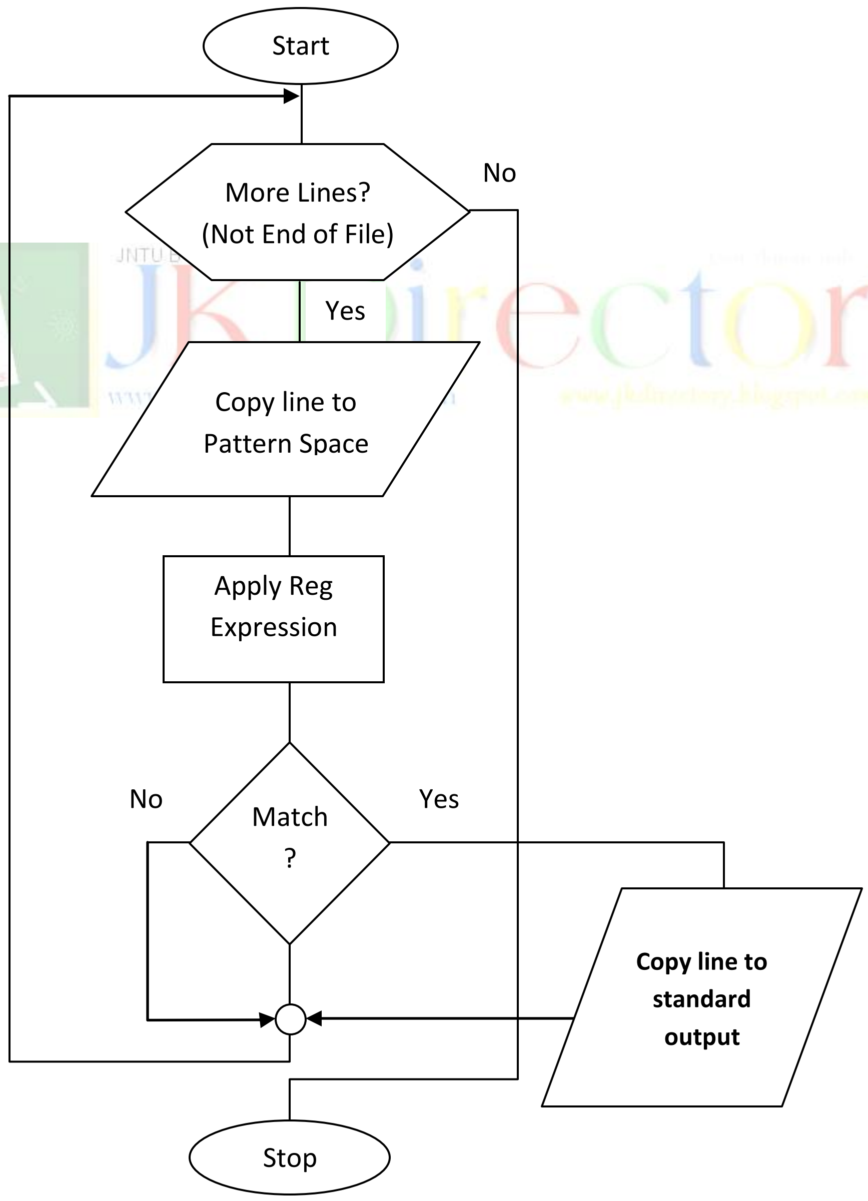
2. Applies the regular expression to the pattern space.

3. If there is a match, the line is copied from the pattern space to the standard output.

The grep utilities repeat these three operations on each line in the input.

grep flowchart:

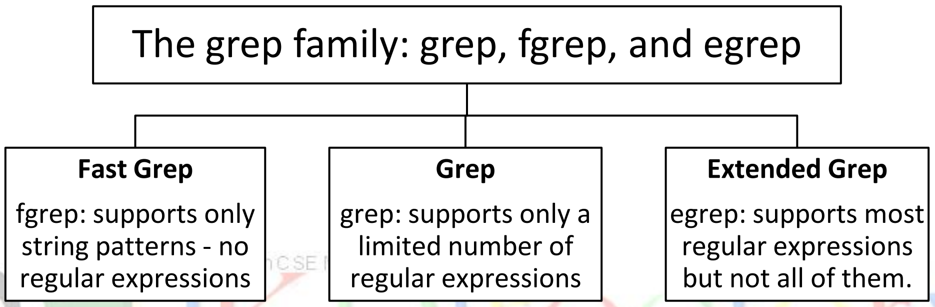
Another way to look at how grep works is to study the flowchart of its operations. Two points about the grep flowchart in the figure below need to be noted. First, the flowchart assumes that no options were specified. Selecting one or more options will change the flowchart. Second although grep keeps a current line counter so that it always knows which line is being processed, the current line number is not reflected in the flowchart.


Start
More Lines? (Not End of File)
No
Yes
Copy line to Pattern Space
Apply Reg Expression
No
Match ?
Yes
Copy line to standard output
Stop

grep Family:

There are three utilities in the grep family: grep, egrep, and fgrep. All these search one or more files and output lines that contain the text that matches criteria specified as a regular expression. The whole line does not have to match the criteria; any matching text in the line is sufficient for it to be output.

It examines each line in the file, one by one. When a line contains matching pattern, the line is output. Although this is a powerful capability that quickly reduces a large amount of data to a meaningful set of information, it cannot be used to process only a portion of the data. The grep family appears in the figure given below:


The grep family: grep, fgrep, and egrep
Fast Grep
fgrep: supports only string patterns - no regular expressions
Grep
grep: supports only a limited number of regular expressions
Extended Grep
egrep: supports most regular expressions but not all of them.

grep Family Options:

There are several options available to the grep family. A summary is found in the table given below:

Option	Explanation
-b	Precedes each line by the file block number in which it is found.
-c	Prints only a count of the number of lines matching the pattern.
-i	Ignores upper-/lowercase in matching text.
-l	Prints a list of files that contain at least one line matching the pattern.
-n	Shows line number of each line before the line.
-s	Silent mode. Executes utility but suppresses all output.
-v	Inverse output. Prints lines that do not match pattern.
-x	Prints only lines that entirely match pattern.
-f file	List of strings to be matched are in file.

grep Family Expressions:

The fast grep (fgrep) uses only sequence operators in a pattern; it does not support any of the other regular expression operators. Basic **grep** and extended grep (**egrep**) both accept regular expressions. As you can see from the table below not all expressions are available.

Atoms	**grep**	**fgrep**	**egrep**	**Operators**	**grep**	**fgrep**	**egrep**
Character	√	√	√	Sequence	√	√	√
Dot	√		√	Repetition	all but ?		* ? +
Class	√		√	Alternation			√
Anchors	√		^ $	Group			√
Back Reference	√			Save	√		

Expressions in the **grep** utilities can become quite complex, often combining several atoms and/or operators into one large expression. When operators and atoms are combined, they are generally enclosed in either single quotes or double quotes. Technically the quotes are needed only when there is blank or other character that has a special meaning to the **grep** utilities. The combined expression format is shown below:

```
$grep 'Forouzan, *Behrouz' file1
```

grep:

The original of file matching utilities, grep handles most of the regular expressions. grep allows regular expressions but is generally slower than egrep. Use it unless you need to group expressions or use repetition to match one or more occurrences of a pattern. It is the only member of the grep family that allows saving the results of a match for later use.

In the following example we use grep to find all the lines that end in a semicolon (;) and then pipe the results to **head** and print the first two.

```
$ grep –n ";$" TheRaven | head -2
8:Ah, distinctly I remember it was in the bleak December;
16:Thrilled me - - filled me with fantastic terrors never felt before;
```

From the table above we see the –n option requests that the line numbers from the original file be included in the output. They are seen at the beginning of each line. The regular expression **;$** looks for a semicolon (;) at the end of the line ($). The file name is "TheRaven", and the output of the **grep** execution is piped to the head utility where only the first two lines are printed (-2).

Fast grep:

If your search criteria require only sequence expressions, fast grep (**fgrep**) is the best utility. Because its expressions consist of only sequence operators, it is also easiest to use if you are searching for text characters that are the same as regular expression operators such as the escape, parentheses, or quotes. For example, to extract all lines of the file that contain an apostrophe, we could use **fgrep** as $ **fgrep –n " ' " file name**

Extended grep:

Extended grep (egrep) is the most powerful of the three grep utilities. While it doesn't have the save option, it does allow more complex patterns. Consider the case where we want to extract all lines that start with a capital letter and end in an exclamation point (!). Our first attempt at this command is shown as follows: $ **egrep –n '^[A-Z].*!$' filename**

The first expression starts at the beginning of the line (^) and looks at the first character only. It uses a set that consists of only uppercase letters ([A-Z]). If the first character does not match the set, the line is skipped and the next line is examined.

If the first character is a match, the second expression (.*) matches the rest of the line until the last character, which must be an exclamation mark; the third expression examines the character at the end of the line ($). It must be an explanation point (a bang). The complete expression therefore matches any line starting with an uppercase letter, that is followed by zero or more characters, and that ends in a bang.

Finally note that we have coded the entire expression in a set of single quotes even though this expression does not require it.

Examples:

1. Count the number of blank lines in the file → $ **egrep –c '^$' testFile**
2. Count the number of nonblank lines in the file → $ **egrep –c '.' testFile**
3. Select the lines from the file that have the string UNIX → $ **fgrep 'UNIX' testFile**
4. Select the lines from the file that have only the string UNIX → $ **egrep '^UNIX$' testFile**
5. Select the lines from the file that have the pattern UNIX at least two times. $ **egrep 'UNIX.*UNIX' testFile**

SEARCHING FOR FILE CONTENTS:

Some modern operating systems allow us to search for a file based on a phrase contained in it. This is especially handy when we have forgotten the filename but know that it contains a specific expression or set of words. Although UNIX doesn't have this capability, we can use the grep family to accomplish the same thing.

Search a Specific Directory:

When we know the directory that contains the file, we can simply use grep by itself. For example, to find a list of all files in the current directory that contain "Raven," we would use the search in the example given below:

```
$ ls
RavenII        TheRaven      man.ed         regexp.dat
$ grep –l 'Raven' *
RavenII
TheRaven
```

The option l prints out the filename of any file that has at least one line that matches the grep expression.

Search All Directories in a Path:

When we don't know where the file is located, we must use the **find** command with the execute criterion. The **find** command begins by executing the specified command, in this case a grep search, using each file in the current directory. It then moves through the subdirectories of the current file applying the grep command. After each directory, it processes its subdirectories until all directories have been processed. In the example below, we start with our home directory (~).

```
$ find ~ –type f –exec grep –l "Raven" {} \;
```

Overview of sed and awk:

Sed is an acronym for **s**tream **ed**itor. Although the name implies editing, it is not a true editor; it does not change anything in the original file. Rather sed scans the input file, line by line, and applies a list of instructions (called a sed script) to each line in the input file. The script, which is usually a separate file, can be included in the sed command line if it is a one-line command. The format is shown below:

sed options script file list

The sed utility has three useful options. Option –n suppresses the automatic output. It allows us to write scripts in which we control the printing. Option –f indicates that there is a script file, which immediately follows on the command line. The third option –e is the default. It indicates that the script is on the command line, not in a file. Because it is the default, it is not required.

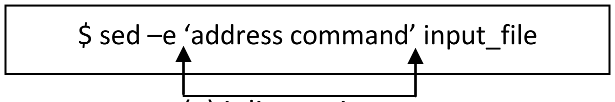
Scripts:

The sed utility is called like any other utility. In addition to input data, sed also requires one or more instructions that provide editing criteria. When there is only one command it may be entered from the keyboard. Most of the time, however, instructions are placed in a file known as sed script (program).

Each instruction in a sed script contains an address and a command.

Script format:

When the script fits in a few lines, its instructions can be included in the command line as shown in figure (a) below. Note that in this case, the script must be enclosed in quotes.



```
$ sed –e ‘address command’ input_file
```

(a) inline script

```
$ sed –f script.sed input_file
```

(b) script file

For longer scripts or for scripts that are going to be executed repeatedly over time, a separate script file is preferred. The file is created with a text editor and saved. In our discussion we suffix the script filename with .sed to indicate that it is a **sed** script. This is not a requirement, but it does make it easier to identify executable scripts. The figure (b) above is an example of executing a sed script.

INSTRUCTION FORMAT:

As previously stated that each instruction consists of an address and a command (figure below :)

address	!	command

The address selects the line to be processed (or not processed) by the command. The exclamation point (!) is an optional address complement. When it is not present, the address must exactly match a line to select the line. When the complement operator is present, any line that does not match the address is selected; lines that match the address are skipped. The command indicates the action that sed is to apply to each input line that matches the address.

Comments:

A comment is a script line that documents or explains one or more instructions in a script. It is provided to assist the reader and is ignored by sed. Comment lines begin with a comment token, which is the pound sign (#). If the comment requires more than one line, each line must start with the comment token.

OPERATION:

Each line in the input file is given a line number by sed. This number can be used to address lines in the text. For each line, sed performs the following operations:

1. Copies an input line to the pattern space. The pattern space is a special buffer capable of holding one or more text lines for processing.

2. Applies all the instructions in the script, one by one, to all pattern space lines that match the specified addresses in the instruction.

3. Copies the contents of the pattern space to the output file unless directed not to by the –n option flag.

When all of the commands have been processed, sed repeats the cycle starting with 1. When you examine this process carefully, you will note that there are two loops in this processing cycle. One loop processes all of the instructions against the current line (operation 2 in the list). The second loop processes all lines.

A second buffer the hold space, is available to temporarily store one or more lines as directed by the sed instructions.

ADDRESSES

The address in an instruction determines which lines in the input file are to be processed by the commands in the instruction. Addresses in sed can be one of four types: ***single line, set of lines, range of lines, nested addresses.***

Single line Addresses:

A single line address specifies one and only one line in the input file. There are two single-line formats: a line number or a dollar sign ($), which specifies the last line in the input file.

Example:

4command$_1$ → command$_1$ applies only to line 4.

16command$_2$ → command$_2$ applies only to line 16.

$command$_3$ → command$_3$ applies only to last line.

Set-of-Line Addresses:

A set-of-line address is a regular expression that may match zero or more lines, not necessarily consecutive, in the input file. The regular expression is written between two slashes. Any line in the input file that matches the regular expression is processed by the instruction command.

Two important points need to be noted: First, the regular expression may match several lines that may or may not be consecutive. Second, even if a line matches, the instruction may not affect the line.

Example:

/^A/command$_1$	→	Matches all lines that start with “A”.
/B$/command$_2$	→	Matches all lines that end with “B”.

Range Addresses:

An address range defines a set of consecutive lines. Its format is start address, comma with no space, and end address:

start-address,end-address

The start and end address can be a sed line number or a regular expression as in the example below:

line-number,line-number

line-number,/regexp/

/regexp/,line-number

/regexp/,/regexp/

When a line that is in the pattern space matches a start range, it is selected for processing. At this point, sed notes that the instruction is in a range. Each input line is processed by the instruction’s command until the stop address matches a line. The line that matches the stop address is also processed by the command, but at that point, the range is no longer active.

If at some future line the start range again matches, the range is again active until a stop address is found. Two important points need to be noted: First, while a range is active, all other instructions are also checked to determine if any of them also match an address.

Second, more than one range may be active at a time. A special case of range address is 1, $, which defines every line from the first line (1) to the last line ($). However, this special case address is not the same as the set-of-lines special case address, which is no address. Given the following two addresses:

(1) Command (2) 1, $command

sed interprets the first as a set of line address and the second as a range address. Some commands, such as insert (i) and append (a), can be used only with a set of line address. These commands accept no address but do not accept 1, $ addresses.

Nested Addresses:

A nested address is an address that is contained within another address. While the outer (first) address range, by definition, must be either a set of lines or an address range, the nested addresses may be either a single line, a set of lines, or another range.

Example-1:

To delete all blank lines between lines 20 and 30

```
20, 30{
        /^$/d
        }
```

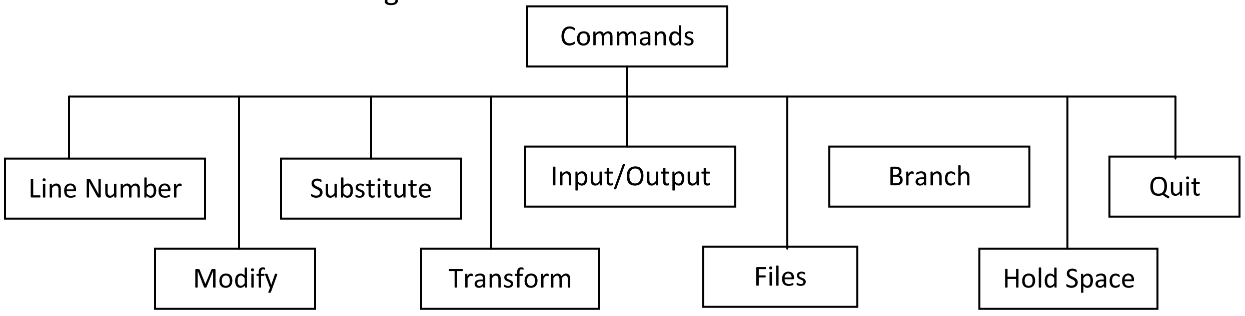
The first command specifies the line range; it is the outer command. The second command, which is enclosed in braces, contains the regular expression for a blank line. It contains the nested address.

Example-2:

To delete all lines that contain the word Raven, but only if the line also contains the word Quoth; In this case, the outer address searches for lines containing Raven, while the inner address looks for lines containing Quoth. Here the interesting thing is that the outer address is not a block of lines but a set of lines spread throughout the file.

```
/Raven/{
                /Quoth/d
        }
```

COMMANDS: There are 25 commands that can be used in an instruction. We group them in to nine categories based on how they perform their task. The following figure summarizes the command categories.

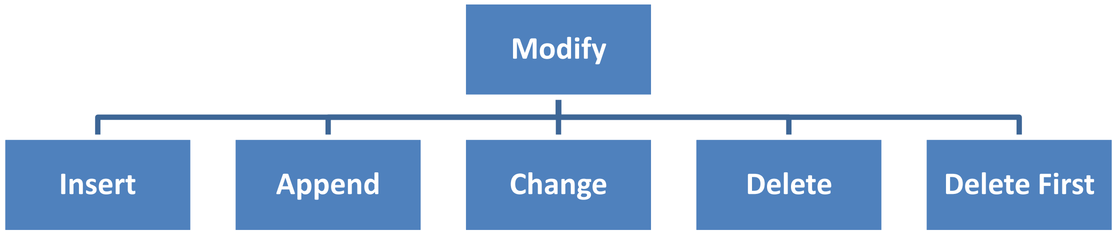


LINE NUMBER COMMAND:

The line number command (=) writes the current line number at the beginning of the line when it writes the line to the output without affecting the pattern space. It is similar to the grep –n option. The only difference is that the line number is written on a separate line.

MODIFY COMMAND:

Modify commands are used to insert, append, change, or delete one or more whole lines. The modify commands require that any text associated with them be placed on the next line in the script. Therefore the script must be in a file; it cannot be coded on the shell command line.



In addition, the modify commands operate on the whole line. In other words, they are line replacement commands. This means that we can’t use these sed commands to insert text in to the middle of a line. *All modify commands apply to the whole line. You cannot modify just part of a line.*

Insert Command (i):

Insert adds one or more lines directly to the output before the address. This command can only be used with the single line and a set of lines; it cannot be used with a range. If you used the inset command with the all lines address, the lines are inserted before every line in the file. This is an easy way to quickly double space a file.

Append Command (a):

Append is similar to insert command except that it writes the text directly to the output after the specified line. Like insert, append cannot be used with a range address.

Inserted and appended text never appears in sed’s pattern space. They are written to the output before the specified line (insert) or after the specified line (append), even if the pattern space is not itself written. Because they are not inserted in to the pattern space, they cannot match a regular expression, nor do they affect sed’s internal line counter.

Change Command (c):

Change replaces a matched line with new text. Unlike insert and append, it accepts all four address types.

Delete Pattern Space Command (d):

The delete command comes in two versions. When a lowercase delete command (d) is used, it deletes the entire pattern space. Any script commands following the delete command that also pertain to the deleted text are ignored because the text is no longer in the pattern space.

Delete Only First Line Command (D):

When an uppercase delete command (D) is used, only the first line of the pattern space is deleted. Of course, if the only line in the pattern space, the effect is the same as the lowercase delete.

SUBSTITUTE COMMAND (S):

Pattern substitution is one of the most powerful commands in sed. In general substitute replaces text that is selected by a regular expression with a replacement string. Thus it is similar to the search and replace found in text editors. With it, we can add, delete or change text in one or more lines. The format of substitute command is given below:

Optional

Address	s	/	**Pattern**	/	Replacement String	/	Flag(s)

Search Pattern: The sed search pattern uses only a subset of the regular expression atoms and patterns. The allowable atoms and operators are listed in table below:

Atoms	**Allowed**	**Operators**	**Allowed**
Character	√	Sequence	√
Dot	√	Repetition	* ? \ { . . . \ }
Class	√	Alternation	√
Anchors	^ $	Group	
Back Reference	√	Save	√

When a text line is selected, its text is matched to the pattern. If matching text is found, it is replaced by the replacement string. The pattern and replacement strings are separated by a triplet of identical delimiters, slashes (/) in the preceding example. Any character can be used as the delimiters, although the slash is the most common.

Replace String:

The replacement text is a string. Only one atom and two metacharacter's can be used in the replacement string. The allowed replacement atom is the back reference. The two metacharacter tokens are the ampersand (&) and the back slash (\). The ampersand is used to place the pattern in the replacement string; the backslash is used to escape an ampersand when it needs to be included in the substitute text (if it's not quoted, it will be replaced by the pattern).

The following example shows how the metacharacter's are used. In the first example, the replacement string becomes ** * * UNIX * * **. In the second example, the replacement string is *now & forever*.

```
$ sed 's/UNIX/*** & ***/' file1
```

```
$ sed '/now/s//now \& forever/' file1
```

TRANSFORM COMMAND (Y):

It is sometimes necessary to transform one set of characters to another. For example, IBM mainframe text files are written in a coding system known as Extended Binary Coded Decimal Interchange Code (EBCDIC). In EBCDIC, the binary codes for characters are different from ASCII. To read an EBCDIC file, therefore, all characters must be transformed to their ASCII equivalents as the file is read.

The transform command (y) requires two parallel sets of characters. Each character in the first string represents a value to be changed to its corresponding character in the second string. This concept is presented as shown below:

Address	y	/	Source Characters	/	Replacement Characters	/

As an example to transform lowercase alphabetic characters to their matching uppercase characters, we would make the source set all of the lowercase characters and the replacement set their corresponding uppercase letters. These two sets would transform lowercase alphabetic characters to their uppercase form. Characters that do not match a source character are left unchanged.

INPUT AND OUTPUT COMMANDS:

The sed utility automatically reads text from the input file and writes data to the output file, usually standard output. There are five input/output commands: next (n), append next(N), print (p), print first line(P), and list (l).

Next Command (n):

The next command (n) forces sed to read the next text line from the input file. Before reading the next line, however, it copies the current contents of the pattern space to the output, deletes the current text in the pattern space, and then refills it with the next input line. After reading the input line, it continues processing through the script.

Append Next Command (N):

Whereas the next command clears the pattern space before inputting the next line, append next command (N) does not. Rather, it adds the next input line to the current contents of the pattern space. This is especially useful when we need to apply patterns to two or more lines at the same time.

Print Command (p):

The print command copies the current contents of the pattern space to the standard output file. If there are multiple lines in the pattern space, they are all copied. The contents of the pattern space are not deleted by the print command.

Print First Line Command (P):

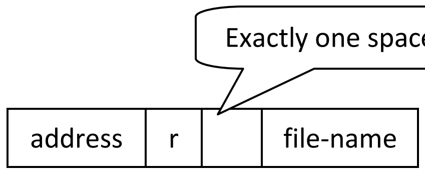
Whereas the print command prints the entire contents of the pattern space, the print first line command (P) prints only the first line. That is it prints the contents of the pattern space up to and including a newline character. Anything following the first newline is not printed.

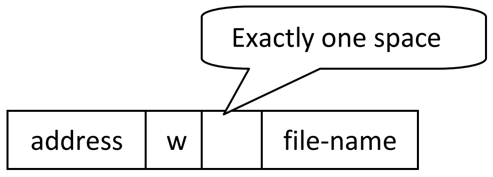
List Command (l):

Depending on the definition of ASCII, there are either 128 (standard ASCII) or 256 (extended ASCII) characters in the character set. Many of these are control characters with no associated graphics. Some, like tab, are control characters that are understood and are actually used for formatting but have no graphic. Others print as spaces because a terminal doesn't support the extended ASCII characters. *The list command (l) converts the unprintable characters to their octal code.*

FILE COMMANDS:

There are two file commands that can be used to read and write files. The basic format for read and write commands is shown below:





Note that there must be exactly one space between the read or write command and the filename. This is one of those sed syntax rules that must be followed exactly.

Read File Command (r):

The read file command reads a file and places its contents in the output before moving to the next command. It is useful when you need to insert one or more common lines after text in a file. The contents of file appear after the current line (pattern space) in the output.

Write File Command:

The write file command (w) writes (actually appends) the contents of the pattern space to a file. It is useful for saving selected data to a file.

Branch Commands:

The branch commands change the regular flow of the commands in the script file. Recall that for every line in the file, sed runs through the script file applying commands that match the current pattern space text. At the end of the script file, the text in the pattern space is copied to the output file, and the next text line is read into the pattern space replacing the old text.

The branch commands allow us to do just that, skip one or more commands in the script file. There are two branch commands: branch (b) and branch on substitution (t).

Branch Label

Each branch command must have a target, which is either a label or the last instruction in the script (a blank label). A label consists of a line that begins with a colon (:) and is followed by up to seven characters that constitute the label name. Example: **:comHere**

Branch Command

The branch command (b) follows the normal instruction format consisting of an address, the command (b) and an attribute (target) that can be used to branch to the end of the script or to a specific location within the script.

The target must be blank or match a script label in the script. If no label is provided, the branch is to the end of the script (after the last line), at which point the current contents of the pattern space are copied to the output file and the script is repeated for the next input line.

Branch on Substitution Command: Rather than branch unconditionally, we may need to branch only if a substitution has been made. In this case we use the branch on substitution or, as it is also known, the test command (t). The format is the same as the branch command.

Hold Space Commands:

The hold buffer is used to save the pattern space. There are five commands that are used to move text back and forth between the pattern space and the hold space: hold and destroy (h), hold and append (H), get and destroy (g), get and append (G) and exchange (x).

Hold and Destroy Command:

The hold and destroy command copies the current contents of the pattern space to the hold space and destroys any text currently in the hold space.

Hold and Append Command:

The hold and append command appends the current contents of the pattern space to the hold space.

Get and Destroy Command:

The get and destroy command copies the text in the hold space to the pattern space and destroy any text currently in the pattern space.

Get and Append Command:

The get and append command appends the current contents of the hold space to the pattern space.

Exchange Command:

The exchange command swaps the text in the pattern and hold space. That is the text in the pattern space is moved to the hold space, and the data that were in the hold space are moved to the pattern space.

Quit: The quit command (q) terminates the sed utility.

awk:

The awk utility which takes its name from the initial of its authors (Alfred V. Aho, Peter J. Weinberger and Brian W. Kernighan), is a powerful programming language disguised as a utility. It behavior is to some extent like sed. It reads the input file, line by line, and performs an action on a part of or on the entire line. Unlike sed, however, it does not print the line unless specifically told to print it.

The format of awk is shown below:

awk options script files

Options:

-F: input field separator
-f: script file

EXECUTION:

The awk utility is called like any other utility. In addition to input data, awk also requires one or more instructions that provide editing instructions. When there are only a few instructions, they may be entered at the command line from the keyboard. Most of the time, however, they are placed in a file known as an awk script (program). Each instruction in awk script contains a pattern and an action.

If the script is short and easily fits on one line, it can be coded directly in the command line. When coded on the command line, the script is enclosed in quotes. The format for the command line script is:

```
$ awk 'pattern[{action}' input-file
```

For longer scripts or for scripts that are going to be executed repeatedly over time, a separate script file is preferred. To create the script, we use a text editor, such as vi or emacs. Once the script has been created we execute it using the file option (-f), which tells awk that the script is in a file. The following example shows how to execute an awk script:

```
$ awk –f scriptFile.awk input-file
```

FIELDS AND RECORDS:

The awk utility vies a file as a collection of fields and records. A field is a unit of data that has informational content. For example, in UNIX list command (ls) output; there are several informational pieces of data, each of which is a field. Among list's output are the permissions, owner, date created, and filename. In awk each field of information is separated from the other fields by one or more whitespace characters or separators defined by the user.

Each line in awk is a record. A record is a collection of fields treated as a unit. In general all of the data in a record should be related. Referring again to the list command output, we see that each record contains data about a file.

When a file is made up of data organized in to records, we call it as a data file, as contrasted with a text file made up of words, lines, and paragraphs. Although awk looks at a file as a set of records consisting of fields, it can also handle a text file. In this case however, each text line is the record, and the words in the line are fields.

BUFFERS AND VARIABLES:

The awk utility provides two types of buffers: record and field. A buffer is an area of memory that holds data while they are being processed.

Fields Buffers:

There are as many field buffers available as there are fields in the current record of the input file. Each field buffer has a name, which is the dollar sign ($) followed by the field number in the current record. Field numbers begin with one, which gives us $1 (the first field buffer), $2 (the second field buffer) and so on.

Record Buffer:

There is only one record buffer available. Its name is $0. It holds the whole record. In other words, its content is the concatenation of all field buffers with one field separator character between each field.

As long as the contents of any of the fields are not changed, $0 holds exactly the same data as found in the input file. If any fields are changed, however, the contents of the $0, including the field separators, are changed.

Variables:

There are two different types of variables in awk: system variables and user-defined variables.

System Variables:

There are more than twelve system variables used by awk; we discuss some of them here. Their names and function are defined by awk. Four of them are totally controlled by awk. The others have standard defaults that can be changed through a script. The system variables are defined in the table given below:

Variable	Function	Default
FS	Input field separator	Space or tab
RS	Input record separator	Newline
OFS	Output field separator	Space or tab
ORS	Output record separator	Newline
NF[a]	Number of nonempty fields in current record	
NR[a]	Number of records read from all files	

Variable	Function	Default
FNR[a]	File number of records read-record number in current file	
FILENAME[a]	Name of the current file	
ARGC	Number of command-line arguments	
ARGV	Command-line argument array	
RLENGTH	Length of string matched by a built-in string function	
RSTART	Start of string matched by a built-in string function	

User Defined Variables:

We can define any number of user defined variables within an awk script. They can be numbers, strings or arrays. Variable names start with a letter and can be followed by any sequence of letters, digits, and underscores. They do not need to be declared; they simply come in to existence the first time they are referenced. All variables are initially created as strings and initialized to a null string (“”).

SCRIPTS:

Data processing programs follows a simple design: preprocessing or initialization, data processing and post processing or end of job. In a similar manner, all awk scripts are divided in to three parts: begin, body, and end (shown in figure below).

BEGIN	{Begin’s Actions}	Preprocessing
	Pattern {Action} Pattern {Action} Pattern {Action}	Body
END	{End’s Actions}	Postprocessing

Initialization Processing (BEGIN): The initialization processing is done only once, before awk starts reading the file. It is identified by the keyword BEGIN, and the instructions are enclosed in a set of branches. The beginning instructions are used to initialize variables, create report headings and perform other processing that must be completed before the file processing starts.

Body Processing: The body is a loop that processes the data in a file. The body starts when awk reads the first record or line from the file. It then processes the data through the body instructions, applying them as appropriate. When end of body instructions is reached, awk repeats the process by reading next record or line & processing it against the body instructions.

End Processing (END): The end processing is executed after all input data have been read. At this time, information accumulated during the processing can be analyzed and printed or other end activities can be conducted.

PATTERNS:

The pattern identifies which records in the file are to receive an action. The awk utility can use several different types of patterns. As it executes a script, it evaluates the patterns against the records found in the file. If the pattern matches the record, the action is taken. If the pattern doesn't match the records, the action is skipped. A statement without a pattern is always true, and the action is always taken. We divide awk patterns in to two categories: simple and range.

Simple Patterns:

A simple pattern matches one record. When a pattern matches a record, the result is true and the action statement is executed. There are four types of simple patterns: BEGIN, END, expression, and nothing (no expression).

BEGIN and END:

BEGIN is true at the beginning of the file before the first record is read. It is used to initialize the script before processing any data; for example it sets the field separators or other system variables. In the below example we set the field separator (FS) and the output field separator (OFS) to tabs.

```
BEGIN
{
FS      =       "\t"
OFS     =       "\t"
} # end BEGIN
.
.
.
END
{
printf("Total Sales:", totalSales)
} # end END
```

END is used at the conclusion of the script. A typical use prints user-defined variables accumulated during the processing.

Expressions:

The awk utility supports four expressions: regular, arithmetic, relational, and logical.

Regular Expressions: The awk regular expressions (regexp) are those defined in egrep. In addition to the expression, awk requires one of two operators: match (~) or not match (!~). When using a regular expression, remember that it must be enclosed in /slashes/.

Example:

```
$0 ~ /^A.*B$/        # Record must begin with ‘A’ and end with ‘B’
$3 !~ /^ /           # Third field must not start with a space
$4 !~ /bird/         # Fourth field must not contain “bird”
```

Arithmetic Expressions: An arithmetic expression is the result of an arithmetic operation. When the expression is arithmetic, it matches the record when the value is nonzero, either plus or minus; it does not match the record when it is zero (false). The following table list the operators used by awk in arithmetic expressions.

Operator	**Example**	**Explanation**
* / % ^	a^2	Variable a is raised to power 2 (a^2)
++	++a, a++	Adds 1 to a.
--	--a, a--	Subtracts 1 from a.
+ -	a + b, a – b	Adds or subtracts two values.
+	+a	Unary plus: value is unchanged
-	-a	Unary minus: value is complemented
=	a = 0	a is assigned the value 0
*=	x *= y	The equivalent of x = x * y
/=	x /= y	The equivalent of x = x / y
%=	x %= y	The equivalent of x = x % y
+=	x += 5	The equivalent of x = x + 5
-=	x -= 5	The equivalent of x = x – 5

Relational Expressions: Relational expressions compare two values and determine if the first is less than, equal to or greater than the second.

Operator	**Explanation**
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal
>	Greater than
>=	Greater than or equal

Logical Expressions: A logical expression uses logical operator to combine two or more expressions.

Operator	**Explanation**
!expr	Not expression
$expr_1$ && $expr_2$	Expression 1 and expression 2
$expr_1$ || $expr_2$	Expression 1 or expression 2

Nothing (No Pattern):

When no address pattern is entered, awk applies the action to every line in the input file. This is the easiest way to specify that all lines are to be processed.

Range Patterns:

A range pattern is associated with a range of records or lines. It is made up of two simple patterns separated by a comma as shown here:

start-pattern, end-pattern

The range starts with the record that matches the start pattern and ends with the next record that matches the end pattern. If the start and end patterns are the same, only one record is in the range.

Each simple pattern can be only one expression; the expression cannot be BEGIN or END. If a range pattern matches more than one set of records in the file, then the action is taken for each set. However, the sets cannot overlap. Thus, if the start range occurs twice before the end range, there is only one matching set starting from the first start record through the matching end record. If there is no matching end range, the matching set begins with the matching start record and end with the last record in the file.

ACTIONS:

In programming languages, actions are known as instructions or statements. They are called actions in awk because they act when the pattern is true. Virtually all of the C language capabilities have been incorporated in to awk and behave as they do in C.

In awk an action is one or more statements associated with a pattern. There is a one-to-one relationship between an action and a pattern: One action is associated with only one pattern. The action statements must be enclosed in a set of braces; the braces are required even if there is only one statement. A set of braces containing pattern/action pairs or statements is known as a block. When an action consists of several statements they must be separated by a statement separator.

In awk the statement separators are a semicolon, a newline, or a set of braces (block).

Pattern/Action Syntax:

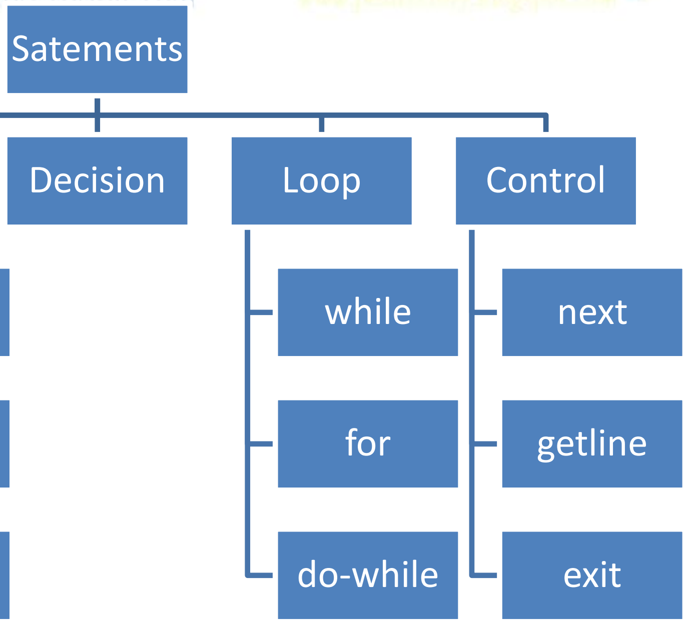
One Statement Action: pattern {statement}

Multiple Statements Separated by Semicolons: pattern {statement1; statement2; statement3}

Multiple Statements Separated by Newlines:

```
pattern
{
        Statement1
        Statement2
        Statement3
}
```

The awk utility contains a rich set of statements that can solve virtually any programming requirement.



Example to print fields:

```
$ awk '{print}' sales.dat
```

Output:

```
1         clothing        3141
1         computers       9161
2         clothing        3252
```

Example to print selected fields:

```
$ awk '{print $1, $2, $3}' sales2.dat | head -2
```

Output:

```
1 clothing 3141
1 computers 9161
```