1. three types of command

the $ character indicates the prompt. Wherever you see a prompt, you can type the name of a command and press Enter.

- simple command

$ command

- complex command

$ command argument1 argument2 argument3 … argumentN

- compound command

$ command1 ; command2 ; command3 ; ... ; commandN ;

The order of execution is command1, followed by command2, followed by command3, and so on. When commandN finishes executing, the prompt returns.

```
$ date
Thu  2 Oct 2014 15:32:48 EST
$ who
users    terminals    time
========================
may      console  Sep 29 12:19
may      ttys000  Oct  2 15:32
$ who am i
may      ttys000  Oct  2 15:32
$ ls -Fa
./            .gitconfig       Adlm/
../           .idlerc/     Applications/
.CFUserTextEncoding  .ipynb_checkpoints/  Desktop/
.DS_Store         .ipython/        Documents/
.Trash/           .matplotlib/     Downloads/
.Xauthority       .pip/            Library/
.bash_history         .rnd         Movies/
.bash_profile         .spyder2/        Music/
.cache/           .ssh/            Pictures/
.config/      .viminfo     Public/
.continuum/       .vimrc           code/
.fontconfig/      .wiznote/        install/
```

```
$ wc -c .viminfo
   10137 .viminfo
```

2. what is a shell?

The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input. When a program has finished executing, it displays that program's output. The shell is sometimes called a command interpreter.

The real power of the UNIX shell lies in the fact that it is much more than a command interpreter. It is also a powerful programming language, complete with conditional statements, loops, and functions.

Two major types of shells:
The different Bourne-type shells follow:

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow:

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

If you are using a Bourne-type shell, the default prompt is the $ character. If you are using a C-type shell, the default prompt is the % character. This book covers only Bourne-type shells because the C-type shells are not powerful enough for shell programming.

In UNIX there are two types of accounts, ***regular user accounts*** and the ***root account***. Normal users are given regular user accounts. The root account is an account with special privileges the administrator of a UNIX system (called the sysadmin) uses to perform maintenance and upgrades.

If you are using the root account, both the Bourne and C shells display the # character as a prompt. Be extremely careful when executing commands as the root user because your commands effect the whole system.

3. kernel, utility and logging in

Utilities are programs you can run or execute. The programs who and date that you saw in the previous chapter are examples of utilities. Almost every program that you know is considered an utility.

The term utility refers to the name of a program, whereas the term command refers to the program and any arguments you specify to that program to change its behaviour.

The kernel is the heart of the UNIX system. It provides utilities with a means of accessing a machine's hardware. It also handles the scheduling and execution of commands.

When a machine is turned off, both the kernel and the utilities are stored on the machine's hard disks. But when the computer is booted, the kernel is loaded from disk into memory. The kernel remains in memory until the machine is turned off.

Utilities, on the other hand, are stored on disk and loaded into memory only when they are executed. For example, when you execute the command $ who, the kernel loads the who command from the machine's hard disk, places it in memory, and executes it. **When the program finishes executing, it remains in the machine's memory for a short period of time before it is removed.** This enables frequently used commands to execute faster.

The shell is a program similar to the who command. The main difference is that the shell is loaded into memory when you log in.

When you first connect to a UNIX system, you usually see a prompt such as the following:

```
login:
```

You need to enter your username at this prompt. After you enter your username, another prompt is presented:

```
login: ranga
Password:
```

You need to enter your password at this prompt.

These two prompts are presented by a program called `getty`. These are its tasks:

1. Display the prompt `login`.
2. Wait for a user to type a username.
3. After a username has been entered, display the password prompt.
4. Wait for a user to enter a password.
5. Give the username and password entered by the user to the `login` command and exit.

After `login` receives your username and password, it looks through the file `/etc/passwd` for an entry matching the information you provided. If it finds a match, `login` executes a shell and exits.
As an example, on my system the matching entry for my username, `ranga`, in file `/etc/passwd` is: `ranga:x:500:100:Sriranga Veeraraghavan:/home/ranga:/bin/bash` **`(will be explained later)`**

If no match is found, the `login` program issues an error message and exits. At this point the `getty` program takes over and displays a new `login` prompt.
The shell that `login` executes is specified in the file `/etc/passwd`. Usually this is one of the shells that I covered in the previous chapter.

In this book I assume that the shell started by the `login` program is `/bin/sh`.
Depending on the version of UNIX you are running, this might or might not be the Bourne shell:

- On Solaris and FreeBSD, it is the Bourne shell.
- On HP-UX, it is the POSIX shell.
- On Linux, it is the Bourne Again shell.

4. shell script and initialisation

Scripts are the power behind the shell because they enable you to group commands together to create new commands.

To ensure that the correct shell is used to **run the script**, you must add the following "magic" line to the beginning of the script:
`#!/bin/sh`

```
#!/bin/sh
# print out the date and who's logged on
date ; who ;
```

The shell can be run in another mode, called *noninteractive mode* . In this mode, the shell does not interact with you; instead it reads commands stored in a file and executes them. When it reaches the end of the file, the shell exits.

You can start the shell noninteractively as follows:
`$ /bin/sh` *`filename`*
Here *`filename`* is the name of a file that contains commands to execute. As an example, consider the
compound command:
`$ date ; who`

Put these commands into a file called `logins`. First open a file called `logins` in an editor and type the command shown previously. Assuming that the file is located in the current directory, after the file is saved, the command can run as

```
$ /bin/sh logins
```

When the login program executes a shell, that shell is *uninitialized*. When a shell is uninitialized, important parameters required by the shell to function correctly are not defined.

The shell undergoes a phase called *initialization* to set up these parameters. This is usually a two step process that involves the shell reading the following files:

```
/etc/profile
profile
```

The process is as follows:

1. The shell checks to see whether the file `/etc/profile` exists.

2. If it exists, the shell reads it. Otherwise, this file is skipped. No error message is displayed.

3. The shell checks to see whether the file `.profile` exists in your home directory. Your *home directory* is the directory that you start out in after you log in.

4. If it exists, the shell reads it; otherwise, the shell skips it. No error message is displayed. As soon as both of these files have been read, the shell displays a prompt:

```
$
```

The file `.profile` is under your control. You can add as much shell customisation information as you want to this file. The minimum set of information that you need to configure includes

- Setting the Terminal Type
- Setting the PATH
- Setting the MANPATH

If needed, go to Page 35 to get some details about how to configure the information above.

5. Working with Files (Ordinary Files) ls, cat, wc, cp, rm, mv

In UNIX there are three basic types of files:

- Ordinary Files

- Directories
- Special Files

When the `-F` option is specified to `ls`, it appends a character indicating the file type of each of the items it lists. The exact character depends on your version of `ls`. For ordinary files, no character is appended. For special files, a character such as `!`, `@`, or `#` is appended to the filename.

Some of the items have a `/` at the end: each of these items is a directory. The other items, such as `hw1`, have no character appended to them. This indicates that they are ordinary files.

```
$ ls -F

bin/            hosts           lib/
ch07            hw1             pub/
ch07.bak        hw2             res.01
docs/           hw3             res.02
res.03
test_results
users          work/
```

Other arguments, for example:

<u>$ ls -1 (The numeric digit “one”.) Force output to be one entry per line</u>

<u>$ ls -a list invisible files</u>

UNIX programs (including the shell) use most of these files to store configuration information. Some common examples of hidden files:

- `.profile`, the Bourne shell (`sh`) initialization script
- `.kshrc`, the Korn shell (`ksh`) initialization script
- `.cshrc`, the C shell (`csh`) initialization script
- `.rhosts`, the remote shell configuration file

All files that do not start with the `.` character are considered visible.

<u>$ ls -a -F list all files with their file types</u>

```
./        .profile    docs/     lib/
../       .rhosts     hosts     pub/
.emacs    bin/        hw1       res.01
.exrc     ch07        hw2       res.02
```

.kshrc ch07.bak hw3 res.03

From the output above, there are two hidden directories (. and ..). These two directories are special entries that are present in all directories. The first one, ., represents the current directory. The second one, .., represents the parent directory.

The commands

```
$ ls -aF
$ ls -Fa
```

are the same as the command

```
$ ls -a -F
```

As you can see, the order of the options does not matter to `ls`. As an example of option grouping, consider the equivalent following commands:

```
ls -1 -a -F
ls -1aF
ls -a1F
ls -Fa1
```

Any combination of the options `-1`, `-a`, and `-F` produces identical output:

```
./
../
.emacs
.exrc
.kshrc
.profile
.rhosts
bin/
ch07
ch07.bak
docs/
hosts
hw1
hw2
hw3
lib/
pub/
res.01
res.02
res.03
```

$ cat filename1 filename2 … viewing the content of files

$ cat -b filename1 filename2 the lines of output are numbered

If you specify more than one file, `wc` gives the individual counts along with a total. For example, the command

```
$ wc .rhosts .profile
```

produces the following output:

```
7 14 179 .rhosts
133 405 2908 .profile
140 419 3087 total
```

$ wc -l	Counts the number of lines
$ wc -w	Counts the number of words
$ wc -m or -c	Counts the number of characters

$ cp source destination

Here *source* is the name of the file that is copied and *destination* is the name of the copy.

`Interactive mode -i applies for cp, mv and rm.`

No error message is generated if the destination already exists. In this case, the destination file is automatically overwritten.

To avoid this behavior you can specify the `-i` (*i* as in *interactive*) options to `cp`. If the file `test_results.orig` exists, the command

```
$ cp -i test_results test_results.orig
```

results in a prompt something like the following:

```
overwrite test_results.orig? (y/n)
```

If you choose `y` (yes), the file will is overwritten. If you choose `n` (no), the file `test_results.orig` isn't changed.

If the *destination* is a directory, the copy has the same name as the *source* but is located in the *destination* directory. For example, the command

```
$ cp test_results work/
```

If more than two inputs are given, `cp` treats the last argument as the *destination* and the other files as *sources*. This works only if the *sources* are files and the *destination* is a directory, as in the following example:

```
$ cp res.01 res.02 res.03 work/
```

$ mv source destination

Here *source* is the original name of the file and *destination* is the new name of the file.

6. Working With Directories (In UNIX/Linux everything's a file:)

UNIX uses a hierarchical structure for organising files and directories. This structure is often referred to as a directory tree . The tree has a single root node, the slash character (/), and all other directories are contained below it. You can use every directory, including /, to store both files and other directories. Every file is stored in a directory, and every directory except / is stored in another directory.

In order to access a file or directory, its pathname must be specified. As you have seen, a pathname consists of two parts: the name of the directory and the names of its parents. UNIX offers two ways to specify the names of the parent directory. That means two types of pathnames:

- Absolute
- Relative

Look at an example that illustrates how relative pathnames are used. Assume that the current directory is
/home/ranga/work
Then the relative pathname
../docs/ch5.doc
represents the file
/home/ranga/docs/ch5.doc

whereas
./docs/ch5.doc
represents the file
/home/ranga/work/docs/ch5.doc
You can also refer to this file using the following relative path:
docs/ch5.doc

cd change directory (or go back to current user's main directory)
mkdir create new directory

list the files in a directory:
$ ls /usr/local
$ ls ../../usr/local

```
$ ls -aF /usr/local
```

Sometimes when you want to create a directory, its parent directory or directories might not exist. In this case, `mkdir` issues an error message. Here is an illustration of this:

```
$ mkdir /tmp/ch04/test1
```

```
mkdir: Failed to make directory "/tmp/ch04/test1"; No such file or directory
```

In such cases, you can specify the `-p` (p as in parent) option to the `mkdir` command. It creates all the necessary directories for you. For example

```
$ mkdir -p /tmp/ch04/test1
```

creates all the required parent directories.

An error also occurs if you try to create a directory with the same name as a file. For example, the following commands

```
$ ls -F docs/names.txt
names
$ mkdir docs/names
```

result in the error message

```
mkdir: cannot make directory 'docs/names': File exists
```

```
$ cp -r docs/book /mnt/zip
```

copies the directory `book` located in the `docs` directory to the directory `/mnt/zip`. It creates a new directory called `book` under `/mnt/zip`.

```
$ cp -r docs/book docs/school work/src /mnt/zip
```

copies the directories `school` and `book`, located in the directory `docs`, to `/mnt/zip`. It also copies the directory `src`, located in the directory `work`, to `/mnt/zip`. After the copies finish, `/mnt/zip` looks like the following:

```
$ ls -aF /mnt/zip
./   ../   book/   school/   src/
```

You can also mix files and directories in the argument list. For example
`$ cp -r .profile docs/book .kshrc doc/names work/src /mnt/jaz` copies all the requested files and directories to the directory `/mnt/jaz`.
If your argument list consists only of files, the `-r` option has no effect.

```
$ mv work/ docs/ .profile pub/
```

moves the directories `work` and `docs` along with the file `.profile` into the directory `pub`.

You can use two commands to remove directories:

```
rmdir
rm -r
```

Use the first command to remove empty directories. It is considered "safe" because in the worst case, you can accidentally lose an empty directory, which you can quickly re-create with `mkdir`.

The second command removes directories along with their contents. It is considered "unsafe" because in the worst case of `rm -r`, you could lose your entire system.

7. File Types

Table 5.1 Special Characters for Different File Types

Character	File Type
-	Regular file
l	Symbolic link
c	Character special
b	Block special
p	Named pipe
s	Socket
d	Directory file

The `ls -l` output for a regular file:

```
-rw-------  1 may  staff  10137  6 Sep 12:14 .viminfo
```

A **symbolic link** is a special file that points to another file on the system. The `ls -l` output for a symbolic link looks like this:

```
lrwxrwxrwx 1 root root 9 Oct 23 13:58 /bin/ -> ./usr/bin/
```

The output indicates that the directory `/bin` is really a link to the directory `./usr/bin`.

Create symbolic links using the `ln` command with the `-s` option. The syntax is as follows: `ln -s` *`source destination`*

Here, *`source`* is either the absolute or relative path to the original version of the file, and *`destination`* is the name you want the link to have. e.g.

```
$ ln -s ../httpd/html/users/ranga ./public_html
```

You can see the relative path by using `ls -l`: `$ ls -l ./public_html`

```
lrwxrwxrwx   1 ranga   users   26 Nov  9  1997
public_html -> ../httpd/html/users/ranga
```

You can access UNIX devices through reading and writing to **device files**. These device files are access points to the device within the file systems. Usually, device files are located under the `/dev` directory. The two main types of device files are

Character special files:

Character special files provide a mechanism for communicating with a device one character at a time. The output of ls-l of a character special file e.g.

```
crw-------   1 ranga    users      4,    0 Feb  7 13:47 /
dev/tty0
```

you also see two extra numbers before the date. The first number is called the *major* number and the second number is called the *minor* number. UNIX uses these two numbers to identify the device driver that this file communicates with.

Block special files:

Block special files also provide a mechanism for communicating with device drivers via the file system. These files are called *block devices* because they transfer large blocks of data at a time. This type of file typically represents hard drives and removable media.

Look at the `ls -l` output for a typical block device.

```
brw-rw---- 1 root disk 8, 0 Feb 7 13:47 /dev/sda
```

Here the first character is `b`, indicating that this file is a block special file. Just like the character special files, these files also have a major and a minor number.

Named Pipe

One of the greatest features of UNIX is that you can redirect the output of one program to the input of another program with very little work. For example, the command `who | grep ranga` takes the output of the `who` command and makes it the input to the `grep` command. This is called *piping* the output of one command into another. You will examine input and output redirection in great detail in Chapter "Input/Output."

Socket files are another form of interprocess communication, but sockets can pass data and information between two processes that are not running on the same machine.

*extras in advance:
1. http://docstore.mik.ua/orelly/unix/upt/ch44_02.htm
2. http://stackoverflow.com/questions/21640837/mxpost-bash-mxpost-bin-kshm-bad-interpreter-no-such-file-or-directory

3. check whether a file exists under the current directory. If not, displays a message and then exits.

```
#!/bin/sh
#more comment here
if [ ! -f ./records.txt ]; then
    echo "File not found!"
fi
```

8. Owners, Groups, and Permissions

Table 5.3 who

Letter	Represents
u	Owner
g	Group
o	Other
a	All

Table 5.4 actions

Symbol	Represents
+	Adding permissions to the file
-	Removing permission from the file
=	Explicitly set the file permissions

Table 5.5 permissions

Letter	Represents
r	Read
w	Write
x	Execute
s	SUID or SGID

Three kinds of permissions:
- Owner permissions
- Group permissions
- All other users permissions

You can perform the following actions on a file:
- Read
- Write
- Execute

You can display the permissions of a file using the `ls -l` command.

Additional permissions are given to programs via a mechanism known as the Set User ID (SUID) and Set Group ID (SGID) bits. When you execute a program that has the SUID bit enabled, you inherit the permissions of that program's owner. Programs that do not have the SUID bit set are run with the permissions of the user who started the program.

To give the "world" read access to all files in a directory, you can use one of the following commands:
`$ chmod a=r *` or `$ chmod guo=r *`

To stop anyone except the owner of the file .profile from writing to it, try this:
`$ chmod go-w .profile`

If you need to apply more than one set of permissions changes to a file or files, use a comma separated list. For example
`$ chmod go-w,a+x a.out`

if the directory pub contains the following directories:
$ ls pub
./ ../ README faqs/ src/
you can change the permission read permissions of the file README along with the files contained in the directories faqs and src with the following command: `$ chmod -R o+r pub`

The `chown` command stands for "change owner" and is used to change the owner of a file.

`chown ranga: /home/httpd/html/users/ranga` changes the owner of the given directory to the user `ranga`.

The `chown` command will recursively change the ownership of all files when the `-R` option is included. For example, the command
`chown -R ranga: /home/httpd/html/users/ranga`
changes the owner of all the files and subdirectories located under the given directory to be the user `ranga`.

The `chgrp` command stands for "change group" and is used to change the group of a file.

As an example
`chgrp authors /home/ranga/docs/ch5.doc`
changes the group of the given file to be the group authors. Just like `chown`, all versions of `chirp` understand the `-R` option also.
On systems without this command, you can use `chown` to change the group of a file. For example, the command
`chown :authors /home/ranga/docs/ch5.doc`
changes the group of the given file to the group authors.

9. Processes

In UNIX every program runs as a process.

● Starting processes
Whenever you issue a command in UNIX, it creates, or starts, a new process. When you tried out the ls command to list directory contents, you started a process (the ls command).

The operating system tracks processes through a five digit ID number known as the pid or process ID . Each process in the system has a unique pid. Pids eventually repeat because all the possible numbers are used up and the next pid rolls or starts over. At any one time, no two processes with the same pid exist in the system because it is the pid that UNIX uses to track each process. You might be interested in the fact that the pid usually rolls over at the 16-bit signed boundary. The highest it gets before rolling over is 32,767.

When you start a process (run a command), there are two ways you can run it--in the foreground or background. The difference is how the process interacts with you at the terminal.

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (&) at the end of the command.

completion message:

```
[1] +  Done                     ls ch0*.doc &
$
```

The first line tells you that the `ls` command background process finishes successfully. The second is a prompt for another command.

enable monitoring with the following:
set -o monitor
To disable the monitoring messages, you use +o:
set +o monitor
You can also check all the shell options (settings) with the following: set -o

How to move a foreground process to the background:
When the foreground process is running, press **Ctrl + Z** to stop it, then enter the **`bg`** command.
command **`fg %1`** does the contrary (move background to foreground).

- Listing running processes

The jobs command shows you the processes you have suspended and the ones running in the background. Because the jobs command is a foreground process, it cannot show you your active foreground processes.

In the following example, I have three jobs. The first one (job 3) is running, the second (job 2) is suspended (a foreground process after I used Ctrl+Z), and the third one (job 1) is stopped in the background to wait for keyboard input:

```
$ jobs
[3] +  Running
[2] - Stopped (SIGTSTP)
[1]   Stopped (SIGTTIN)
first_one &
second_one
third_one &
```

Another command that shows all processes running is the **ps** (Process Status) command.

For UNIX based OS, the basic ps command offers four pieces of information: the pid, the TTY (terminal running this process), the Time or amount of CPU consumed by this process, and the command name running.

```
$ ps
  PID TTY           TIME CMD
43232 ttys000    0:00.01 -bash
```

- Killing processes

The job number is prefixed with a percent sign. To kill job number 1:

```
$ kill %1
[1] - Terminated                  third_one &
$
```

You can also kill a specific process by specifying the process ID on the command line without the percent sign used with job numbers. To kill job number 2 (process `6738`) in the earlier example using process ID, I use the following:

```
$ kill 6739
$
```

In reality, `kill` does not physically kill a process; it sends the process a signal. By default, it sends the `TERM` (value `15`) signal. A process can choose to ignore the `TERM` signal or use it to begin an orderly shut down (flushing buffers, closing files, and so on). If a process ignores a regular `kill` command, you can use `kill -9` or `kill -KILL` followed by the process ID or job number (prefixed with a percent sign). This forces the process to end.

● Parent and child processes

In the **ps -f** example in the **ps** command section, each process has two ID numbers assigned to it: process ID (pid) and parent process ID (ppid).

Each user process in the system has a parent process. Most commands that you run have the shell as their parent. The parent of your shell is usually the operating system or the terminal communications process.

When a child is forked, or created, from its parent, it receives a copy of the parent's environment, including environment variables. The child can change its own environment, but those changes do not reflect in the parent and go away when the child exits.

10. Variables & Arrays

```
$ FRUIT=apple
$ FRUIT[1]=peach
```

the element `FRUIT` has the value `apple`. At this point any accesses to the scalar variable `FRUIT` are treated like an access to the array item `FRUIT[0]`.

The one thing to be careful about is using values that have spaces.
For example,
$ FRUIT=apple orange plum
results in the following error message:
sh: orange: not found.
In order to use spaces you need to quote the value.
For example, both of the following are valid assignments:
$ FRUIT="apple orange plum"
$ FRUIT='apple orange plum’

`$ set -A band derri terry mike gene` or
`$ band=(derri terry mike gene)`

is equivalent to the following commands:

```
$ band[0]=derri
$ band[1]=terry
$ band[2]=mike
$ band[3]=gene
```

To access the array item at index 5 use the following:

```
${adams[5]}
```

To access every item in the array use the following:

```
${adams[@]}
```

readonly NAME

often used in scripts to make sure that critical variables are not overwritten accidentally.

```
$ FRUIT=kiwi
$ readonly FRUIT
$ echo $FRUIT
kiwi
$ FRUIT=cantaloupe
```

The last command results in an error message:

```
/bin/sh: FRUIT: This variable is read only.
```

```
unset FRUIT
```

unsets the variable `FRUIT`.

You cannot use the `unset` command to unset variables that are marked `readonly`.

When a shell is running, three main types of variables are present:

- Local Variables
- Environment Variables
- Shell Variables

A local variable is a variable that is present within the current instance of the shell. It is **not** available to programs that are started by the shell. The variables that you looked at previously have all been local variables.

An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.

A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

How to make environment variables? ***exporting*** them.
`name=value; export name`
An example of this is `PATH=/sbin:/bin; export PATH`
export more than one variable to the environment:
`export PATH HOME UID`

11. Filename Substitution – Globbing with * ? [] !

Globbing is Case Sensitive.

*****

Matching a File Prefix. e.g.
`$ ls ch1*`
matches all the files and directories in the current directory that start with the letters ch1. The output is similar to the following:
`ch10-01  ch10-02  ch10-03  ch11-01  ch11-02  ch11-03`

Matching a File Suffix. e.g.
`$ ls *doc`
matches all the files and directories in the current directory that end with the letters doc

Matching Suffixes and Prefixes. e.g.
`$ ls Backup*doc`
matches all the files in the current directory that start with the letters *Backup* and end with the letters *doc*

or even
`$ ls CGI*st*java`

?

One limitation of the * wildcard is that it matches one or more characters each time. (In computing, wildcard means a character that will match any character or sequence of characters in a search.) In order to match only one character, use the ? wildcard. Each ? represents for one character.

```
$ ls ch0?.doc
$ ls ch??.doc
```

[]

```
$ ls [a-z]*
```
lists all the files starting with a lowercase letter.
```
$ ls [A-Z]*
```
lists all the files starting with uppercase letters.
```
$ ls [a-zA-Z]*
```
matches all files that start with a letter.
```
$ ls *[a-zA-Z0-9]
```
matches all files ending with a letter or a number.

!

```
$ ls [!a]*
```
list all files except those that start with the letter a

12. Quoting

a list of most of the **shell special characters** (also called metacharacters):

```
* ? [ ] ' " \ $ ; & ( ) | ^ < > new-line space tab
```

- backslash: use before a single special character. e.g.

```
echo Hello\; world
```
The backslash causes the ; character to be handled as any other normal character. The resulting output is
```
Hello; world
```

- single quote : quote a large group of characters.

exception: Single quotes must be entered in pairs. You cannot get around by putting a backslash before an embedded single quote. e.g.
```
echo 'It's Friday'
```
should be corrected to
```
echo It\'s Friday
```

- double quote: take away the special meaning of all characters except the following

- $ for parameter substitution ($variable name for actual variable values).
- ´ Backquotes for command substitution.
- \$ to enable literal dollar signs.
- \´ to enable literal backquotes.
- \" to enable embedded double quotes.
- \\ to enable embedded backslashes.
- All other \ characters are literal (not special).

e.g.
```
echo "The DOS directory is \"\\windows\\temp\""
```
The output looks like this:
```
The DOS directory is “\windows\temp"
```

13. Flow Control

Two powerful flow control mechanics are available in the shell:
- The if statement
- The case statement

The basic if statement syntax follows:
```
if list1
then
    list2
elif list3
then
    list4
else
    list5
fi
```

File test expressions test whether a file fits some particular criteria. The general syntax for a file test is
```
test option file
```
or
```
[ option file ]
```

examples:
```
$ if [ -d /home/ranga/bin]; then PATH=“$PATH:/home/ranga/
bin"
fi
```

testing whether the directory /home/ranga/bin exists. If it does, append it to the variable PATH.

```
if [ -s $HOME/.bash_aliai]; then . $HOME/.bash_aliai ; fi
```

execute commands stored in the file $HOME/.bash_aliai if it exists.

Option	Description
`-b file`	True if `file` exists and is a block special file.
-c file	True if file exists and is a character special file.
-d file	True if file exists and is a directory.
-e file	True if file exists.
-f file	True if file exists and is a regular file.
-g file	True if file exists and has its SGID bit set.
-h file	True if file exists and is a symbolic link.
-k file	True if file exists and has its "sticky" bit set.
-p file	True if file exists and is a named pipe.
-r file	True if file exists and is readable.
-s file	True if file exists and has a size greater than zero.
-u file	True if file exists and has its SUID bit set.
-w file	True if file exists and is writable.
-x file	True if file exists and is executable.
`-O file`	True if `file` exists and is owned by the effective user ID.

Can also use `test` to compare strings and numerical variables.

Option	Description
`-z string`	True if `string` has zero length.
`-n string`	True if `string` has nonzero length.
`string1 = string2`	True if the strings are equal.
`string1 != string2`	True if the strings are not equal.

Operator	Description
`int1 -eq int2`	True if `int1` equals `int2`.
`int1 -ne int2`	True if `int1` is not equal to `int2`.
`int1 -lt int2`	True if `int1` is less than `int2`.
`int1 -le int2`	True if `int1` is less than or equal to `int2`.
`int1 -gt int2`	True if `int1` is greater than `int2`.
`int1 -ge int2`	True if `int1` is greater than or equal to `int2`.

The case statement syntax:

```
case word in
    pattern1)  list1;;
    pattern2)  list2;;
esac
```

An example of a simple `case` statement that uses patterns is

```
case "$TERM" in
      *term)
           TERM=xterm ;;
      network|dialup|unknown|vt[0-9][0-9][0-9])
           TERM=vt100 ;;
esac
```

Here the string contained in `$TERM` is compared against two patterns. If this string ends with the string `term`, `$TERM` is assigned the value `xterm`. Otherwise , `$TERM` is compared against the strings `network`, `dialup`, `unknown`, and `vtXXX`, where *`XXX`* is some three digit number, such as 102. If one of these strings matches, `$TERM` is set to `vt100`.

Given the following variable declarations,

```
HOME=/home/ranga
BINDIR=/home/ranga/bin
```

the output of the following `if` statement

```
if [ $HOME/bin = $BINDIR ] ; then
   echo "Your binaries are stored in your home
directory."
fi
```

is what echo contains.

14. Loops

- The while loop
- The for loop

The basic syntax of the `while` loop is

```
while command
do
    list
done
```

Here is a simple example that uses the `while` loop to display the numbers zero to nine:

```
x=0
while [ $x -lt 10 ]     #lt means less than
do
    echo $x
     x=´echo "$x + 1" | bc´
done
```

example:

```
RESPONSE=
while [ -z "$RESPONSE" ] ;
do
    echo "Enter the name of a directory where your files
are located:\c "
    read RESPONSE
    if [ ! -d "$RESPONSE" ] ; then
        echo "ERROR: Please enter a directory pathname."
RESPONSE= fi
done
```

Here you store the user's response in the variable `RESPONSE`. Initially this variable is set to null, enabling the `while` loop to begin executing.

*until loop

```
until command
do
    list
done
```

for loop

```
for name in word1 word2 ... wordN
do
    list
done
```

example:
```
for FILE in $HOME/.bash*
do
    cp $FILE ${HOME}/public_html
    chmod a+r ${HOME}/public_html/${FILE}
done
```

*select loop
*break continue

15. Output & Input

In UNIX, the process of capturing the output of a command and storing it in a file is called output redirection because it redirects the output of a command into a file instead of the screen.

- the output redirection operator >
- appends output to a file >>
  - (output is appended to the end of the specified file)
- Redirecting Output to a File and the Screen **command | tee** file
- input redirection: command < file
- read *contents* like scanf in C

16. Function & Text Filters

“PATH is an environment variable on Unix-like operating systems, DOS, OS/2, and Microsoft Windows, specifying a set of directories where executable programs are located. In general, each executing process or user session has its own PATH setting. “

function definition & invoking:
```
name () { list ; }
$ name
```

The grep command displays every line in file that contains word.
```
grep word file
```
Can specify more than one file.
If grep cannot find a line in any of the specified files that contains the requested word, no output is produced.
To match words regardless of the case that you specify, use the -i option.