CHAPTER 13

Shell Programming

13. 1 INTRODUCTION

Why Shell Programming? A working knowledge of shell scripting is essential to everyone wishing to become reasonably adept at system administration, even if they do not anticipate ever having to actually write a script. Consider that as a Linux machine boots up, init process is initiated first then it executes the shell scripts in /etc/rc.d to restore the system configuration and set up services. A detailed understanding of these startup scripts is important for analyzing the behavior of a system, and possibly modifying it [Kernigham].

Writing shell scripts is not hard to learn, since the scripts can be built in bite-sized sections and there is only a fairly small set of shell-specific operators and options to learn. The syntax is simple and straightforward, similar to that of invoking and chaining together utilities at the command line, and there are only a few "rules" to learn. Most short scripts work right the first time, and debugging even the longer ones is straightforward. A shell script is a "quick and dirty" method of prototyping a complex application. Getting even a limited subset of the functionality to work in a shell script, even if slowly, is often a useful first stage in project development. This way, the structure of the application can be tested and played with, and the major pitfalls found before proceeding to the final coding in C, C++, Java, or Perl. Shell scripting hearkens back to the classical UNIX philosophy of breaking complex projects into simpler subtasks, of chaining together components and utilities. Many consider this a better, or at least more esthetically pleasing approach to problem solving than using one of the new generation of high powered all-in-one languages, such as Perl, which attempt to be all things to all people, but at the cost of forcing you to alter your thinking processes to fit the tool.

When we want to execute some set of commands one after another without users physical intervention and presence (batch operations), shell scripts are very handy.

Moreover, for small scale database applications where precision, speed and security is little botheration, shell scripts are very preferable and SW project cost may tremendously reduces.

Shell scripts are very much employed in developing automatic SW installation scripts and for fine tuning the SW's installed.

When not to use shell scripts :

- resource-intensive tasks, especially where speed is a factor (sorting, hashing, etc.)
- procedures involving heavy-duty math operations, especially floating point arithmetic, arbitrary precision calculations, or complex numbers (use C++ or FORTRAN instead)
- cross-platform portability required (use C instead)
- complex applications, where structured programming is a necessity (need type checking of variables, function prototypes, etc.)
- mission-critical applications upon which you are betting the ranch, or the future of the company
- situations where security is important, where you need to guarantee the integrity of your system and protect against intrusion, cracking, and vandalism
- project consists of subcomponents with interlocking dependencies
- extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion)
- need multi-dimensional arrays
- need data structures, such as linked lists or trees
- need to generate or manipulate graphics or GUIs
- need direct access to system hardware
- need port or socket I/O
- need to use libraries or interface with legacy code
- proprietary, closed-source applications (shell scripts are necessarily Open Source)

If any of the above applies, consider a more powerful scripting language, perhaps Perl, Tcl, Python, or possibly a high-level compiled language such as C, C++, or Java. Even then, prototyping the application as a shell script might still be a useful development step.

Shell programs also called as shell scripts. In the simplest case, a script is nothing more than a list of system commands stored in a file. If we want to execute a set of commands many times repeatedly, we can write the same in a file and execute which saves the effort of retyping that particular sequence of commands each time they are needed.

13.2 WHICH SHELL ?

You have many to pick from, each with its own peculiarities

Bourne shell (sh)

The Bourne shell (sh) is the original UNIX shell written by Steve Bourne of Bell Labs.

It is available on all UNIX systems.

It does not have the interactive facilities provided by other shells such as the C shell and Korn shell, so you should use another shell for interactive use.

It is the shell of choice developing portable shell scripts.

C Shell (csh) or Total C Shell (tcsh)

This shell was written at the University of California, Berkley.

It provides a C-like language - hence its name.

Most of the features of modern interactive shell were originally developed in the csh including job control and command history and editing.

tcsh is a public domain enhancement to the csh.

It provides all the features of the C shell together with emacs style editing of the command line.

Korn Shell (ksh)

This shell was written by David Korn of Bell labs.

It is now provided as the standard shell on UNIX systems - in fact it is defined by POSIX as the standard shell for UNIX.

It provides all the features of the C and TC shells together with a shell programming language similar to that of the original Bourne shell.

It is the most efficient shell.

Consider using this as your standard interactive shell.

Z Shell (zsh)

zsh is a shell designed for interactive use, although it is also a powerful scripting language.

Many of the useful features of bash, ksh, and tcsh were incorporated into zsh; many original features were added.

Popular among programmers and advanced UNIX users.

Bourne-again Shell (bash)

This is a public domain shell written by the Free Software Foundation under their GNU initiative.

It is a full implementation of the IEEE POSIX Shell and Tools specification.

This shell is widely used within the academic community.

bash provides all the interactive features of the C shell (csh) and the Korn shell (ksh).

Its programming language is compatible with the Bourne shell (sh).

If you use the Bourne shell (sh) for shell programming consider using bash as your complete shell environment.

It is the default shell used by Linux and the most popular shell for new users.

Differences among the UNIX Shells[3]

	sh	csh	ksh	bash	tcsh	zsh	rc	es
Job control	N	Y	Y	Y	Y	Y	N	N
Aliases	N	Y	Y	Y	Y	Y	N	N
Shell functions	Y(1)	N	Y	Y	N	Y	Y	Y
"Sensible" Input/Output redirection	Y	N	Y	Y	N	Y	Y	Y
Directory stack	N	Y	Y	Y	Y	Y	F	F
Command history	N	Y	Y	Y	Y	Y	L	L
Command line editing	N	N	Y	Y	Y	Y	L	L
Vi Command line editing	N	N	Y	Y	Y(3)	Y	L	L
Emacs Command line editing	N	N	Y	Y	Y	Y	L	L
Rebindable Command line editing	N	N	N	Y	Y	Y	L	L
User name look up	N	Y	Y	Y	Y	Y	L	L
Login/Logout watching	N	N	N	N	Y	Y	F	F
Filename completion	N	Y(1)	Y	Y	Y	Y	L	L
Username completion	N	Y(2)	Y	Y	Y	Y	L	L
Hostname completion	N	Y(2)	Y	Y	Y	Y	L	L
History completion	N	N	N	Y	Y	Y	L	L
Fully programmable Completion	N	N	N	N	Y	Y	N	N
Mh Mailbox completion	N	N	N	N(4)	N(6)	N(6)	N	N
Co Processes	N	N	Y	N	N	Y	N	N
Builtin artithmetic evaluation	N	Y	Y	Y	Y	Y	N	N
Can follow symbolic links invisibly	N	N	Y	Y	Y	Y	N	N
Periodic command execution	N	N	N	N	Y	Y	N	N
Custom Prompt (easily)	N	N	Y	Y	Y	Y	Y	Y
Sun Keyboard Hack	N	N	N	N	N	Y	N	N
Spelling Correction	N	N	N	N	Y	Y	N	N
Process Substitution	N	N	N	Y(2)	N	Y	Y	Y
Underlying Syntax	sh	csh	sh	sh	csh	sh	rc	rc
Freely Available	N	N	N(5)	Y	Y	Y	Y	Y
Checks Mailbox	N	Y	Y	Y	Y	Y	F	F
Tty Sanity Checking	N	N	N	N	Y	Y	N	N
Can cope with large argument lists	Y	N	Y	Y	Y	Y	Y	Y
Has non-interactive startup file	N	Y	Y(7)	Y(7)	Y	Y	N	N
Has non-login startup file	N	Y	Y(7)	Y	Y	Y	N	N
Can avoid user startup files	N	Y	N	Y	N	Y	Y	Y
Can specify startup file	N	N	Y	Y	N	N	N	N
Low level command redefinition	N	N	N	N	N	N	N	Y
Has anonymous functions	N	N	N	N	N	N	Y	Y
List Variables	N	Y	Y	N	Y	Y	Y	Y
Full signal trap handling	Y	N	Y	Y	N	Y	Y	Y
File no clobber ability	N	Y	Y	Y	Y	Y	N	F

Local variables	N	N	Y	Y	N	Y	Y	Y
Lexically scoped variables	N	N	N	N	N	N	N	Y
Exceptions	N	N	N	N	N	N	N	Y

Key to the table above.

Y	Feature can be done using this shell.
N	Feature is not present in the shell.
F	Feature can only be done by using the shells function mechanism.
L	The readline library must be linked into the shell to enable this Feature.

Notes to the table above

1. This feature was not in the original version, but has since become almost standard.
2. This feature is fairly new and so is often not found on many versions of the shell, it is gradually making its way into standard distribution.
3. The Vi emulation of this shell is thought by many to be incomplete.
4. This feature is not standard but unoffical patches exist to perform this.
5. A version called 'pdksh' is freely available, but does not have the full functionality of the AT&T version.
6. This can be done via the shells programmable completion mechanism.
7. Only by specifing a file via the ENV environment variable.

13.3 BOURNE AGAIN SHELL

Bash is largely compatible with sh and incorporates useful features from the Korn shell (ksh) and the C shell (csh). It is intended to be a conformant implementation of the IEEE POSIX Shell and Tools portion of the IEEE POSIX specification (IEEE Standard 1003.1). It offers functional improvements over sh for both interactive and programming use. This is the one which is very widely available in Linux, BSD derivatives. In the following pages we will be explaining the Bourn Shell related concepts.

13.3.1 Invoking the script

Having written the script, we can invoke it by sh scriptname, or alternately bash scriptname.

Much more convenient is to make the script itself directly executable with chmod.

Either

chmod 555 scriptname (gives everyone read/execute permission)

or

chmod +rx scriptname (gives everyone read/execute permission)

chmod u+rx scriptname (gives only the script owner read/execute permission)

Having made the script executable, we may now test it by typing **./scriptname** at command prompt.

As a final step, after testing and debugging, we would likely want to move it to /usr/local/bin (as root, of course), to make the script available to all other users as a system-wide executable. The script could then be invoked by simply typing scriptname from the command line.

It is shell programming practice in which line starting with #! at the head of a script tells our system that this file is a set of commands to be fed to the command interpreter indicated. The #! is actually a two-byte "magic number", a special marker that designates a file type, or in this case an executable shell script . Immediately following the #!, we have to include the path of the program that interprets the commands in the script, whether it be a shell, a programming language, or a utility. This command interpreter then executes the commands in the script, starting at the top (line 1 of the script), ignoring comments.

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/awk -f
```

Each of the above script header lines calls a different command interpreter, be it / bin/sh, the default shell (bash in a Linux system) or otherwise. Using #!/bin/sh, the default Bourne Shell in most commercial variants of UNIX, makes the script portable to non-Linux machines, though you may have to sacrifice a few Bash-specific features (the script will conform to the POSIX sh standard).

#! can be omitted if the script consists only of a set of generic system commands, using no internal shell directives.

Variables are at the heart of every programming and scripting language. They appear in arithmetic operations and manipulation of quantities, string parsing, and are indispensable for working in the abstract with symbols - tokens that represent something else. A variable is nothing more than a location or set of locations in computer memory holding an item of data.

Unlike many other programming languages, Bash does not segregate its variables by "type". Essentially, Bash variables are character strings, but, depending on context, Bash permits integer operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits.

13.3.2 Shell Builtins

This section describes builtin commands which are unique to or have been extended in Bash. Some of these commands are specified in the POSIX standard.

alias

alias [-p] [*name*[=*value*] ...]

Without arguments or with the -p option, alias prints the list of aliases on the standard output in a form that allows them to be reused as input. If arguments are supplied, an alias is defined for each *name* whose *value* is given. If no *value* is given, the name and value of the alias is printed.

bind

bind [-m *keymap*] [-lpsvPSV]
bind [-m *keymap*] [-q *function*] [-u *function*] [-r *keyseq*]
bind [-m *keymap*] -f *filename*

bind [-m *keymap*] -x *keyseq:shell-command*
bind [-m *keymap*] *keyseq:function-name*
bind *readline-command*

Display current Readline key and function bindings, bind a key sequence to a Readline function or macro, or set a Readline variable. Each non-option argument is a command as it would appear in a Readline initialization file, but each binding or command must be passed as a separate argument; e.g., '"\C-x\C-r":re-read-init-file'.

Options, if supplied, have the following meanings :

-m *keymap*

Use *keymap* as the keymap to be affected by the subsequent bindings. Acceptable *keymap* names are emacs, emacs-standard, emacs-meta, emacs-ctlx, vi, vi-move, vi-command, and vi-insert. vi is equivalent to vi-command; emacs is equivalent to emacs-standard.

-l

List the names of all Readline functions.

-p

Display Readline function names and bindings in such a way that they can be used as input or in a Readline initialization file.

-P

List current Readline function names and bindings.

-v

Display Readline variable names and values in such a way that they can be used as input or in a Readline initialization file.

-V

List current Readline variable names and values.

-s

Display Readline key sequences bound to macros and the strings they output in such a way that they can be used as input or in a Readline initialization file.

-S

Display Readline key sequences bound to macros and the strings they output.

-f *filename*

Read key bindings from *filename*.

-q *function*

Query about which keys invoke the named *function*.

-u *function*

Unbind all keys bound to the named *function*.

-r *keyseq*

Remove any current binding for *keyseq*.

-x *keyseq:shell-command*

Cause *shell-command* to be executed whenever *keyseq* is entered. When *shell-command* is executed, the shell sets the READLINE_LINE variable to the contents of the Readline line buffer and the READLINE_POINT variable to the current location of the insertion point. If the executed command changes the value of READLINE_LINE or READLINE_POINT, those new values will be reflected in the editing state.

The return status is zero unless an invalid option is supplied or an error occurs.

builtin

builtin [*shell-builtin* [*args*]]

Run a shell builtin, passing it *args*, and return its exit status. This is useful when defining a shell function with the same name as a shell builtin, retaining the functionality of the builtin within the function. The return status is non-zero if *shell-builtin* is not a shell builtin command.

caller

caller [*expr*]

Returns the context of any active subroutine call (a shell function or a script executed with the . or source built-ins).

Without *expr*, caller displays the line number and source filename of the current subroutine call. If a non-negative integer is supplied as *expr*, caller displays the line number, subroutine name, and source file corresponding to that position in the current execution call stack. This extra information may be used, for example, to print a stack trace. The current frame is frame 0.

The return value is 0 unless the shell is not executing a subroutine call or *expr* does not correspond to a valid position in the call stack.

command

command [-pVv] *command* [*arguments* ...]

Runs *command* with *arguments* ignoring any shell function named *command*. Only shell builtin commands or commands found by searching the PATH are executed. If there is a shell function named ls, running 'command ls' within the function will execute the external command ls instead of calling the function recursively. The -p option means to use a default value for PATH that is guaranteed to find all of the standard utilities. The return status in this case is 127 if *command* cannot be found or an error occurred, and the exit status of *command* otherwise.

If either the -V or -v option is supplied, a description of *command* is printed. The -v option causes a single word indicating the command or file name used to invoke *command* to be displayed; the -V option produces a more verbose description. In this case, the return status is zero if *command* is found, and non-zero if not.

declare

declare [-aAfFilrtux] [-p] [*name*[=*value*] ...]

Declare variables and give them attributes. If no *name*s are given, then display the values of variables instead.

The -p option will display the attributes and values of each *name*. When -p is used with *name* arguments, additional options are ignored.

When -p is supplied without *name* arguments, declare will display the attributes and values of all variables having the attributes specified by the additional options. If no other options are supplied with -p, declare will display the attributes and values of all shell variables. The -f option will restrict the display to shell functions.

The -F option inhibits the display of function definitions; only the function name and attributes are printed. If the extdebug shell option is enabled using shopt, the source file name and line number where the function is defined are displayed as well. -F implies -f. The following options can be used to restrict output to variables with the specified attributes or to give variables attributes :

-a

Each *name* is an indexed array variable (see Arrays).

-A

Each *name* is an associative array variable (see Arrays).

-f

Use function names only.

-i

The variable is to be treated as an integer; arithmetic evaluation is performed when the variable is assigned a value.

-l

When the variable is assigned a value, all upper-case characters are converted to lower-case. The upper-case attribute is disabled.

-r

Make *name*s readonly. These names cannot then be assigned values by subsequent assignment statements or unset.

-t

Give each *name* the trace attribute. Traced functions inherit the DEBUG and RETURN traps from the calling shell. The trace attribute has no special meaning for variables.

-u

When the variable is assigned a value, all lower-case characters are converted to upper-case. The lower-case attribute is disabled.

-x

Mark each *name* for export to subsequent commands via the environment.

Using '+' instead of '-' turns off the attribute instead, with the exceptions that '+a' may not be used to destroy an array variable and '+r' will not remove the readonly attribute. When used in a function, declare makes each *name* local, as with the local command. If a variable name is followed by =*value*, the value of the variable is set to *value*.

The return status is zero unless an invalid option is encountered, an attempt is made to define a function using '-f foo=bar', an attempt is made to assign a value to a readonly variable, an attempt is made to assign a value to an array variable without using the compound assignment syntax (see Arrays), one of the *names* is not a valid shell variable name, an attempt is made to turn off readonly status for a readonly variable, an attempt is made to turn off array status for an array variable, or an attempt is made to display a non-existent function with -f.

echo

echo [-neE] [*arg* ...]

Output the *args*, separated by spaces, terminated with a newline. The return status is always 0. If -n is specified, the trailing newline is suppressed. If the -e option is given, interpretation of the following backslash-escaped characters is enabled. The -E option disables the interpretation of these escape characters, even on systems where they are interpreted by default. The xpg_echo shell option may be used to dynamically determine whether or not echo expands these escape characters by default. echo does not interpret — to mean the end of options.

echo interprets the following escape sequences :

\a

alert (bell)

\b

backspace

\c

suppress further output

\e

escape

\f

form feed

\n

new line

\r

carriage return

\t

horizontal tab

\v

vertical tab

\

backslash

\0*nnn*

the eight-bit character whose value is the octal value *nnn* (zero to three octal digits)

\x*HH*

the eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits)

enable

enable [-a] [-dnps] [-f *filename*] [*name* ...]

Enable and disable builtin shell commands. Disabling a builtin allows a disk command which has the same name as a shell builtin to be executed without specifying a full pathname, even though the shell normally searches for built-ins before disk commands. If -n is used, the *name*s become disabled. Otherwise *name*s are enabled. For example, to use the test binary found via $PATH instead of the shell builtin version, type 'enable -n test'.

If the -p option is supplied, or no *name* arguments appear, a list of shell built-ins is printed. With no other arguments, the list consists of all enabled shell built-ins. The -a option means to list each builtin with an indication of whether or not it is enabled.

The -f option means to load the new builtin command *name* from shared object *filename*, on systems that support dynamic loading. The -d option will delete a builtin loaded with -f.

If there are no options, a list of the shell built-ins is displayed. The -s option restricts enable to the POSIX special built-ins. If -s is used with -f, the new builtin becomes a special builtin (see Special Built-ins).

The return status is zero unless a *name* is not a shell builtin or there is an error loading a new builtin from a shared object.

help

help [-dms] [*pattern*]

Display helpful information about builtin commands. If *pattern* is specified, help gives detailed help on all commands matching *pattern*, otherwise a list of the built-ins is printed.

Options, if supplied, have the following meanings :

-d

Display a short description of each *pattern*

-m

Display the description of each *pattern* in a manpage-like format

-s

Display only a short usage synopsis for each *pattern*

The return status is zero unless no command matches *pattern*.

let

let *expression* [*expression*]

The let builtin allows arithmetic to be performed on shell variables. Each *expression* is evaluated according to the rules given below. If the last *expression* evaluates to 0, let returns 1; otherwise 0 is returned.

local

local [*option*] *name*[=*value*] ...

For each argument, a local variable named *name* is created, and assigned *value*. The *option* can be any of the options accepted by declare. local can only be used within a function; it makes the variable *name* have a visible scope restricted to that function and its children. The return status is zero unless local is used outside a function, an invalid *name* is supplied, or *name* is a readonly variable.

logout

logout [*n*]

Exit a login shell, returning a status of *n* to the shell's parent.

mapfile

mapfile [-n *count*] [-O *origin*] [-s *count*] [-t] [-u *fd*] [-C *callback*] [-c *quantum*] [*array*]

Read lines from the standard input into array variable *array*, or from file descriptor *fd* if the -u option is supplied. The variable MAPFILE is the default *array*. Options, if supplied, have the following meanings :

-n

Copy at most *count* lines. If *count* is 0, all lines are copied.

-O

Begin assigning to *array* at index *origin*. The default index is 0.

-s

Discard the first *count* lines read.

-t

Remove a trailing line from each line read.

-u

Read lines from file descriptor *fd* instead of the standard input.

-C

Evaluate *callback* each time *quantum*P lines are read. The -c option specifies *quantum*.

-c

Specify the number of lines read between each call to *callback*.

If -C is specified without -c, the default quantum is 5000. When *callback* is evaluated, it is supplied the index of the next array element to be assigned as an additional argument. *callback* is evaluated after the line is read but before the array element is assigned.

If not supplied with an explicit origin, mapfile will clear *array* before assigning to it. mapfile returns successfully unless an invalid option or option argument is supplied, or *array* is invalid or unassignable.

printf

printf [-v *var*] *format* [*arguments*]

Write the formatted *arguments* to the standard output under the control of the *format*. The *format* is a character string which contains three types of objects: plain characters, which are simply copied to standard output, character escape sequences, which are converted and copied to the standard output, and format specifications, each of which causes printing of the next successive *argument*. In addition to the standard printf(1) formats, '%b' causes printf to expand backslash escape sequences in the corresponding *argument*, (except that '\c' terminates output, backslashes in '\'', '\"', and '\?' are not removed, and octal escapes beginning with '\0' may contain up to four digits), and '%q' causes printf to output the corresponding *argument* in a format that can be reused as shell input.

The -v option causes the output to be assigned to the variable *var* rather than being printed to the standard output.

The *format* is reused as necessary to consume all of the *arguments*. If the *format* requires more *arguments* than are supplied, the extra format specifications behave as if a zero value or null string, as appropriate, had been supplied. The return value is zero on success, non-zero on failure.

read

read [-ers] [-a *aname*] [-d *delim*] [-i *text*] [-n *nchars*] [-p *prompt*] [-t *timeout*] [-u *fd*] [*name* ...]

One line is read from the standard input, or from the file descriptor *fd* supplied as an argument to the -u option, and the first word is assigned to the first *name*, the second word to the second *name*, and so on, with leftover words and their intervening separators assigned to the last *name*. If there are fewer words read from the input stream than names, the remaining names are assigned empty values. The characters in the value of the IFS variable are used to split the line into words. The backslash character '\' may be used to remove any special meaning for the next character read and for line continuation. If no names are supplied, the line read is assigned to the variable REPLY. The return code is zero, unless end-of-file is encountered, read times out (in which case the return code is greater than 128), or an invalid file descriptor is supplied as the argument to -u.

Options, if supplied, have the following meanings :

-a *aname*

The words are assigned to sequential indices of the array variable *aname*, starting at 0. All elements are removed from *aname* before the assignment. Other *name* arguments are ignored.

-d *delim*

The first character of *delim* is used to terminate the input line, rather than newline.

-e

Readline is used to obtain the line. Readline uses the current (or default, if line editing was not previously active) editing settings.

-i *text*

If Readline is being used to read the line, *text* is placed into the editing buffer before editing begins.

-n *nchars*

read returns after reading *nchars* characters rather than waiting for a complete line of input.

-p *prompt*

Display *prompt*, without a trailing newline, before attempting to read any input. The prompt is displayed only if input is coming from a terminal.

-r

If this option is given, backslash does not act as an escape character. The backslash is considered to be part of the line. In particular, a backslash-newline pair may not be used as a line continuation.

-s

Silent mode. If input is coming from a terminal, characters are not echoed.

-t *timeout*

Cause read to time out and return failure if a complete line of input is not read within *timeout* seconds. *timeout* may be a decimal number with a fractional portion following the decimal point. This option is only effective if read is reading input from a terminal, pipe, or other special file; it has no effect when reading from regular files. If *timeout* is 0, read returns success if input is available on the specified file descriptor, failure otherwise. The exit status is greater than 128 if the timeout is exceeded.

-u *fd*

Read input from file descriptor *fd*.

readarray

readarray [-n *count*] [-O *origin*] [-s *count*] [-t] [-u *fd*] [-C *callback*] [-c *quantum*] [*array*]

Read lines from the standard input into array variable *array*, or from file descriptor *fd* if the -u option is supplied.

A synonym for mapfile.

source

source *filename*

A synonym for . (see Bourne Shell Built-ins).

type

type [-afptP] [*name* ...]

For each *name*, indicate how it would be interpreted if used as a command name.

If the -t option is used, type prints a single word which is one of 'alias', 'function', 'builtin', 'file' or 'keyword', if *name* is an alias, shell function, shell builtin, disk file, or shell reserved word, respectively. If the *name* is not found, then nothing is printed, and type returns a failure status.

If the -p option is used, type either returns the name of the disk file that would be executed, or nothing if -t would not return 'file'.

The -P option forces a path search for each *name*, even if -t would not return 'file'.

If a command is hashed, -p and -P print the hashed value, not necessarily the file that appears first in $PATH.

If the -a option is used, type returns all of the places that contain an executable named *file*. This includes aliases and functions, if and only if the -p option is not also used.

If the -f option is used, type does not attempt to find shell functions, as with the command builtin.

The return status is zero if all of the *names* are found, non-zero if any are not found.

typeset

typeset [-afFrxi] [-p] [*name*[=*value*] ...]

The typeset command is supplied for compatibility with the Korn shell; however, it has been deprecated in favor of the declare builtin command.

ulimit

ulimit [-abcdefilmnpqrstuvxHST] [*limit*]

ulimit provides control over the resources available to processes started by the shell, on systems that allow such control. If an option is given, it is interpreted as follows :

-S

Change and report the soft limit associated with a resource.

-H

Change and report the hard limit associated with a resource.

-a

All current limits are reported.

-b

The maximum socket buffer size.

-c

The maximum size of core files created.

-d

The maximum size of a process's data segment.

-e

The maximum scheduling priority ("nice").

-f

The maximum size of files written by the shell and its children.

-i

The maximum number of pending signals.

-l

The maximum size that may be locked into memory.

-m

The maximum resident set size (many systems do not honor this limit).

-n

The maximum number of open file descriptors (most systems do not allow this value to be set).

-p

The pipe buffer size.

-q

The maximum number of bytes in POSIX message queues.

-r

The maximum real-time scheduling priority.

-s

The maximum stack size.

-t

The maximum amount of cpu time in seconds.

-u

The maximum number of processes available to a single user.

-v

The maximum amount of virtual memory available to the process.

-x

The maximum number of file locks.

-T

The maximum number of threads.

If *limit* is given, it is the new value of the specified resource; the special *limit* values hard, soft, and unlimited stand for the current hard limit, the current soft limit, and no limit, respectively. A hard limit cannot be increased by a non-root user once it is set; a soft limit may be increased up to the value of the hard limit. Otherwise, the current value of the soft limit for the specified resource is printed, unless the -H option is supplied. When setting new limits, if neither -H nor -S is supplied, both the hard and soft limits are set. If no option is given, then -f is assumed. Values are in 1024-byte increments, except for -t, which is in seconds, -p, which is in units of 512-byte blocks, and -n and -u, which are unscaled values.

The return status is zero unless an invalid option or argument is supplied, or an error occurs while setting a new limit.

unalias

unalias [-a] [*name* ...]

Remove each *name* from the list of aliases. If -a is supplied, all aliases are removed. Aliases are described in Aliases.

13.4 VARIABLES

Shell programming supports prominently the following type of variables :

- Shell Variables
- Environment Variables
- Positional Variables

13.4.1 Shell Variables

X=Hello (no spaces before and after =)

The above statement at the bash prompt defines a shell variable X and assigns a value for it. Anywhere, $X indicates the value of the variable X.

Very often shell variables are used to reduce typing burden. For example, in the following examples after defining shell variable DIR the same can be used where ever we need to type /usr/lib.

DIR=/usr/lib	
ls $DIR	displays listing of /usr/lib directory
cd $DIR	moves to /usr/lib directory
ls $DIR/libm*.so	displays all files /usr/lib which satisfies libm*.so model

We have discussed about one of the shell built-in known as read. With that also, we can define shell variables. For example, in the following shell program we are defining two shell variables X, Y and Z.

```
echo Enter Name
read X
echo Enter Age and Tel No
read Y Z
echo $X $Y $Z
```

13.4.2 Environnent Variables

Variables that affect the behavior of the shell and user interface. In a more general context, each process has an "environment", that is, a group of variables that hold information that the process may reference. In this sense, the shell behaves like any other process. Every time a shell starts, it creates shell variables that correspond to its own environmental variables. Updating or adding new shell variables causes the shell to update its environment, and all the shell's child processes (the commands it executes) inherit this environment. *Caution:* The space allotted to the environment is limited. Creating too many environmental variables or ones that use up excessive space may cause problems.

If we execute "env" command at the dollar prompt we may find the details of all the environment variables defined in our current shell. The output may look like

```
PATH=/bin:/sbin:/usr/local/bin
MANPATH=/usr/man:/usr/man/man1:/usr/man/man2
IFS=
TERM=VT100
HOST=darkstar
USER=guest
HOME=/usr/guest
MAIL=/var/spool/mail/guest
MAILCHECK=300
```

Environment variables are used by shell and other application programs. For example, the value of MAILCHECK, i.e. 300 indicates that the mailer has to check for every 300 seconds for new arrivals and intimate the same to the user. A dynamic business user can set this variable value to a low value such that the mailer informs the user within the specified time period about new mails arrival.

Similarly, PATH environment variable is used by shell in locating the executable file of the commands typed by the user. System will check for the executable files in the directories of the PATH variable and if found it will be loaded and executed. Otherwise, we may get error "bad command or file not found".

Let the following C language file named "a.c" :

```
#include<stdio.h>
main()
{
printf("Hello\n");
}
```

To compile :

```
gcc -o aa a.c
```

The file a.c is the C language source file and "aa" will become executable file.

Very often (if PATH is set properly) by simply typing "aa" at the $ prompt we can run the above program. If in the value of PATH variable dot (".") is not available then we may get error "bad command or file not found" as the system is not in position to identify the file "aa". By typing ./aa we can run program (this problem is very much seen Redhat Linux distributions).

Similarly, if we created executable file name as "test" (normally, new users behavior) then if we type "test" at the dollar prompt the above program may not run. This is because, there exists a UNIX command "test". Thus when you try to start "test" command instead of running our developed program, UNIX command test runs. This may be also attributed to PATH problem only. When we type test, the system will check first say program "test" in/bin or /usr/bin then system and the same is executed. Thus, never our can run. Thus, we can add . (dot) to PATH in the following manner such that the above problem is not seen.

```
PATH=.:$PATH
```

If a script sets environmental variables, they need to be "exported", that is, reported to the environment local to the script. This is the function of the export command.

Main difference between shell variables and environment variables is that the latter are inheritable to sub-shells. Environment variables defined in a shell or modified in a shell are visible in its sub-shells only. That is, parent shells do not see the environment variables defined in its sub-shell or the modifications done to environment variables in the sub-shells. Please note that when you see $ prompt, you are in bash shell.

```
X=Hello
Y=How
echo $PATH   // displays value of PATH environment variable
echo $X      // displays value of X shell variable
echo $Y      // displays value of X shell variable
export Y     // makes Y as environment variable
bash         // a sub-shell bash is created run ps -Al in other terminal to see
echo $PATH   // displays value of PATH environment variable which is same as
             above
echo $X          // displays nothing as X shell variable is not inherited
echo $Y          // displays how as Y is environment variable
Z=Raj
export Z
```

```
echo $Z          // displays Z variable value
csh           // another sub shell is initiated
echo $PATH    // displays value of PATH environment variable which is same as
              above
echo $X       // displays nothing as X shell variable is not inherited
echo $Y          // displays how as Y is environment variable
Z=Raj
export Z
echo $Z          // displays Z variable value
exit or ^c          // to come out from C shell
^d                  // to come out from bash sub-shell
echo $PATH    // displays value of PATH environment variable which is same as
              above
echo $X       // displays X shell variable value
echo $Y          // displays how as Y is environment variable
Z=Raj
export Z
echo $Z          // displays nothing as Z is not visible
```

Note

A script can export variables only to child processes, that is, only to commands or processes which that particular script initiates. A script invoked from the command line cannot export variables back to the command line environment. Child processes cannot export variables back to the parent processes that spawned them.

How to change our prompt?

We can change our shell's prompt by changing the value of environment variable PS1. If we execute the command PS1=Enter at the command prompt, our prompt will be changed to Enter.

We do have many options to be used with PS1 assignment. They are :

\a : an ASCII bell character (07)

\d : the date in "Weekday Month Date" format (e.g., "Tue May 26")

\D{format} : the format is passed to strftime(3) and the result is inserted into the prompt string; an empty format results in a locale-specific time representation. The braces are required

\e : an ASCII escape character (033)

\h : the hostname up to the first '.'

\H : the hostname

\j : the number of jobs currently managed by the shell

\l : the basename of the shell's terminal device name

\n : newline

\r : carriage return

\s : the name of the shell, the basename of $0 (the portion following the final slash)

\t : the current time in 24-hour HH:MM:SS format

\T : the current time in 12-hour HH:MM:SS format

\@ : the current time in 12-hour am/pm format

\A : the current time in 24-hour HH:MM format

\u : the username of the current user

\v : the version of bash (e.g., 2.00)

\V : the release of bash, version + patch level (e.g., 2.00.0)

\w : the current working directory, with $HOME abbreviated with a tilde

\W : the basename of the current working directory, with $HOME abbreviated with a tilde

\! : the history number of this command

\# : the command number of this command

\$: if the effective UID is 0, a #, otherwise a $

\nnn : the character corresponding to the octal number nnn

\\ : a backslash

\[: begin a sequence of non-printing characters, which could be used to embed a terminal control sequence into the prompt

\] : end a sequence of non-printing characters

Let us try to set the prompt so that it can display today's date and hostname :

PS1="\d \h $ "

Output :

```
Wed May 20 compaq-laptop $
```

Now, we will set the prompt to display date/time, hostname and current directory and a beep sound by executing the following command.

$ PS1="[\d \t \u@\h:\w \a] $ "

Output :

```
[Wed May 20 08:06:09 Administrator@compaq-laptop:~ ] $
```

The following code fragment if added to /etc/profile, for super user we will get a colored prompt and for others normal prompt will come. Of course, your terminal drivers should support color options.

```
# If id command returns zero, you've root access.
if [ $(id -u) -eq 0 ];
then # you are root, set red colour prompt
  PS1="\\[$(tput setaf 1)\\]\\u@\\h:\\w #\\[$(tput sgr0)\\]"
else # normal
  PS1="[\\u@\\h:\\w] $"
fi
```

If this prompt to be permanent, we have to enter this line either in .profile or .bashrc.

The secondary prompt PS2

In most of the shells, value of this variable is >. Whenever shell considers that the command is not yet completed it gives this prompt to indicate that the command is not completed and user can continue to complete the command. If we happened to have a very large command whose size may exceed the command buffer size, we may split the command between multiple lines. In the following interactive session, we may find shell considers cp command is not completed if we just type cp only at the dollar prompt. Thus, it shows PS2 prompt. We can complete the command by typing source and destination filenames and press enter.

```
$cp
> rao raj
$
```

Similarly, we can split the command ls-al between multiple lines as follows. If we enter \ character at the end of a line, we indicating the shell that the command is not yet completed. Thus, shell gives PS2 prompt to inform us to complete the same in the next line.

```
$ls\
> -a\
>l
$
```

13.4.3 Positional Parameters

These parameters are arguments passed to the script from the command line - $0, $1, $2, $3... Here, $0 is the name of the script itself, $1 is the first argument, $2 the second, $3 the third, and so forth. After $9, the arguments must be enclosed in brackets, for example, ${10}, ${11}, ${12}.

Also, the following parameters can be also used in shell scripts

$#	number of command line arguments
$*	list of command line arguments
$@	list of command line arguments
$$	PID of the current shell
$?	Exit status of most recent command. Usually it is zero if the command is successful.
$!	PID of most recent background job

Example

The following shell program (abc) displays the program name, number of command line arguments, and list of command line arguments.

```
echo $0
echo $#
echo $*
echo $@
```

Now, we assume that we have given user executable permissions to the file abc by executing the command **chmod u+x abc**. We can test abc as :

```
$abc rao raj 123 Abhi
abc
4
rao raj 123 Abhi
rao raj 123 Abhi
```

We have shell built-in known as shift with which we shift positional arguments. If we call shift $3 becomes $2, $4 becomes $3 and vice versa. As mentioned earlier, we can use only nine command line arguments in some versions of Bourne shells. In those shells, we can use this shift to access command line arguments of any number. The following program positional1 demonstrates the use of shift in shell program.

```
$ cat>positional1
echo $1 $9
shift
echo $1 $9
shift 3
echo $1 $9
^d
$ chmod u+x positional1
$ ./positional1 r1 r2 r3 r4 r5 r6 r7 r8 r9 s1 s2 s3 s4 s5 s6
r1 r9
r2 s1
r5 s4
```

Remember, we can shift the command line arguments left to right. That is, we can not make $1 as $2, $2 as $3 and vice versa.

The following shell program (positional2) executes iteratively and prints the message Hello. That is we are calling this program positional2 from itself.

```
$cat>positional2
echo Hello
positional2
^d
$chmod u+x positional2
$positional2
```

The following shell program (positional3) also works in the same fashion as $0 indicates the program name (positional3) itself.

```
$cat>positional3
echo Hello
$0
^d
$chmod u+x positional3
$positional3
```

Now we may get doubt how many times the above programs runs recursively. It depends on our and system and its parameters. We have a system limit known as nproc which indicates how many child processes a process can create. If this limit is crossed, we may get an error such as "unable to fork". We may get error such as "unable to swap" or "swap space exhausted". This happens if there is no space in the swap memory to create process structures for each of the recursive calls.

Understand the working of the following program.

```
$ cat>positional4
X=$0
echo $*
shift
$X "$@"
^d
$chmod u+x positional4
$positional4 rao raj ravi
rao raj ravi
raj ravi
ravi
```

Identify what happens if we replace $@ with $* in the above program?.

We wanted to write a shell program (cx) which reads a filename along the command line and gives user executable permissions to it.

```
cat>cx
chmod u+x $1
^d
```

Now, let we give user executable permissions for cx by executing :

chmod u+x cx

Now on wards, we can use cx to change permissions of any file. For example, to change permissions of a file xyz, we can execute

cx xyz

Of course, this cx command will work only if we are in the same directory as that of cx file. However, if we wanted to use cx command while we are in other directories then we can add path of the cx file to the environment variable PATH. If we assume, cx is available in /home/rao/shell then we can execute the following :

PATH=$PATH:/usr/rao/shell

If we want the cx command to be available for us for every login session, then we have enter the above line either in .profile or .bashrc.

Example

Let us assume that we wanted to have an illusion that we are working under DOS!. That is, we wanted to use DOS command names in UNIX itself. This can be achieved by simply writing shell program with the names of DOS commands. Of course, we should not forget to give permissions for all these files. We are worrying about having commonly used DOS commands with most command styles.

DOS command Name (also filename)	Shell Code
COPY	cp $1 $2
MOVE	mv $1 $2
DEL	rm $1
DELETE	rm $1
ERASE	rm $1
DIR	ls -l
Tree	ls –lR $1
Deltree	rm –R $1
TYPE	cat $1
XCOPY	cp –r $1 $2
REN	mv $1 $2
RENAME	mv $1 $2
MKDIR	mkdir $1
MD	mkdir $1
RMDIR	rmdir $1
RD	rmdir $1
CD	cd $1

Now, let us understand about $?. We have mentioned that it will be having the exit status of most recent command. In UNIX, programs are designed such that if they run successfully they will be having their exit status as 0; otherwise a number other than zero (often a negative integer). Manual pages of every command contains details about the exit status and the possible reason for failure. Thus, if any one wants to know the reason for any commands failure, they can simply note the value of $? and then refer that command's manual pages for reason against this value.

To support our statements, see the following interactive session details. We can verify that the exit status of ls, grep etc., are showing as 0 if succeeded otherwise other than 0.

```
$ ls
A D aaa abc m2 mm mmmm pal pp s1 s6 x1
B Pqr aaaa employee m3 mm1 mno positional1 ppp s3 s8 x2
C D aa ab hist m4 mm2 nbv positional4 rrr s5 volume xx
$ echo $?
0
$ls /tmpp
ls: can not access /tmpp: No such file or directory
$echo $?
2
```

```
$ grep "if" *
Pqr:if [ $# -eq 0 ]
m3:if [ $x -eq 10 ]
positional1:shift
positional1:shift 3
positional4:shift
s1:if [ $# -lt 1 ] ; then
s5:if [ $# -eq 1 ]
s5: if [ -f $1 ]
s8:shift
volume:if [ $# -ne 6 ]
$ echo $?
0
$ grep "RamaRao" *
$ echo $?
1
```

In fact, tomorrow if we are required to develop a program under Linux/UNIX we have to also see that if it is successful it should return 0 else other than zero. Because of this reason only, in most of the C and C++ books main will be having return 0 statement at the end. That indicates, the zero returned by main will be assigned to this positional variable $? when the main is executed. Let us do some simple experiment.

Example

Let us have a simple C program with the name a.c and with the following content.

```
#include<stdio.h>
int main()
{
return 10;
}
```

We will compile this by executing the following command.

```
$gcc –o aa a.c
```

Or

```
$cc –o aa a.c
```

We run this program:

```
$aa
```

Or

```
$./aa
```

Now, we execute echo $? command at the dollar prompt. We will use output as 10, which is returned by our main.

```
$echo $?
10
```

Checkup whether we can get same behavior if we replace **return 10** statement with **exit(10)** or not?.

We have mentioned that whenever we enter a command at the shell prompt, a new process is created and in it the command will be executed. To support this we propose to do a simple experiment. We will create a shell program aa with a single statement echo $$. That is, it displays the PID of the current shell. First, we execute echo $$ statement in the current shell and note the number returned. Then, we will start this aa program which displays sub-shells PID. This will be different from previous. After completion of the command aa, control returns to parent shell. Again, we will execute echo $$ and note the number. This will be same as first number. This experiment will support our statement that whenever a command is executed a sub-shell is created.

```
$cat>aa
echo $$
^d
$chmod u+x aa
$echo $$
$aa
Or
$./aa
$echo $$
```

Now let us talk about another positional variable, $!, whose value will be PID of the most recent back ground job. Just to see its effect, we have carried out the following experiment at the command prompt.

```
$ ls&
[1] 3396
$ A D aaa abc m2 mm mmmm pal pp s1 s6 x
1
B Pqr aaaa employee m3 mm1 mno positional1 ppp s3 s8 x2
C D aa ab hist m4 mm2 nbv positional4 rrr s5 volume xx
[1]+ Done ls
$echo $!
3396
```

If we verify the above work out, we see that PID's are same. That is, when we run a command in background, shell displays the Job number and its PID. In this case, they are 1 and 3396. The result of **echo $!** is also showing 3396.

Example

Let us assume that we wanted to write a shell program which takes a filename along the command line and adds a new line to that file at the beginning with that file name.

Here, we propose to use here the strings operator, <<<. We know $1 will be having names of the file for which a line has to be added. The following line in a file (addnewlineatbeg) will help us to achieve the same. Here, - indicates interactive input for $1 value is assigned now because of <<< operator. Thus, file name and content of the file will be sent to $1.new file.

```
$cat>>addnewlineatbeg
cat - $1 <<<$1 >$1.new
^d
$chmod addnewlineatbeg
$cat aa
Rao
Raj
$./newlineatbeg aa
$cat aa.new
aa
Rao
Raj
```

Let us assume that we wanted to modify the above shell script to add filename at the end of the file.

```
$cat>>addnewlineatend
cat $1 - <<<$1 >$1.new
^d
$chmod addnewlineatend
$./newlineatend aa
$cat aa.new
Rao
Raj
aa
```

13.4.5 The Shell Quoting

We have already explained about what happens if we enclose between single quotes, double quotes or back quotes in second chapter. However, for the sake of continuity we shall recall them here also.

- A backslash (\) protects the next character, except if it is a newline. If a backslash precedes a newline, it prevents the newline from being interpreted as a command separator, but the backslash-newline pair disappears completely.
- Single quotes ('...') protect everything (even backslashes, newlines, etc.) except single quotes, until the next single quote.
- Double quotes ("...") protect everything except double quotes, backslashes, dollar signs, and back quotes, until the next double quote. A backslash can be used to protect ", \, $, or ` within double quotes. A backslash-newline pair disappears completely; a backslash that does not precede ", \, $, `, or newline is taken literally.

Note that it is perfectly legal to quote the third character with double quotes, the fifth to seventh with single quotes, and the rest not at all, so that

12'4"$\89

is quoted like

```
      12"''"4'"$\'89
        \_/  \___/
```

Before we dwelve into details about shell quoting, let us discuss about the stages in shell interpretation of commands.

Input interpretation: an overview

Before the shell executes a command, it performs the following operations :

1. Syntax analysis (Parsing) :

remove comments, split input into words, detect quoting, detect variables, detect keywords, analyze control structures, ...

concerns: #, Space, Tab, Newline, ‵, ‶, \, ` , $VAR, =, ;, &, |, >, >>, (, {, for, while, do, if, ...

2. Parameter (variable) and command substitution :

$HOME -> /usr/home/walter

‵date‵ -> Thu Jun 18 12:40:45 MET DST 1998

concerns: $VAR, ‵...‵

3. Blank interpretation (Word Splitting) :

if previous substitutions have introduced further whitespace characters, split into words.

concerns: Space, Tab, Newline (cf. Sect. 3.1.4)

4. Filename generation (Globbing) :

*.c -> input.c main.c output.c

concerns: *, ?, [...]

5. Quote removal :

remove all quoting characters detected at parsing time.

concerns: ‵, ‶, \

The order is important: Characters that are introduced during, say, step 2 may be processed further during step 3 and 4, but they are irrelevant for step 1.

How does quoting influence the behavior of the shell? If a string/character is quoted by single quotes or by a backslash, then none of the operations above occurs (except for quote removal, of course). If a string is quoted by double quotes, then parameter and command substitution still occur, but syntax analysis, blank interpretation and filename generation are prohibited.

As a consequence, there *is* a difference between $XYZ and "$XYZ", (or between `date` and "`date`"), though in both cases, the parameter is replaced by its value: If the value of $XYZ contains globbing or whitespace characters, they are left unchanged in "$XYZ". In $XYZ, however, globbing characters are expanded and whitespace is interpreted as a separator.

13.4.5.1 The Set Builtin

This builtin is so complicated that it deserves its own section. set allows us to change the values of shell options and set the positional parameters, or to display the names and values of shell variables.

set

set [—abefhkmnptuvxBCEHPT] [-o *option*] [*argument* ...]

set [+abefhkmnptuvxBCEHPT] [+o *option*] [*argument* ...]

If no options or arguments are supplied, set displays the names and values of all shell variables and functions, sorted according to the current locale, in a format that may be reused as input for setting or resetting the currently-set variables. Read-only variables cannot be reset. In POSIX mode, only shell variables are listed.

When options are supplied, they set or unset shell attributes. Options, if specified, have the following meanings :

-a

Mark variables and function which are modified or created for export to the environment of subsequent commands.

-b

Cause the status of terminated background jobs to be reported immediately, rather than before printing the next primary prompt.

-e

Exit immediately if a pipeline, which may consist of a single simple command, a sub-shell command enclosed in parentheses, or one of the commands executed as part of a command list enclosed by braces returns a non-zero status. The shell does not exit if the command that fails is part of the command list immediately following a while or until keyword, part of the test in an if statement, part of any command executed in a && or || list except the command following the final && or ||, any command in a pipeline but the last, or if the command's return status is being inverted with !. A trap on ERR, if set, is executed before the shell exits.

This option applies to the shell environment and each sub-shell environment separately , and may cause sub-shells to exit before executing all the commands in the sub-shell.

-f

Disable file name generation (globbing).

-h

Locate and remember (hash) commands as they are looked up for execution. This option is enabled by default.

-k

All arguments in the form of assignment statements are placed in the environment for a command, not just those that precede the command name.

-m

Job control is enabled (see Job Control).

-n

Read commands but do not execute them; this may be used to check a script for syntax errors. This option is ignored by interactive shells.

-o *option-name*

Set the option corresponding to *option-name* :

allexport

Same as -a.

braceexpand

Same as -B.

emacs

Use an emacs-style line editing interface. This also affects the editing interface used for read -e.

errexit

Same as -e.

errtrace

Same as -E.

functrace

Same as -T.

hashall

Same as -h.

histexpand

Same as -H.

history

Enable command history. This option is on by default in interactive shells.

ignoreeof

An interactive shell will not exit upon reading EOF.

keyword

Same as -k.

monitor

Same as -m.

noclobber

Same as -C.

noexec

Same as -n.

noglob

Same as -f.

nolog

Currently ignored.

notify

Same as -b.

nounset

Same as -u.

onecmd

Same as -t.

physical

Same as -P.

pipefail

If set, the return value of a pipeline is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands in the pipeline exit successfully. This option is disabled by default.

posix

Change the behavior of Bash where the default operation differs from the POSIX standard to match the standard. This is intended to make Bash behave as a strict superset of that standard.

privileged

Same as -p.

verbose

Same as -v.

vi

Use a vi-style line editing interface. This also affects the editing interface used for read -e.

xtrace

Same as -x.

-p

Turn on privileged mode. In this mode, the $BASH_ENV and $ENV files are not processed, shell functions are not inherited from the environment, and the SHELLOPTS, CDPATH and GLOBIGNORE variables, if they appear in the environment, are ignored. If the shell is started with the effective user (group) id not equal to the real user (group) id, and the -p option is not supplied, these actions are taken and the effective user id is set to the real user id. If the -p option is supplied at startup, the effective user id is not reset. Turning this option off causes the effective user and group ids to be set to the real user and group ids.

-t

Exit after reading and executing one command.

-u

Treat unset variables as an error when performing parameter expansion. An error message will be written to the standard error, and a non-interactive shell will exit.

-v

Print shell input lines as they are read.

-x

Print a trace of simple commands, for commands, case commands, select commands, and arithmetic for commands and their arguments or associated word lists after they are expanded and before they are executed. The value of the PS4 variable is expanded and the resultant value is printed before the command and its expanded arguments.

-B

The shell will perform brace expansion. This option is on by default.

-C

Prevent output redirection using '>', '>&', and '<>' from overwriting existing files.

-E

If set, any trap on ERR is inherited by shell functions, command substitutions, and commands executed in a sub-shell environment. The ERR trap is normally not inherited in such cases.

-H

Enable '!' style history substitution. This option is on by default for interactive shells.

-P

If set, do not follow symbolic links when performing commands such as cd which change the current directory. The physical directory is used instead. By default, Bash follows the logical chain of directories when performing commands which change the current directory.

For example, if /usr/sys is a symbolic link to /usr/local/sys then :

```
$ cd /usr/sys; echo $PWD
/usr/sys
$ cd ..; pwd
/usr
```

If set -P is on, then :

```
$ cd /usr/sys; echo $PWD
/usr/local/sys
$ cd ..; pwd
/usr/local
```

-T

If set, any trap on DEBUG and RETURN are inherited by shell functions, command substitutions, and commands executed in a sub-shell environment. The DEBUG and RETURN traps are normally not inherited in such cases.

—

If no arguments follow this option, then the positional parameters are unset. Otherwise, the positional parameters are set to the *arguments*, even if some of them begin with a '-'.

-

Signal the end of options, cause all remaining *arguments* to be assigned to the positional parameters. The -x and -v options are turned off. If there are no arguments, the positional parameters remain unchanged.

Using '+' rather than '-' causes these options to be turned off. The options can also be used upon invocation of the shell. The current set of options may be found in $-.

The remaining N *arguments* are positional parameters and are assigned, in order, to $1, $2, ... $N. The special parameter # is set to N.

The return status is always zero unless an invalid option is supplied.

To know what are the available positional variables, shell variables, functions, etc., we can run set command with out any arguments. For example, on our machine we have got the following results.

```
$ set
!C:='C:\newcygwin\bin'
ALLUSERSPROFILE='C:\Documents and Settings\All Users'
APPDATA='C:\Documents and Settings\Administrator\Application Data'
BASH=/usr/bin/bash
BASH_ARGC=()
BASH_ARGV=()
BASH_LINENO=()
BASH_SOURCE=()
```

```
    BASH_VERSINFO=([0]=”3" [1]=”2" [2]=”39" [3]=”19" [4]=”release” [5]=”i686-pc-
cygw
    in”)
    BASH_VERSION=’3.2.39(19)-release’
    CLIENTNAME=Console
    COMMONPROGRAMFILES=’C:\Program Files\Common Files’
    COMPUTERNAME=COMPAQ-LAPTOP
    COMSPEC=’C:\WINDOWS\system32\cmd.exe’
    CVS_RSH=/bin/ssh
    DIRSTACK=()
    EUID=500
    FP_NO_HOST_CHECK=NO
    GROUPS=()
    HISTFILE=/home/Administrator/.bash_history
    HISTFILESIZE=500
    HISTSIZE=500
    HOME=/home/Administrator
    HOMEDRIVE=C:
    HOMEPATH=’\Documents and Settings\Administrator’
    HOSTNAME=compaq-laptop
    HOSTTYPE=i686
    IFS=$’ \t\n’
    INFOPATH=/usr/local/info:/usr/share/info:/usr/info:
    LOGONSERVER=’\\COMPAQ-LAPTOP’
    MACHTYPE=i686-pc-cygwin
    MAILCHECK=60
    MAKE_MODE=UNIX
    MANPATH=/usr/local/man:/usr/share/man:/usr/man:
    NUMBER_OF_PROCESSORS=2
    OLDPWD=/usr/bin
    OPTERR=1
    OPTIND=1
    OS=Windows_NT
    OSTYPE=cygwin
    PATH=’/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/cygdrive/c/WINDOWS/system32:/
    cygdrive/c/WINDOWS:/cygdrive/c/WINDOWS/System32/Wbem:/cygdrive/c/Program
Files/C
    epstral/bin:/cygdrive/c/Program Files/java/jdk15/bin:/cygdrive/c/javaexamples’
    PATHEXT=’.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH’
    PIPESTATUS=([0]=”127")
    PPID=1
    PRINTER=’EPSON Stylus CX1500 Series (Copy 1)’
    PROCESSOR_ARCHITECTURE=x86
```

```
PROCESSOR_IDENTIFIER='x86 Family 6 Model 14 Stepping 8, GenuineIntel'
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=0e08
PROGRAMFILES='C:\Program Files'
PROMPT='$P$G'
PS1='\[\e]0;\w\a\]\n\[\e[32m\]\u@\h \[\e[33m\]\w\[\e[0m\]\n\$ `
PS2='> `
PS4='+ `
PWD=/home/Administrator
SESSIONNAME=Console
SHELL=/bin/bash
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-
comments:moni
tor
SHLVL=1
SYSTEMDRIVE=C:
SYSTEMROOT='C:\WINDOWS'
TEMP=/cygdrive/c/DOCUME~1/ADMINI~1/LOCALS~1/Temp
TERM=cygwin
TMP=/cygdrive/c/DOCUME~1/ADMINI~1/LOCALS~1/Temp
UID=500
USER=Administrator
USERDOMAIN=COMPAQ-LAPTOP
USERNAME=Administrator
USERPROFILE='C:\Documents and Settings\Administrator'
WINDIR='C:\WINDOWS'
_=cal
__COMPAT_LAYER='EnableNXShowUI `
f=
```

We can use set to initialize positional variables. For example, in the following command we are initializing Hello How are you as positional variables. Thus, $1 becomes Hello, $2 becomes How and vice versa.

```
$ set Hello How are you
$ echo $1 $#
Hello 4
```

In the following command, we are initializing the results of date command to positional variables with the help of set command.

```
$ set `date`
$ echo $*
Thu May 21 07:39:03 IST 2009
$ echo $1
Thu
```

Now, we take IFS as : and then set positional variables with $4 as shown below. After this, if we display $1, we will get hours in the current time.

```
$IFS=:
$set $4
$echo $1
7
```

We know that the value of $PATH environment variable is colon separated list of directories. If we assign $PATH to positional variables, whole value of PATH is assigned to $1. However, we want to assign individual directories to each of the positional variables. To achieve this, we have initialized IFS value as colon (:) and then $PATH value is initialized through set command. See the following work out at the dollar prompt.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/cygdrive/c/WINDOWS/system32:/
cygdri
ve/c/WINDOWS:/cygdrive/c/WINDOWS/System32/Wbem:/cygdrive/c/Program
Files/Cepstra
l/bin:/cygdrive/c/Program Files/java/jdk15/bin:/cygdrive/c/javaexamples
$set $PATH
$echo $1
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/cygdrive/c/WINDOWS/system32:/
cygdri
ve/c/WINDOWS:/cygdrive/c/WINDOWS/System32/Wbem:/cygdrive/c/Program
Files/Cepstra
l/bin:/cygdrive/c/Program Files/java/jdk15/bin:/cygdrive/c/javaexamples
$ IFS=:
$ set $PATH
$ echo $1
/usr/local/bin
$ echo $2
/usr/bin
```

Now, let us one more experiment with IFS. See the following steps to know the current hour in the output of date command. Here, we have initialized output of date to positional variables through set command. Thus, fourth positional variable value will be having current time in hours, minutes and seconds format separated by colon. Thus, we initialize IFS as : and set current $4 value to positional variables. Thus, $1 becomes current hours, $2 becomes current minutes and $3 becomes current seconds. See the following work out at the dollar prompt.

```
$ set `date`
$ echo $1
Thu
$ echo $4
08:18:24
$ IFS=:
$ set $4
$ echo $1
08
```

Let us assume that we wanted to separate /usr/local/bin as usr, local and bin. Can you do it?.

Yes. We can set IFS value as / and then we initialize positional variables with /usr/local/bin using set command. Then, $1, $2 and $3 are the one's what we want.

```
$ IFS="/"
$ set /usr/local/bin
$ echo $1 $2 $3
 usr local bin
```

Check whether you are getting same on your shell or not.

13.4.5.2 Quoting Continued

We have already discussed about escape characters, quoting using single or double quotes. We have mentioned that quoting is used to take care of special characters or strings while working the shell. Here, we wanted to generalize the same.

In a nutshell, quoting is required to be used at various levels. Evidently the following forms of quoting is needed with most of the shells.

1. Terminal Quoting
2. Application Quoting
3. Shell Quoting

Terminal Quoting

Terminal quoting is very much specific to the terminal driver. They consider some characters and character sequences as special. For example, Ctrl-C (^C) is considered as a signal (SIGINT) which makes the current program to get terminated. If one wants this to be in a filename, then they have to pre-pend with Ctrl-V. However, this may vary from terminal driver to driver.

Application Quoting

Here, applications indicates either external or internal (built-in) commands. In general, these applications will be having different interpretations for different characters and character sequences. Generalizing may not be practical.

Consider that we have created a file with its name starting with '-'. At a later stage, we wanted to remove the same. Thus, when we try to execute rm command on that file, we face problem. This happens because rm commands assume filename as some option. To avoid this, we can pre-pend ./ before '-' in the filename. See the following work out at the command prompt.

```
$ cat >-a
asas
asa
^d
$ rm -a
rm: invalid option — a
Try 'rm ./-a' to remove the file '-a'.
Try 'rm —help' for more information.
$ rm ./-a
```

Also, a wide-spread convention is to interpret the special option — as the end of the option list and every further argument as a non-option. This convention is very handy but, unfortunately, there are still many commands around that do not obey it. One should not even rely on the fact that all implementations of a certain command behave in the same way. The rm command is a well-known example: Some rm implementations interpret — as the end of option list, some use - instead (which is a bad idea since - commonly denotes standard input), some rm implementations accept both, and some accept neither.

Consider the following work out at the command prompt. If –a option is after —, its effect is not seen with ls command. If it is before, we can see that it is printing all files including dot files also of directory nbv.

```
$ ls nbv
a1 a2 aa
$ ls nbv -a
. .. a1 a2 aa
$ ls nbv — -a
ls: cannot access -a: No such file or directory
nbv:
a1 a2 aa
$ ls nbv -a —
. .. a1 a2 aa
$ ls -l -a nbv (All files including dot files in long fashion will be displayed)
total 3
drwxr-xr-x+ 2 Administrator None 0 May 17 09:29 .
drwxrwxrwx+ 6 Administrator None 0 May 21 12:06 ..
-rwxr—r— 1 Administrator None 9 May 13 07:44 a1
-rw-r—r— 1 Administrator None 12 May 13 07:44 a2
-rwxr—r— 1 Administrator None 119 May 13 07:44 aa
$ ls -a nbv — -l ( -l effect will not be seen as it is after —)
ls: cannot access -l: No such file or directory
nbv:
. .. a1 a2 aa
$ ls -l nbv — -a ( -a effect will not be seen as it is after — )
ls: cannot access -a: No such file or directory
```

```
nbv:
total 3
-rwxr—r— 1 Administrator None 9 May 13 07:44 a1
-rw-r—r— 1 Administrator None 12 May 13 07:44 a2
-rwxr—r— 1 Administrator None 119 May 13 07:44 aa
```

Even this – will have special implication with set command also.

```
$ set Hello How are you
$ echo $1
Hello
$ echo $2
How
$ echo $3
are
$ echo $4
you
$ set —                    ( All positional variables will be unset )
$ echo $1 $2 $3 $4
```

13.4.6 Echo command

The echo command exists in numerous incompatible versions. Some implementations accept the option -n to suppress the trailing newline; some implementations accept various backslash-escapes for control characters (including \c to suppress the trailing newline), and some accept both. In those implementations that accept backslash-escapes, there is usually no way to disable the interpretation of backslash. To output arbitrary strings in a reliable manner, it is therefore recommended to use printf instead of echo.

Also, the read command of the Bourne shell reads a line from standard input, splits it into words using $IFS, and assigns the words to variables. Backslashes in the input escape the next character; a backslash at the end of the line can be used for line continuation. In most cases, this behavior is not desired. Newer versions of the Bourne shell implement an option -r for read that prevents interpretation of the backslash as an escape or line continuation character.

Example :

Suppose that we wanted to find whether a file mm contains the string abc* or not. Thus, the following commands we have executed. While assigning abc* for XYZ, we made mistake with an extra space in between abc and * as shown below.

```
$ XYZ='abc *'
$ grep $XYZ mm
```

Here, the parameter $XYZ is first replaced by **abc ***, then the space inside the value of $XYZ is interpreted as a separator, and finally the * expands to all files of C.W.D. So this is equivalent to

```
$ grep abc * mm
```

Thus, we have got the following type output :

```
mmmm:abc
```

```
mmmm:abcc
s6:while getopts "abc:" flag
sssss:ram abc11
sssss:ravi abc2
```

Now let's put the parameter into double quotes :

```
$ XYZ='abc *'
$ grep "$XYZ" mm
```

The parameter substitution occurs even within double quotes, but the space and the * inside the value of $XYZ are now protected by the quotes, so no further interpretation takes place here. This is therefore equivalent to

```
$ grep 'abc *' mm
```

Common misconceptions

It is important to realize that parsing takes place before parameter and command substitution. The result of parameter or command substitution is therefore subject to blank interpretation and filename generation (unless protected by double quotes), but it is not re-parsed.

Example :

Suppose that the current directory contains the files foo.1, foo.2, and bar (and nothing else). Then in the command sequence

```
$ XYZ='f* ; cat bar'
$ echo $XYZ
foo.1 foo.2 ; cat bar
```

parameter substitution produces

```
echo f* ; cat bar
```

and filename generation yields

```
echo foo.1 foo.2 ; cat bar
```

Since the semicolon was not present at parsing time, it is taken literally and does not work as a command separator. Therefore »echo« is called with the five arguments foo.1, foo.2, ;, cat, and bar.

Similarly, single/double quotes or backslashes count as quoting characters only if they are detected at parsing time, not when they are the result of parameter or command substitution :

Example :

Consider the command sequence

```
$ XYZ='ghi jkl'
$ cat abc\ def $XYZ
```

Parameter substitution in the second command produces

```
cat abc\ def ghi jkl
```

The backslash quotes the space before »def«. As $XYZ was not enclosed in double quotes, the space before jkl is subject to blank interpretation; hence cat is invoked with three arguments abc def, ghi, and jkl.

What happens if we precede the space between ghi and jkl in the assignment to $XYZ with a backslash?

```
$ XYZ='ghi\ jkl'
$ cat abc\ def $XYZ
```

Now parameter substitution produces

```
cat abc\ def ghi\ jkl
```

The first backslash was already present when the cat command was parsed, thus is quotes the following space. The second backslash, however, is the result of parameter substitution and was not present at parsing time. It is taken literally and does *not* quote the following space. So the space before jkl is still subject to blank interpretation and cat is invoked with three arguments abc def, ghi\, and jkl. There is no way to include quoting in the value of a parameter to compensate for the missing double quotes around $XYZ.

To force one more run through the parsing/substitution/globbing procedure, the eval command can be used :

Example :

Let us replace cat in the previous example by eval cat :

```
$ XYZ='ghi\ jkl'
$ eval cat abc\ def $XYZ
```

As above, parameter substitution produces

```
eval cat abc\ def ghi\ jkl
```

where the first backslash quotes the following space and the second backslash is taken literally. Thus eval is invoked with four arguments cat, abc def, ghi\, and jkl. Now eval concatenates its arguments separated by spaces, yielding

```
cat abc def ghi\ jkl
```

This string is parsed once more. At this time, the *quoted* backslash becomes a *quoting* backslash and thus the following space becomes protected, whereas the space after abc is no longer protected. Hence cat is called with three arguments abc, def, and ghi jkl.

Blank interpretation revisited

In previous Sections, we have written that during the blank interpretation phase, spaces, tabs and newlines are interpreted as separators between words. This is indeed the default behavior, but it can be changed: The characters at which the input is split are exactly those found in the value of the shell parameter $IFS (IFS = Internal Field Separators). For instance in the command sequence

```
$ A='a:b:c d'
$ echo $A
a:b:c d
```

echo is invoked with the two arguments a:b:c and d. Now let's change $IFS from its default value (a string consisting of a space, a tab, and a newline) to a colon, *i.e.,*

```
$ A='a:b:c d'
$ IFS=:
$ echo $A
a b c d
```

Then the value of $A, *i.e.,* a:b:c d is split no longer at the spaces, but at the colons, thus »echo« is invoked with three arguments a, b, and c d. In particular, setting IFS='' prevents blank interpretation completely.

Disabling filename generation

The -f option to the »set« command makes it possible to disable filename generation :

```
$ echo *
bar foo.1 foo.2
$ set -f
$ echo *
*
```

Using the command

```
    $ set +f
```

the default behavior can be restored.

This feature was added to the Bourne shell in System III; almost all Bourne shells used today support it. It is particularly useful in situations, where we *want* blank interpretation (so that enclosing a certain parameter in double quotes is not a solution) but nevertheless want to prevent filename generation.

Assignments, case commands, indirection

There are some particular situations in which neither blank interpretation nor filename generation occurs: in assignments, between the keywords »case« and »in« of case commands, and after input/output redirection operators such as »<« or »>>«. Note that in all these cases the shell syntax allows only a single word, not a sequence of words, so that blank interpretation or expansion of globbing characters might result in something syntactically illegal. Double quotes around parameters or back quoted commands serve only to prevent blank interpretation and filename generation, hence they are unnecessary in the situations mentioned above. For instance, the double quotes in the following three examples may be omitted :

```
    case "$A" in ...
    D="$A/$B/$C"
    cat result > "`date`"
```

On the other hand, quotes remain necessary if, say, the value after »=« contains literal whitespace (which is detected already at parsing time, not during blank interpretation) :

```
    D="$A $B $C"
```

The list of positional parameters: $* and $@

In the Bourne shell, the positional parameters (or command line arguments) can be accessed individually as $1, $2, ... To access the whole list of positional parameters, the two special parameters $* and $@ are available. Outside of double quotes, these two are equivalent: Both expand to the list of positional parameters starting with $1 (separated by spaces). Within double quotes, however, they differ: $* within a pair of double quotes is equivalent to the list of positional parameters, separated by *quoted* spaces, i.e., "$1 $2 ...". On the other hand, $@ within a pair of double quotes is equivalent to the list of positional parameters, separated by *unquoted* spaces, i.e.,

"$1" "$2" .... (This is the behavior if $IFS has its default value (space, tab, newline). If $IFS has a non-standard value, the evaluation of $*, $@, "$*", and "$@" is highly obscure, non-intuitive, badly documented, and varying between different shells. Avoid it.)

What happens if the list of positional parameters is empty (i.e., $# equals 0)? As one should expect, $* and $@ expand to nothing and "$*" expands to one empty argument. The question is: what should "$@" be? Originally, it expanded to one empty argument, just as "$*". But this is somewhat inconsistent: it contradicts the usual rule that "$@" yields exactly the list of all positional parameters originally passed to the current program, i.e., a list whose length equals $#. In most Bourne shells used today, the evaluation of "$@" has been regularized, so that "$@" behaves in the way a programmer expects: it expands to the list of all positional parameters, and in particular it is expands to nothing if the list of positional parameters is empty.

Shell Parameter Expansion

The '$' character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name.

When braces are used, the matching ending brace is the first '}' not escaped by a backslash or within a quoted string, and not within an embedded arithmetic expansion, command substitution, or parameter expansion.

The basic form of parameter expansion is ${*parameter*}. The value of *parameter* is substituted. The braces are required when *parameter* is a positional parameter with more than one digit, or when *parameter* is followed by a character that is not to be interpreted as part of its name.

If the first character of *parameter* is an exclamation point, a level of variable indirection is introduced. Bash uses the value of the variable formed from the rest of *parameter* as the name of the variable; this variable is then expanded and that value is used in the rest of the substitution, rather than the value of *parameter* itself. This is known as indirect expansion. The exceptions to this are the expansions of ${!*prefix**} and ${!*name*[@]} described below. The exclamation point must immediately follow the left brace in order to introduce indirection.

In each of the cases below, *word* is subject to tilde expansion, parameter expansion, command substitution, and arithmetic expansion.

When not performing substring expansion, using the form described below, Bash tests for a parameter that is unset or null. Omitting the colon results in a test only for a parameter that is unset. Put another way, if the colon is included, the operator tests for both *parameter*'s existence and that its value is not null; if the colon is omitted, the operator tests only for existence.

${*parameter*:"*word*}

If *parameter* is unset or null, the expansion of *word* is substituted. Otherwise, the value of *parameter* is substituted.

${*parameter*:=*word*}

If *parameter* is unset or null, the expansion of *word* is assigned to *parameter*. The value of *parameter* is then substituted. Positional parameters and special parameters may not be assigned to in this way.

${*parameter*:?*word*}

If *parameter* is null or unset, the expansion of *word* (or a message to that effect if *word* is not present) is written to the standard error and the shell, if it is not interactive, exits. Otherwise, the value of *parameter* is substituted.

${*parameter*:+*word*}

If *parameter* is null or unset, nothing is substituted, otherwise the expansion of *word* is substituted.

${*parameter*:*offset*}

${*parameter*:*offset*:*length*}

Expands to up to *length* characters of *parameter* starting at the character specified by *offset*. If *length* is omitted, expands to the substring of *parameter* starting at the character specified by *offset*. *length* and *offset* are arithmetic expressions This is referred to as Substring Expansion.

length must evaluate to a number greater than or equal to zero. If *offset* evaluates to a number less than zero, the value is used as an offset from the end of the value of *parameter*. If *parameter* is '@', the result is *length* positional parameters beginning at *offset*. If *parameter* is an indexed array name subscripted by '@' or '*', the result is the *length* members of the array beginning with ${*parameter*[*offset*]}. A negative *offset* is taken relative to one greater than the maximum index of the specified array. Substring expansion applied to an associative array produces undefined results.

Note that a negative offset must be separated from the colon by at least one space to avoid being confused with the ':-' expansion. Substring indexing is zero-based unless the positional parameters are used, in which case the indexing starts at 1 by default. If *offset* is 0, and the positional parameters are used, $@ is prefixed to the list.

${!*prefix}**

${!*prefix*@}

Expands to the names of variables whose names begin with *prefix*, separated by the first character of the IFS special variable. When '@' is used and the expansion appears within double quotes, each variable name expands to a separate word.

${!*name*[@]}

${!*name*[*]}

If *name* is an array variable, expands to the list of array indices (keys) assigned in *name*. If *name* is not an array, expands to 0 if *name* is set and null otherwise. When '@' is used and the expansion appears within double quotes, each key expands to a separate word.

${#*parameter*}

The length in characters of the expanded value of *parameter* is substituted. If *parameter* is '*' or '@', the value substituted is the number of positional parameters. If *parameter* is an array name subscripted by '*' or '@', the value substituted is the number of elements in the array.

${*parameter#word*}

${*parameter##word*}

The *word* is expanded to produce a pattern just as in filename expansion. If the pattern matches the beginning of the expanded value of *parameter*, then the result of the expansion is the expanded value of *parameter* with the shortest matching pattern (the '#' case) or the longest matching pattern (the '##' case) deleted. If *parameter* is '@' or '*', the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '@' or '*', the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.

${*parameter%word*}

${*parameter%%word*}

The *word* is expanded to produce a pattern just as in filename expansion. If the pattern matches a trailing portion of the expanded value of *parameter*, then the result of the expansion is the value of *parameter* with the shortest matching pattern (the '%' case) or the longest matching pattern (the '%%' case) deleted. If *parameter* is '@' or '*', the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '@' or '*', the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.

${*parameter/pattern/string*}

The *pattern* is expanded to produce a pattern just as in filename expansion. *Parameter* is expanded and the longest match of *pattern* against its value is replaced with *string*. If *pattern* begins with '/', all matches of *pattern* are replaced with *string*. Normally only the first match is replaced. If *pattern* begins with '#', it must match at the beginning of the expanded value of *parameter*. If *pattern* begins with '%', it must match at the end of the expanded value of *parameter*. If *string* is null, matches of *pattern* are deleted and the / following *pattern* may be omitted. If *parameter* is '@' or '*', the substitution operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '@' or '*', the substitution operation is applied to each member of the array in turn, and the expansion is the resultant list.

${*parameter^pattern*}

${*parameter^^pattern*}

${*parameter,pattern*}

${*parameter,,pattern*}

This expansion modifies the case of alphabetic characters in *parameter*. The *pattern* is expanded to produce a pattern just as in pathname expansion. The '^' operator converts lowercase letters matching *pattern* to uppercase; the ',' operator converts matching uppercase letters to lowercase. The '^^' and ',,' expansions convert each matched character in the expanded value; the '^' and ',' expansions match and convert only the first character in the expanded value. If *pattern* is omitted, it is treated like a '?', which matches every character. If *parameter* is '@' or '*', the case modification operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '@' or '*', the case modification operation is applied to each member of the array in turn, and the expansion is the resultant list.

${VAR+...}

If the parameter $XYZ is unset or set to the empty string, then $XYZ expands to nothing, but "$XYZ" expands to an empty argument (just as '' or ""). What can we do, if we want to have the usual effect of double quotes (i.e., no blank interpretation, no filename generation), except that an unset parameter should expand to nothing?

The ${VAR+...} construct can be used to solve our problem. Generally, ${XYZ+word} is evaluated to »word«, if the parameter $XYZ is set, and to nothing, otherwise. In particular, ${XYZ+"$XYZ"} is evaluated to "$XYZ", if the parameter $XYZ is set, and to nothing, otherwise. It is also possible to treat a parameter set to the empty string in the same way as an unset parameter; in this case we have to put a »:« before the plus sign.

Using ${VAR+...} it is also possible to pass the list of all positional parameters to a subprogram in a portable way :

```
${1+"$@"}
```

expands to nothing, if $# equals 0 (i.e., if $1 is undefined), and otherwise to "$@", that is "$1" "$2" .... This works also for shells in which "$@" expands to one empty argument if $# equals 0.

Here documents

A *here document*

```
    command <<word
...
...
```

is a special type of redirection that instructs the shell to take some part of the current source file as standard input for »command«. The delimiter »word« is subject to parameter and command substitution. All lines of the current source up to (and not including) the first following line containing only »word« constitute the standard input for »command«.

If any part of »word« is quoted, then no additional processing is done on the lines constituting the here document, hence the output of

```
$ A=123
$ cat <<'qwerty'
bcd\
$A
qwerty
```

consists of the two lines

```
bcd\
$A
```

If no part of word is quoted, then parameter and command substitution occurs, every newline preceded by a backslash is removed, and every dollar sign, back quote, or backslash must be quoted by a backslash, thus

```
$ A=123
$ cat <<qwerty
bcd\
$A
qwerty
```

produces the output

```
bcd123
```

If the special form <<- of the redirection operator is used, then all leading tab characters are stripped from the input lines (before the input lines are compared with »word«.)

Details of command substitution

When the shell encounters a string between back quotes

`cmd`

it executes cmd and replaces the back quoted string with the standard output of cmd, with any trailing newlines deleted. (There is no way to preserve these trailing newlines!)

Quoting inside back quoted commands is somewhat complicated, mainly because the same token is used to start and to end a back quoted command. As a consequence, to nest back quoted commands, the back quotes of the inner one have to be escaped using backslashes. Furthermore, backslashes can be used to quote other backslashes and dollar signs (the latter are in fact redundant). In the back quoted command is contained within double quotes, a backslash can also be used to quote a double quote. All these backslashes are removed when the shell reads the back quoted command. All other backslashes are left intact. (Usually, nested back quoted commands can be avoided using variable assignments.)

The golden rules of Bourne shell quoting

So, what are the *right* quotes to use in a shell script? It depends. But for those who prefer an easy rule to a complicated explanation, there are some rules of thumb that work very well in practice :

- Parameters and back quoted commands that should be interpreted by the shell are enclosed in double quotes. Single quotes are also protected by double quotes (or by a backslash).
- The proper way to pass the list of all positional parameters (or command line arguments) to a subprogram is some_prg ${1+"$@"}«. (For most newer Bourne shells, »some_prg "$@" has the same effect.)
- Everything else that might be maltreated by the shell is protected by single quotes.

For those who know what they are doing :

- If you are absolutely sure that the value of the parameter contains neither blanks nor globbing characters, you *may* omit the quotes. This applies for instance to the parameters $$, $#, $?, and $!. (Never do this for command line arguments!)
- If you really want the value of the parameter or back quoted command to be interpreted as a list, with embedded blanks as separators (and with expansion of globbing characters), you *must* omit the double quotes. (This occurs rather infrequently.) If you want blank interpretation but no filename generation, use

the »set -f« command; if you want filename generation but no blank interpretation, set the parameter $IFS temporarily to ‘’.

- If the parameter $XYZ is unset or set to the empty string, then $XYZ expands to nothing, but "$XYZ" expands to an empty argument (just as ‘’ or ""). If you want to have the usual effect of double quotes, except that an unset parameter should expand to nothing, use ${XYZ+"$XYZ"}. If additionally you want an empty parameter to be treated in the same way as an unset parameter, put a : before the plus sign.
- In assignments and case statements, neither blank interpretation nor filename generation takes place; thus double quotes around parameters or back quoted commands are redundant. (In the Bourne shell, the same applies to i/o redirections. Notice, however, that bash and ksh differ here, hence for portability reasons it is better not to omit the double quotes.)

Bash vs ksh

Quoting in bash or ksh is similar to Bourne shell quoting. Some differences are due to the fact that the number of interpretation steps in these shells is significantly larger: Apart from parameter and command substitution, there is also alias substitution, history substitution (only bash), brace expansion (only bash), tilde expansion, arithmetic expansion, and process substitution (not on all UNIX's). Arithmetic expansion behaves in the same way as parameter and command substitution: it takes place also within double quotes. The rules for alias substitution and history substitution are slightly confusing. Consult the manual for details. In bash, both are enabled by default only in interactive shells.

Both bash and ksh offer an alternative syntax for back quoted commands. Instead of

```
`command`
```

they allow us to write

```
$(command).
```

This form avoids most of the quoting troubles of back quoted commands.

Some further differences

Both bash and ksh perform blank interpretation only if the argument is non-constant, that is, if it contains a parameter or back quoted command :

```
bash$ IFS=,
bash$ A='a,b,c,d'
bash$ set $A
bash$ echo "$1"
a
bash$ set x,y,$A
bash$ echo "$1"
x
bash$ set x,y
bash$ echo "$1"
x,y
```

By contrast, the Bourne shell splits on $IFS even if the argument is constant :

```
$ IFS=,
$ set x,y
$ echo "$1"
x
```

In the Bourne shell, $* within double quotes expands to all positional parameters separated by quoted spaces. In both bash and ksh, the positional parameters are separated by the first character of $IFS :

```
$ set q w e r t
$ IFS=,:
$ echo "$*"
q w e r t
bash$ set q w e r t
bash$ IFS=,:
bash$ echo "$*"
q,w,e,r,t
```

In contrast to the Bourne shell, both bash and ksh perform filename generation after input/output redirection operators; bash also performs blank interpretation. If the result of (blank interpretation and) filename generation is more than one word, they produce an error. For instance, if the current directory contains exactly one file foobar whose name matches foo*, then in bash or ksh,

```
echo > foo*
```

overwrites foobar, whereas the same command in sh creates a new file foo*.

Neither bash nor ksh perform parameter or command substitution on the delimiter of a here document: The output of

```
bash$ A=abc
bash$ cat <<$A
q
abc
$A
```

consists of the two lines

```
q
abc
```

In the Bourne shell, line 4 (containing abc) would already have terminated the here document.

13.5 SHELL ARITHMETIC

The shell allows arithmetic expressions to be evaluated, as one of the shell expansions or by the let and the -i option to the declare built-ins. Evaluation is done in fixed-width integers with no check for overflow, though division by 0 is trapped and flagged as an error. The operators and their precedence, associativity, and values are the same as in the C language. The following list of operators is grouped into levels of equal-precedence operators. The levels are listed in order of decreasing precedence.

id++ id—
variable post-increment and post-decrement

++*id* —*id*
variable pre-increment and pre-decrement

- +
unary minus and plus

! ~
logical and bitwise negation

******
exponentiation

*** / %**
multiplication, division, remainder

+ -
addition, subtraction

<< >>
left and right bitwise shifts

<= >= < >
comparison

== !=
equality and inequality

&
bitwise AND

^
bitwise exclusive OR

|
bitwise OR

&&
logical AND

| |
logical OR

expr ? expr : expr
conditional operator

= *= /= %= += -= <<= >>= &= ^= |=
assignment

expr1 , expr2

comma

Shell variables are allowed as operands; parameter expansion is performed before the expression is evaluated. Within an expression, shell variables may also be referenced by name without using the parameter expansion syntax. A shell variable that is null or unset evaluates to 0 when referenced by name without using the parameter expansion syntax. The value of a variable is evaluated as an arithmetic expression when it is referenced, or when a variable which has been given the *integer* attribute using 'declare -i' is assigned a value. A null value evaluates to 0. A shell variable need not have its integer attribute turned on to be used in an expression.

Constants with a leading 0 are interpreted as octal numbers. A leading '0x' or '0X' denotes hexadecimal. Otherwise, numbers take the form [*base#*]*n*, where *base* is a decimal number between 2 and 64 representing the arithmetic base, and *n* is a number in that base. If *base#* is omitted, then base 10 is used. The digits greater than 9 are represented by the lowercase letters, the uppercase letters, '@', and '_', in that order. If *base* is less than or equal to 36, lowercase and uppercase letters may be used interchangeably to represent numbers between 10 and 35.

Operators are evaluated in order of precedence. Sub-expressions in parentheses are evaluated first and may override the precedence rules above.

The following is a simple work out along with explanation about how to use arithmetic operators in Bash.

$ let "A=10"	Declares a variable A and assigns 10 to it.
$ let "B=5"	Declares a variable B and assigns 5 to it.
$ let "C=A+B"	C value becomes 15.
$ echo $C15	
$ let "C=A-B"	C value becomes 5.
$ echo $C5	
$ let "C=A*B"	C value becomes 50.
$ echo $C50	
$ let "C=A/B"	C value becomes 2.
$ echo $C2	
$ let "C=A+"	A value is assigned to C and then it is incremented by 1.
$ echo $C10	
$ echo $A11	
$ ((A++))	A value is incremented by one.
$ echo $A12	
$ ((C=++A))	A value is first incremented by one and then the same is assigned to C.
$ echo $C13	
$ echo $A13	

$ ((C+=A))	A value is added to C.
$ echo $C26	
$ let "C/=B"	C is divided by B.
$ echo $C5	
$: $((C=C+B))	B value is added to C.
$ echo $C10	
$: $((++C))	C value is incremented by one.
$ echo $C11	
$: $[++C]	C value is incremented by one.
$ echo $C12	
$ let "D=2#10011"	Declares a variable D and assigns 10011 (19) to it.
$ echo $D19	
$ let "D=8#103"	Declares a variable D and assigns octal 103 to it.
$ echo $D67	
$ let "D=16#103"	Declares a variable D and assigns hexadecimal 103 to it.
$ echo $D259	
$ let "D=1.76" bash: let: D=1.76: syntax error: invalid arithmetic operator (error token is ".76")	We can not assign float to a variable.
We can use comma operator also with arithmetic expressions: $let "C=((A+B, A-B, A*B))" $echo $C65 $let "C=((D=5, A*B))" $echo $C $D65 5	Last value (13*5=65) is assigned to C.

13.6 PROGRAMMING CONSTRUCTS

Like all programming languages, Shell also supports variety of programming constructs such as loops, if conditions, arrays, etc. In the following sections, we explain the same.

13.6.1 If-then-else-fi condition

Like high level languages Shell supports if condition. The syntax is as follows :

-
  ```
  if [expr]
  then
      statements
  fi
  ```

-
  ```
  if [expr]
  then
      statements
  else
      statements
  fi
  ```

-
  ```
  if [expr]
  then
      statements
  elif [ expr ]
  then
      statements
  elif [expr]
  then
      statements
  else
      statements
  fi
  ```

The expressions can be using the variables as described or numbers or filenames and relational operators. Any number of elif clauses can be used in third style which is commonly called as nested if statement. However, it has to terminate with an else block.

Also, the following style of if command is also acceptable. That is, we check whether a command is successfully executed or not. If executed successfully we can execute some set of commands else some other set of statements will be executed.

```
if command
then
Some set of statements
else
Some set of statements
fi
```

Of course, if the output of the command is of not our interest, we can redirect the same to null device as shown below.

```
if command > /dev/null
then
Some set of statements
else
Some set of statements
fi
```

Example

The following program takes two integers along the command line and prints maximum of them.

```
$cat>aa
if [ $1 -gt $2 ]
then
    echo $1
else
    echo $2
fi
^d
$chmod u+x aa
$aa 10 70
70
$aa 98 01
98
```

The above program takes two numbers along the command line and displays the maximum of them. **Check what happens if we give strings instead of numbers?.**

Similar to -gt we can also use -ge, -lt, -le, -ne, and -eq to compare numeric values of two arguments. See the following for the available relational operators to be used with if.

Option	Meaning	Example
-eq	Equal to	*test a -eq b*
-gt	Greater than	*test a -gt b*
-lt	Less than (small L and small T)	*test a -lt b*
-ge	Greater than or equal to	*test a -ge b*
-le	Less than or equal to (small L and small E)	*test a -le b*
-ne	Not equal to	*test a -ne b*

We can also use logical AND and OR like operators given in the following table.

Option	Meaning	Example
-a	Logical AND	*test expression1 -a expression2*
-o	Logical OR	*test expression1 -o expression2*
!	Logical negation	*test !expression*

Example

Now consider another program which reads two integers interactively and displays the maximum of them.

```
echo Enter Two integers
read x y
if [ $x -gt $y ]
then
    echo "Maximum=" $x
else
    echo "Maximum=" $y
fi
```

Assuming that the above program is stored in a file xyz, we can change its permissions to user executable by running chmod command such as :

chmod u+x xyz

Now, by simply typing xyz at the command prompt we can run the above program.

We can also use if command between environment variables. Carry out the following experiment at the command prompt.

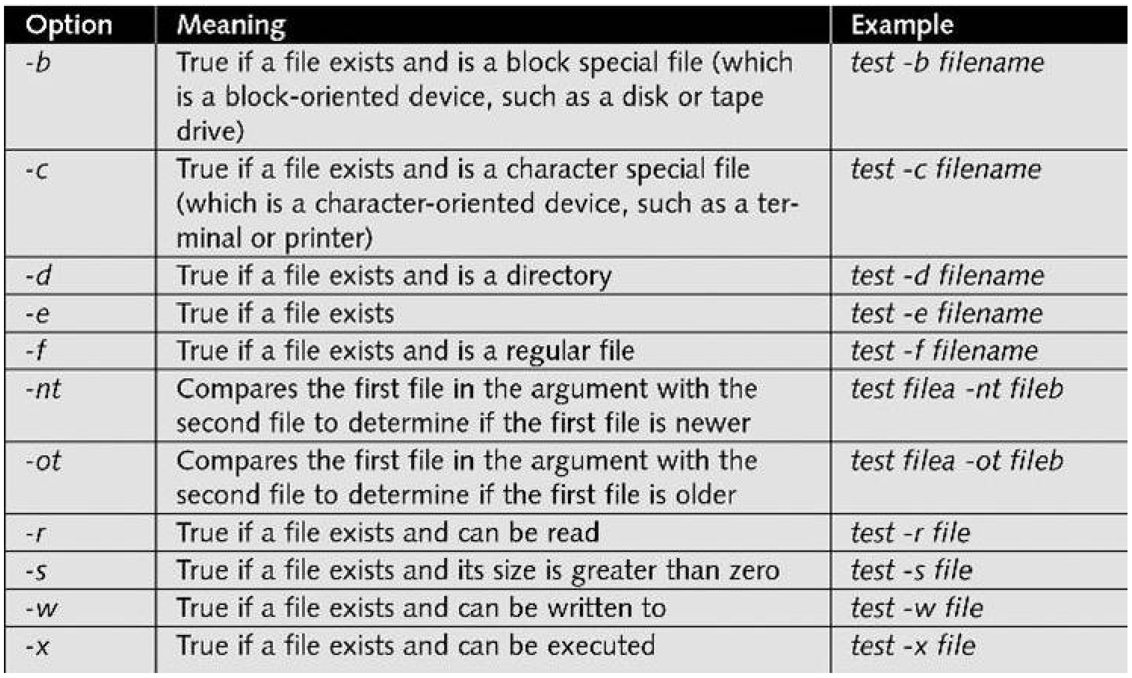
$cat>ppp

```
if [ $X -gt $Y ]
then
echo "Maximum=" $X
else
echo "Maximum=" $Y
fi
^d
$chmod u+x ppp
$X=10
$export X
$Y=90
$export Y
$ppp
Maximum=90
$X=100
$ppp
Maximum=100
```

File testing operations

Some times, we may required to find our whether given file is having reading permissions or writing permissions, etc or we may required to check whether given name is a file or a directory etc. The following can be used if conditions expression with the argument.

-r	true if the file/directory is having reading permissions
-w	true if the file/directory is having writing permissions
-x	true if the file/directory is having execution permissions
-f	true if the given argument is file
-d	true if the given argument is directory
-c	true if the argument if character special file
-b	true if the given argument is block special file



Option	Meaning	Example
-b	True if a file exists and is a block special file (which is a block-oriented device, such as a disk or tape drive)	*test -b filename*
-c	True if a file exists and is a character special file (which is a character-oriented device, such as a terminal or printer)	*test -c filename*
-d	True if a file exists and is a directory	*test -d filename*
-e	True if a file exists	*test -e filename*
-f	True if a file exists and is a regular file	*test -f filename*
-nt	Compares the first file in the argument with the second file to determine if the first file is newer	*test filea -nt fileb*
-ot	Compares the first file in the argument with the second file to determine if the first file is older	*test filea -ot fileb*
-r	True if a file exists and can be read	*test -r file*
-s	True if a file exists and its size is greater than zero	*test -s file*
-w	True if a file exists and can be written to	*test -w file*
-x	True if a file exists and can be executed	*test -x file*

```
if [ -f $1 ]
then
   echo Regular file
elif [ -d $1 ]
then
   echo Directory
elif [ -c $1 ]
then
   echo character special file
elif [ -b $1 ]
then
   echo Block special file
else
   echo others
fi
```

For the above shell script if we give /etc/passwd as argument we will get message "Regular file". If we give /etc as argument we will get message "Directory". If we give /dev/ttyS0 as argument we will get message "character special file". If we give /dev/hda1 as argument we will get message "block special file".

String comparison

= is equal to

Example :

```
if [ “$a” = “$b” ]
```

== is equal to

Example : if [“$a” == “$b”] This is a synonym for =.

Example : [$a == z*] # true if $a starts with an “z” (pattern matching)

Example : [$a == “z*”] # true if $a is equal to z*

Example : [“$a” == “z*”] # true if $a is equal to z*

!= is not equal to

Example :

```
if [ “$a” != “$b” ] #true if both the strings are different
```

This operator uses pattern matching within a [[...]] construct.

-z string is “null”, that is, has zero length

Example : if [-z “$1”] #true if $1 is null

-n string is not “null”.

Example : if [-n “$1”] # true if $1 is not null

Option or Expression	Meaning	Example
-z	Tests for a zero-length string	*test -z string*
-n	Tests for a nonzero string length	*test -n string*
string1 = string2	Tests two strings for equality	*test string1 = string2*
string1 != string2	Tests two strings for inequality	*test string1 != string2*
string	Tests for a nonzero string length	*test string*

Example

Write a shell program which takes two file names and if their contents are same then second one will be deleted.

```
if diff $1 $2
then
    rm $2
fi
```

Example

Write a shell script which says Good Morning, Good Evening, Good Afternoon depending on the present time.

```
x=‘date|awk ‘{ print $4 }’ |awk -F: ‘{ print $1 }^
if [ $x -lt 3 ]
then
    echo “Good Night”
elif [ $x -lt 12 ]
then
    echo “Good Morning”
elif [ $x -lt 16 ]
then
```

```
    echo "Good Evening"
elif [ $x -lt 22 ]
then
    echo "Good Night"
fi
```

Example

The following code if added to any script at the beginning, it allows only the specified (venkat) user to run that script.

```
if [ 'whoami' != "venkat" ];
then
echo "Sorry You are not eligible to run this program.
exit 1
fi
```

Example

The following script if added to the beginning of any shell program, it allows to run on a specified host (production) only.

```
THEHOST='uname –a|awk '{print $1}'`
if [ $THEHOST != $PRODSERV ]
then
echo "This is supposed to run on production server only. Sorry. Exiting"
exit 1
fi
```

Example

Write a program which reads three integers along the command line and prints maximum of them.

```
if [ $1 –gt $2 -a $1 –gt $3 ]
then
    echo $1
elif [ $2 –gt $3 ]
    echo $2
else
    echo $3
fi
```

The following version also works.

```
if [ $1 –gt $2]
then
    if [ $1 –gt $3 ]
    then
        echo $1
    else
        echo $3
    fi
```

```
else
    if [ $2 –gt $3 ]
    then
    echo $2
    else
    echo $3
    fi
fi
```

Example

Repeat the above program assuming that the data is read into three shell variables interactively.

```
echo Enter Three integers
read X Y Z
if [ $X –gt $Y -a $X –gt $Z ]
    then
        echo $X
    elif [ $Y –gt $Z ]
        echo $Y
    else
        echo $Z
    fi
```

The following version also works.

```
echo Enter Three integers
read X Y Z
if [ $X –gt $Y ]
then
    if [ $X –gt $Z ]
    then
        echo $X
    else
        echo $Z
    fi
else
    if [ $Y –gt $Z ]
    then
    echo $Y
    else
    echo $Z
    fi
fi
```

Example

Write a shell program which displays either of the following message depending on the number of users working on the machine.

No of Users	Message
>100	Heavily Loaded
>50	Mediumly Loaded
<50	Lowly Loaded

First, we will find out number of users working currently on the machine by piping who command output to wc –l command. Then, using if command we will display the appropriate message.

```
N=`who|wc –l`
if [ $N –gt 100 ]
then
    echo Heavily Loaded
elif [ $N –gt 50 ]
then
    echo Mediumly Loaded
else
    echo Lowly Loaded
fi
```

Example

Assume that our organization is having less bandwidth. Thus, it is decided not to allow more than 4 ftp sessions to be active at any time. If a user runs ftp command (see previous chapters on ftp command) and already 4 users are running ftp then this user should get an error message. Otherwise, ftp command should be allowed.

We do have the following data. That is, value of PATH environment variable looks like :

/bin:/sbin:/usr/bin:….

Machine language file for the ftp command is available /usr/bin with the name ftp. To start ftp program, user has to supply IP address of the remote machine or machine name.

We have to write a shell program which sees that no more than 4 users will be using ftp command at any time.

We create a shell program with the name ftp and store in /bin directory such that whenever any user types ftp, this will be executed instead of the system level ftp program which is in /usr/bin directory. This shell program first calculates number users using ftp from ps –al command output. Based on this, either he will be allowed or dis-allowed. If the number of users who are using ftp command is less than four then /usr/bin/ftp command is executed for which first command line argument is supplied as argument.

The shell program is as follows :

```
NU=`ps –al|grep "ftp"|wc –l`
if [ $NU –ge 4 ]
then
```

```
        echo Try Later
        exit
    else
    /usr/bin/ftp $1
    fi
```

13.6.2 Test command

This command can be used to check file types and compare values. We can use this with if also construct explained above.

The syntax of the usage of this command is :

```
test EXPRESSION
test [ EXPRESSION ]
```

An omitted EXPRESSION defaults to false. Otherwise, EXPRESSION is true or false and sets exit status. It is one of :

(EXPRESSION)	EXPRESSION is true
! EXPRESSION	EXPRESSION is false
EXPRESSION1 -a EXPRESSION2	both EXPRESSION1 and EXPRESSION2 are true
EXPRESSION1 -o EXPRESSION2	either EXPRESSION1 or EXPRESSION2 is true
-n STRING	the length of STRING is nonzero
-z STRING	the length of STRING is zero
STRING1 = STRING2	the strings are equal
STRING1 != STRING2	the strings are not equal
INTEGER1 -eq INTEGER2	INTEGER1 is equal to INTEGER2
INTEGER1 -ge INTEGER2	INTEGER1 is greater than or equal to INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1 is greater than INTEGER2
INTEGER1 -le INTEGER2	INTEGER1 is less than or equal to INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1 is less than INTEGER2
INTEGER1 -ne INTEGER2	INTEGER1 is not equal to INTEGER2
FILE1 -ef FILE2	FILE1 and FILE2 have the same device and i-node numbers
FILE1 -nt FILE2	FILE1 is newer (modification date) than FILE2
FILE1 -ot FILE2	FILE1 is older than FILE2
-b FILE	FILE exists and is block special
-c FILE	FILE exists and is character special
-d FILE	FILE exists and is a directory
-e FILE	FILE exists
-f FILE	FILE exists and is a regular file
-g FILE	FILE exists and is set-group-ID
-G FILE	FILE exists and is owned by the effective group ID
-h FILE	FILE exists and is a symbolic link (same as -L)
-k FILE	FILE exists and has its sticky bit set
-L FILE	FILE exists and is a symbolic link (same as -h)
-O FILE	FILE exists and is owned by the effective user ID

-p FILE	FILE exists and is a named pipe
-r FILE	FILE exists and read permission is granted
-s FILE	FILE exists and has a size greater than zero
-S FILE	FILE exists and is a socket
-t FD	file descriptor FD is opened on a terminal
-u FILE	FILE exists and its set-user-ID bit is set
-w FILE	FILE exists and write permission is granted
-x FILE	FILE exists and execute (or search) permission is granted

Except for -h and -L, all FILE-related tests dereference symbolic links. Beware that parentheses need to be escaped (e.g., by back- slashes) for shells. INTEGER may also be -l STRING, which evaluates to the length of STRING.

Example

We can modify the above shell program to find maximum of two command line arguments using test command as :

```
if test $1 -gt $2
    then
        echo $1
    else
        echo $2
    fi
```

Example

Write a shell program which takes two file's names along the command line and print the oldest file name first followed by the youngest.

To achieve this we propose to use test command with –ot option.

```
if test $1 -ot $2
    then
        echo $1
        echo $2
    else
        echo $2
        echo $1
    fi
```

13.6.3 Case construct

The following lines in file abc and is having world permissions and its name is entered in /etc/profile file. What happens?

```
case $LOGNAME in
guest) echo "It is common directory. don't disturb files" ; ;
  root) echo "Don't be Biased"; ;
  *) echo "Don't waste your time on internet" ; ;
esac
```

Answer :

If the username is guest first message will display at the login time, whereas root user logs in, the second message is displayed otherwise the third one is displayed.

Example

Explain what happen if you run this shell script?.

```
#!/bin/sh
usage=”usage:
          —help display help
      —opt display options”
case $# in
   1)
   case “$1” in
   —help) echo “$usage”; exit 0; ;
   —opt) echo “1 for kill”; ;
      exit 0;;
      *) echo “$usage”; exit 0;–;
esac
```

Answer :

If the above shell program name is assumed as **XX,** if you enter XX at command line without arguments or with option **—help it** will display the following message.

```
—help display help
—opt display options
```

otherwise it will display the following message.

```
1 for kill
```

What is Interactive and Non-Interactive Shell

While we are in interactive shell, it takes our input from key board and gives output on to the terminal in which we are working. Interactive shells uses the configuration files during their starting where as non-interactive shell (sub-shells) are not. We can run either of the following shell programs to check whether your shell is interactive or not.

To determine within a startup script whether or not Bash is running interactively, test the value of the ‘-’ special parameter. It contains i when the shell is interactive.

```
case “$-” in
*i*) echo This shell is interactive ;;
*) echo This shell is not interactive ;;
esac
```

Alternatively, startup scripts may examine the variable PS1; it is unset in non-interactive shells, and set in interactive shells. Thus :

```
if [ -z “$PS1” ]; then
    echo This shell is not interactive
else
    echo This shell is interactive
fi
```

Interactive shell's changes their behavior in several ways.

1. Startup files are read and executed as described in Bash Startup Files.
2. Job Control is enabled by default. When job control is in effect, Bash ignores the keyboard-generated job control signals SIGTTIN, SIGTTOU, and SIGTSTP.
3. Bash expands and displays PS1 before reading the first line of a command, and expands and displays PS2 before reading the second and subsequent lines of a multi-line command.
4. Bash executes the value of the PROMPT_COMMAND variable as a command before printing the primary prompt, $PS1.
5. Readline is used to read commands from the user's terminal.
6. Bash inspects the value of the ignoreeof option to set -o instead of exiting immediately when it receives an EOF on its standard input when reading a command.
7. Command history and history expansion are enabled by default. Bash will save the command history to the file named by $HISTFILE when an interactive shell exits.
8. Alias expansion is performed by default.
9. In the absence of any traps, Bash ignores SIGTERM.
10. In the absence of any traps, SIGINT is caught and handled. SIGINT will interrupt some shell built-ins.
11. An interactive login shell sends a SIGHUP to all jobs on exit if the huponexit shell option has been enabled.
12. The -n invocation option is ignored, and 'set -n' has no effect.
13. Bash will check for mail periodically, depending on the values of the MAIL, MAILPATH, and MAILCHECK shell variables.
14. Expansion errors due to references to unbound shell variables after 'set -u' has been enabled will not cause the shell to exit.
15. The shell will not exit on expansion errors caused by *var* being unset or null in ${*var*:?*word*} expansions.
16. Redirection errors encountered by shell built-ins will not cause the shell to exit.
17. When running in POSIX mode, a special builtin returning an error status will not cause the shell to exit.
18. A failed exec will not cause the shell to exit.
19. Parser syntax errors will not cause the shell to exit.
20. Simple spelling correction for directory arguments to the cd builtin is enabled by default (see the description of the cdspell option to the shopt builtin in The Shopt Builtin).
21. The shell will check the value of the TMOUT variable and exit if a command is not read within the specified number of seconds after printing $PS1.

13.6.4 While loop

Like any other high level language, shell also supports loops which can be used to execute some set of instructions repeatedly, probably in given number of times.

The following styles of while loop are used to execute a group of statements eternally.

```
while :
do
—
—
done
or
while true
do
—
—
done
```

The following while loop structure is used execute a group of statements as long as the expression is true.

```
while [ expr ]
do
—
—
done
```

Here, the expr can be having relational or string comparison operations between command line arguments, environment variables, shell variables or literals both numbers or strings. As long as the expr is true the statements between do and done will be executed.

```
while command
do
—
—
done
```

The above style of while loop execute the group of statements as long as the given command is executed successfully.

```
while test command
do
—
—
done
```

This version of while loop also behaves similar to the above while loop.

Example

Write a shell program which informs as soon as a specified user whose name is given along the command line is logged into the system.

```
while :
do
    if who|grep $1 >/dev/null
    then
        echo $1 is logged in
```

```
            exit
        else
            sleep 60
        fi
    done
```

Assuming that the above code is available in a file ckuser. After giving user executable permissions, we can run the same along with a legal username. If we want we can run this program in background. It informs us as soon as that particular user log's.

The above program can be modified using case construct as follows.

```
    while :
    do
        who|grep $1 >/dev/null
        case "$"" in
        0)echo $1 is logged in
            exit;;
        *) sleep 60;;
        esac
    done
```

In the above program, we have used the exit status of the piping command in designing the case construct. If the specified user is currently logged in, then the piping command successfully completes. Thus, its exit status value will be 0.

Example

Let us assume that we wanted to establish a talk (chatting) session with the user as soon as he log's in.

```
    while :
    do
        if who|grep $1 >/dev/null
        then
            talk $1
            exit
        else
            sleep 60
        fi
    done
```

Example

Write a shell program which takes a source file name and other duplicate file names as command line arguments and creates the duplicate copies of the first file with the names given as subsequent command line arguments. For example, if we assume that the shell program file name is MCP and file for which we wanted to create the duplicates is PP, then the following command at the dollar prompt has to create duplicates of file PP with the names s1, s2, s3, s4.

$MCP PP s1 s2 s3 s4

Solution 1 :

```
while [ “$2” ]
do
    cp $1 $2
    shift
done
```

Solution 2 :

```
X=$1
shift
while [ “$1” ]
do
    cp $X $1
    shift
done
```

Solution 3 :

```
X=$1
shift
while [ $# -ne 0 ]
do
    cp $X $1
    shift
done
```

Example

Write a shell program which takes a source file name and directories names as command line arguments and prints message yes if the file is found in any of the given directories.

Solution 1 :

```
X=$1
shift
while [ “$1” ]
do
if [ -f $1/$X ]
        then
        echo Yes
        exit
    else
      shift
    fi
done
echo No
```

We can even remove the else part in the if condition.

```
X=$1
shift
while [ “$1” ]
do
if [ -f $1/$X ]
      then
      echo Yes
      exit
   fi
     shift
   done
     echo No
```

Example

The following program takes primary name of a C language program and it executes the same if it compiles successfully otherwise automatically it brings the vi editor to edit the C language program. This repeats till the program is corrected to have no compile time errors.

```
while true
do
gcc -o $1 $1.c
case “$?” in
0)echo executing
   $1
   exit ;;
   *)vi $1.c ;;
   esac
   done
```

Assuming that the above program is in a file COMP and we have a C program file with the name abc.c; then we can run the above program as :

$COMP abc

Example

Write a shell script to lock our terminal till you enter a password.

```
trap “ “1 2 3
echo terminal locked
read key
pw=xxxxxx
while [ “$pw” = “xxxxxx” ]
do
echo Enter password
stty -echo
read pw
stty sane
done
```

Example

The following shell script (m3) executes inner for loop as long as we enter 10. If we enter any number other than 10, control exits from the outer while loop.

```
while true
do
while true
do
read x
if [ $x -eq 10 ]
then
break 1
else
break 2
fi
done
done
```

$ ./m3

```
10
10
10
9
```

Here, break 1 means exiting from the inner while loop. While break 2 means exiting from the outer while loop.

13.6.5 Until loop

```
until [ expr ]
do
—
—
done
```

Here, the expr can be having relational or string comparison operations between command line arguments, environment variables, shell variables or literals both numbers or strings. As long as the expr is false the statements between do and done will be executed.

```
until command
do
—
—
done
```

The group of statements between do and done will be executed as long the command is failure.

Example

Write a shell program which informs as soon as a specified user whose name is given along the command line is logged into the system.

```
until who|grep $1 >/dev/null
do
sleep 60
done
echo $1 is logged in
```

Example

Write a shell program which takes a source file name and other duplicate file names as command line arguments and creates the duplicate copies of the first file with the names given as subsequent command line arguments.

Solution 1 :

```
until [ $# -eq 1 ]
do
    cp $1 $2
    shift
done
```

Solution 2 :

```
X=$1
shift
until [ $# -eq 0 ]
do
    cp $X $1
    shift
done
```

Example

Write a shell program which takes a source file name and directories names as command line arguments and prints message yes if the file is found in any of the given directories.

```
X=$1
shift
until [ $# -ne 0 ]
do
if [ -f $1/$X ]
    then
        echo Yes
        exit
else
    shift
fi
 done
 echo No
```

Example

The following program takes primary name of a C language program and it executes the same if it compiles successfully otherwise automatically it brings the vi editor to edit the C language program. This repeats till the program is corrected to have no compile time errors.

```
until gcc -o $1 $1.c
vi $1.c
done
echo executing
$1
```

13.6.6 For loop

```
for var in list
do
——
——
done
```

Example

What is the **output** of the following for loop?.

```
for x in .
do
ls $x
done
```

Answer: lists all file names in P.W.D.

Example

What is the output of the following program?.

```
for x in *
do
ls $x
done
```

Answer: lists all file names in P.W.D.

Example

What is the output of the following program?.

```
for x in `ls ..`
do
ls $x
done
```

Answer: lists all file names of parent directory of P.W.D..

Example

What is the output of the following program

```
IFS=#
for x in .#..
do
ls $x
done
```

Answer: lists file names in P.W.D and its parent directory.

Example

Write a shell program which takes a source file name and other duplicate file names as command line arguments and creates the duplicate copies of the first file with the names given as subsequent command line arguments.

```
X=$1
shift
for Y in $*
do
    cp $X $Y
    shift
done
```

Example

Write a shell program which takes a source file name and directories names as command line arguments and prints message yes if the file is found in any of the given directories else prints no.

```
X=$1
shift
for Y in $*
do
    if [ -f $Y/$X ]
then
        echo Yes
        exit
    fi
done
echo No
```

Example

What does the following script does?.

```
    a=”$1"
    shift
    readonly a
    for I in $*
    do
cp $a $I
    shift
        done
```

Answer: - Makes the first command line argument as readonly. Then duplicates of the same will be created with the names $2 $3... and so on.

Example

What is the **output of following** shell script?.

```
set `who am i`
for in i *
do
    mv $i $i.$1
done
```

Answer: - It adds username as extension to files of **P.W.D.**

Example

What does the following shell script?.

```
for x in `ls`
do
chmod u=rwx $x
done.
```

Answer: - changes permissions of files in **P.W.D** as **rwx** for users.

Example

What does the following shell script does?.

```
for x in *.ps
do
compress $x
mv $x.ps.Z /backup
done
```

Answer : - It compresses all postscript files in P.W.D and moves to /backup directory.

Example

What does the following shell script does?.

```
for i in $*
do
cc -C $i.c
done
```

Answer : - Creates object files for those c program files whose primary names are given along the command line to the above shell script.

Example

What does the following shell script does.

```
for i in *.dvi
do
dvips $i.dvi | lpr
done
```

Answer : - It converts all dvi files in P.W.D and converts to postscript and redirects to printer.

Example

Explain what happens if you run the following shell script.

```
I=1
    for i in $*
    do
    J=I
    for j in $*
    if [ $I -ne $J ]
    then
        if diff $i $j
            then
                rm $j
        else
            J=`expr $J + 1`
        fi
    fi
done
I=`expr $I + 1`
done
echo $I
```

Answer : Takes a set of file names along the command line and removes if there exists duplicate files.

Example

Write a shell program such that files (only) of P.W.D will contain PID of the current shell (in which shell script is running) as their extension.

```
for x in `ls`
    do
if [ ! -d $x ]
        then
            mv $x $x.$$
    fi
done
```

Example

Write a program which takes a filename (of C.W.D) along command line and prompt the user whether to delete or retain duplicate files (of C.W.D) of this file. Based on user's response the duplicate file can be retained or removed.

```
for X in *
do
if [ -f $X ]
then
if [ "$X" != "$1" ]
then
```

```
    if diff $X $1 >/dev/null
    then
        echo Do you want to delete $X
        read ans
        case "$ans" in
        [yY]|[yY][eE][sS])rm $X;;
        esac
    fi
fi
fi
done
```

Example

Write a program which traverses the files of C.W.D directory and for each file prompt the user whether to compress the file or not. Based on user's response the file will be compressed or retained as it is.

```
for X in *
do
if [ -f $X ]
        echo Do you want to compress $X
        read ans
        case "$ans" in
        [yY]|[yY][eE][sS])gzip $X;;
        esac
    fi
done
```

Example

Two files contains a list of words to be searched and list of filenames respectively. Write a shell script which display search word and its number of occurrences over all the files as a tabular fashion.

```
echo "Word Filename Occurrences"
for x in `cat $file1`
    do
        for y in `cat $file2`
            do
                I=0
            for z in `cat $y`
                do
                    if [ "$x" == "$y" ]
                      then
                        I=`expr $I + 1`
                    fi
                done
        echo $x $y $I
    done
```

Example

Two files contains a list of words to be searched and list of filenames respectively. Write a shell script which display search word over all the files and display as a table with yes or no for each word and file combination respectively.

```
echo "Word Filename Occurrences"
for x in 'cat $file1'
do
    for y in 'cat $file2'
        do
        done
        echo $x $y $I
    done
```

Example

Write a shell script which accepts in command line user's name and informs you as soon as he/she log into system.

```
uname=$1
while :
do
who | grep "$uname">/dev/null
if [ $? -eq 0 ]
  then
    echo $uname is logged in
    exit
    else
    sleep 60
done
```

Example

Write a shell script which lists the filenames of a directory (reading permissions are assumed to be available) which contains more than specified no of characters.

```
read size
foreach x
do
y='wc -c $x'
if [ $y -gt $size ]
echo $x
fi
done
```

Example

Write a shell script which displays names of c programs which uses a specified function.

```
read functname
for prog in *.c
do
if grep $functname $prog
then
echo $prog
fi
done
```

Example

Write a shell script which displays names of the directories in PATH one line each.

Answer **:**

```
IFS=:
set `echo $PATH`
for i in $*
do
    echo $i
done
IFS=:
for i in $PATH
do
    echo $i
done
```

Example

A file (ABC) having a list of search words. Write a program that takes a file name as command line argument and print's success if at least one line of the file contains all the search words of ABC otherwise display failure.

```
cat $1 |
while read xx
do
FLAG=1
for y in `cat ABC`
do
if ! grep $y $xx
then
FLAG=0
break
fi
done
if $FLAG -eq 1
then
echo "SUCCESS"
exit
```

```
fi
done
echo “FAILURE”
```

Example

Write a shell script which removes empty files from PWD and changes other files time stamps to current time.

```
for x in .
do
if [ -f $x ]
then
    if [ -s $x ]
        then
            touch $x
    else
        rm $x
    fi
fi
done
```

Example

Write a program to calculate factorial value

```
#!/bin/sh
factorial()
{
 if [ “$1” -gt “1” ]; then
    i=‘expr $1 - 1‘
    j=‘factorial $i‘
    k=‘expr $1 \* $j‘
    echo $k
 else
    echo 1
 fi
}
while :
do
    echo “Enter a number:”
    read x
    factorial $x
done
```

Example

Write a program which reads a digit and prints its BCD code.

```
#!/bin/sh
convert_digit()
{
case $1 in
    0) echo “0000 \c” ;;
    1) echo “0001 \c” ;;
    2) echo “0010 \c” ;;
    3) echo “0011 \c” ;;
    4) echo “0100 \c” ;;
    5) echo “0101 \c” ;;
    6) echo “0110 \c” ;;
    7) echo “0111 \c” ;;
    8) echo “1000 \c” ;;
    9) echo “1001 \c” ;;
    *) echo
      echo “Invalid input $1, expected decimal digit”
      ;;
    esac
}

ecimal=$1
stringlength=‘echo $decimal | wc -c‘
char=1
while [ “${char}” -lt “${stringlength}” ]
do
  convert_digit ‘echo $decimal|cut -c ${char}‘
  char=‘expr ${char} + 1‘
done
echo
```

Example

Write a program which reads a filename along the command line and prints frequency of the occurrence of words.

```
#!/bin/sh
# Count the frequency of words in a file.
# Syntax: frequency.sh textfile.txt
INFILE=$1
WORDS=/tmp/words.$$.txt
COUNT=/tmp/count.$$.txt
if [ -z “$INFILE” ]; then
    echo “Syntax: `basename $0` textfile.txt”
    echo “A utility to count frequency of words in a text file”
    exit 1
```

```
fi
if [ ! -r $INFILE ]; then
    echo "Error: Can't read input file $INFILE"
    exit 1
fi
> $WORDS
> $COUNT
# First, get each word onto its own line...
# Save this off to a temporary file ($WORDS)
# The "tr '\t' ' '" replaces tabs with spaces;
# The "tr -s ' '" removes duplicate spaces.
# The "tr ' ' '\n' replaces spaces with newlines.
# Note: The "tr "[:punct:]"" requires GNU tr, not UNIX tr.
cat $INFILE | tr "[:punct:]" ' '| tr '\t' ' ' | tr -s ' ' | tr ' ' '\n'| while read f
do
    echo $f >> $WORDS
done
# Now read in each line (word) from the temporary file $WORDS ...
while read f
do
    # Have we already encountered this word?
grep — " ${f}$" $COUNT > /dev/null 2>&1
if [ "$?" -ne "0" ]; then
    # No, we haven't found this word before... count its frequency
    NUMBER=`grep -cw — "${f}" $WORDS`
    # Store the frequency in the $COUNT file
    echo "$NUMBER $f" >> $COUNT
fi
done < $WORDS
# Now we have $COUNT which has a tally of every word found, and how
# often it was encountered. Sort it numerically for legibility.
# We can use head to limit the number of results - using 20 as an
example.
echo "20 most frequently encountered words:"
sort -rn $COUNT | head -20
# Now remove the temporary files.
#rm -f $WORDS $COUNT
```

Example

The following shell program creates a duplicate file for each of the files of the sub-directories of current working directory.

```
for f in */*
do
cp $f $f.bak
done
```

Example

The following program generates checksum using md5sum command for each of the files of the sub-directories of current working directory.

```
for f in */*
do
md5sum $f > $f.sum
done
```

13.6.7 Select Command

The select construct allows the easy generation of menus. It has almost the same syntax as the for command :

select *name* [in *words* ...]; do *commands*; done

The list of words following in is expanded, generating a list of items. The set of expanded words is printed on the standard error output stream, each preceded by a number. If the 'in *words*' is omitted, the positional parameters are printed, as if 'in "$@"' had been specified. The PS3 prompt is then displayed and a line is read from the standard input. If the line consists of a number corresponding to one of the displayed words, then the value of *name* is set to that word. If the line is empty, the words and prompt are displayed again. If EOF is read, the select command completes. Any other value read causes *name* to be set to null. The line read is saved in the variable REPLY.

The *commands* are executed after each selection until a break command is executed, at which point the select command completes.

Example

The following script displays a menu.

```
select x in rao raj
do
case $x in
rao)echo RAO;;
raj)echo RAJ;;
*)No Problem;;
esac
done
```

Assuming that the above program is in a file named pp, we can execute the same as :

```
$ ./pp
Output:
1) rao
2) raj
#? 1 <- User Input
RAO
#? 2 <- User Input
RAJ
#?
```

13.6.8 Arrays

The original Bourne shell does not have arrays. Bash version 2.x does have arrays, however. An array can be assigned from a string of words separated by white spaces or the individual elements of the array can be set individually.

```
colours=(red white green)
    colours[3]="yellow"
```

An element of the array must be referred to using curly braces.

```
echo ${colours[1]}
white
```

Note that the first element of the array has index 0. The set of all elements is referred to by ${colours[*]}.

```
echo ${colours[*]}
red white green yellow
echo ${#colours[*]}
4
```

As seen the number of elements in an array is given by ${#colours[*]}.

We can define arrays and initialize them using read and here the strings as shown below.

Here, a variable X is defined. The operator <<< is known as here the strings. It expands the variable and gives as the input for the command (here the command is read). The –a option read command is used to define an array. Here, we are declaring array W. Thus, the words of the string X is assigned to the elements of the array.

```
$X="Hello How are you"
$ read -r -a W<<<$X
$ echo $[W[0]}
>
>^d
$ echo ${W[0]}
Hello
$ echo ${W[1]}
How
$ echo ${W[2]}
are
$ echo ${W[3]}
You
```

Example

The following program takes a matrix into a 1-D array and then prints its transpose.

```
echo "Enter Number of rows"
read r
echo "Enter Number of columns"
read c
i=1
echo "Enter elements"
```

```
            N=`expr $r \* $c `
            until [ $i –gt $N ]
            do
                read p;
                a[$i]=$p;
                i=`expr $i + 1`
            done
            i=1
            echo "Transpose of the Matrix"
            while [ $i –le $c ]
            do
                j=0;
                k=$i
                until [ $j –eq $r ]
                do
                    echo -n -e ${a[$k]} "\t"
                    k=`expr $k + $c`
                    j=`expr $j + 1`
                done
                i=`expr $i + 1`
                echo
            done
```

Sample workout at the command prompt assuming that the program is in the file mattranspose and having user executable permissions.

```
$ ./mattranspose
    Enter Number of rows
    3
    Enter Number of columns
    2
    Enter elements
    2
    2
    3
    3
    4
    4
    Transpose of the Matrix
            2       3       4
            2       3       4
```

A useful facility in the C-shell is the ability to make arrays out of strings and other variables. The round parentheses '(..)' do this. For example, look at the following commands.

```
set array = ( a b c d )
echo $array[1]
a
echo $array[2]
b
echo $array[$#array]
d
set noarray = ( "a b c d" )
echo $noarray[1]
a b c d
echo $noarray[$#noarray]
a b c d
```

The first command defines an array containing the elements 'a b c d'. The elements of the array are referred to using square brackets '[..]' and the first element is '$array[1]'. The last element is '$array[4]'. *NOTE: this is not the same as in C or C++ where the first element of the array is the zero'th element!*

The special operator '$#' returns the number of elements in an array. This gives us a simple way of finding the end of the array. For example

```
echo $#path
23
echo "The last element in path is $path[$#path]"
```

The last element in path is.

13.6.9 Getopts and getopt with Bash

With the help of Bourn shells built-in getopts and an external program getopt, we can design command line interfaces for shell programs. Of course, getopt program may not be available in some old UNIX versions.

We know that UNIX commands are having variety of options. Based on the options given by the user, the style of working of a command changes. For example, if we –l option with ls command we will get details of the files in long fashion (one file or directory information in one line). Similarly, for example, we know —help option with most of the UNIX commands displays terse help of them. Also, with some commands we will be specifying an option and followed by a value for it along the command line. For example,

sort –o abc xyx

gcc –o abc abc.c

Here, abc is the value for option –o.

Also, we know that "ls –l –a –t" is equivalent to "ls –lat". That is, a group of options are grouped together.

Like this, options are specified as hyphen followed a single character, two hyphens followed by a single character, hyphen followed by a word, or hyphen and a character followed by a word (like –o in the above statements). We may need such a type of interface to the shell programs also. Here, these getopts and getopt will be useful.

getopts

getopts *optstring name* [*args*]

getopts is used by shell scripts to parse positional parameters. *optstring* contains the option characters to be recognized; if a character is followed by a colon, the option is expected to have an argument, which should be separated from it by white space. The colon (':') and question mark ('?') may not be used as option characters. Each time it is invoked, getopts places the next option in the shell variable *name*, initializing *name* if it does not exist, and the index of the next argument to be processed into the variable OPTIND. OPTIND is initialized to 1 each time the shell or a shell script is invoked. When an option requires an argument, getopts places that argument into the variable OPTARG. The shell does not reset OPTIND automatically; it must be manually reset between multiple calls to getopts within the same shell invocation if a new set of parameters is to be used.

When the end of options is encountered, getopts exits with a return value greater than zero. OPTIND is set to the index of the first non-option argument, and name is set to '?'.

getopts normally parses the positional parameters, but if more arguments are given in *args*, getopts parses those instead.

getopts can report errors in two ways. If the first character of *optstring* is a colon, *silent* error reporting is used. In normal operation diagnostic messages are printed when invalid options or missing option arguments are encountered. If the variable OPTERR is set to 0, no error messages will be displayed, even if the first character of optstring is not a colon.

If an invalid option is seen, getopts places '?' into *name* and, if not silent, prints an error message and unsets OPTARG. If getopts is silent, the option character found is placed in OPTARG and no diagnostic message is printed.

If a required argument is not found, and getopts is not silent, a question mark ('?') is placed in *name*, OPTARG is unset, and a diagnostic message is printed. If getopts is silent, then a colon (':') is placed in *name* and OPTARG is set to the option character found.

Assume that we wanted to have a shell program (Pqr) which prints its name. If we use –l or –L option with it, its name will be printed in lower case. If we use –u or –U with it, its name will be printed in upper case. Otherwise, it should prints its name as it is.

That is it should have the following command line interface.

Pqr	Should display Pqr
Pqr –l	Should display pqr
Pqr –L	Should display pqr
Pqr –u	Should display PQR
Pqr –U	Should display PQR

Example

The following shell program with the name Pqr does the same. Here, we propose to use getopts command with optstring value is lLuU as we will be using the option –l, -L, -u and –U.

```
X=`basename $0`
if [ $# -eq 0 ]
then
echo $X
else
while getopts lLuU val
do
case "$val" in
 [lL])echo $X|tr A-Z a-z ;;
 [uU])echo $X|tr a-z A-Z;;
esac
done
fi
```

Example

Let us assume that we want a shell program (CAT) to display a given files content with a different command line interface given as :

CAT filename	This should work like usual cat command.
CAT –l filename	It should display the content of the given file in lowercase
CAT –L filename	It should display the content of the given file in lowercase
CAT –u filename	It should display the content of the given file in uppercase
CAT –lUfilename	It should display the content of the given file in uppercase

To achieve this, we have written the following shell program with the name CAT. Here, with getopts we have used optstring as l:L:u:U: as each of the options –l, -L, -u and –U needs a value or argument.

```
if [ $# -eq 1 ]
then
    if [ -f $1 ]
    then
     cat $1
    fi
else
while getopts l:L:u:U: val
do
case "$val" in
 [lL])cat $OPTARG|tr A-Z a-z ;;
 [uU])cat $OPTARG|tr a-z A-Z;;
esac
done
fi
```

Example

Now let us assume that we wanted to write a shell program to calculate room volume and surface area to be white washed. User is supposed to specify the dimensions along with the options –l, -L, -b, -B, -h or –H.

```
if [ $# -ne 6 ]
then
echo Wrong usage
echo Correct usage is: $0 –l 10 –b 8 –H 4
else
while getopts l:b:h: val
do
    case "$val" in
    [lL])l=$OPTARG;;
    [bB])b=$OPTARG;;
    [hH])h=$OPTARG;;
    esac
done
echo Volume=`expr $l \* $b \* $h`
echo Surface Area = ` expr 2 \* $h \* \( $l + $b \) + $l \* $b `
fi
```

Assuming that the above shell code is in a file volume, the following is our workout at the command prompt.

```
$ ./volume -b 8 -l 10 -h 6
    Volume=480
    Surface Area = 296
$ ./volume -b 8 -l 10 -h
    Wrong usage
    Correct usage is: ./volume -l 10 -b 8 -H 4
```

The built-in "getopts" is called each time you want to process an argument, and it doesn't change the original arguments . A simple script, s6 is written to explain. It is having the following lines of code. Remember that the "$OPTIND" will contain the index of the argument that will be examined next.

```
#!/bin/bash
while getopts "abc:" flag
do
    echo "$flag" $OPTIND $OPTARG
done
```

We change the permissions of s6 such that it will have user executable.

```
$ chmod u+x s6
$ ./s6 -c rao -a -b
    c 3 rao
    a 4
    b 5
```

```
$ ./s6 -abc rao
    a 1
    b 1
    c 3 rao
$ ./s6 -bac rao
    b 1
    a 1
    c 3 rao
$ ./s6 -ab -c rao
    a 1
    b 2
    c 4 rao
$ ./s6 -aab -c rao
    a 1
    a 1
    b 2
    c 4 rao
$ ./s6 -b -a -c rao
    b 2
    a 3
    c 5 rao
```

The getopts raises an error if we have not supplied a value for an option. We know –a and –b options does not require value. While –c needs a value. We have tried to execute the above program as follows. We have got an error indicating that –c needs a value. The flag variable value becomes "?".

```
$ ./s6 -b -a -c
    b 2
    a 3
    ./s6: option requires an argument — c
    ? 4
```

The getopts raises an error if we give an extra option. For example:

```
$ ./s6 -b -A -c rao
    b 2
    ./s6: illegal option — A
    ? 3
    c 5 rao
```

If needed, we can reset OPTIND value and reprocess the arguments. Assume that the following lines are in a file s7 and executable permissions are given to the same.

Example

```
#!/bin/bash
while getopts "abc:def:ghi" flag
do
    echo "$flag" $OPTIND $OPTARG
done
echo "Resetting"
OPTIND=1
while getopts "abc:def:ghi" flag
do
echo "$flag" $OPTIND $OPTARG
done
```

We'll give it more arguments so that you can observe it at work :

$./s7 -a -bc foo -f "foo bar" -h -gde

```
a 2
b 2
c 4 foo
f 6 foo bar
h 7
g 7
d 7
e 8
Resetting
a 2
b 2
c 4 foo
f 6 foo bar
h 7
g 7
d 7
e 8
```

Example

Now let assume that we wanted to write another shell program to join files either vertically or horizontally. If user enters file names along with first argument as –H or –h, we will join the file using cat command. If the first argument is –v or –V then we will join the file using join command. Let us assume that this program is available in a file s7 with properly set permissions.

```
separator="\t"   # default separator is tab
X=$1
shift
while getopts vVhH val $X
do        case "$val" in
```

```
                h|H)  cat "$@";;
                v|V)  paste –d "$seperator" "$@";;
                [?])  echo "Usage: $0 [-v] [-V] [-h][-H] files ..."
                      exit 1;;
                esac
     done
```

Let us our workout at the command prompt.

```
$ cat A
    Hi
$ cat B
    Hello
$ cat D
    This is D
$ ./s8 -H A B D
    Hi
    Hello
    This is D
$ ./s8 -V A B D
    HiHello This is D
```

The getopt command

The getopt command is often used in shell scripts to parse command line options. The first command argument, optiondesc, contains each option letter that is valid in the following command argument strings. An option letter followed by a colon (:) means that the preceding option letter requires a further argument (as in -o file).

getopt considers each argument that begins with a - to be a potential option and prints an error if it does not find the argument in optiondesc. Scanning for further options stops at the first argument which does not begin with - or with an argument that is —. In either case, the options are separated from the rest of the non-option argument strings by a — string.

The most common construct for using getopt is

```
set - - $(getopt [-c cmdname] optiondesc "$@")
```

This may be used inside the MKS Korn shell to parse the arguments to a shell script; see sh for more about the shell.

Options

-c cmdname

uses cmdname rather than getopt when displaying error messages.

Example

The simple use of "getopt" is shown in the following mini-script program which is in the file s9. We assume that we have given u+x permissions for the same.

```
#!/bin/bash
echo "Before getopt"
for i
do
    echo $i
done
args=`getopt abc:d $*`
set — $args
echo "After getopt"
for i
do
    echo "—>$i"
done
```

What we have said is that any of -a, -b, -c or -d will be allowed, but that -c is followed by an argument (the "c:" says that).

$ ./s9 -abc foo

```
Before getopt
-abc
foo
After getopt
—>-a
—>-b
—>-c
—>foo
—>—
```

We start with two arguments, and "getopt" breaks apart the options and puts each in its own argument. It also added "—".

Of course "getopt" doesn't care that we didn't use "-d"; if that were important, it would be up to your script to notice and complain. However, "getopt" will notice if we try to use a flag that wasn't specified :

The getopt also identifies illegal options. If we run the above s9 command as :

./s9 –abc rao –f

We get an error "illegal option".

However, if you preface the option string with a colon:

args=`getopt **:**abc:d $*`

Example

The following is another shell program which can be used to parse the command line arguments using getopt.

```
aflag=no
bfalg=no
flist=""
set — `getopt abf: "$@"`
while [ $# -gt 0 ]
do
case "$1" in
-a)aflag=yes;;
-b)bflag=yes;;
-f)flist="$flist $2"; shift;;
—)shift;break;
*)echo "$0 error - unrecognised option $1" 1>&2; exit 1;;
esac
shift
done
```

13.6.10 Shell Functions

Shell functions are a way to group commands for later execution using a single name for the group. They are executed just like a "regular" command. When the name of a shell function is used as a simple command name, the list of commands associated with that function name is executed. Shell functions are executed in the current shell context; no new process is created to interpret them.

Functions are declared using this syntax :

[function] *name* () *compound-command* [*redirections*]

This defines a shell function named *name*. The reserved word function is optional. If the function reserved word is supplied, the parentheses are optional. The *body* of the function is the compound command *compound-command*. That command is usually a *list* enclosed between { and }, but may be any compound command listed above. *compound-command* is executed whenever *name* is specified as the name of a command. Any redirections associated with the shell function are performed when the function is executed.

A function definition may be deleted using the -f option to the unset built-in.

The exit status of a function definition is zero unless a syntax error occurs or a readonly function with the same name already exists. When executed, the exit status of a function is the exit status of the last command executed in the body.

Note that for historical reasons, in the most common usage the curly braces that surround the body of the function must be separated from the body by blanks or newlines. This is because the braces are reserved words and are only recognized as such when they are separated from the command list by whitespace or another shell metacharacter. Also, when using the braces, the *list* must be terminated by a semicolon, a '&', or a newline.

When a function is executed, the arguments to the function become the positional parameters during its execution. The special parameter '#' that expands to the number of positional parameters is updated to reflect the change. Special parameter 0 is unchanged. The first element of the FUNCNAME variable is set to the name of the function while the function is executing. All other aspects of the shell execution environment are identical between a function and its caller with the exception that the DEBUG and RETURN traps are not inherited unless the function has been given the trace attribute using the declare built-in or the -o functrace option has been enabled with the set built-in, (in which case all functions inherit the DEBUG and RETURN traps). If the built-in command return is executed in a function, the function completes and execution resumes with the next command after the function call. Any command associated with the RETURN trap is executed before execution resumes. When a function completes, the values of the positional parameters and the special parameter '#' are restored to the values they had prior to the function's execution. If a numeric argument is given to return, that is the function's return status; otherwise the function's return status is the exit status of the last command executed before the return.

Variables local to the function may be declared with the local built-in. These variables are visible only to the function and the commands it invokes.

Function names and definitions may be listed with the -f option to declare or typeset built-in commands. The -F option to declare or typeset will list the function names only (and optionally the source file and line number, if the extdebug shell option is enabled). Functions may be exported so that sub-shell's automatically have them defined with the -f option to the export built-in. Note that shell functions and variables with the same name may result in multiple identically-named entries in the environment passed to the shell's children. Care should be taken in cases where this may cause a problem.

Functions may be recursive. No limit is placed on the number of recursive calls.

The following session output shows the definition of functions dir, dir1, and dir2 along with their usage.

```
$ function dir ()
> {
> ls -l
> }
$ dir1()
> {
> ls
> }
$ function dir2()
> {
> ls -l $1
> }
$ dir
total 3
-rwxr—r— 1 Administrator None 9 May 13 07:44 a1
-rw-r—r— 1 Administrator None 12 May 13 07:44 a2
-rwxr—r— 1 Administrator None 119 May 13 07:44 aa
```

```
$ dir1
a1 a2 aa
$ dir2
total 3
-rwxr—r— 1 Administrator None 9 May 13 07:44 a1
-rw-r—r— 1 Administrator None 12 May 13 07:44 a2
-rwxr—r— 1 Administrator None 119 May 13 07:44 aa
```

In the following, we are defining some functions NU (to display number of users currently working on the machine), NF(to display number of files of the current directory), and NMU (to change all files names of current working directory such that they will have PID of the shell as their extension).

```
$ function NU()
> {
> who|wc -l
> }
$ NU
5
$ who
$ function NF() { ls -l|grep "^-"|wc -l; }
$ NF
3
$ function NMU()
> {
> for X in *
> do
> mv $X $X.$$
> done
> }
$ ls
a1 a2 aa
$ NMU
$ ls
a1.3256 a2.3256 aa.3256
```

13.7 CONCLUSIONS

In this chapter shell programming is explained in detail. It emphasizes the need for shell programming and its limitations. Shell constructs such as if, while, until and for loop etc., are explained. How arrays can be used in shell also dealt in a nutshell fashion. Also, user configuration is explained in detail.