Student Solutions for

UNIX and Shell Programming

A Textbook

Behrouz A. Forouzan and Richard F. Gilberg
Brooks/Cole Publishing
(ISBN 0 534-95159-7)

Contents

This document is provided to students using ***UNIX and Shell Programming A Textbook***. All reprints in any form must cite the Wadsworth Group copyright on the previous page.

Student solutions contain the answers to the odd numbered questions. Before checking answers, be sure to download the latest errata from the Brooks/Cole web site. It's address is http://www.brookscole.com/compsci_d/.

Behrouz Forouzan
Richard Gilberg

Chapter 1: Introduction

Review Questions

1. Portable refers to the ability of the operating system to be moved to new platforms without the need for a major rewrite. Portable operating systems are typically implemented in a high level language such as C, which is available on many platforms.
3. In a time-sharing environment many users are connected to one or more computers, and all users share output and storage devices. All computing and shared resources are controlled by a central computer. In a client-server environment computing functions are split between a central computer (server) and a users' computer (client).
5. Verb (command name), options, and arguments.
7. Bourne shell, Korn shell, and C shell.
9. Logon, interaction, and logout.
11. This chapter discussed one option for **date** — u. It is used do display the date in GMT time. Using the **man** command you may discover other options depending on the particular UNIX implementation.
13. **whoami** has no options. On some systems the **whoami** has been obsoleted by the **id** utility.
15. This chapter did not discuss any options for **passwd**. Using the **man** command you may discover other options depending on the particular UNIX implementation.
17. This chapter did not discuss any options for **man**. Using the **man** command itself, you may discover other options depending on the particular UNIX implementation.
19. **who**:
    a. user name
    b. terminal
    c. time of login
21. **date**:
    a. one
    b. 28
    c. no [@@ fix question]
    d. 10: day mmm dd
    e. 12: hh:mm:ss zzz

Exercises

23. **cal**:
    a. No error.
    b. Invalid option.

c. **cat** should be **cal**
d. Invalid syntax.

25. **echo**:
    a. No error.
    b. No error, unless "man echo" was intended.
    c. The option –u is not recognized as an option to **echo**.
    d. No error.

27. **man**:
    a. Should be **man** passwd.
    b. No error.
    c. No error — **man** accepts multiple keywords.
    d. Invalid option.

29. **cal** month:
    a. Prints calendar for August 2000.
    b. Prints calendar for August 2000.
    c. Error: bad month. Range 1..12.
    d. Prints calendar for year 0009, not September.

31. **date** format:
    a. Prints two digit month, space, two digit minute.
    b. Prints the date in hh:mm:ss [AM | PM] hh:mm.
    c. Prints the abbreviated weekday name, space, full weekday name.
    d. Prints the abbreviated mont name, space, full month name.

33. **man**:
    a. Shows **man** page for **cal** command.
    b. Shows **man** page for calendar, which is different than **cal**.
    c. Shows **man** page for **date**.
    d. Shows **man** page for **whoami**, which is different than **who**.

Chapter 2: Basic vi Editor

Review Questions

1. An editor is a UNIX utility that can be used to create and change text files. It differs from a word processor in that it does not perform typographical formatting such as bolding, centering, underlining, or varying fonts within a document.
3. **sed** and **ex**
5. It is a screen editor.
7. The current character is the character at the cursor.
9. Two **vi** modes were discussed: command mode and text mode.
11. **vi** starts in command mode.
13. When moving the cursor, you are in command mode.
15. This chapter gave six commands that transition from command mode to text mode (`a`, `A`, `i`, `I`, `o`, and `O`).
17. Frequently saving avoids loss of data in case of a system failure.
19. Yes. Start **vi** with the new filename as the argument.

Exercises

21. b. To delete the current character, use the `x` key.
23. d. All of the listed keys move from the command mode to the text mode.
25. d. All of the listed keys move from the command mode to the text mode.
27. To move from the eighth to the ninth character on a line, use the l (lower case L) key.
29. To move to the beginning of the current line, use the 0 (zero) key.
31. After all four keystrokes, the cursor would be on line 12 character 3.
33. After the two keystrokes, the cursor is on the fifth character of line 11.
35. The number of the last line on the screen is 25.
37. To discard all changes, use **c** (`:q!`).
39. To save and continue editing, use **a** (`:w`).

Chapter 3: File Systems

Review Questions

1. Some rules for naming files:
   a. Start your names with an alphabetic character.
   b. Use dividers to separate parts of the name. Good dividers are the underscore, period, and the hyphen.
   c. Use an extension at the end of the file name, even though UNIX doesn't recognize extensions. Some applications, such as compilers, require file extensions. In addition, file extensions help you classify files so that they can be easily identified.
   d. Never start a file name with a period. File names that start with a period are hidden files in UNIX. Generally, hidden files are created and used by the system. They will not be seen when you list your files.

3. The `*` matches zero or more characters, the `?` matches any single character, and `[...]` matches a single character in the given set.

5. The `[...]` matches a single character from the set of characters within brackets

7. The seven UNIX file types are: regular, directory, character special, block special, symbolic link, FIFO, and socket

9. A directory is a file that contains the names and locations of files stored in a file system. A user's home directory is one such example.

11. A block-special file represents a physical device, such as a disk, that reads or writes data a block at a time.

13. A FIFO, or first-in, first-out file—also known as a named pipe, is a file that is used for inter-process communication.

15. Regular files are divided into two categories — text files and binary files. UNIX views both formats as a collection of bytes and leaves the interpretation of the file format to the program that processes it.

17. A directory hierarchy is an organizational structure in which directories can contain files and other subdirectories.

19. The working or current directory is the directory that the user is in at any point in a session. It is denoted by a dot (.).

21. The parent directory is the directory immediately above the working directory. It is denoted by two dots (. .).

23. An absolute pathname starts at the root directory. A relative pathname is the path from the working directory to the file. An absolute pathname starts with a "/", such as `/etc/profile`.

25. A relative pathname starts from the working directory.

27. No. For example, if we are in the `profile` subdirectory of `/etc`, the relative pathname would be represented as `../etc/profile`. So in this case, the relative pathname is longer and the absolute pathname, `/etc/profile`.
29. A directory contains a mapping of names to inodes. The relationship of file names to inodes is a many to one relationship, that is a physical file has only one inode, but it can have many names. For example, two filenames can both be linked to the same file.
31. The file system areas are the boot block, the super block, the inode block, and the data block.
33. The inode fields are owner, group, type, permission, time, address.
35. A symbolic link is a logical file that defines the location of another file somewhere else in the system. Soft links can be used for both files and directories.
37. With a basic list command, five entries (3 subdirectories and 2 files) are displayed. With the all option, seven entries are displayed: the three subdirectories, the two files, the current directory (`.`) and the parent directory (`..`).
39. The four operations that can be used only with files are create (**vi** etc), edit (**vi**, etc), display (**more**), and print (**lpr**).
41. Copying a file duplicates the contents. Moving a file, changes a link to the contents.
43.
    a. **ls** lists the contents of a directory.
    b. **ln** links one file to another file.
    c. **cp** copies a file.
    d. **pwd** displays the absolute pathname of the current directory.
45.
    a. **rm** -r deletes a file or a directory.
    b. **cp** -r copies a directory.
    c. **cd** changes the current working directory.
    d. **mkdir** creates a directory.

Exercises

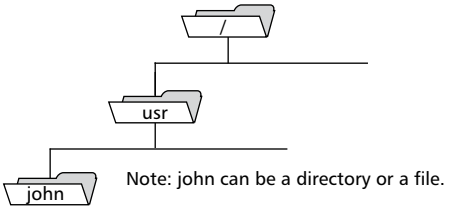
47. Only ‘a’ is a valid file name.
49.
    a. file[1-9]
    b. file[1-9]*
    c. file2[0-9]
    d. *.c
    e. f*
    f. [a-z]*
    g. [a-zA-Z]*
    h. [0-9]*

i. *[0-9][0-9]
j. ./*
k. ../*
l. [!0-9]*

51. The answers a, b, f, and g are absolute pathnames.

53. The relative pathname to a user's home directory is the tilde (~).

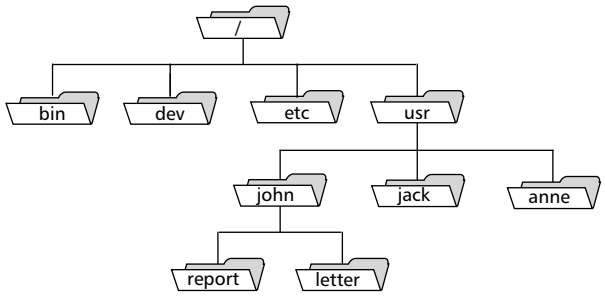
55. /usr/user/john as shown in the following figure.



57. The absolute pathnames are:

a. /
b. /bin
c. /usr/jack
d. /usr/jack/letter
e. /usr/jack/report

59. The relative pathnames from the john directory are:



a. ../..
b. ../../bin
c. .
d. ../jack/letter
e. ../jack/report

61. The absolute pathnames are:

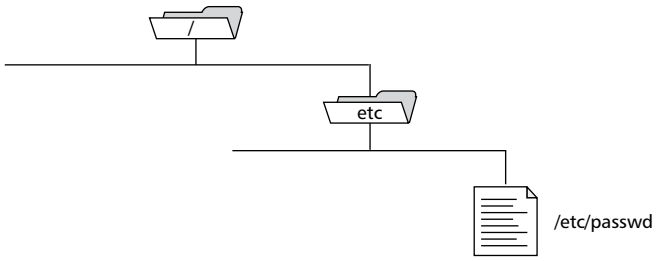
a. /
b. /bin
c. /usr/jack/report
d. /usr/john

63. The errors in the commands are:
    a. There is an extra argument (hello).
    b. There is a missing argument.
    c. There is an extra argument (file3).
    d. There is an illegal option (-s).
    e. The ordering is wrong; options should come first.
    f. There is a missing argument (pathname).
    g. There is no error.

Chapter 4: Security and File Permission

Review Questions

1. In UNIX, everyone who logs into the system is called a user.
3. Yes. For example a user can be in a group called "staff" and a group called "admin".
5. The home directory of a superuser user is `root`.
7. The three security levels are: system level, directory level, and file level.
9. At the system level, who accesses the system is controlled by log ins and passwords. At the directory and file levels, security is controlled by permission codes to determine who can access and manipulate a directory or file.
11. There are seven fields in `/etc/passwd`: login name, password, user-id, group-id, user info, home directory, login shell.
13. The field lengths in the `/etc/passwd` file cannot be predetermined since they contain variable length information.
15. The directory structure is shown in the following figure:



17. Each permission code contains three triplets.
19. The second permissions triplet controls the group read/write/execute permissions.
21. A directory has read (`r`), write (`w`), and execute (`x`) permissions. Although for a directory, execute really means permission to navigate to and through a directory.
23. Write permission for a directory allows entries to be added to and deleted from a directory.
25. A file has read (`r`), write (`w`), and execute (`x`) permissions.
27. Permission `w` for a file allows a file's contents to be changed.
29. The command to change a directory or file permission is **chmod**.
31. Yes, symbolic codes can be used to totally reset permissions.
33. Yes. When using symbolic codes, unreferenced permissions are unchanged.
35. Yes. For example

```
chmod 666 filename
```

will set the read and write permissions, and clear the execute permissions for user, group, and others.

37. No. The octal code always uses three octal digits when specifying file permissions, so all permissions are assigned by the three octal digits.
39. To copy the contents of a directory requires read and execute permissions (`o+rx`).
41. To execute a program, read permission (`o+r`) is required for the directory and execute permission (`o+x`) is required for the programs in the directory.
43. The system default permission for a new directory is 777.
45. If there is no user mask, the permissions for a newly created directory is 777. If there is a user mask, it must be *xor*'d by 777 to get the new user mask.
47. The command to set the user mask is **umask**.
49. The command to change the ownership of a file is **chown**.
51. There can be multiple permission sets for a file or a directory in the sense that there are separate read/write/execute permissions for user, group, and others.
53. To restrict file deletions to the owner requires u=rwx,g=rx,o=rx permissions.
55. It is possible to destroy a file even if it cannot be deleted. To delete a file, you must have write permission for the containing directory. But to "destroy" a file, you only need to be able to change the file's contents, which is determined by a separate write permission for the file.

Exercises

57. Symbolic format:
    a. `--x --x --x` is `a=x`
    b. `rwx --x --x` is `u=rwx,g=x,o=x`
    c. `--x rwx --x` is `u=x,g=rwx,o=x`
    d. `r-x r-x rwx` is `u=rx,g=rx,o=rwx`
59. The symbolic format is:
    a. `u=r,g-rwx,o-rwx`
    b. `u=rw,o-rwx`
    c. `u=r,g-rwx,o=x`
    d. `u=rwx,g=r,o=x`
61. The octal format is:
    a. 711
    b. 6?? (can't be done)
    c. 7?? (can't be done)
    d. 7?? (can't be done)
63. The symbolic code is:
    a. `ug=x,o-rwx`
    b. `a=rwx`
    c. `uo=rx,g-rwx`

d. `a=x`

65. For George to copy the file, the following permissions are required: (1) `/usr/john/report` should have read permission for others; (2) the root directory should have read and execute permission for others; and `/usr/john` should have read and execute permission for others.

67. The permissions for George to copy the file `/usr/john/report` to his own directory are:
    a. `report:` others: `o+r`
    b. `root:` others: `o+rx`
    c. `/usr:` others: `o+rx`
    d. `/usr/john:` others: `o+rx`
    e. `~george:` others: `u+wx`

69. Permissions for Anne to copy the file `/usr/anne/sales to /usr/john` are:
    a. `sales`: users: `u+r`
    b. `/`: others: `o+rx`
    c. `/usr`: others: `o+rx`
    d. `/usr/john`: others: `o+wx`
    e. `/usr/anne`: users: `u+rx`

71. The file has the following permissions: user—execute, group—execute, others:—execute.

73. The file has the following permissions: `rw- rwx r--`

75. The default permissions after umask 022 are:
    a. `rw- r-- r--`
    b. `rwx r-x r-x`

Chapter 5: Introduction to Shells

Review Questions

1. The first is the interpreter. The interpreter reads user commands and works with the kernel to execute them. The second part of the shell is programming capability that allows you to write a shell (command) script.
3. The login shell can be verified by entering `echo $SHELL` at the command line.
5. No, the login shell is not always the same as the current shell. The user can change shells.
7. Yes. Shells can be nested.
9. The file descriptor for standard input is `0`, standard output is `1`, and standard error is `2`.
11. The standard output stream is normally associated with the monitor
13. Redirection is the process by which a user specifies that a file is to be used in place of one of the standard streams.
15. To redirect standard output use `1> filename`.
17. Pipe is an operator that temporarily saves the output of one command in a buffer that is being used at the same time as the input of the next command. The pipe token is the vertical bar '|'.
19. A sequence of commands can be entered on one line separated by semicolons (;). Commands can be grouped together using parentheses. Commands can be chained together using pipes.
21. Command line editing is a shell facility that permits the user to edit commands using one of the screen editors (**vi** or **emacs**).
23. The command line editors are normally **vi** or **emacs**.
25. The backslash metacharacter (\) changes the interpretation of the character that follows. Double quotes remove the special interpretation of most metacharacters. Single quotes operate like double quotes, but their effect is stronger. They preserve only the meaning of single quotes.
27. A job is a command or a set of command entered on one line. A foreground job is any job running under the supervision of the user. A background job is any job running on the background without the direct supervision of the user.
29. The six different job states are: (1) Running in foreground. (2) Stopped (suspended). (3) Running in background. (4) Done (completed successfully). (5) Exit (completed unsuccessfully). (6) Terminated (killed).
31. An alias provides a means of creating customized commands by assigning a name to a command. Aliases are available in the Bash, Korn, and C Shells. To use an alias to rename files in Bash and Korn, use `alias rename=mv`. In the C Shell's, the syntax is slightly different, `alias rename mv`, or to use the C Shell positional argument substitution, the user could write `alias rename 'mv \!:1 \!:2'`.

33. A user can store a value in a user variable, for example `msg="Hello world"` (or for C Shell: `set msg = "Hello world"`), and recall the variable's value, as in `echo $msg`. A predefined variable is similar in concept, but is already set up for the user. For example the primary prompt (**PS1** in Korn and Bash Shells, or **prompt** for C Shell) is a predefined variable.
35. Options are on/off switches that are used to control the shell. Three such options are **noclobber**—do not allow redirection to clobber existing files, **ignoreeof**—do not allow `ctrl+d` to exit the shell, and `verbose`—print commands before executing them.
37. The startup files for the Korn shell are the system wide `/etc/profile` file maintained by the system administrator, and the local .profile file, an optional file maintain by the user. In addition, the Korn shell allows for an environment file to run whenever a new shell is started. The file name is determined by the exported shell variable **ENV**. The Bash Shell is similar, except that the user `.profile` file is searched for under several names in the following order: `~/bash_profile`, `~/bash_login`, and finally `~/.profile`. The C shell is similar, but the file names are different. The system wide login file is `/etc/csh.login`, and the system wide environment file (for all new subshells) is `/etc/csh.cshrc`. The personal login file is `~/.login`, and the personal environment file is `~/.chsrc`.

Exercises

39. b: `echo $SHELL`
41. b: bash
43. c: csh
45. b: 1
47. None of these commands read from standard input.
49. Yes. The **lpr** command accepts standard input.
51. **lpr** input redirection:
    a. **lpr** prints from standard input (keyboard).
    b. **lpr** prints file1.
    c. **lpr** prints file1.
    d. **lpr** prints file1.
53. input redirection errors:
    a. **cp** requires filename arguments.
    b. **ls** should be followed by zero or more directory or file names.
    c. **mkdir** should be followed by a new directory name.
    d. **rm** should be followed by a file name.
55. Commands that don't take input redirection:
    a. **lpr** can be used with redirection (see `man lpr`), but it is generally used with file-name arguments.
    b. **ln** is used to link files. Using it with redirection makes no sense.

c. cd is used to change directories. It does not operate on standard input.

d. **more** browses through a text file. Since it uses terminal input to control paging/navigation, it cannot also take input from the terminal. However, it can still use input redirection from a file, as in `more < someFile`.

57. Commands that can be used with output redirection:

a. a) **cal** can be used with output redirection.

b. b) **date** can be used with output redirection.

c. c) **echo** can be used with output redirection.

d. d) **man** can be used with output redirection.

e. e) **passwd** cannot be used with output redirection.

f. **who** can be used with output redirection.

g. **whoami** can be used with output redirection.

59. **cal** command usage:

a. **cal** results (calendar of current month) are sent to standard output.

b. Sends **cal** results (calendar of current month) to Outfile

c. Sends `cal 1` results (calendar of year 1) to Outfile. Because there is a space between the '1' and the '>' this is not the same as redirecting standard output.

d. Sends `cal 2` results (calendar of year 2) to Outfile. Because there is a space between the '2' and the '>' this is not the same as redirecting standard error.

61. **date** command usage:

a. Sends **date** output to file1 (unless noclobber option is enabled).

b. Sends **date** output to file1, overriding noclobber setting.

c. Appends **date** output to file1.

d. syntax error.

63. Command errors:

a. No error.

b. **more** and **date** are swapped.

c. **more** cannot take input from the terminal.

d. **man** requires an argument and it does not make sense to **pipe** to **man**.

65. Chained commands:

a. Command1 writes to standard output.

b. Command2 reads from standard input.

67. The command `date | more` works. The command `more | date` does not work because **more** needs to have input and **date** does not read from standard input.

69. In the following command: `ls -l | tee`:

a. Input comes from the system.

b. No output file is created.

71. Yes, the **tee** command can be used to create a file. The statement `tee file1` sends keyboard input to a file1.

73. In the command `ls -l | tee > file1`:
    a. The output will go to both standard output (monitor) and to the file file1.
    b. One file is created.
75. C shell errors:
    a. C requires the set command: `set a = 24`.
    b. OK.
    c. OK.
    d. C shell does not allow the '$' in target variable of **set** command.
77. **echo** command output: The only differences are in the quoting of UNIX.
    a. Hello to the user of UNIX operating system
    b. Hello to the user of "UNIX" operating system
    c. Hello to the user of 'UNIX' operating system
    d. Hello to the user of "UNIX" operating system
79. Command errors:
    a. There should not be a '$' prefix on `$a` and **cp** does not generate output.
    b. **lpr** does not write to standard out.
    c. No '$' prefix on `$a`.
    d. The **cal** command's formatting gets removed. To preserve the formatting, use echo `"`cal`"`.
81. Command errors:
    a. No '$' prefix for `$a`. The output is sent to a file, so it is not available for the variable assignment.
    b. There is nothing to **echo** because the **cal** output is sent to **lpr**.
    c. No '$' prefix for `$a` and file1 is not a command.
    d. The **cal** command's formatting gets removed. Use quotes to preserve it.

Chapter 6: Filters

Review Questions

1. In UNIX, a filter is any command that gets its input from the standard input stream, manipulates the input, and then sends the result to the standard output stream.
3. Filters discussed in this chapter: **cat**, **cmp**, **comm**, **cut**, **diff**, **head**, **paste**, **sort**, tail, **tr**, **uniq**, and **wc**. Other filters mentioned in this chapter: **more**, **grep**, **sed**, and **awk**.
5. `head [-options] [file-list]`—The **head** command outputs lines at the beginning of a file. Input can come from the keyboard, a file, or several files. Output can go to the monitor or be redirected to a file. The **head** command does not output to multiple files.
7. `cut [-options] [file-list]`—The **cut** command select portions of each line of a file. Input can come from the keyboard, a file, or several files. Output can go to the monitor or be redirected to a file. The **cut** command does not output to multiple files.
9. `sort [-options] [field-specifiers] [file-list]`—The **sort** command arranges data in ascending or descending order. Input can come from the keyboard, a file, or several files. Output can go to the monitor or be redirected to a file, or a specified output file that can even be the same as one of the input files. The **sort** command does not output to multiple files.
11. `uniq [-options] [file-list]`—The **uniq** command displays unique lines in a file (adjacent repeated lines are treated as one line). Input can come from the keyboard, a file, or input redirection. Output can go to the monitor or be redirected to a file, or can go to an output file specified in a command line argument. The **uniq** command does not output to multiple files.
13. The **paste** and **cat** commands both combine files. The **cat** command concatenates files vertically, the **paste** command concatenates files horizontally.
15. The **cmp** command compares two files for equality and if they do not have identical contents, reports where the two files start to differ. The **comm** command compares two files and optionally shows the lines that are only in the first, lines that are only in the second, and lines that are common to both.
17. The **comm** command can be used to show lines that are common to two files, whereas the **diff** command can be used to show the differences between two files.

Exercises

19. a & c display the contents of a file; b & d create files.
21. The input sources are:
    a. file1
    b. keyboard
    c. keyboard
    d. file1

23. Equivalent commands are:
    a. equivalent
    b. not equivalent. The first outputs to file1, the second outputs from file1.
    c. If **noclobber** is not set, then the two are equivalent. Otherwise, the first command will fail but the second overrides **noclobber** and will succeed.
    d. Not equivalent. The first reads from the keyboard, the second reads from file1.
25. Command errors:
    a. **date** does not read from standard input.
    b. OK
    c. Can't **pipe** to a file.
    d. Can't **pipe** to a file.
27. The output of **echo** is piped to `cat file1 > file2`. Since **cat** is reading from file1, it ignores the output from **echo**, and thus file1 is copied to file2.
29. In the first command, "Header" is written to standard output, then the contents of file1 are written to file2. In the second command, "Header" followed by file1 are written to file2. In the third command the contents of file1 is written to file2, and nothing goes to standard output.
31. d copies file 1 to file 2.
33. `cat <file1 >file2`
35. Not possible.
    ```
    tail -r file1 | tail -50r > file2
    ```
37. Use tail to copy last 50 lines: `tail +50 file1 > file2`
39. Copy line one: `head -1 file1 > file2`.
41. Copy and append lines from two files:
    ```
    head -50 file1 | tail -21 > file3; head -60 file2 | tail -41 >> file3
    ```
43. Differences in commands:
    a. The first command counts characters, words, and lines from file1 and annotates the output with the filename "file1." The second counts characters, words, and lines from standard input, which was redirected from file1, so it provides the same count results without the filename annotation.
    b. The first counts characters, words, and lines from file1 then from file2. The second processes the files in the reverse order.
    c. The first counts characters, words, and lines from file2. The second processes the files file1 and file2.
45. Count characters in first line of a file: `head -1 file | wc -c`.
47. Find errors:
    a. Illegal option: (`-l`).
    b. OK.
    c. The `c` option supersedes the `u` option. In some implementations, this is an illegal usage.
    d. Missing argument to `-s` option.

49. Change lowercase to uppercase: `tr "[a-z]" "[A-Z]" < filename`.
51. The command works. The **tr** command will map each input character found in the first string "AB" to each corresponding character in the second string "BA".
53. Yes, we can change uppercase and lowercase in one command. The command is:
    ```
    tr "[a-z][A-Z]" "[A-Z][a-z]"
    ```
55. The command will squeeze multiple sequential `A` and `B` characters into the character `B`. It will also map original `X` characters to `B`. For example if the input were `ABAABB-B-A-X`, the output would be `B-B-B-B`.
57. To create a file of 10 lines, enter the command `tail +1 > filename` and enter at least 10 lines from standard input.
59. Enter the command `paste - > filename` and enter as many lines as you wish from standard input. The option `-` means read from standard input instead of a file.
61. Yes, the sort can be used to copy a file without actually sorting by using the `-m` sort option: `sort -m file1 > file2`. The `-m` option tells sort that the input is assumed to be sorted (whether or not it really is), so it can be used in a merge only mode to prevent sorting.
63. The command `head -5 file1 | head -3 | head -2` demonstrates that **head** is a filter because it is used on the left of a pipe, between two pipes, and on the right of a pipe.
65. The command `cut -f1,3 file1 | cut -f1,2 | cut -f1` demonstrates that **cut** is a filter because it is used on the left of a pipe, between two pipes, and on the right of a pipe.
67. The command `tr "l" "L" file1 | tr "i" "I" | tr "n" "N"` demonstrates that **tr** is a filter because it is used on the left of a pipe, between two pipes, and on the right of a pipe.
69. The command `wc -l file1 | wc -l | wc -l` demonstrates that **wc** is a filter because it is used on the left of a pipe, between two pipes, and on the right of a pipe. The result is always one since the output of the `wc -l` is a single line, and the final `wc -l` provides a count of the one line, which is 1.
71. None of the commands are correct.
73. None of the commands are correct.
75. None of the commands are correct.

Chapter 7: Communications

Review Questions

1. The five communications utilities are: **talk**, **write**, **mail**, **telnet**, and **ftp**.
3. Yes, **talk** can be used by users on different systems.
5. To prevent **talk** sessions, enter the command `mesg n`.
7. The **talk** command is a two-way communication utility.
9. No. The **write** command can only be used to communicate with users on the same system.
11. The **write** command is a one-way communication utility.
13. The **mail** utility is used to send and receive electronic mail. It is a store and forward based system, which means that the receiver does not need to be logged in at the time the message arrives.
15. The body contains the text of the message. The header consists of the subject, the addressees (To:), sender (From:), a list of open copy recipients (Cc), and a list of blind copy recipients (Bcc).
17. A **mail** address is made up of two parts: local address and domain name separated by a n at-sign (`@`) character.
19. The domain address uniquely defines the destination mail server anywhere in the world.
21. The original seven generic domain labels are: `com`, `edu`, `gov`, `int`, `mil`, `net`, and `org`. The labels approved in November 2000 are: `aero`, `biz`, `coop`, `info`, `museum`, `name`, and `pro`.
23. Yes. To send mail, the **mail** utility needs the username(s) of the message recipient(s).
25. In **mail**, when we are in the read mode, we can type a command without using a tilde (~) character.
27. When composing a document in **vi** text mode, the escape character is used to toggle to command mode. When composing a message **mail**, a line starting with tilde (~) is used to indicate a command. In **vi**, there are a number of commands (such as insert and append) that transition from command to text mode. In **mail**, a reply (`r` or `R`) command, or a new message command (`m`) can be used to transition from read mode to send mode. The send mode in **mail** is a very primitive editor; it is not a general purpose text editor.
29. In a distribution list, an alias is formatted as: `alias <alias-name> <alias-list>`.
31. To forward mail, we need to use the `m` command to move from the *read mode* to the *send mode*. While in the send mode, we can use `~m` to retrieve received mail and send it. Of course, when we send it we can add comments.
33. **telnet** is a general-purpose client/server utility because it lets a user access any application program on a remote computer; in other words, it allows the user to log into

a remote computer. After logging on, a user can use the services available on the remote computer and transfer the results back to the local computer.

35. A terminal driver is the interface between a terminal and the operating system. In the case of a remote terminal (e.g. **telnet** client), no physical terminal (or terminal driver) is used. Since there is no terminal driver, a faux terminal is needed for applications to communicate with. This is done using a pseudoterminal driver.
37. **Synopsis: ftp [-option] [host]—ftp** (File Transfer Protocol) allows the user to transfer files between the local and remote computers.
39. The **ftp** utility uses two connections: control connection and data connection

Exercises

41. The **mail** utility does not use a prompt while composing a mail message, but you may be prompted for header items such as subject.
43. To read message 7, key either `7` or `t 7`.
45. To list the headers for all messages in the read mode, key `h`.
47. To reply to only the sender, use `R`.
49. To forward a message, use `m` to send a new message or `~m` to read in a message to forward.
51. To include a file in a message key `~r <filename>`.
53. To save a file before it is sent, key `~w <filename>`.
55. No, you can use **vi** as your e-mail editor when connected remotely using **ftp**.

Chapter 8: vi and ex

Review Questions

1. There are three operating modes in **vi**: command, text, and **ex**.
3. Yes. Enter **ex** from the command line.
5. Yes. Enter the command **vi** from the colon (`:`) prompt.
7. To directly exit from **ex**, key `wq`, `q`, or `q!`.
9. In command mode, a powerful set of **vi** commands are available to move and make changes.
11. A number of **vi** commands—such as insert (`i` and `I`), append (`a` and `A`), and new line (`o` and `O`)—enter text mode.
13. Yes. The colon (`:`) key moves from the command mode to the **ex** mode..
15. No. To move from the **vi** mode to the **ex** mode, you must pass through the **vi** command mode.
17. There is no text mode prompt.
19. Yes. The **ex** mode uses a colon (`:`) prompt.
21. Local **vi** commands are applied to the text at the current cursor location.
23. Global commands are applied to all of the text in the whole buffer, as contrasted to the current window.
25. A range command is applied on a text object.
27. There are 17 local **vi** commands described: `i`, `I`, `a`, `A`, `o`, `O`, `r`, `R`, `s`, `S`, `x`, `X`, `p`, `P`, `m`, `~`, and `J`.
29. Four range commands were described: "nothing" (moves the cursor), `d` (delete), `c` (change), and `y` (yank).
31. Fifteen global commands were described: `ctrl+y`, `ctrl+e`, `ctrl+u`, `ctrl+d`, `ctrl+b`, `ctrl+f`, `u`, `U`, `z<return>`, `z.`, `z-`, `ctrl+L`, `.` (dot), `ctrl+G`, and `ZZ`.
33. A single-line address in **ex** defines one line for processing.
35. A set-of-line address in **ex** defines one or more lines that are not necessarily consecutive. The lines are chosen, not based on their relative positions, but based on their contents.
37. A range address in **ex** defines one or more consecutive lines. A special range address is the percent sign (`%`), which defines the entire file, or a range address consisting of two addresses separated by a comma or a semicolon.
39. A nested address defines a set-of-line address inside a range.
41. There are 26 alphabetic named buffers that are controlled by the user. They are identified by the lowercase letters a through z.

Exercises

43. The number of lines deleted are:
    a. dd: 1
    b. 4dd: 4
    c. 4dd2: 4
    d. dd5: 1
45. Only (b), the `r` command, needs a text object.
47. The **ex** address types are:
    a. `5`: single-line address
    b. `20,30`: range address
    c. `$`: single-line address
    d. `/pattern/`: set-of-lines address
49. The **ex** address types are:
    a. `20,60g/pattern/`: nested address
    b. `?pattern?;?pattern?`: range address
    c. `/pattern/,?pattern?`: range address
    d. `45,80v?pattern?`: nested address
51. The **ex** that defines a block containing lines 30 to 50: `30,50`
53. The difference between the two **ex** commands is:
    a. deletes all lines with UNIX
    b. deletes the next line with UNIX
55. The difference between the two **ex** commands is:
    a. copies lines 20-50 after line 1
    b. copies lines 20-50 before line 1
57. The lines deleted are:
    a. 1—line 20
    b. 11—lines 20 through 30
    c. 11—lines 20 through 30
    d. 31—lines 20 through 51

Chapter 9: Regular Expressions

Review Questions

1. A regular expression is a character pattern that is matched against text. A regular expression is like a mathematical expression. A mathematical expression is made of operands (data) and operators. Similarly, a regular expression is made of atoms and operators.

3. An atom specifies what we are looking for and where in the text the match is to be made. Atoms corresponds to operands in a mathematical expression.

5. A single character atom matches itself. Its length is one.

7. A class atom defines a set of characters, any one of which may match the corresponding character in text. Its length is one.

9. There are four types of anchors: `^`, `$`, `\<`, and `\>`.

11. Operators are used to combine atoms into more complex regular expressions. The regular expression operators play the same role as mathematical operators. Mathematical expression operators combine mathematical operands; regular expression operators combine regular expression atoms.

13. The sequence operator concatenates a series of atoms. It is conceptual. It has no symbolic representation.

15. The repetition operator specifies that the atoms or expressions may be repeated. Its symbolic formats are as follows: `\{m,n\}`—match previous atom *m* to *n* times; `\{m\}`—match previous atom exactly *m* times; `\{m,\}`—match previous atom at least *m* times; `\{,n\}`—matches previous atom at most *n* times. There are also several short forms for the repetition operator.

17. The save operator saves one or more characters in a buffer to be matched later with its corresponding back-reference atom. Its symbolic format is `\(…\)`.

19. The `?` operator's format is: \{0,1\}

21. The `+` operator's format is: \{1,\}

Exercises

23. The number of characters matched is:
    a. 1
    b. 1
    c. zero or more
    d. 3

25. The minimum length to be matched is:

a. 3
b. 0
c. 3
d. 0

27. The number of characters matched is:
    a. 4
    b. 4-7
    c. 0-7
    d. 8 or more
29. The expressions can be rewritten as:
    a. A\{5\}
    b. A\{2,4\}
    c. A\{2\}| A\{5\}| A\{8\}
    d. A
31. The expressions can be rewritten as:
    a. Can't rewrite using +
    b. A+
    c. Can't rewrite using +
    d. Can't rewrite using +
33. `[ABC]` matches all but the last one. The matched characters are underlined.
    A
    AB
    ADBC
35. `[A-G]` matches all lines. The matched characters are underlined.
    A
    AB
    MBC
    D
37. `[ABC][AB]` matches only one line. The matched characters are underlined.
    AB
39. `^[ABC][AB]$` matches only one line. The matched characters are underlined.
    AB
41. `bc.*` matches three lines. The matched characters are underlined.
    aaabbbccdddd
    aaaabcsssss
    aaaaabc
43. `^[a-z]...` matches two lines. The matched characters are underlined.
    abcdefg
    a:237efg
45. `s:?s*` matches three lines. The matched characters are underlined.
    efgs:sgfe
    sssssssss

rsts

47. `[^\$]$` matches two lines. The matched characters are underlined.

    $$$$$$aaaaaa

    abc

49. `^[^\$]$` does not match any lines.

51. `^\$[0-9][0-9]$` matches one line. The matched characters are underlined.

    $10

53. Match blank line: `^$`

55. Matches every line: `.*`

57. Matches a line with at least three characters `.\{3,\}`

59. Matches date (Month dd, yy): `[A-Z][a-z]+ [0-9][0-9], [0-9][0-9]`

61. Matches Social Security Number: `[0-9]\{3\}-[0-9]\{2\}-[0-9]\{4\}`

63. Matches any digit: `[0-9]`

65. Matches Zip Code (5 digits): `[0-9]\{5\}`

67. Matches HTML tags: `<.+>`

Chapter 10: grep

Questions

1. The term **grep** is an acronym for Global Regular Expression Print; **egrep** is Extended grep; and, **fgrep** is Fast grep.
3. There are three members of the **grep** family: **grep**, **egrep**, and **fgrep**.
5. **fgrep** cannot accept regular expressions.
7. The atoms supported by **grep** varies depending on the **grep** implementation you are using. Some older implementations may not support the begin/end word anchors `\<` and `\>`.
9. No. Although the degree of operator support could vary based on the particular implementation of **grep/egrep**.
11. Not all **egrep** implementations support the save operator. Some implementations support save by overloading the group syntax, so (…) provides the functionality of `\(…\)`.

Exercises

13. Select all lines with exactly three characters: `grep "^...$" file1`
15. Select lines that have three or less characters: `grep "^.\{0,3\}$" file1`
17. Count blank lines: `grep -c "^$" file1`
19. Select lines starting with any uppercase character: `grep "^[A-Z]" file1`
21. Select lines containing UNIX or variation: `egrep "UNIX|Unix|unix" file1`
23. Select lines ending with UNIX: `grep "UNIX$" file1`
25. Select lines with UNIX twice: `grep "UNIX.*UNIX" file1`
27. Write command copy a file: `grep ".*" file1 > file2`
29. Select lines that start with space: `egrep "^ +" file1`
31. Select lines that start and end with space: `egrep "^ .* $" file1`
33. Select lines that end with an uppercase character followed by zero or more spaces:
    ```
    grep "[A-Z] *$" file1
    ```
35. Select lines with two consecutive digits: `grep "[0-9]\{2,\}" file1`
37. Select lines in which first nonblank character is uppercase:
    ```
    grep "^[ ]*A" file1
    ```
39. Select lines that do not start with A: `grep "^[^A]" file1`
41. Select lines that start with A, B, or C: `grep "^[ABC]" file1`
43. Select lines that start with a character: `grep "^[a-zA-Z]" file1`
45. Select lines that contain only one floating point number:
    ```
    egrep "^[0-9]+[.][0-9]*$" file1
    ```

47. Select lines that contain only one hex number:
    ```
    egrep "^0[xX][0-9A-Fa-f]+$" file1
    ```
49. Copy only lines with at least four digits:
    ```
    grep "[0-9].*[0-9].*[0-9].*[0-9]" file1 > file2
    ```
51. Copy palindromes: `grep "^\(.\)\(.\).\2\1" file1 > file2`
53. Simulate commands:
    a. **cat**: `grep ".*" > file1`
    b. **cut**: Not possible.
    c. **paste**: Not possible.
    d. It is not possible because if we use `grep ".*" file1 file2`, each output line will be preceded by the file name.

Chapter 11: sed

Questions

1. **sed** [options] script [file-list]—**sed** edits specified lines in the input files and process them.
3. The **sed** command allows for an "in-line script," in which the script is specified on the command line and also a file script, in which the script is stored in a file.
5. A script file is a text file that contains a **sed** script.
7. A script file can be created with a standard editor such as **vi**.
9. The hold space is a buffer used to temporarily store and retrieve lines as directed by **sed** instructions.
11. The address selects the line(s) to be processed (or not processed) by the command.
13. A single-line address specifies one and only one line in the input file. It is either a line number or the $ (last line) symbol.
15. The single-line addresses are: a & d.
17. The range addresses are: b & d.
19. The two scripts are equivalent.
21. The addresses and commands are:
    a. address: `25`; command: `d`
    b. no address; command: `d`
    c. address: `20,50`; command: s/A/B/g
    d. address: `25,/^A.*A$/`; command: `s/A//g`
23. If no address is mentioned, the command applies to every line (a).
25. No, the **sed** command does not change the file.
27. The `N` command appends the next line to the pattern space while the `D` command deletes the first line of the pattern space.
29. The `p` command prints the entire pattern space; The `P` command prints only the first line of the pattern space.
31. The `h` command copies the pattern space to the hold space whereas the `H` command appends the pattern space to the hold space.
33. The command comparisons are shown in the following table.

Command	ex	sed
Line number	#	=
Insert	i	i
Append	a	a
Change	c	c
Delete	d	d
Substitute	s	s

Command	ex	sed
Translate	n/a	y
Input next line	n/a (no pattern space)	n, N
Print	p	p, P
List	l	l
Read	r	r
Write	w	w
Label	n/a	:label
Branch	n/a	b, T

Exercises

35. The result of the command: `sed "s/bc*./Z/"`
    ```
    aaaZbcccddd
    aaaaZssss
    aaaaaZ
    aaZss
    ```
37. The result of the command: `sed "s/\..*\$/Z/"`
    ```
    rsZ
    abcZ
    abcZ
    abc
    ```
39. The result of the command: `sed "s/:?.?/Z/"`
    ```
    :a??????????
    eeeeeeffffff?hhhh
    aaa:Z???
    :?.\?
    ```
41. The result of the command: `sed "s/^[^$]*$/Z/"`
    ```
    $$$$$$aaaaaa
    ```
    bcdef$

    $

    Z
43. The result of the command: `sed "s/^\$\d\d$/Z/"`
    ```
    $$$$$$$
    $10
    abc$
    $
    ```
45. This command is effectively a no-op.
47. Deletes all blank lines.
49. Prints the first 80 lines.
51. Each line is repeated three times.
53. Removes every third line (printing lines 1, 2, 4, 5, 7, 8, 10, …).
55. Moves lines 10-15 after line 20 while removing lines 11-15.
57. Delete second character: `sed "s/^\(.\)./\1/" file1`

59. Delete character before last character: `sed "s/.\(.\)$/\1/" file1`

61. Delete second word: `sed "s/^\([^ ]* *\)[^ ]* */\1/" file1`

63. Delete word before last word: `sed "s/ *[^ ]*\( *[^ ]*\)$/\1/" file1`

65. Swap first and last characters:

```
sed "s/^\(.\)\(.*\)\(.\)$/\3\2\1/" file1
```

67. Delete all digits: `sed "s/[0-9]//g" file1`

69. Replace single spaces at beginning of line with tabs:

```
sed "s/^ \([^ ]*\)/      \1/" file1
```

To only replace a lines starting with single space we find the start of line followed by one space and zero or more non-space characters. Just finding start of line followed by space could match lines that start with two or more spaces.)

71. Print every other line: `p;p;p;d`

This command works whether or not the `-n` option is used.

73. Copy two lines and delete the third: `n;n;d`

75. Move lines 22-33 after line 56: `22h;23,33H;22,33d;56G`

77. Delete all trailing spaces: `sed "s/ *$//" file1`

79. Change date mm/dd/yy to yy/mm/dd:

```
sed       "s/\([0-9][0-9]\)\/\([0-9][0-9]\)\/\([0-9][0-9]\)/\3\/\1\/\2/g" file1
```

81. Extract month from mm/dd/yy:

```
sed -n "s/[^0-9]*\([0-9][0-9]\)\/\([0-9][0-9]\)\/\([0-9][0-9]\).*/\1/p" file1
```

83. Extract year from mm/dd/yy:

```
sed -n "s/[^0-9]*\([0-9][0-9]\)\/\([0-9][0-9]\)\/\([0-9][0-9]\).*/\3/p" file1
```

85. Copy lines containing at least five digits:

```
sed -n "/\([0-9].*\)\{5\}/p" file1 > file2
```

87. Copy file deleting second word:

```
sed "s/\([^ ]\{1,\} *\)\([^ ]\{1,\} *\)/\1/" file1 > file2
```

89. Simulate copy command: `sed -n "p" file1 > file2`

91. Simulate **head** command: `sed "20q" file1`

93. Simulate **tail** command:

```
sed -n '1!G;h;$p' file1 | sed -n "1,40p" | sed -n '1!G;h;$p'
```

This is equivalent to `tail -40 file1`! Analysis: Since we don't know how many lines are in the file, we also don't know the line number that is 40 before the end. Therefore **sed** is not suited to this task. But by reversing the file using sed's hold buffer commands, we can then extract the first 40 lines (which were the last 40 lines) and then reverse the lines again. So there is a tricky solution to this problem, but it is not very practical.

Chapter 12: awk

Questions

1. The **awk** utility, takes its name from the initials of its authors (Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan).
3. The **awk** command supports the regular expressions supported by **egrep**. It does not support word anchors or back reference atoms.
5. The **awk** command can use an inline script specified on the command line or a file based script specified by the `-f` option.
7. A script file is a file that contains a script. This is useful for longer scripts that are too long to reasonably put on a command line.
9. Script files can be created using a standard text editor such as **vi**.
11. Patterns identifies which records in the file are to receive an action.
13. A simple pattern matches one record.
15. There are four types of simple patterns: `BEGIN`, `END`, expression, and nothing (no expression).
17. `END` is true after the last line is read and false everywhere else.
19. An empty expression (no expression) applies to every line.
21. Patterns a, b, and d are valid.
23. Answers b and c are true; answers a and d are false.
25. The pattern `NR == 8 || NR==9` selects lines 8 and 9.
27. The pattern `$0 ~/UNIX/ || $0 ~/DOS/` selects lines containing UNIX or DOS.
29. No, **awk** does not modify the input file..
31. `next` and `getline` both read the next input line, but `getline` continues execution, whereas `next` terminates the processing for the current record and begins processing of the next one.
33. Actions in **awk** are instructions or statements that act when the pattern is true.
35. `print` provides simple unformatted output. `printf` is a formatted print, and `sprintf` is a formatted print that will print to a string instead of standard output.
37. Yes, the *for* loop is a pre-test loop.
39. No, the *do…while* loop is not a pre-test loop.
41. The built-in string functions are: length, index, substring, split, substitute, global substitute, toupper, and tolower.

Exercises

43. Simulate **sed** addresses:
    a. `5: NR==5`
    b. n/a

c. `5!: !(NR==5)`

d. n/a

45. Simulate **sed** addresses:

a. `5,9: NR==5,NR==9`

b. `5,9!: NR<5 || NR>9`

c. `/UNIX/,/DOS/: /UNIX/,/DOS/`

d. `/UNIX/,10: /UNIX/,NR==10`

47. Simulate **sed** script:

```
!/^A/;/^A/ { print $0; print " This is the test to"; print "be added" }
```

49. Simulate **sed** script:

```
!/^A/
```

51. Simulate **sed** script:

```
# simulate sed script /^A/,/B$/y/aeio/AEIO/
/^A/,/B$/ {
    gsub(/a/,"A");
    gsub(/e/,"E");
    gsub(/i/,"I");
    gsub(/o/,"O");
    }
{ print $0 }
```

53. Simulate **sed** script:

```
{print}; NR==20,NR==60 {print $0 > "file2"}
```

55. Simulate **sed** script:

```
{print}; /UNIX/ {exit}
```

57. Simulate **cp** command: `awk '{print > "file2"}' file1`

59. Simulate **head** command: `awk '{ print }; NR==20 { exit }' file1`

61. Simulate **tail** command:

```
awk '{ lines[NR]=$0 }; END { for (i=40; i>0; i--) print lines[NR-i+1] }' file1
```

63. Print eighth line: `NR==8 {print; exit}`

65. Print total input lines: `END {print NR}`

67. Print last field in each line: `{print $NF}`

69. Print lines with more that four fields: `NF>4 {print $0}`

71. Print total number of fields in file:

```
{total_fields += NF}; END {print total_fields}
```

73. Print lines that contain UNIX: `/UNIX/ {print $0}`

75. Print all the lines with field order field 4, field 3, field 2, and field 1.

```
{print $4, $3, $2, $1}
```

77. Print sum of all values in the third field: `{sum3 += $3}; END {print sum3}`

79. Print lines in which third field value is greater than 5.00 and fourth field value is not 0:

```
$3>5.00 && $4!=0 {print $0}
```

81. Add one blank line after each line: `{print $0, "\n"}`

83. Print any line that follows a line whose first field is greater than 9:
```
{ if (printNext) print $0; printNext = ($1 > 9) }
```
85. Print each line three times: `{ print $0; print $0; print $0 }`
87. Copy two lines and delete the third:
```
{ print $0; getline; print $0; getline }
```
89. Copy lines 22 to 23 after line 56:
```
{
if (NR>=22 && NR<=23)
    held[NR]=$0
print $0
if (NR==56)
    {
    print held[22]
    print held[23]
    }
}
```
91. Prints the first and the third word in each line: `{print $1, $3}`
93. Evaluate expressions to true or false:
    a. `!(3 + 3 >= 6)`: false
    b. `1 + 6 == 7 || 3 + 2 == 1`: true
    c. `1 > 5 || 6 < 50 && 2 < 5`: true
    d. `14 != 55 && !(13 < 29) || 31 > 52`: false
95. Write an *if* statement that assigns 1 to `best` if `score` is 90 or greater:
```
if (score >= 90) best = 1
```
97. Find the smallest of ten integers:
```
#Find the smallest in an array
function smallestN(array, size)
    {
    smallest = array[0]
    for (i = 1; i < size; i++)
        if (array[i] < smallest) smallest = array[i]
    return smallest
    }

#find the smallest of 10 integers
function smallest10(n0, n1, n2, n3, n4, n5, n6, n7, n8, n9)
    {
     val[0]=n0; val[1]=n1; val[2]=n2; val[3]=n3; val[4]=n4
     val[5]=n5; val[6]=n6; val[7]=n7; val[8]=n8; val[9]=n9
     return smallestN(val, 10)
    }
```

Chapter 13: Interactive Korn Shell

Questions

1. The Korn shell can be used either interactively to receive user commands and interpret them, or as a programming language to create shell scripts.
3. No. You can change to a different subshell and therefore the current shell will not necessarily be the same as the login shell.
5. To create a child shell, enter the command name for the shell (such as `ksh` or `csh`). Then use **exit** to return to the parent shell.
7. The standard stream file descriptors are: standard input, `0`; standard output, `1`; and standard error, `2`.
9. The standard output stream is normally associated with the monitor.
11. To redirect the standard input stream to a file, use input redirection.

    ```
    command 0< filename
    ```

13. To redirect the standard input stream to a file, use error redirection.

    ```
    command 2> filename
    ```

    Also redirection override (`1>|`) and redirection append (`1>>`) may be used.
15. A sequence of commands can be entered on one line separated by semicolons (`;`). Commands can be grouped together using parentheses. Commands can be chained together using pipes.
17. Quotes are used to change the predefined meanings of characters. Three types of quotes are used for this purpose: backslash (`\`), a pair of double quotes (`"…"`), and a pair of single quotes (`'…'`).
19. Command substitution provides the capability to convert the result of a command to a string. The command substitution operator that converts the output of a command to a string is a dollar sign and a set of parentheses: `$(command)`. Another syntax, `` `command` ``, is also supported, but may be deprecated in the future.
21. To suspend a foreground job, use `ctrl-z`. The **bg** command moves a suspended job to the background. The **fg** command moves a suspended or background job to the foreground.
23. A variable is a location in memory where values can be stored.
25. The shell variables are used to configure the shell. The environmental variables control the user environment and can be exported to subshells.
27. When a command is executed, it returns a value known as the exit status of the command. The exit status of a command is stored in a shell variable called (`?`).
29. The environmental variable holds name of the login shell is: `SHELL`. It is displayed with `print $SHELL`.
31. The environmental variable holds the primary prompt is `PS1`. It is displayed with `print $PS1`.

33. The environmental variable holds the path of your home directory is `HOME`. It is displayed with `print $HOME`.
35. The environmental variable holds the path of your mailbox is `MAIL`. It is displayed with `print $MAIL`.
37. The environmental variable holds your login name is `LOGNAME`. It is displayed with `print $LOGNAME`.
39. Options are on/off switches that are used to control the shell. Three such options are `noclobber`—do not allow redirection to clobber existing files, `ignoreeof`—disallow `ctrl+d` to exit the shell, and `verbose`—print commands before executing them.

Exercises

41. The command that shows your login shell is print `$SHELL` (a).
43. The descriptor of the input standard stream is 0 (a). The descriptor of the output stream is 1 (b). The descriptor of the error stream is 2 (c).
45. The valid Korn shell variables are `cat` (a), `_first` (b), and `2Var` (d).
47. The output is:
    a. `a`
       `44`
    b. <a blank line>
       `a44`
    c. `44a`
49. Use a **grep** command and check its exit status.
    ```
    grep root /etc/passwd > /dev/null; print $?
    ```
    This command should always be successful, so the exit status is zero. If we substitute "no_user" (or some other bogus username) for "root", the **grep** command will be unsuccessful, so the exit status will be non-zero.
51. Use a **awk** command and check its exit status.
    ```
    awk '$0 ~ /root/' /etc/passwd > /dev/null; print $?
    ```
    This command should always be successful, so the exit status is zero. If we introduce an error into the script, such as leave out the `/root/` regular expression, the **awk** command will be unsuccessful, so the exit status will be non-zero.
53. To show the value of all environmental variable, use the **set** command.
55. The first print will print `Bye`, and the second print will print `Hello`. This is because the variable `x` in the subshell is independent of the variable `x` in the parent shell.

Chapter 14: Korn Shell Programming

Questions

1. No, the Korn shell does not support `goto`.
3. Yes, an *until* loop can be coded to execute infinitely. Here is an example.

```
until false; do
    date; sleep 2 # print the time every two seconds
done
```

5. The Korn shell expressions are:
   a. `3 + 4: (( 3 + 4 ))`
   b. `x * 14: (( x * 14 ))`
   c. `4 / x: (( 4 / x ))`
   d. `x % 5: (( x % 5 ))`
7. The Korn shell expressions are:
   a. `x > 9: (( x > 9 ))`
   b. `x < = y: (( x <= y ))`
   c. `x != y: (( x != y ))`
   d. `x = = y: (( x == y ))`
9. Write Korn shell expressions for:
   a. file1 is empty: `[[ ! -s file1 ]]`
   b. file2 is not readable: `[[ ! -r file2 ]]`
11.
   a. while loop repeated 10 times

```
loop_count=10
while (( loop_count > 0 )) ; do
   (( loop_count = loop_count - 1 ))
   echo $loop_count
done
```

   b. until loop repeated 10 times

```
loop_count=10
until (( loop_count == 0 )) ; do
      (( loop_count = loop_count - 1 ))
      echo $loop_count
done
```

   c. for-in loop repeated 10 times

```
for i in 0 1 2 3 4 5 6 7 8 9; do
   echo $i
done
```

   d. Select is used for generating menus, not to control repetitions.

Exercises

13. The value of `x` and `y`:
    a. `(( x = x%2 )):  x = 1, y = 5`
    b. `(( x = x + 4 )): x = 17, y = 5`
    c. `(( x = x * 3 )): x = 39, y = 5`
    d. `(( x = x / y )): x = 2, y = 5`
15. The value of each expression:
    a. `(( x < 12  )):` 1
    b. `(( y > 7  )):` 1
    c. `(( x < y )):` 1
    d. `(( y == x )):` 1
17. The loop prints an infinite sequence of `hello`.
19. The loop prints an infinite sequence of `hello`.
21. This loop prints an infinite sequence of `12` because the expression `(( x > 7 ))` is always true.
23.
    a. This loop prints the date and time followed by `Hello` and infinitely repeats because **date** is executed and always returns a true exit status.
    b. This loop prints an error because the *while* evaluates the output of the **date** command, which is a string instead of an exit status.
25. This loop prints the values of `i` in the list: `12 11 10 8 7`.
27. This prints `Hello` seven times because that is how many values are in the list.
28. This prints `Hello` four times because that is how many values (strings) are in the list.
29.
    a. This loop prints `Hello` one time because the list contains the string date which is not interpreted as a command in this context.
    b. This loop prints `Hello` six times because the list contains the string output of the date command.
31. This loop prints the first three space separated strings of the **date** command on three separate lines. That is, the day of the week, the month, and the date, on separate lines.
33. The script prints five lines of `12345`. The inner loop prints the variable `j` as it changes values from `1` to `5` and the outer loop prints a newline. The outer loop repeats five times.
35. This loop prints the numbers `2` to `11`, which is the data in the *for-in* loop list plus 1. The increment expression in the loop increments each value before being printed.
37. This loop prints the first nine odd numbers, from `1` to `17`. As the outer loop increments `j` from `1` to `9`, the inner loop prints `j*2-1`.
39. Simulate *while* loop with a *for-in* loop.

```
IFS="
"
for line in $(cat file1)
```

```
do
...
done
```

Chapter 15: Korn Shell Advanced Programming

Questions

1. `24`

3. `40`

5. `Hello` is printed twice.

7. This is the world of UNIX

   (The purpose of this question is to show that quotes preserve multiple spaces.)

9. `print "Hello World" | sed "s/\(..\).*/\1/"`

11. `print "Hello World" | sed "s/.*\(..\)$/\1/"`

13. Ignore a `SIGHUP` signal.

    ```
    trap "" SIGHUP
    ```

15. Use **getopts** to capture two options. `getopts a:b variable`

Exercises

17. Write a **sed** script.

    ```
    echo $1 | sed -n '/^\(.\).*\1$/p'
    ```

19. Create a new file containing only one word lines.

    ```
    sed -n '/^[^ ]\{1,\}$/p' < $1 > ${1}.words
    ```

21. Write a script that deletes all even lines

    ```
    sed -n 'N;P' $1
    ```

23. Write a script that combines odd and even lines together.

    ```
    sed -n 'N;s/\n//;p' $1
    ```

25. Write a script that prints the name of the file that is newer.

    ```
    #!/bin/ksh
    #print newer filename
    if [[ $1 -nt $2 ]]; then
        print $1
    else
        print $2
    fi
    ```

27. What is the exit status of the script: `0`

29. What is the exit status of a **sed** command when it finds a pattern? … when it does not find a pattern?

    A legal **sed** script that executes without error will generate an exit status of `0` regardless of whether it finds a pattern. Here is a one line script that searches itself for `sed`, which it finds and shows the exit status of `0`. It also searches itself for `SED`, which it does not find, and shows the exit status of `0`.

    ```
    sed -n "/sed/p" $0; echo $?; sed -n "/\S\E\D/p" $0; echo $?
    ```

31. Store the first argument in `$1` using the **getopts** command.

```
#!/bin/ksh
echo $OPTIND
while getopts stuv variable 2>/dev/null
do
    case $variable in
    s) echo "processing s" ;;
    t) echo "processing t" ;;
    u) echo "processing u" ;;
    v) echo "processing v" ;;
    \?) print "Invalid option: Quitting" ; exit 1
    esac
done
# shift arguments to start at $1
(( argShift=OPTIND-1 ))
shift $argShift
# show arguments
echo $1 $2 $3
```

Chapter 16: Interactive C Shell

Questions

1. The C shell can be used either interactively to receive user commands and interpret them or as a programming language to create shell scripts.
3. No, the login shell is not always the same as your current shell. You can change to a different subshell, and therefore the current shell will not necessarily be the same as the login shell.
5. To return to the parent shell, use the command `exit`.
7. The three standard streams are standard input, standard output, and standard error.
9. The standard output stream is normally associated with the monitor.
11. The standard input stream can be redirected to a file using input redirection:
    `command < filename.`
13. The standard error stream can be redirected to the same file as the standard output by using append redirection: `command >& file1`.
15. A sequence of commands can be entered on one line separated by semicolons (`;`). Commands can be grouped together using parentheses. Commands can be chained together using pipes.
17. Quotes are used to change the predefined meanings of characters. Three types of quotes are used for this purpose: backslash (`\`), a pair of double quotes (`"..."`), and a pair of single quotes (`'...'`).
19. Command substitution provides the capability to convert the output of a command to a string. The command substitution in the C shell consists of backquotes surrounding the command, as in `` `date.` ``
21. To suspend a foreground job key `ctrl-z`. The **bg** command moves a suspended job to the background. The **fg** command moves a suspended or background job to the foreground.
23. A variable is a location in memory where values can be stored.
25. The shell variables are used to configure the shell. The environmental variables control the user environment and can be exported to subshells.
27. When a command is executed, it returns a value known as the exit status of the command. In the C shell, the exit status of a command is stored in a shell variable called `status`.
29. The environmental variable that holds the name of the login shell is `SHELL`. It is displayed with `echo $SHELL`.
31. The environmental variable that holds the primary prompt in the C shell is `PROMPT/prompt`. It is displayed with `echo $prompt`.
33. The mail check interval is not stored in a shell variable in the C shell.

35. The environmental variable that holds the type of your terminal is `TERM/term`. It is displayed with `echo $TERM`.
37. The startup scripts are: `/etc/csh.login`, `~/.login`, `/etc/csh.cshrc`, and `~/.cshrc`.

Exercises

39. The command that shows the login shell is echo `$SHELL` (a).
41. Five commands that cannot be used with input redirection are: **date**, **who**, **cal**, **set**, and **cd**.
43. The display output would be:
    a. a
       44
    b. Undefined
    c. 44a
       4444
45. Use a **grep** command and check its exit status.
    ```
    grep root /etc/passwd > /dev/null; echo $status
    ```
    This command should always be successful, so the exit status is zero. If we substitute "no_user" (or some other bogus username) for "root", the **grep** command will be unsuccessful, so the exit status will be non-zero.
47. Use a **awk** command and check its exit status.
    ```
    awk '$0 ~ /root/' /etc/passwd > /dev/null; echo $status
    ```
    This command should always be successful, so the exit status is zero. If we introduce an error into the script, such as leave out the `/root/` regular expression, the **awk** command will be unsuccessful, so the exit status will be non-zero.

Chapter 17: C Shell Programming

Questions

1. Yes, although goto statements are discouraged.
3. C shell foreach loops are list-controlled and can only loop a finite number of times.
5. The C shell expressions are:
   a. `(3 + 4) * 8:` `@ result = (3 + 4) * 8`
   b. `(x * 14) / y:` `@ result = ($x * 14) / $y`
   c. `z = t * (4 / x) + 6:` `@ z = $t * (4 / $x) + 6`
   d. `x = x + y + z:` `@ x = $x + $y + $z`
7. The C shell expressions are:
   a. `(x == 5) && (y > 10):` `(($x == 5) && ($y > 10))`
   b. `(x < = y) || (y > 10):` `(($x <= $y) || ($y > 10))`
   c. `(x != y) || (x > 7) && (y <= x):`
      `(($x != $y) || ($x > 7) && ($y <= $x))`
   d. `(x = = y) || (x > 5) || (y > 6):`
      `(($x == $y) || ($x > 5) || ($y > 6))`
9.
   a. *while* loop repeated 10 times:

```
set count = 10
while ($count > 0)
    @ count -= 1
    echo $count
end
```

   b. *foreach* loop repeated 10 times

```
foreach i (0 1 2 3 4 5 6 7 8 9)
    echo $i
end
```

   c. *repeat* loop repeated 10 times

```
repeat 10 echo hello
```

Exercises

11. The value of x and y:
    a. `x = 1, y = 5`
    b. `x = 17, y = 5`
    c. `x = 39, y = 5`
    d. syntax error
13. The value of each expression:
    a. `( $x < 12 && $y > 7 ):` `false`
    b. `( $y > 7 || $x < $x ):` `false`

c. `( ! $y < 7 ):` `true` (`!` has high order of precedence.)

d. `( ! $x >= 8 && ! $x < 9 ): false`

15. The loop prints nothing.

17. The loop prints nothing.

19. The loop prints `5`, `<`, and `6` on three separate lines.

21. The prints `12`, `11`, `10`, `9`, and `8` on separate lines as `x` decrements inside the loop.

23. The loop prints `Hello` three times because `x` yields three values.

25. The loop prints `Hello` four times because `x` yields four values.

27.

a. The loop prints `Hello` one time because the *foreach* list consists of one item, the word date (not the command date).

b. The loop prints `Hello` six times, once for each field output by the **date** command.

29. The script segment prints the values `1` through `10`, the values in the foreach list.

31. Each line's contents is controlled by the inner loop which counts from 1 to `i`, while the outer loop increments `i` from `1` to `9`, so this prints the following:

```
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

33. The script segment prints the odd numbers `1` through `9` on separate lines. Each value in the foreach list is multiplied by `2` and decremented and then printed if the result is less than or equal to `9`.

Chapter 18: C Shell Advanced Programming

Questions

1. The command displays `This is the world of UNIX`
   (The purpose of this question is to show that quotes preserve multiple spaces.)
3. There are nine elements in the array.
5. This results in an error since the **echo** command will reference a non-existent element.
7. The code prints `44`, `66`, `77` on separate lines
9. This code prints the number of words in he file.

Exercises

11. Change the name of files passed as arguments to lower-case.

```
#!/bin/csh
#convert filename ($argv[1]) to lowercase
set lowername = `echo $argv[1] | tr "A-Z" "a-z"`
mv $argv[1] $lowername
```

13. Write a script that deletes all odd lines

```
#!/bin/csh
set filename = $argv[1]
sed -n 'n;p' $filename
```

15. Write a script that changes the first three characters of file names to lower-case.

```
#!/bin/csh
# change first three letters of filenames to lowercase
while ( $#argv > 0 )
    set name       = $argv[1]
    set first3 = `echo $name | sed "s/^\(...\).*/\1/"`
    set end    = `echo $name | sed "s/^...\(.*\)/\1/"`
    set lower3 = `echo $first3 | tr "[A-Z]" "[a-z]"`
    set lower3Name = ${lower3}${end}
    mv $name $lower3Name
    shift
end
```

17. The exit value of the script is `0`.
19. A legal **sed** script that executes without error will generate an exit status of `0` regardless of whether it finds a pattern. Here is a one line script that finds a match and shows the exit status of `0`. It then does not find a match and shows the exit status of `0`:

```
   echo xx | sed -n "/xx/p"; echo $status; echo xx | sed -n "/yy/p";
echo $status
```