INTRODUCTION TO SHELLS

- The UNIX operating system contains four distinct parts:- the kernel, the shell, utilities, applications.

- The shell is the part of UNIX that is most visible to the user. It receives and interprets the commands entered by the user.

- There are two major parts to a shell. The **first** is the interpreter reads your commands and works with the kernel and execute them.

- The **second** part of the shell is a programming capability that allows you to write a shell script.

- A **shell script** is a file that contains shell commands that performs a useful function. It is also known as a **shell program**.

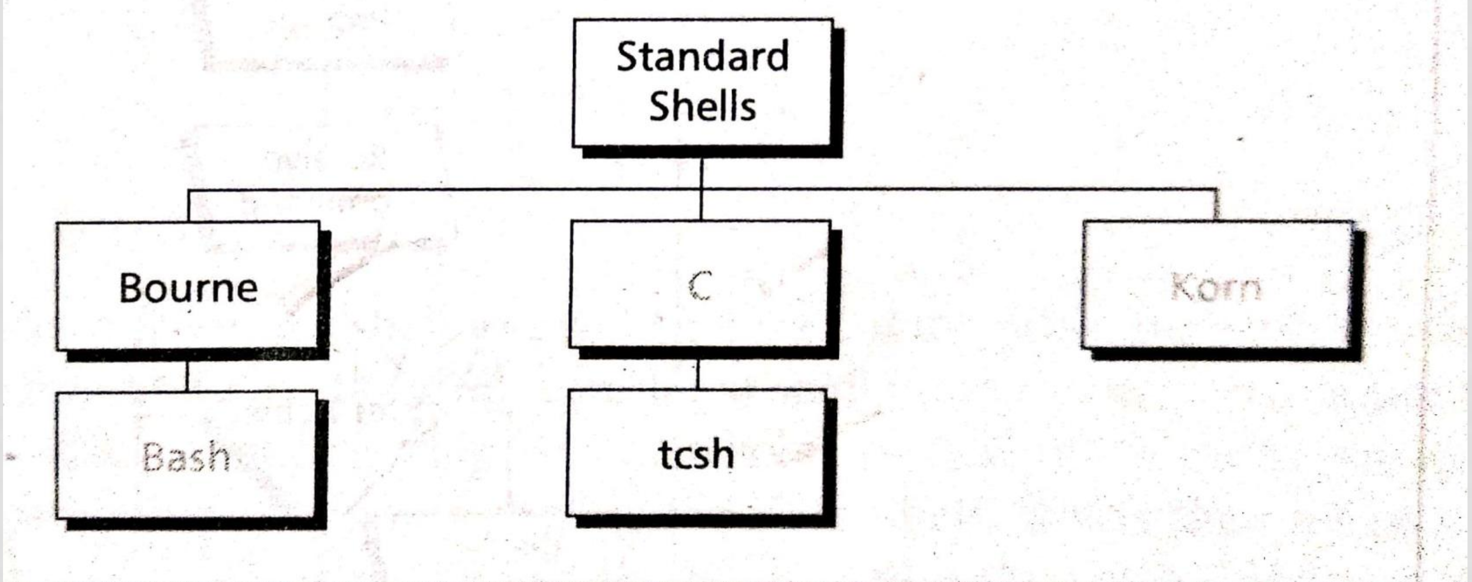


FIGURE 5.1 *Five Standard UNIX Shells*

UNIX SESSION

- $ bash
- $ ksh
- $ csh

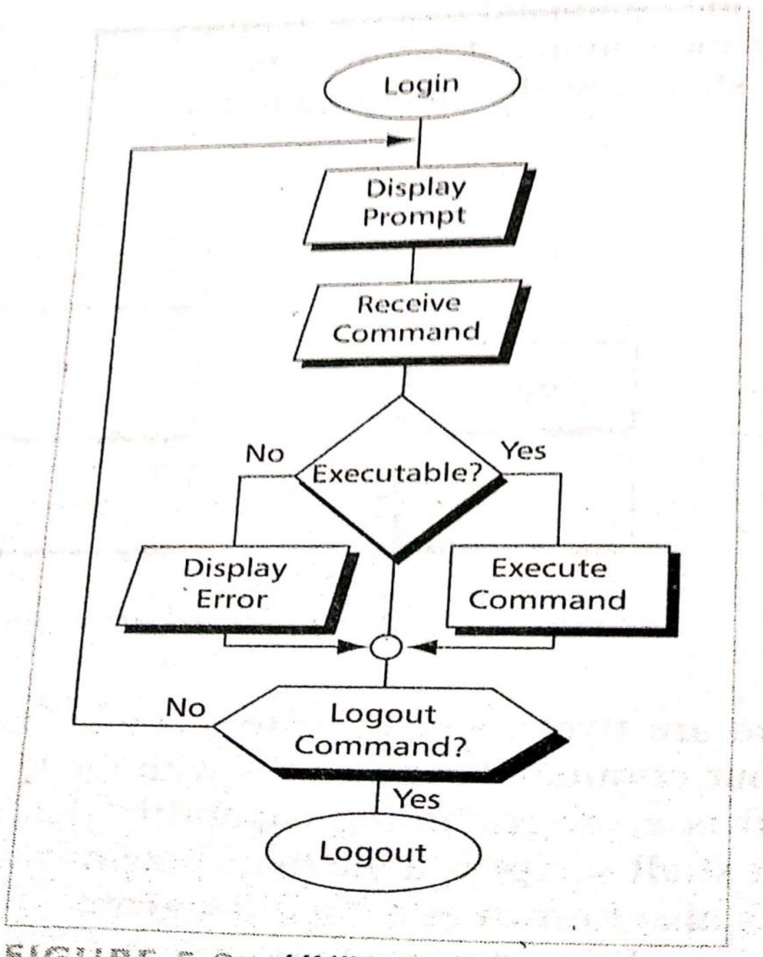


FIGURE 5.2 *UNIX Workflow*

- **Login Shell Verification**
- $ echo $SHELL
- /bin/ksh

- **Current Shell Verification**
- $ echo $0
- ksh

- **Shell Relationships**
- $ exit

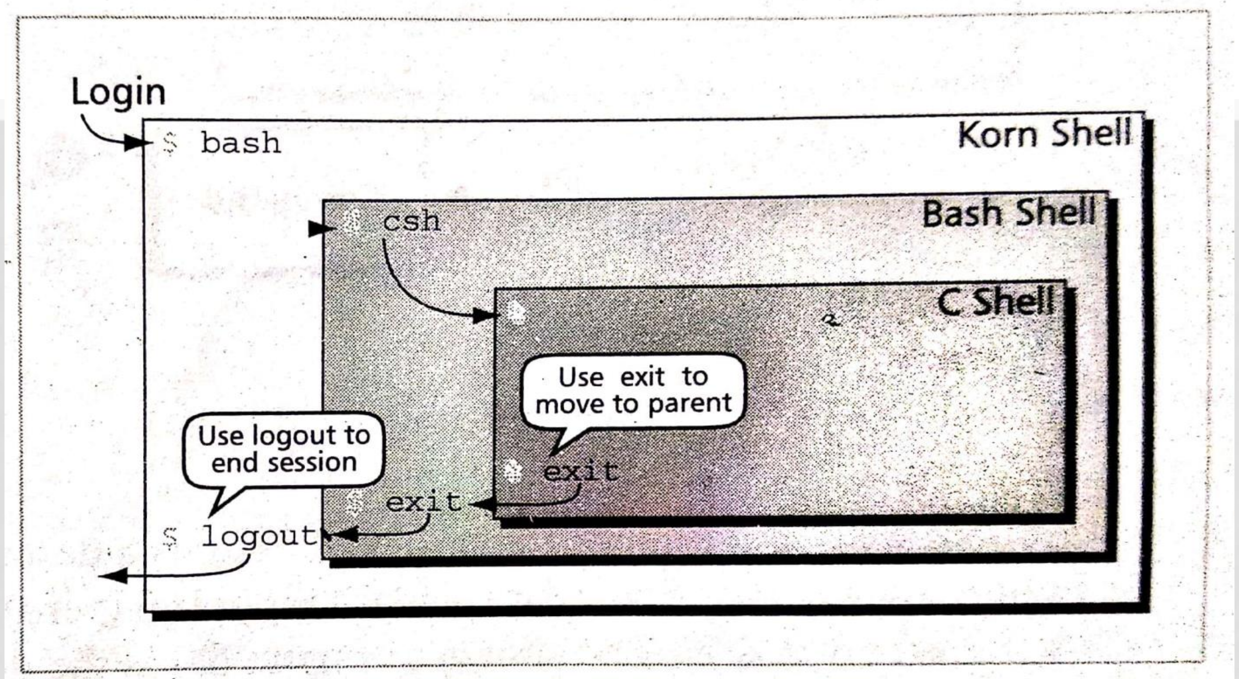


FIGURE 5.3 *Shell Relationships*

- **Logout**
- $ logout

STANDARD STREAMS

- Standard input (0)
- Standard output (1)
- Standard error (2)
- The **lpr** command send its output directly to the printer.

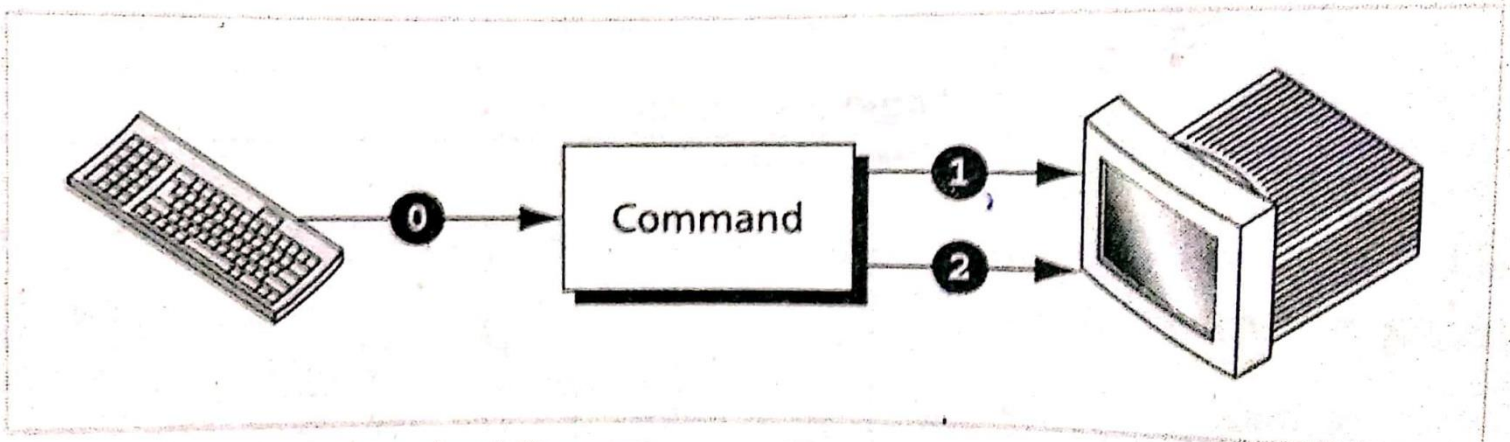


FIGURE 5.4 *Standard Streams*

REDIRECTION

- Redirection is the process by which we specify that a file is to be used in place of one of the standard files.
- **Redirecting input**
- We can redirect the standard input from the keyboard to any text file.
- The input redirection operator is less than character (<).

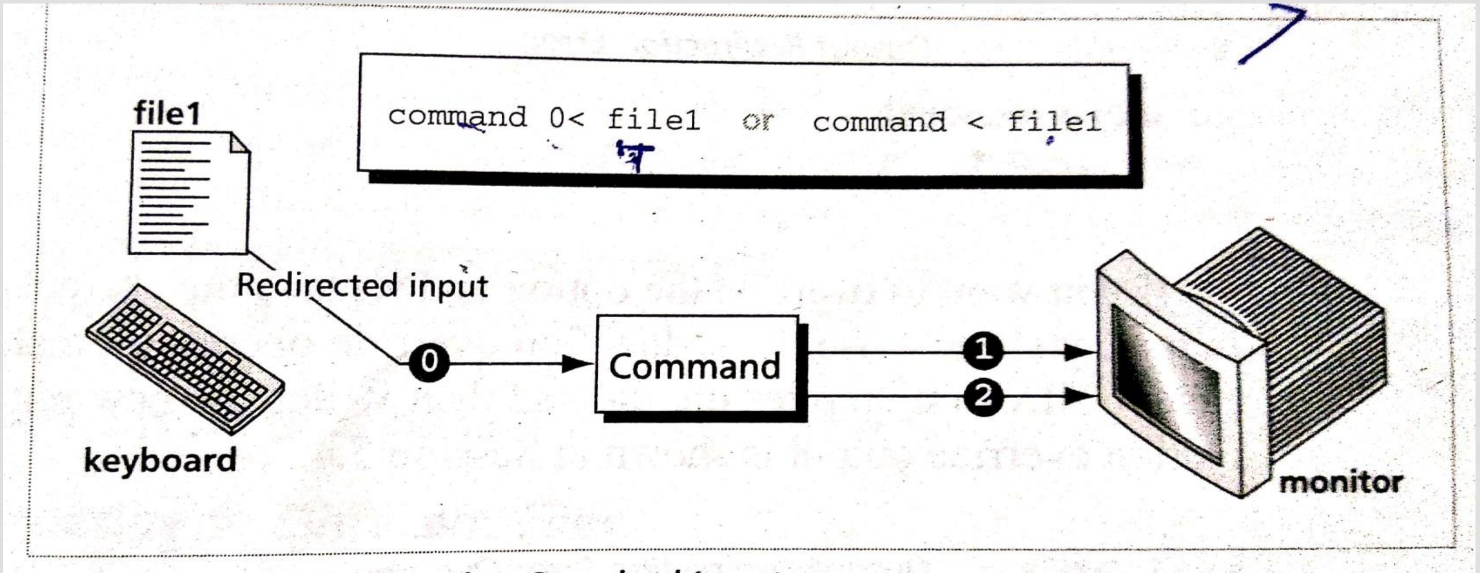
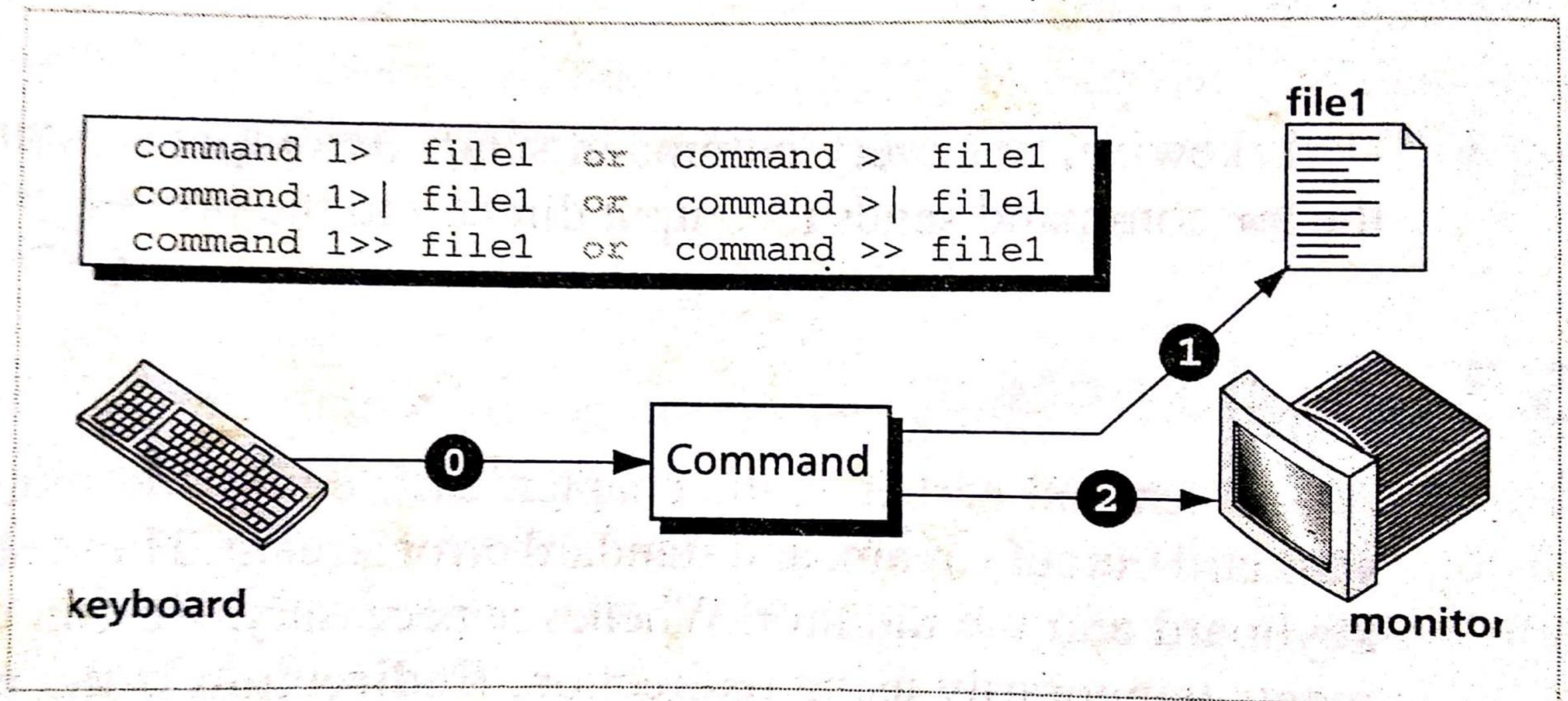


FIGURE 5.5 *Redirecting Standard Input*

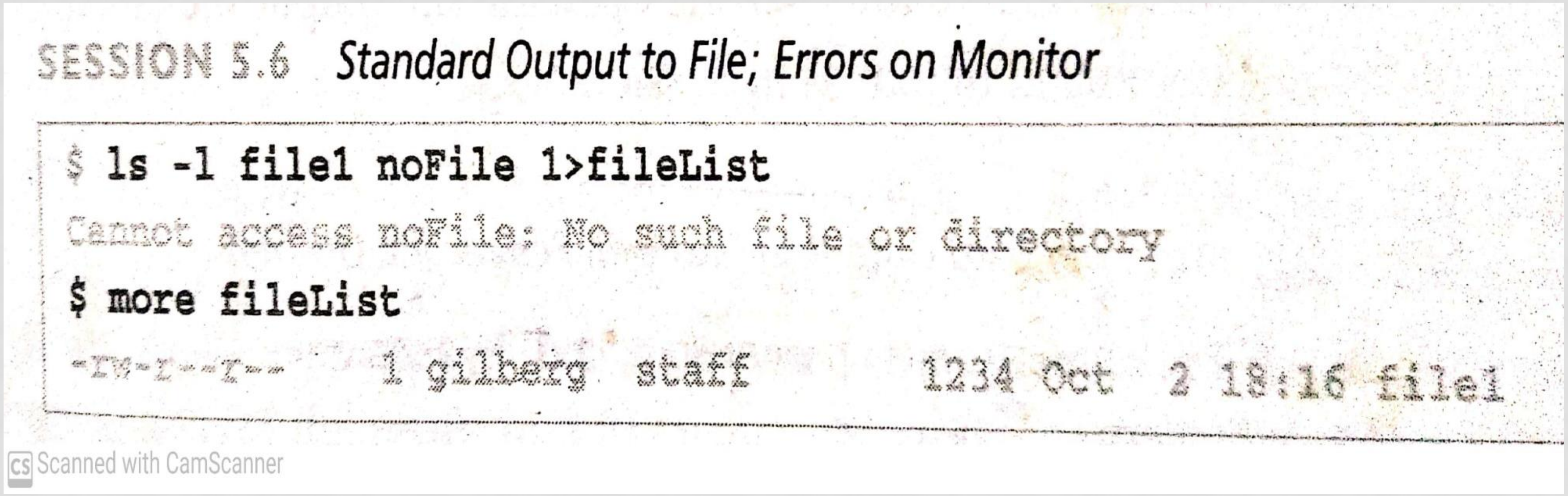
- **Redirecting output**
- When we redirect standard output, the command's output is copied to a file rather than displayed on the monitor.



```
command 1>   file1   or   command >   file1
command 1>|  file1   or   command >|  file1
command 1>>  file1   or   command >>  file1
```

FIGURE 5.6 *Redirecting Standard Output*

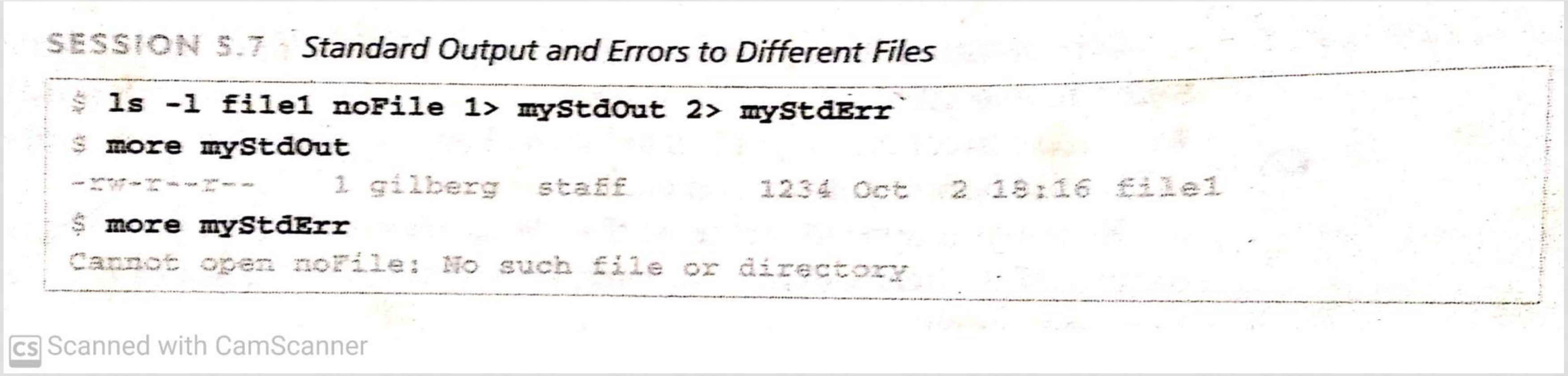
- **Redirecting Errors**
- One of the difficulties with the standard error stream is that it is, by default, combined with standard output stream on the monitor.


SESSION 5.6 *Standard Output to File; Errors on Monitor*

```
$ ls -l file1 noFile 1>fileList
Cannot access noFile: No such file or directory
$ more fileList
-rw-r--r--    1 gilberg  staff        1234 Oct  2 18:16 file1
```

Scanned with CamScanner

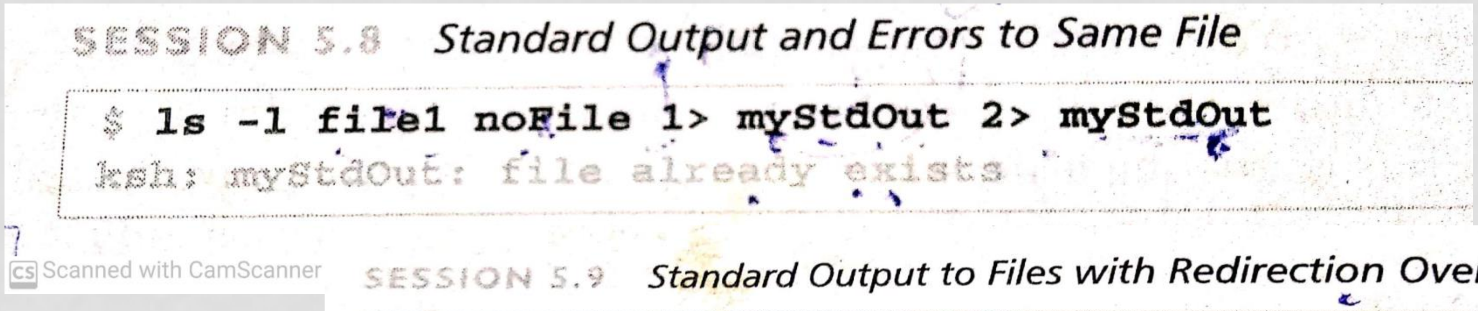
- **Redirecting to Different Files**
- To redirect to different files, we must use the stream descriptors.


SESSION 5.7 *Standard Output and Errors to Different Files*

```
$ ls -l file1 noFile 1> myStdOut 2> myStdErr
$ more myStdOut
-rw-r--r--    1 gilberg  staff        1234 Oct  2 18:16 file1
$ more myStdErr
Cannot open noFile: No such file or directory
```

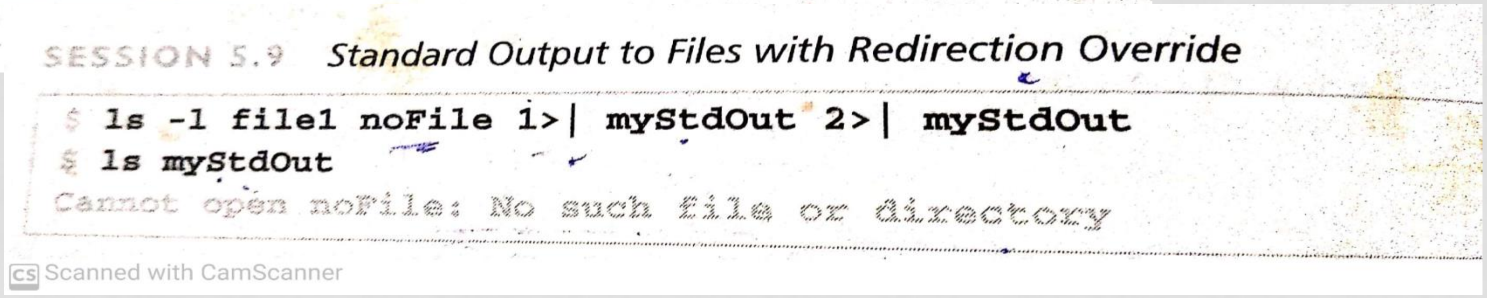
Scanned with CamScanner

- **Redirecting to One Files**


SESSION 5.8 *Standard Output and Errors to Same File*

```
$ ls -l file1 noFile 1> myStdOut 2> myStdOut
ksh: myStdOut: file already exists
```

Scanned with CamScanner


SESSION 5.9 *Standard Output to Files with Redirection Override*

```
$ ls -l file1 noFile 1>| myStdOut 2>| myStdOut
$ ls myStdOut
Cannot open noFile: No such file or directory
```

Scanned with CamScanner

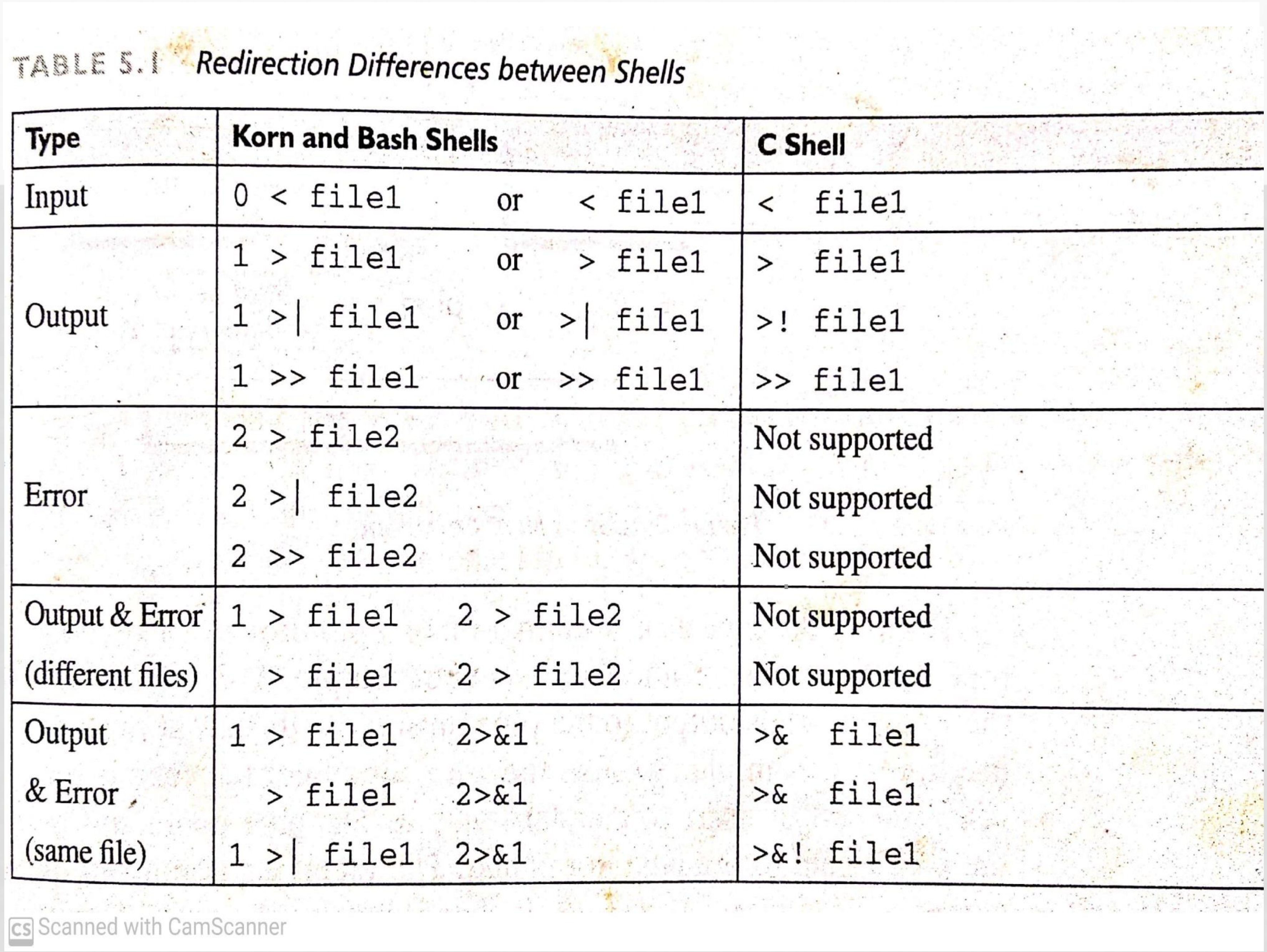


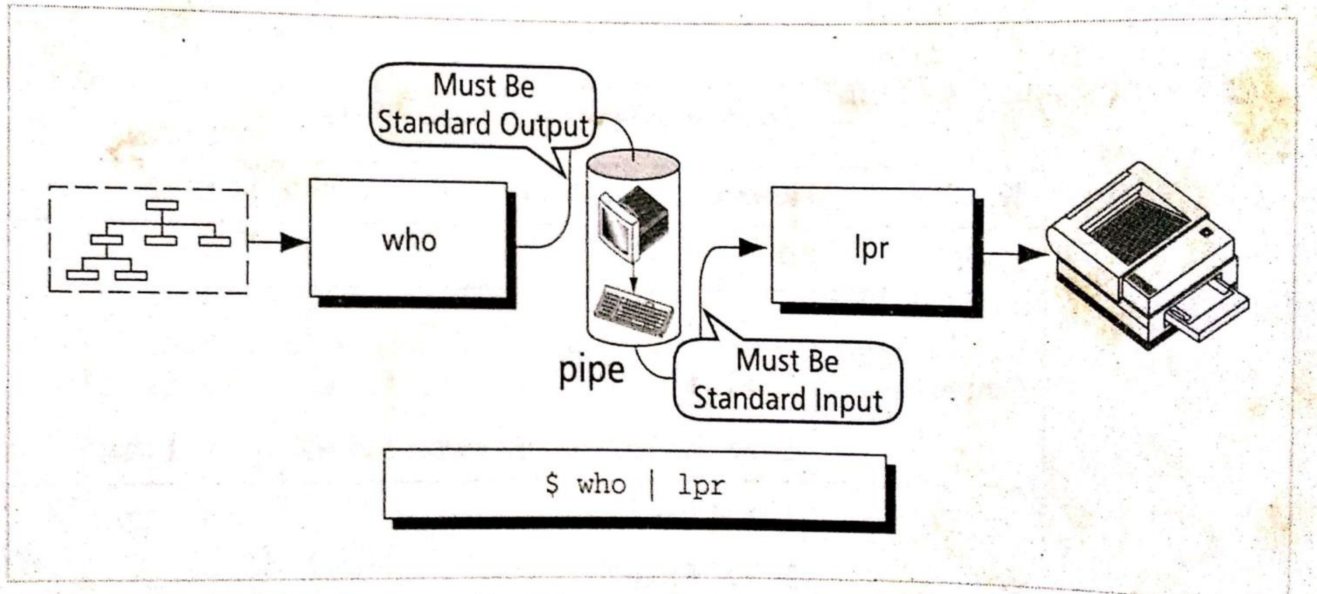
TABLE 5.1 *Redirection Differences between Shells*

Type	Korn and Bash Shells	C Shell
Input	`0 < file1` or `< file1`	`< file1`
Output	`1 > file1` or `> file1`	`> file1`
	`1 >| file1` or `>| file1`	`>! file1`
	`1 >> file1` or `>> file1`	`>> file1`
Error	`2 > file2`	Not supported
	`2 >| file2`	Not supported
	`2 >> file2`	Not supported
Output & Error (different files)	`1 > file1 2 > file2`	Not supported
	`> file1 2 > file2`	Not supported
Output & Error (same file)	`1 > file1 2>&1`	`>& file1`
	`> file1 2>&1`	`>& file1`
	`1 >| file1 2>&1`	`>&! file1`

PIPES

- We often need to use a series of commands to complete a task.
- 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.
- Think of the pipe as a combination of a monitor and a keyboard. The input to the pipe operator must come from standard output.

- The token for a pipe is the vertical har (|).
- The pipe is not a command, it is an operator. It must be placed between two commands.
- The pipe tells the system that these two commands need to share the output of the first command and to pass it directly to the second command.



```
$ who | lpr
```

FIGURE 5.8 *Piping Output to Next Command*

tee COMMAND

- The **tee** command copies standard input to standard output and at the same time copies it to one or more files.
- The first copy goes to standard output, which is usually the monitor. At the same time, the output is sent to the optional files specified in the argument list.
- The **tee** command creates the output files if they do not exist and overwrites them if they already exist.

- To prevent the files from being overwritten, we can use the option **–a**, which tells **tee** to append the output to existing files rather than deleting their current content.

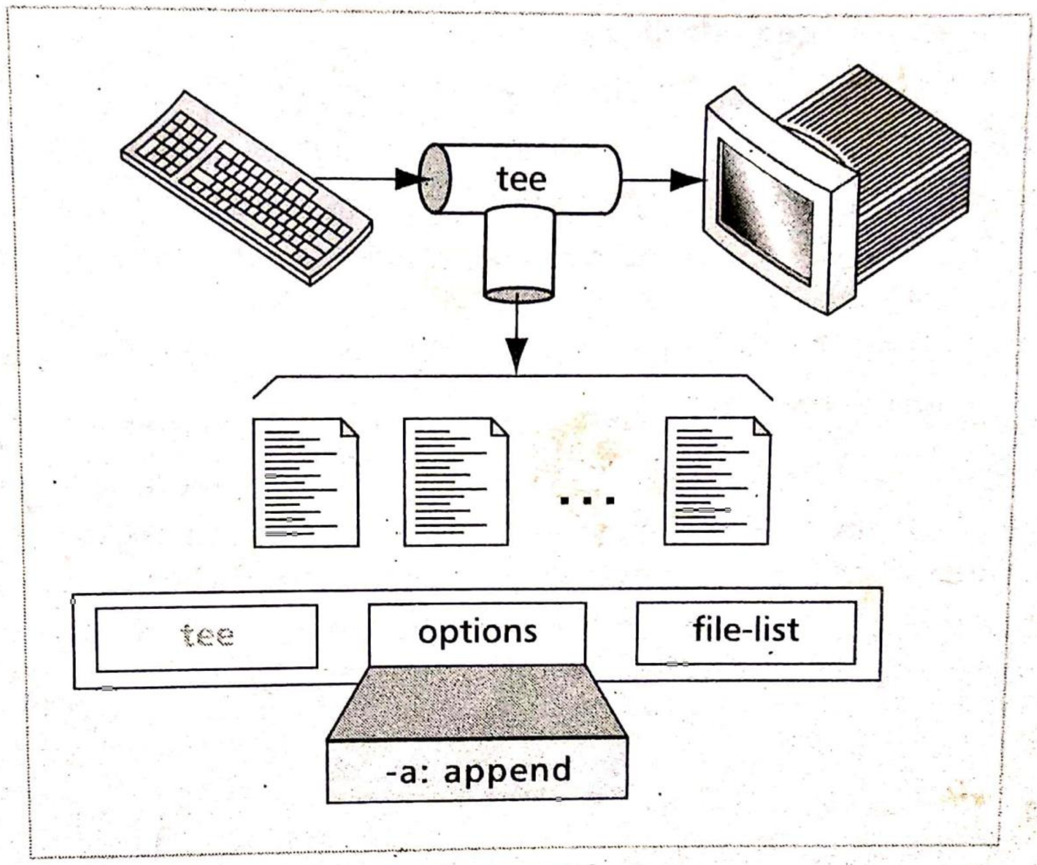
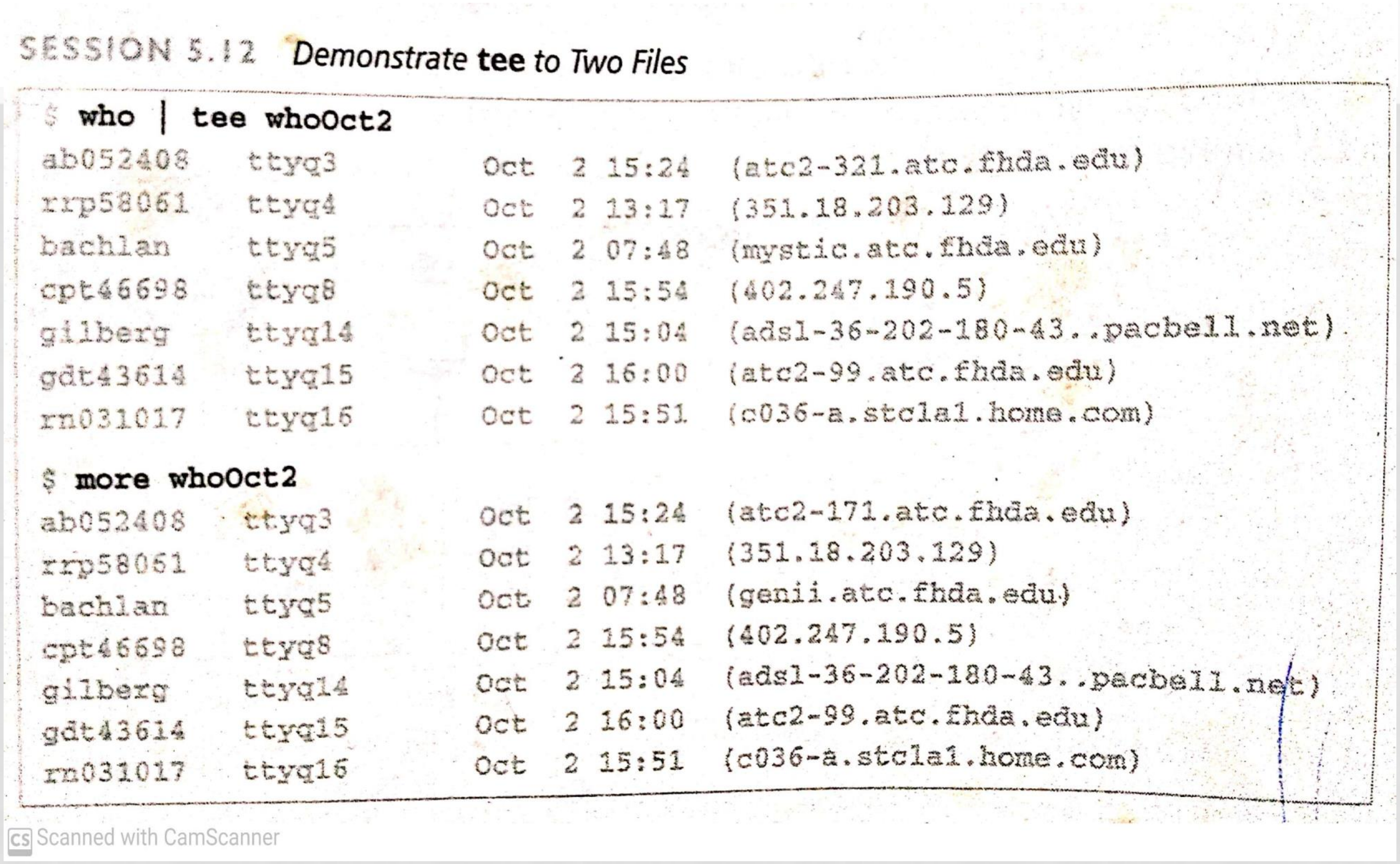


FIGURE 5.9 *The **tee** Command*



SESSION 5.12 *Demonstrate* **tee** *to Two Files*

```
$ who | tee whoOct2
ab052408    ttyq3          Oct  2 15:24  (atc2-321.atc.fhda.edu)
rrp58061    ttyq4          Oct  2 13:17  (351.18.203.129)
bachlan     ttyq5          Oct  2 07:48  (mystic.atc.fhda.edu)
cpt46698    ttyq8          Oct  2 15:54  (402.247.190.5)
gilberg     ttyq14         Oct  2 15:04  (adsl-36-202-180-43..pacbell.net)
gdt43614    ttyq15         Oct  2 16:00  (atc2-99.atc.fhda.edu)
rn031017    ttyq16         Oct  2 15:51  (c036-a.stcla1.home.com)

$ more whoOct2
ab052408    ttyq3          Oct  2 15:24  (atc2-171.atc.fhda.edu)
rrp58061    ttyq4          Oct  2 13:17  (351.18.203.129)
bachlan     ttyq5          Oct  2 07:48  (genii.atc.fhda.edu)
cpt46698    ttyq8          Oct  2 15:54  (402.247.190.5)
gilberg     ttyq14         Oct  2 15:04  (adsl-36-202-180-43..pacbell.net)
gdt43614    ttyq15         Oct  2 16:00  (atc2-99.atc.fhda.edu)
rn031017    ttyq16         Oct  2 15:51  (c036-a.stcla1.home.com)
```

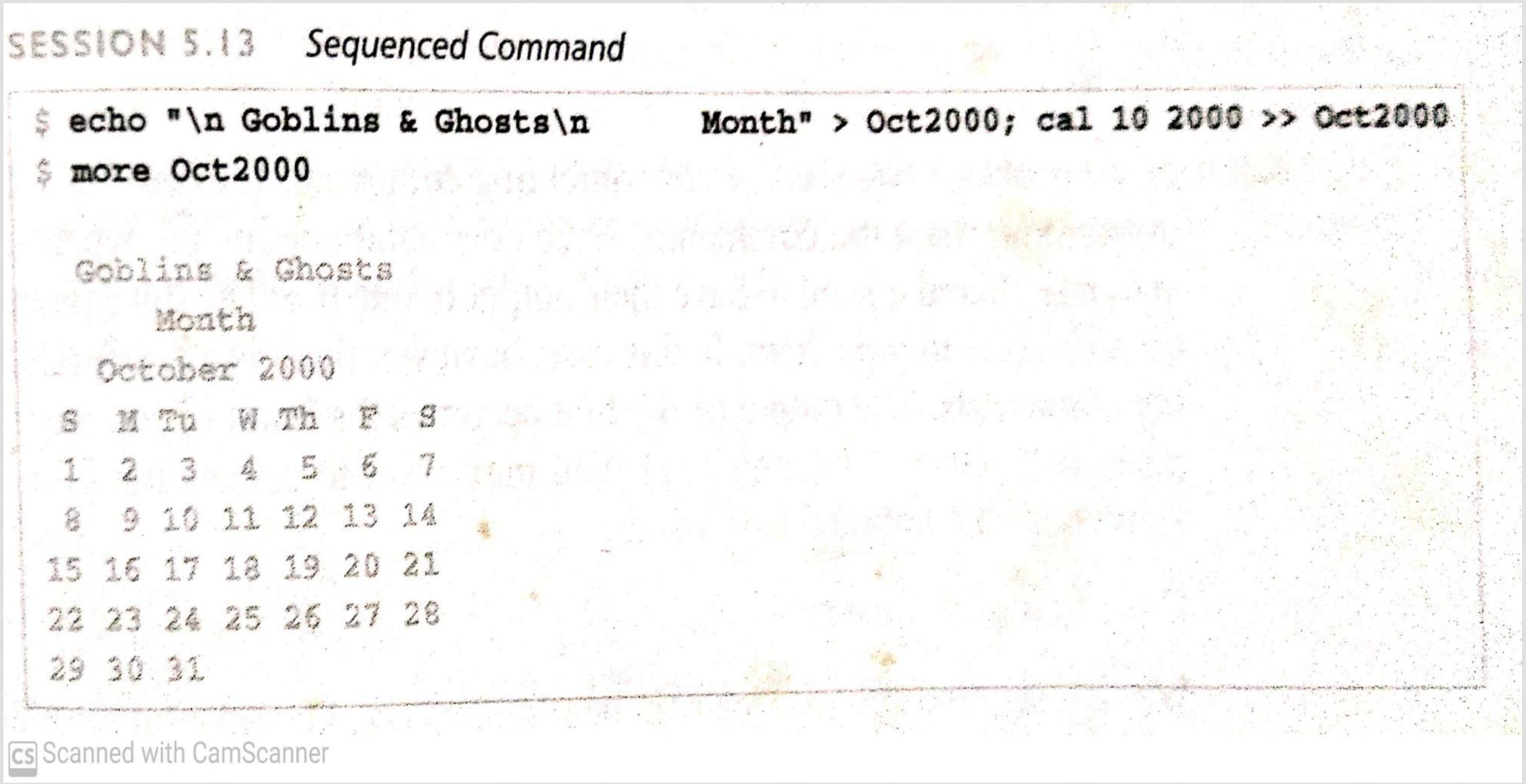
COMMAND EXECUTION

- There are four syntactical format for combining commands into one line :- sequenced, grouped, chained and conditional.

- **Sequenced Commands**

- A sequence of commands can be entered on one line. Each command must be separated from its predecessor by semicolon.

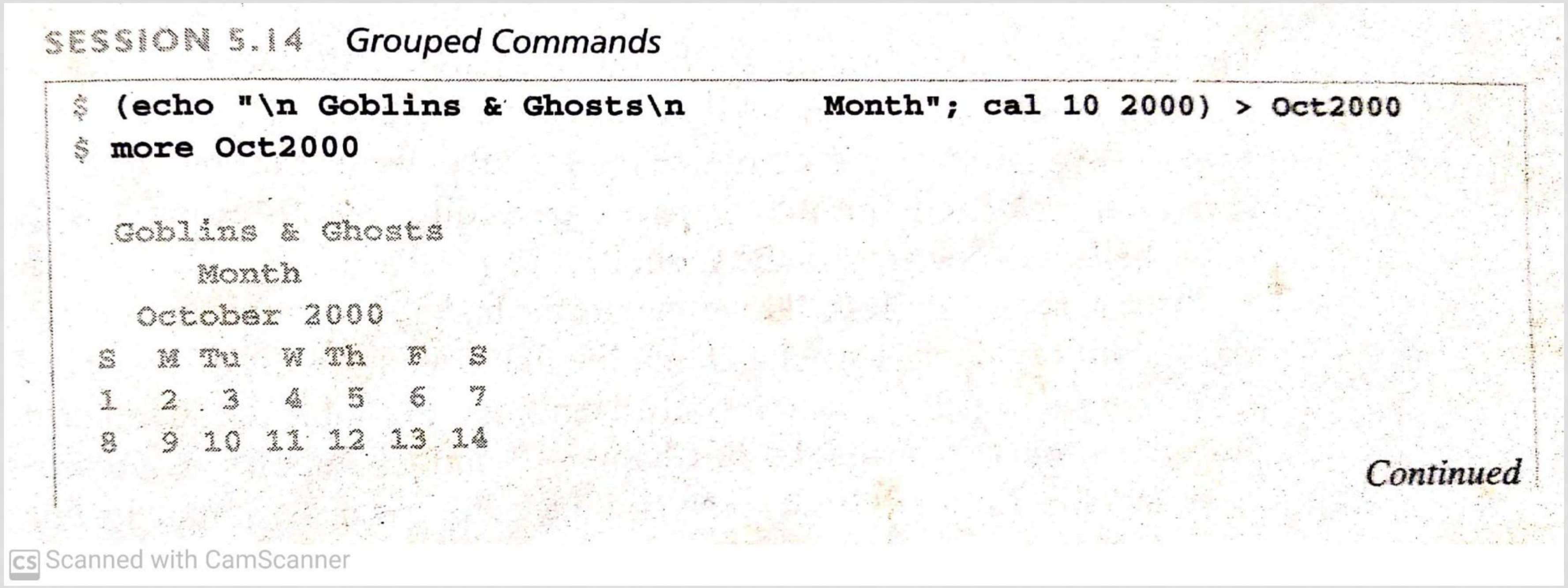
- There is no direct relationship between the commands, that is, one command does not communicate with the other.



SESSION 5.13 *Sequenced Command*

```
$ echo "\n Goblins & Ghosts\n      Month" > Oct2000; cal 10 2000 >> Oct2000
$ more Oct2000

 Goblins & Ghosts
      Month
   October 2000
 S  M Tu  W Th  F  S
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

- **Grouped Commands**
- When we group commands, we apply the same operation to the group.
- Commands are grouped by placing them in parentheses.



SESSION 5.14 *Grouped Commands*

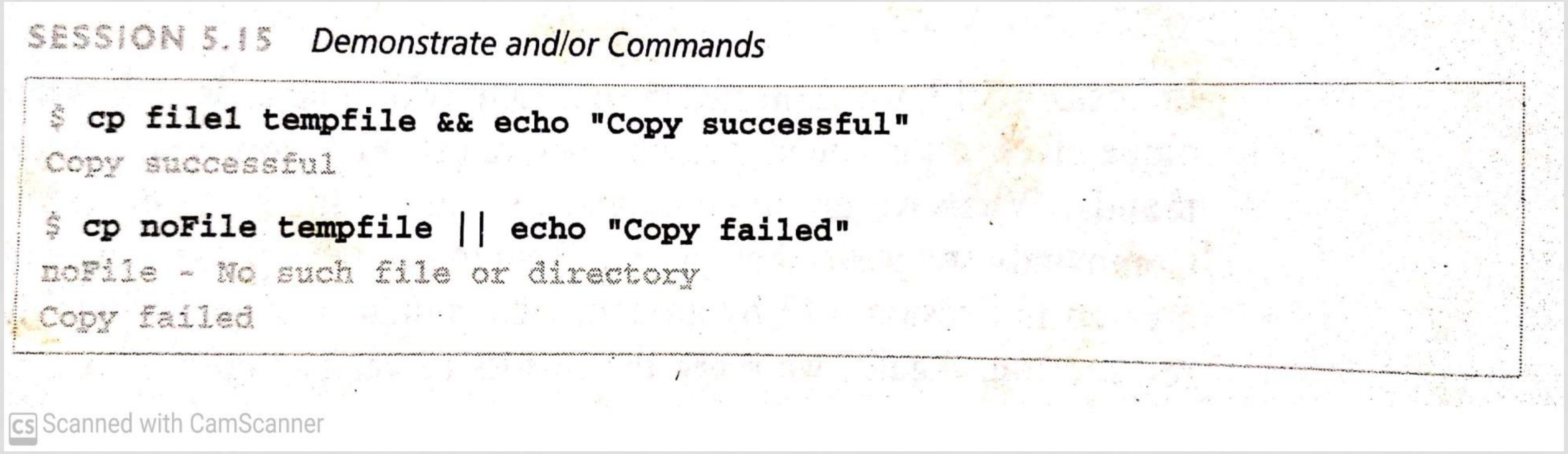
```
$ (echo "\n Goblins & Ghosts\n        Month"; cal 10 2000) > Oct2000
$ more Oct2000

  Goblins & Ghosts
       Month
   October 2000
 S  M Tu  W Th  F  S
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
```

Continued

- **Chained Commands**
- The third method of combining commands is to pipe them. There is a direct relationship between the commands.
- The output of the first becomes the input of the second.

- **Conditional Commands**
- We can combine two or more commands using conditional relationships.
- There are two shell logical operators, *and (&&)* and *or (| |).*
- When two commands are combined with a logical and, the second executes only if the first command is successful.

- If two commands are combined using the logical or, the second command executes only if the first fails.



SESSION 5.15 *Demonstrate and/or Commands*

```
$ cp file1 tempfile && echo "Copy successful"
Copy successful

$ cp noFile tempfile || echo "Copy failed"
noFile - No such file or directory
Copy failed
```

COMMAND – LINE EDITING

- The history file is a special UNIX file that contains a list of commands used during a session.

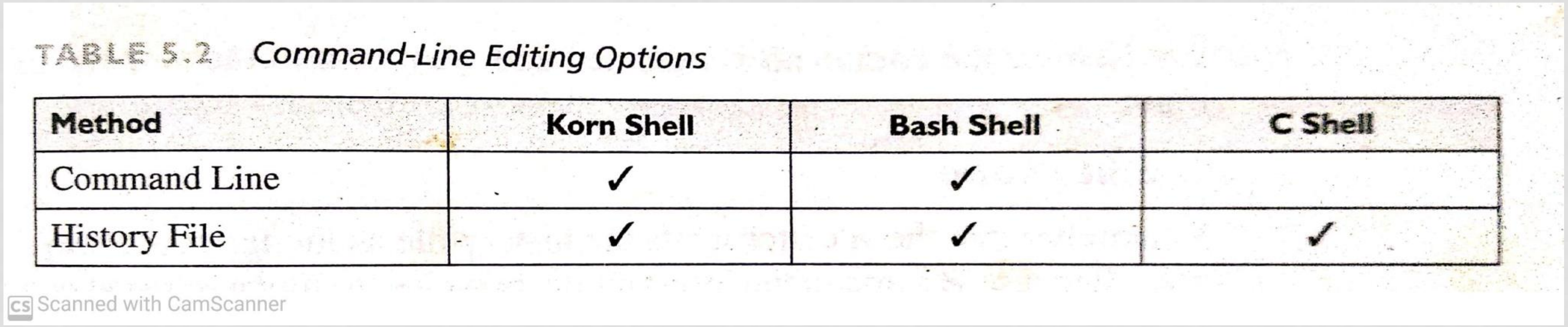


TABLE 5.2 *Command-Line Editing Options*

Method	Korn Shell	Bash Shell	C Shell
Command Line	✓	✓	
History File	✓	✓	✓

- **Command-Line Editing Concept**
- The korn shell copies it to a special file, with command-line editing, we can edit the command using either **vi** or **emacs** without opening the file.

- It’s as though the shell keeps the file in a buffer that provides instant access to our commands.

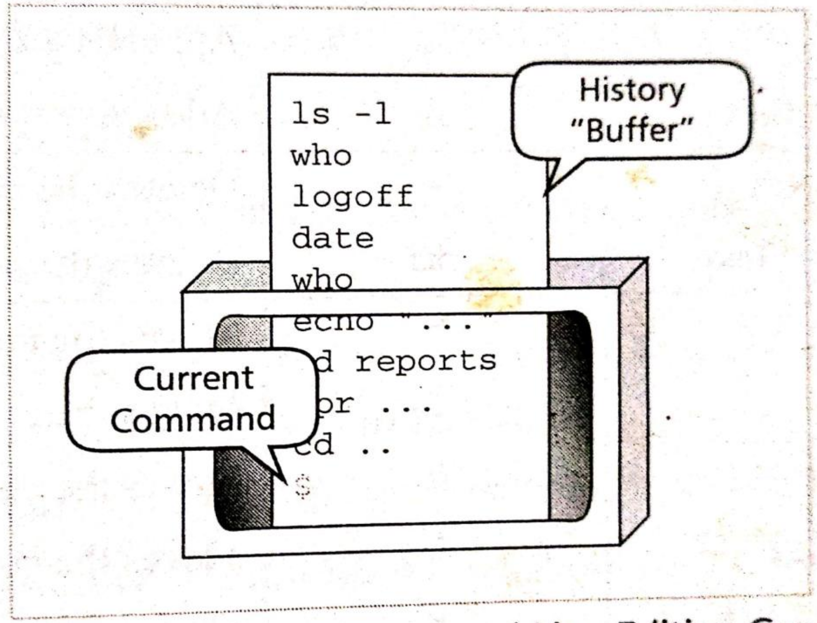


FIGURE 5.10 *Command-Line Editing Concept*

- **Editor Selection**
- The system administrator may set the default command-line editor, most likely in /etc/profile.
- We use the **set** command with the editor.
- $ set –o vi
- $ set –o emacs

- **vi Command-Line Editor**
- The **vi** editor treats the history file as though it is always open and available.

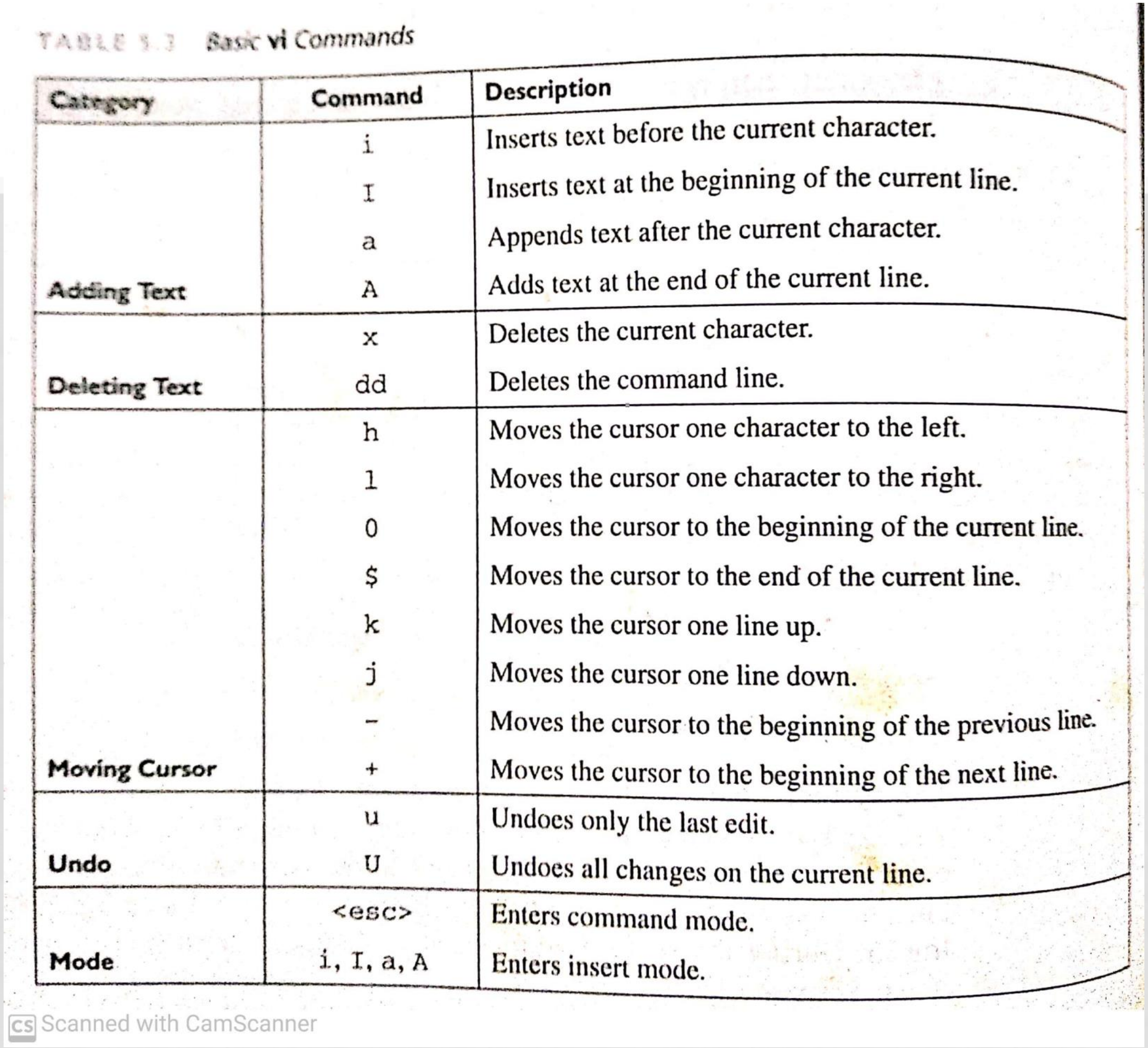


TABLE 5.3 *Basic* **vi** *Commands*

Category	Command	Description
	`i`	Inserts text before the current character.
	`I`	Inserts text at the beginning of the current line.
	`a`	Appends text after the current character.
Adding Text	`A`	Adds text at the end of the current line.
	`x`	Deletes the current character.
Deleting Text	`dd`	Deletes the command line.
	`h`	Moves the cursor one character to the left.
	`l`	Moves the cursor one character to the right.
	`0`	Moves the cursor to the beginning of the current line.
	`$`	Moves the cursor to the end of the current line.
	`k`	Moves the cursor one line up.
	`j`	Moves the cursor one line down.
	`-`	Moves the cursor to the beginning of the previous line.
Moving Cursor	`+`	Moves the cursor to the beginning of the next line.
	`u`	Undoes only the last edit.
Undo	`U`	Undoes all changes on the current line.
	`<esc>`	Enters command mode.
Mode	`i, I, a, A`	Enters insert mode.

- **Move Commands**

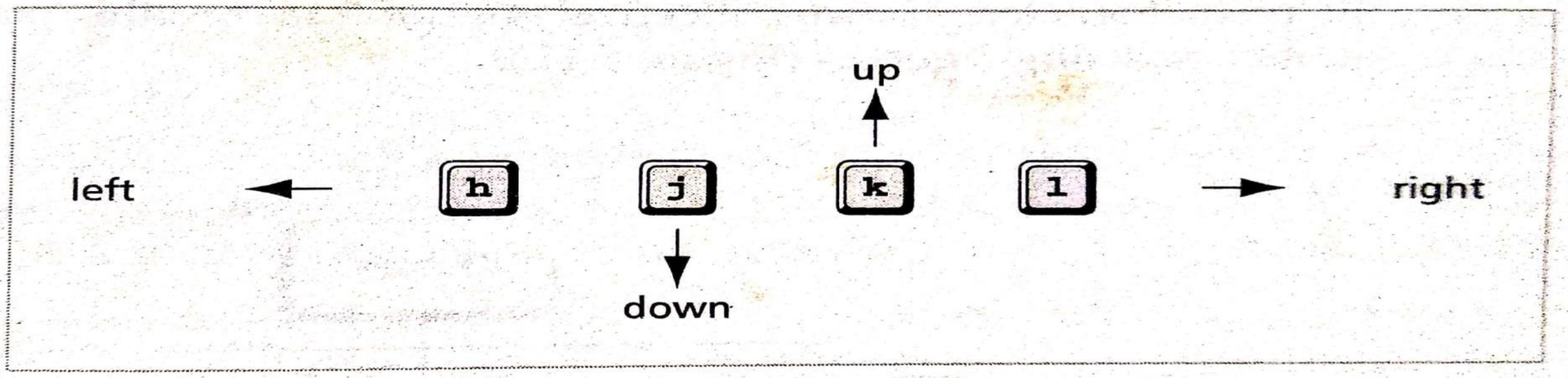


FIGURE 5.11 **vi** *Directional Keys*

QUOTES

- The shells use a selected set of metacharacters in commands.
- **Metacharacters** are characters that have a special interpretation (|).

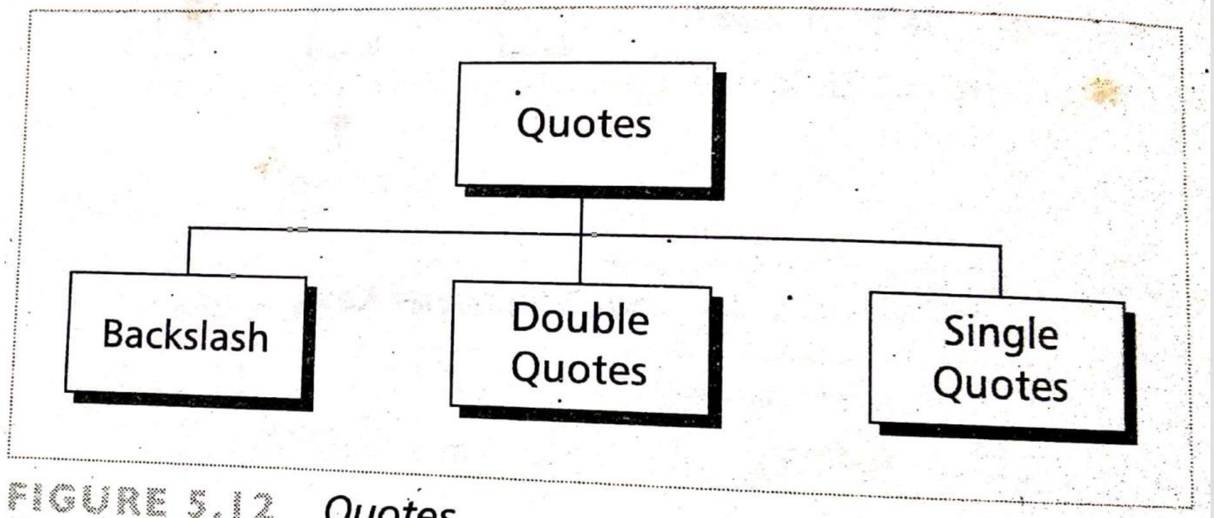


FIGURE 5.12 *Quotes*

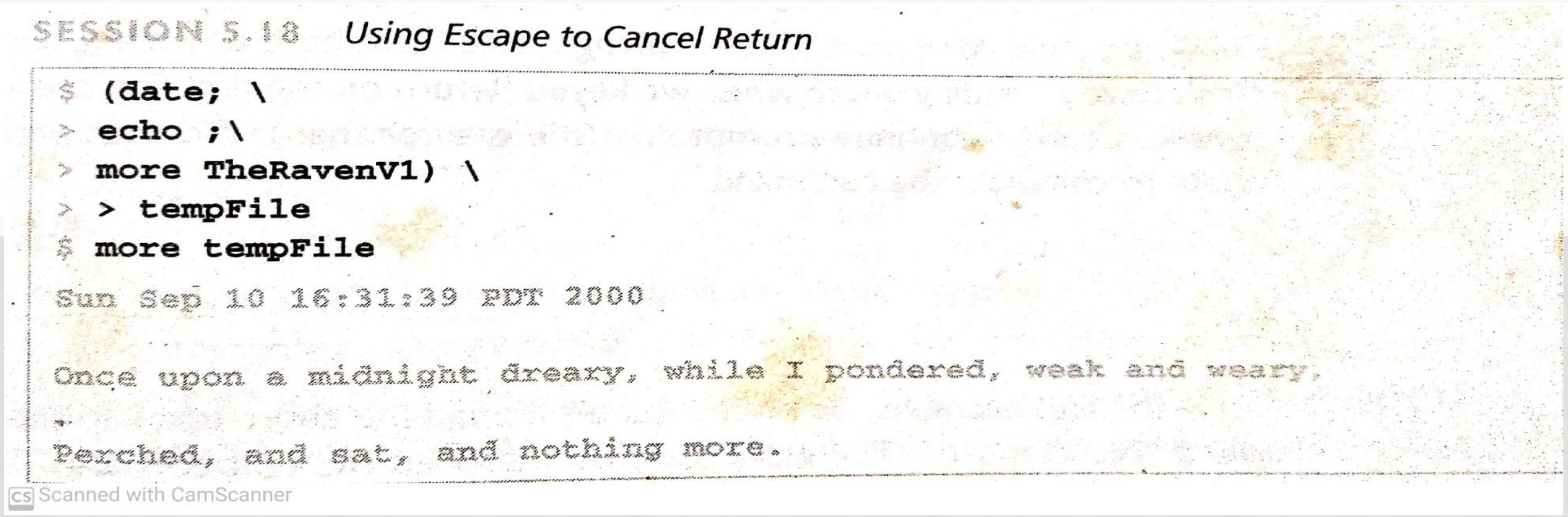
- **Backslash**
- The backslash metacharacter (\) changes the interpretation that follows – it convert literals characters into special characters and special characters into literal characters.
- $ echo < > " ' \ $

Syntax error

- echo \< \> \" \' \\ \$

< > " ' \ $

- The Return key has two effects on a UNIX shell : it is a command separator(end of command) and it is a line separator (end of line).


SESSION 5.18 *Using Escape to Cancel Return*

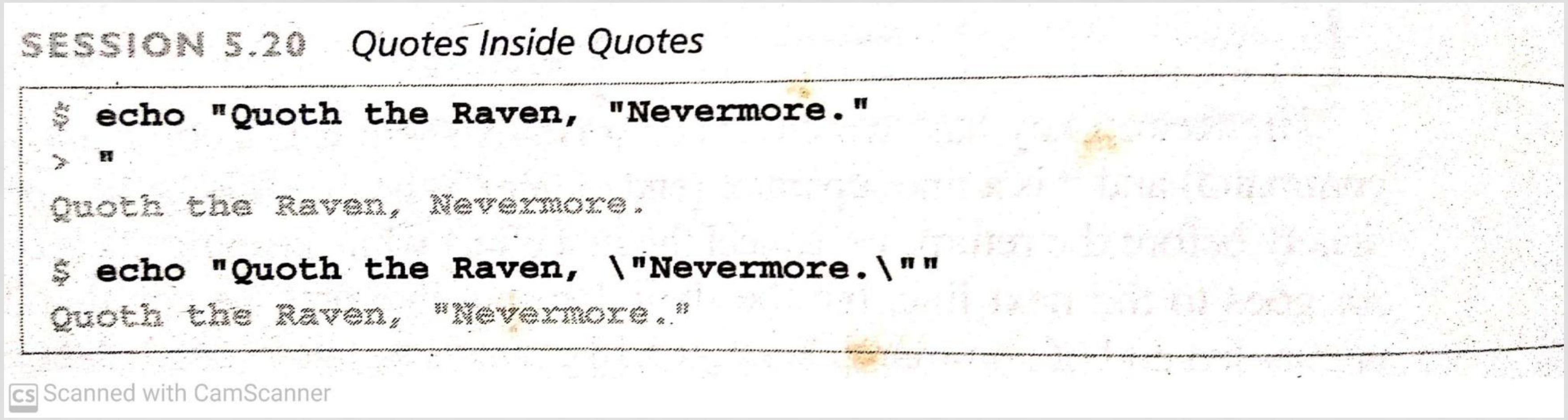
```
$ (date; \
> echo ;\
> more TheRavenV1) \
> > tempFile
$ more tempFile
Sun Sep 10 16:31:39 PDT 2000

Once upon a midnight dreary, while I pondered, weak and weary,
.
Perched, and sat, and nothing more.
```

Scanned with CamScanner

- **<u>Double Quotes</u>**
- When we need to change the meaning of several characters, we can use double quotes.
- Double quotes remove the special interpretation of most metacharacters.
- The exceptions are the dollar sign in front of a variable name, and single quotes.

- $ x=hello
- $ echo “< > $x ‘y’ ? &”

< > hello ‘y’ ? &



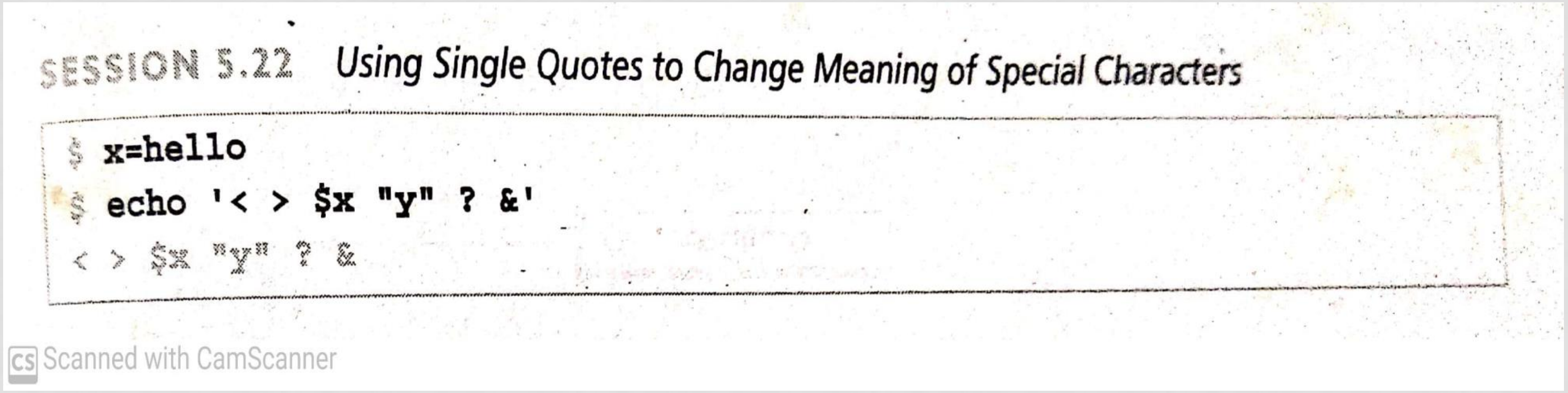
SESSION 5.20 *Quotes Inside Quotes*

```
$ echo "Quoth the Raven, "Nevermore."
> "
Quoth the Raven, Nevermore.
$ echo "Quoth the Raven, \"Nevermore.\""
Quoth the Raven, "Nevermore."
```

Scanned with CamScanner

- Double quotes also preserve whitespace characters in the text. Whitespace characters are the tab, newline, and the blank or space character.

- **Single quotes**
- Single quotes operate like double quotes, but their effect is stronger. They preserve only the meaning of single quotes.



SESSION 5.22 *Using Single Quotes to Change Meaning of Special Characters*

```
$ x=hello
$ echo '< > $x "y" ? &'
< > $x "y" ? &
```

- Also the fact that double quotes lose their special character properties when placed inside single quotes.

COMMAND SUBSTITUTION

- When a shell executes a command, the output is directed to standard output.

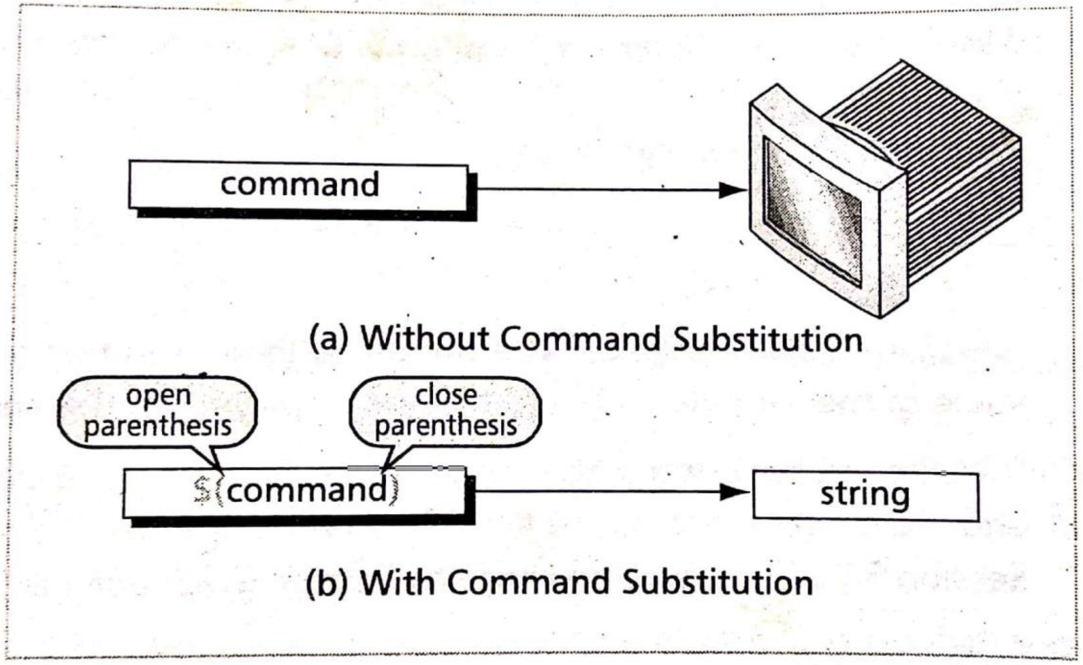

command
(a) Without Command Substitution
open parenthesis
close parenthesis
$(command)
string
(b) With Command Substitution

FIGURE 5.13 *Command Substitution*