Using Scripts to Automate Tasks

Set	Explanation
[b–f]	Denotes any characters in the set b , c , d , e , f
[1–4,7–9]	Denotes any two numbers, the first of which must be 1 , 2 , 3 , or 4 and the second of which must be 7 , 8 , or 9
[aeiou]	Denotes any character that is a vowel
[b,aeiou]	Denotes any character that is either b or a vowel
[aeiou][a–z]	Denotes any vowel followed by any lowercase character between and including a and z

eg :

```
$ ls [b-c]*
basename bash2
bash
bsh
cat
chgrp
chmod
chown
consolechars
cp
cpio
csh
cut
```

The resulting set consists of any file name beginning with b or c.

```
$ ls [b-d,f]?
cp dd df
```

BASH SPECIAL CHARACTERS

Character	Description
*	Multicharacter wildcard
?	Single-character wildcard
<	Redirect input
>	Redirect output
>>	Append output
|	Pipe
{	Start command block
}	End command block
(	Start subshell
)	End subshell

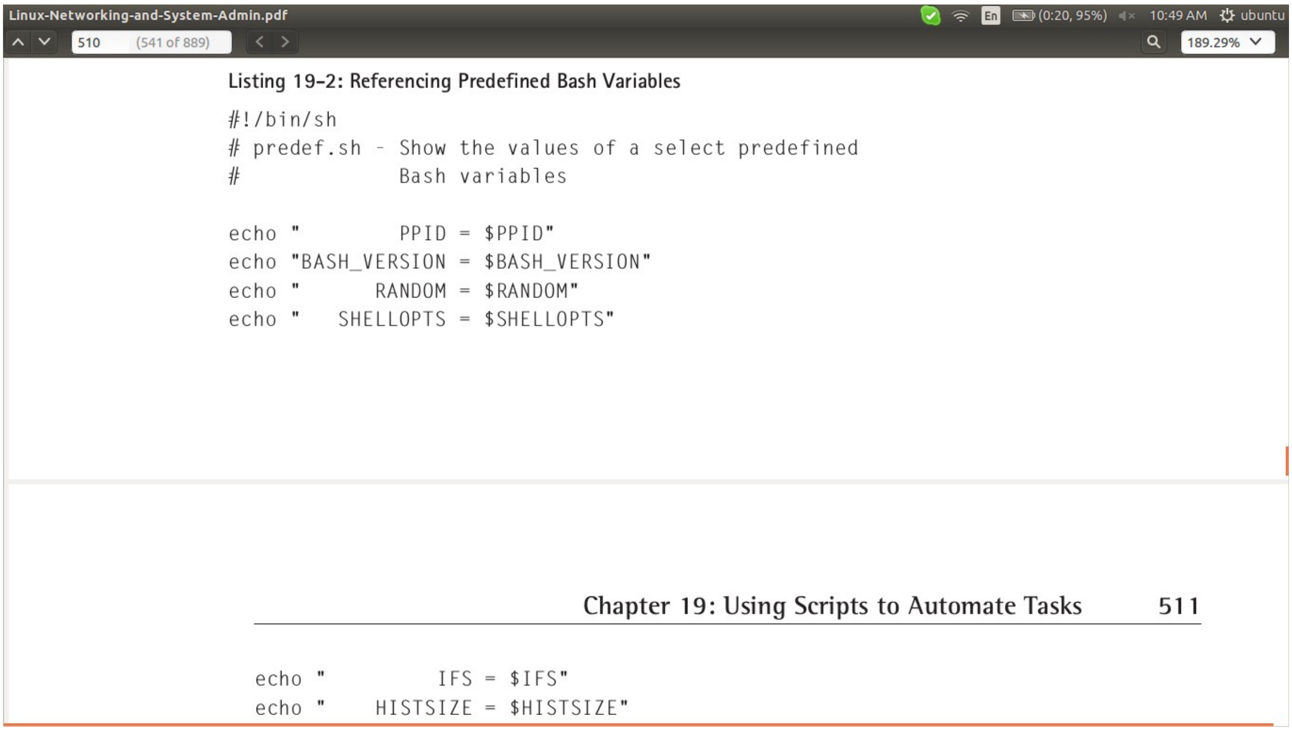
`	Command substitution
$	Variable expression
‘	Strong quote
“	Weak quote
\	Interpret the next character literally
&	Execute command in background
;	Command separator
~	Home directory
#	Comment

You should already be familiar with redirecting input and output and using pipes, but you will see examples of all three operations later in the chapter. Input and output redirection should be familiar to you. Commands in a block, that is, delimited by { and } elicit different behavior from Bash than commands executed in a subshell, that is, commands delimited by (and) .

PREDEFINED VARIABLES

Variable Name	Description
PPID	Stores the process ID of the Bash’s parent process.
PWD	Contains the current working directory (as set by the cd command).
OLDPWD	Stores the most previous working directory (as set by the next most recent cd command).
UID	Contains the user ID of the current user as set when the shell started.
BASH_VERSION	Stores the version string of the current Bash instance.
RANDOM	Returns a random number between 0 and 32,767 each time it is referenced.
OPTARG	Contains the last command line argument read using the getopts built-in command.
OPTIND	Points to the array index of the next argument to be processed using the getopts built-in command.
HOSTNAME	Stores the name of the current host system.
SHELLOPTS	Contains a colon-separated string of the currently enabled shell options (as set using the set -o shell built-in command).
IFS	Stores the value of the Internal Field Separator, used to define word boundaries. The default value is a space, a tab, or a new line.
HOME	The home directory of the current user.
PS1	Defines the primary command prompt. The default value is “\s-\v\$ “ .
PS2	Defines the secondary command prompt, used to indicate that additional input is required to complete a command.
HISTSIZE	The number of commands retained in Bash’s command history. The default value is 500.
HISTFILE	The name of the file in which Bash stores the command history. The default value is $HOME/.bash_history .

Eg :



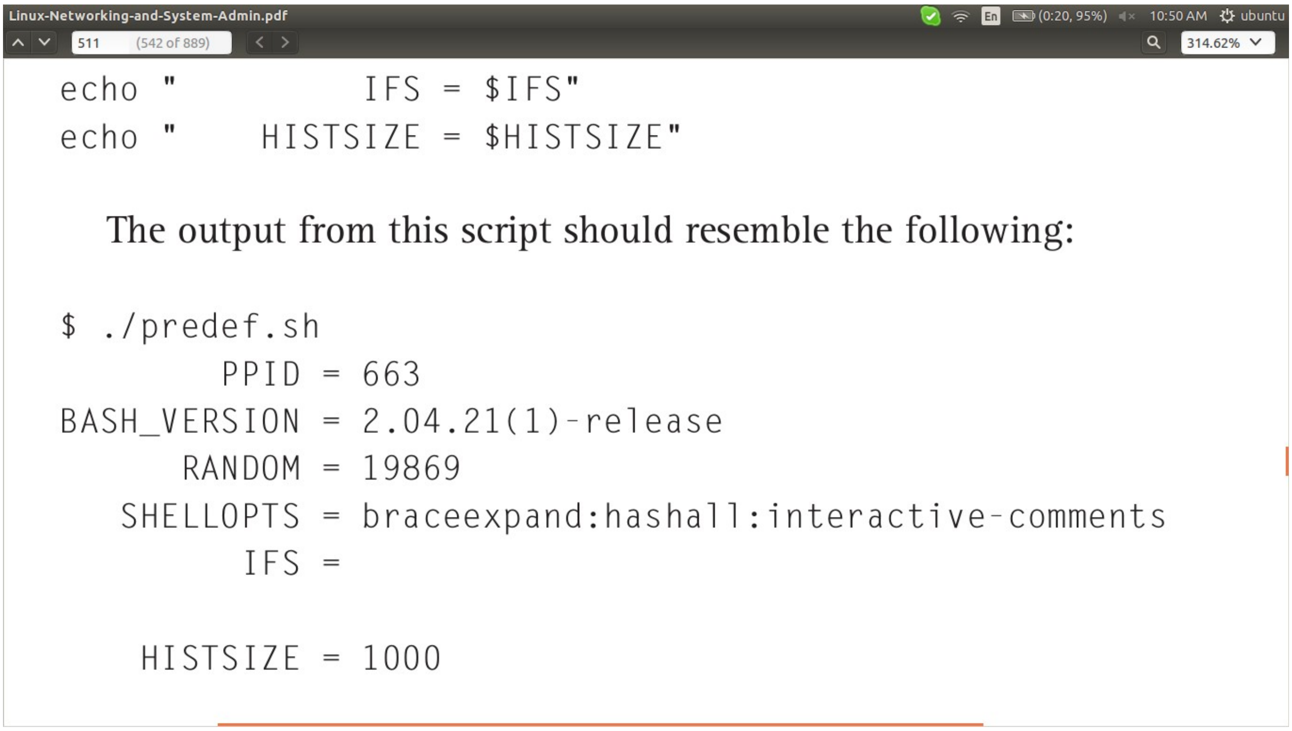
Listing 19-2: Referencing Predefined Bash Variables

```
#!/bin/sh
# predef.sh - Show the values of a select predefined
#             Bash variables

echo "         PPID = $PPID"
echo "BASH_VERSION = $BASH_VERSION"
echo "       RANDOM = $RANDOM"
echo "    SHELLOPTS = $SHELLOPTS"
```

Chapter 19: Using Scripts to Automate Tasks 511

```
echo "          IFS = $IFS"
echo "     HISTSIZE = $HISTSIZE"
```

output :



```
echo "          IFS = $IFS"
echo "     HISTSIZE = $HISTSIZE"
```

The output from this script should resemble the following:

```
$ ./predef.sh
        PPID = 663
BASH_VERSION = 2.04.21(1)-release
      RANDOM = 19869
   SHELLOPTS = braceexpand:hashall:interactive-comments
         IFS =

    HISTSIZE = 1000
```

BASH STRING COMPARISON OPERATORS

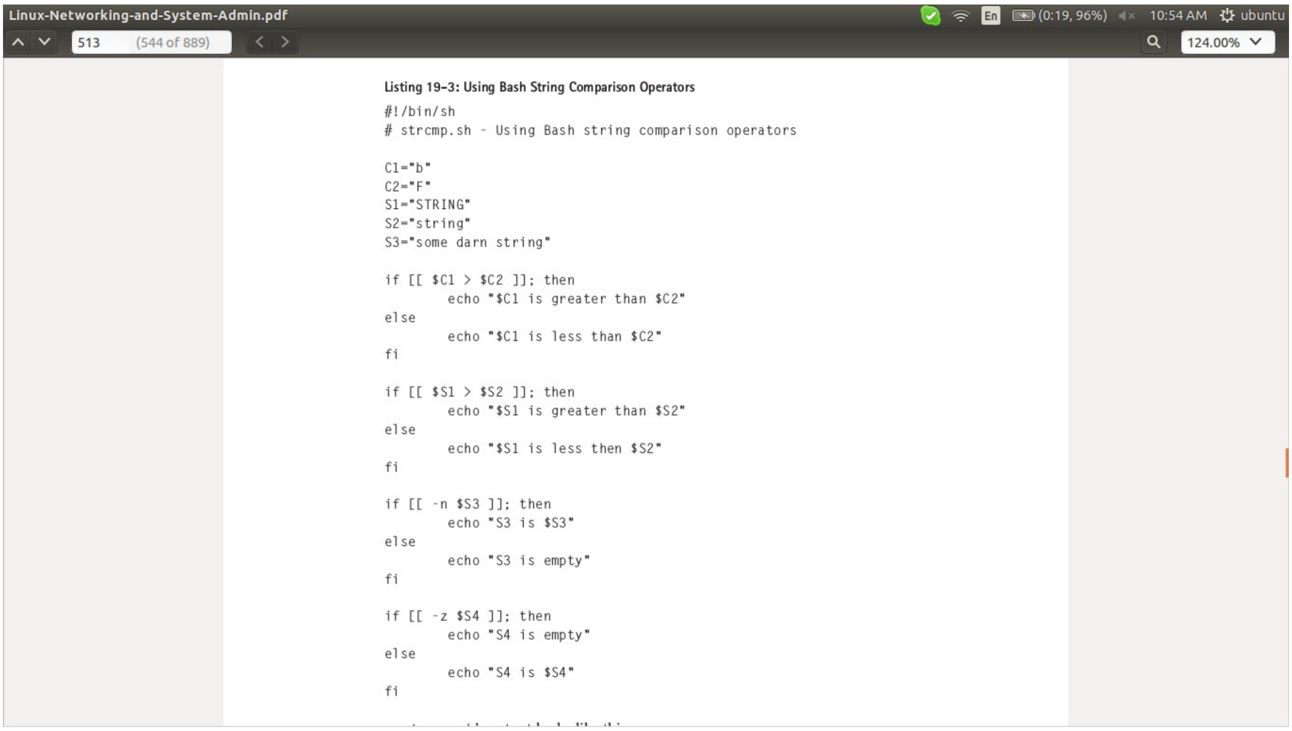
Operator	Expression	True if...
=	str1 = str2	str1 matches str2
==	str1 == str2	str1 matches str2
!=	str1 != str2	str1 does not match str2
<	str1 < str2	str1 is less than str2
>	str1 > str2	str1 is greater than str2
-n	-n str	str ’s length is nonzero
-z	-z str	str ’s length is zero

BASH NUMERIC COMPARISON OPERATORS

Operator	Expression	True if...
-eq	val1 -eq val2	val1 equals val2
-ne	val1 -ne val2	val1 is not equal val2
-ge	val1 -ge val2	val1 is greater than or equal to val2
-gt	val1 -gt val2	val1 is greater than val2
-le	val1 -le val2	val1 is less than or equal to val2
-lt	val1 -lt val2	val1 is less than val2

When comparing strings, keep in mind that A–Z come before a–z in the ASCII character set, so A is “less than” a and foo is “greater than” bar

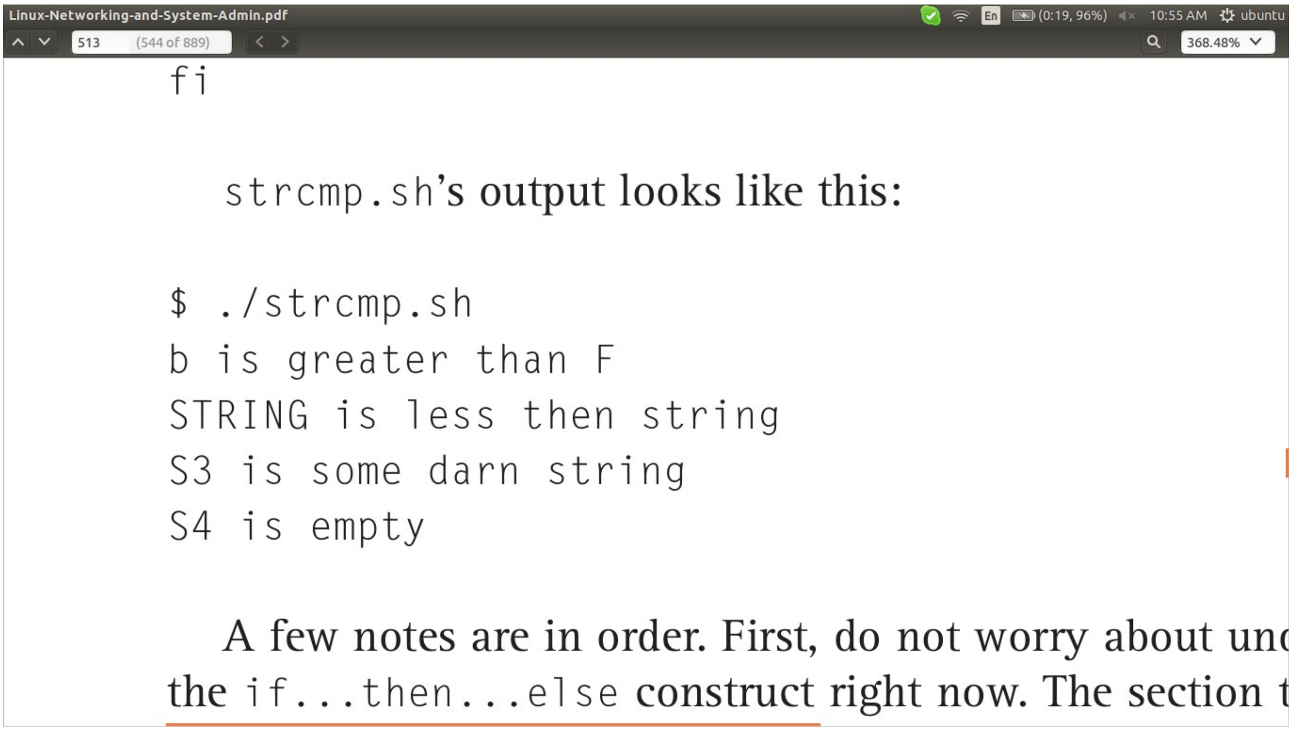
eg : Using Bash String Comparison Operators



Listing 19-3: Using Bash String Comparison Operators

```
#!/bin/sh
# strcmp.sh - Using Bash string comparison operators

C1="b"
C2="F"
S1="STRING"
S2="string"
S3="some darn string"

if [[ $C1 > $C2 ]]; then
        echo "$C1 is greater than $C2"
else
        echo "$C1 is less than $C2"
fi

if [[ $S1 > $S2 ]]; then
        echo "$S1 is greater than $S2"
else
        echo "$S1 is less then $S2"
fi

if [[ -n $S3 ]]; then
        echo "S3 is $S3"
else
        echo "S3 is empty"
fi

if [[ -z $S4 ]]; then
        echo "S4 is empty"
else
        echo "S4 is $S4"
fi
```

output :

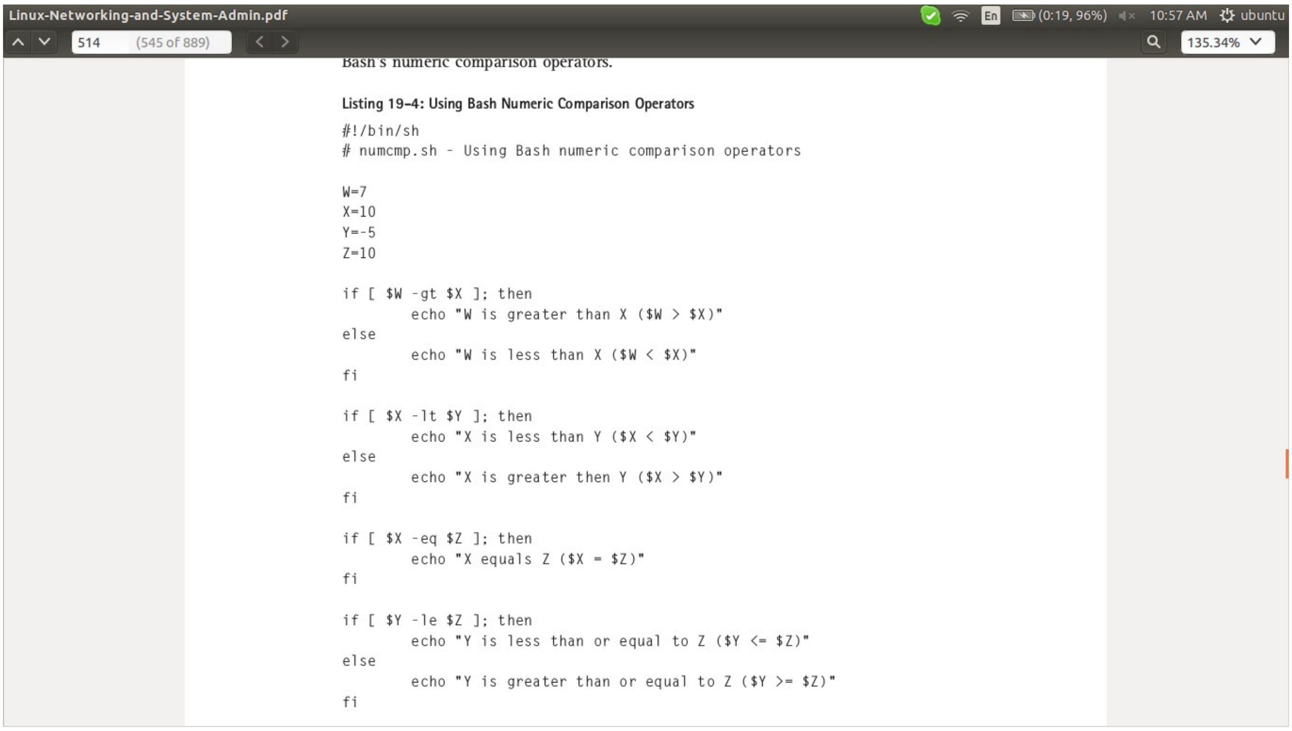


```
fi
```

strcmp.sh's output looks like this:

```
$ ./strcmp.sh
b is greater than F
STRING is less then string
S3 is some darn string
S4 is empty
```

A few notes are in order. First, do not worry about unc the if...then...else construct right now. The section t eg : Using Bash Numeric Comparison Operators

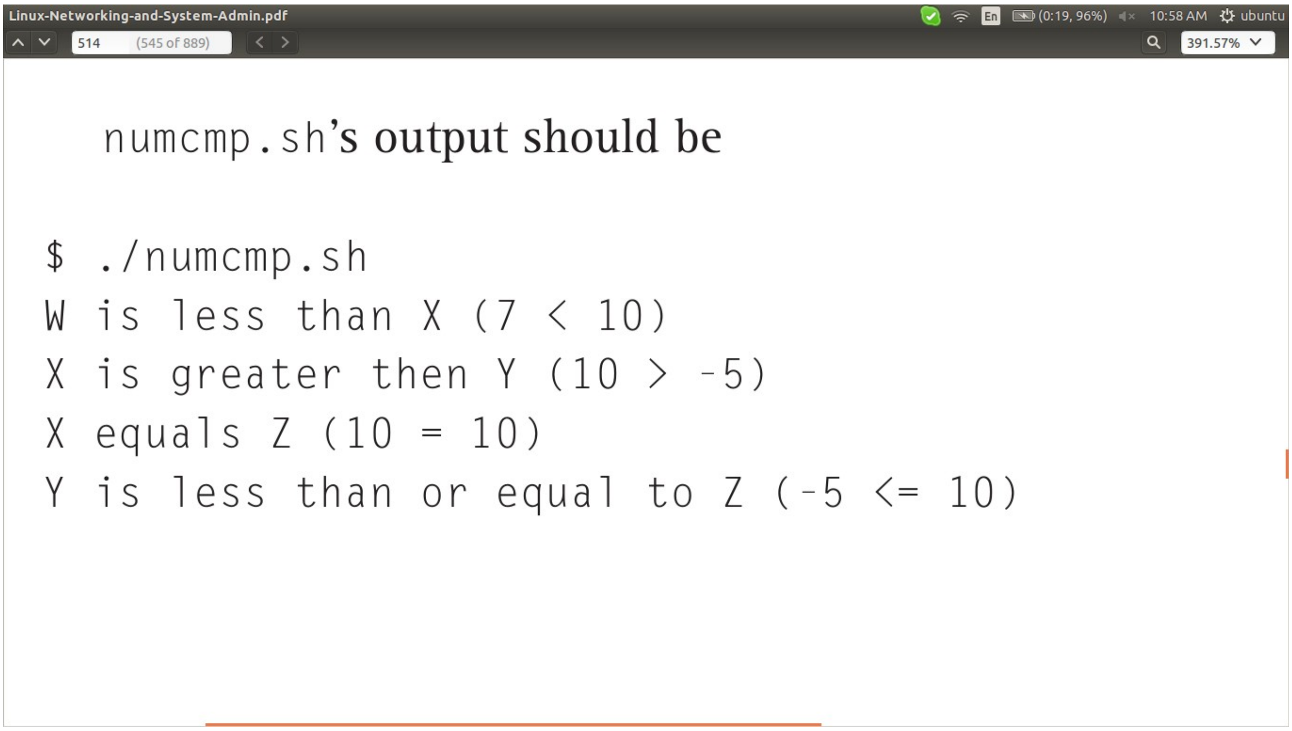


Bash's numeric comparison operators.

Listing 19-4: Using Bash Numeric Comparison Operators

```
#!/bin/sh
# numcmp.sh - Using Bash numeric comparison operators

W=7
X=10
Y=-5
Z=10

if [ $W -gt $X ]; then
        echo "W is greater than X ($W > $X)"
else
        echo "W is less than X ($W < $X)"
fi

if [ $X -lt $Y ]; then
        echo "X is less than Y ($X < $Y)"
else
        echo "X is greater then Y ($X > $Y)"
fi

if [ $X -eq $Z ]; then
        echo "X equals Z ($X = $Z)"
fi

if [ $Y -le $Z ]; then
        echo "Y is less than or equal to Z ($Y <= $Z)"
else
        echo "Y is greater than or equal to Z ($Y >= $Z)"
fi
```

output :



numcmp.sh's output should be

```
$ ./numcmp.sh
W is less than X (7 < 10)
X is greater then Y (10 > -5)
X equals Z (10 = 10)
Y is less than or equal to Z (-5 <= 10)
```

ARITHMETIC OPERATORS

If Bash allows you to compare numeric values and variables, it follows that Bash allows you to perform arithmetic on numeric values and variables. In addition to the comparison operators discussed in the previous section, Bash uses the operators listed in Table 19-6 to perform arithmetic operations in shell scripts.

ARITHMETIC OPERATORS

Operator	Expression	Description
++	var++	Increments var after obtaining its value
++	++var	Increments var before obtaining its value
--	var--	Decrements var after obtaining its value
--	--var	Decrements var before obtaining its value
+	var1 + var2	Adds var1 and var2
-	var1 - var2	Subtracts var2 from var1
*	var1 * var2	Multiplies var1 by var2
/	var1 / var2	Divides var1 by var2
%	var1 % var2	Calculates the remainder of dividing var1 by var2
=	var = op	Assigns result of an arithmetic operation op to var

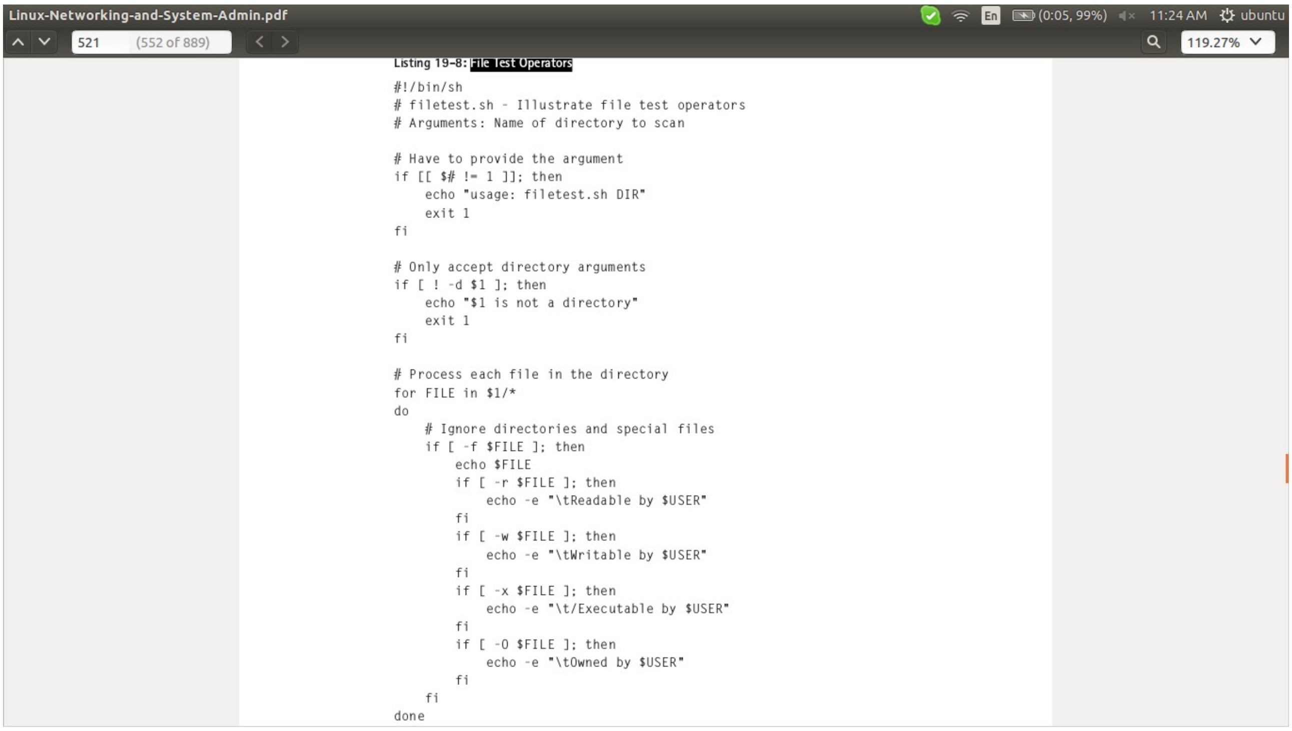
FILE TEST OPERATORS

Many common administrative tasks involve working with files and directories. Bash makes it easy to perform these tasks because it has a rich set of operators that perform a variety of tests. You can, for example, determine if a file exists at all or if it exists but is empty, if a file is a directory, and what the file’s permissions are. Table 19-9 lists commonly used file test operators.

BASH FILE TEST OPERATORS

Operator	Description
-d	file file exists and is a directory.
-e	file file exists.
-f	file file exists and is a regular file (not a directory or special file).
-g	file file exists and is SGID (set group ID).
-r	file You have read permission on file.
-s	file file exists and is not empty.
-u	file file exists and is SUID (set user ID).
-w	file You have write permission on file.
-x	file You have execute permission on file or, if file is a directory, you have search permission on it.
-O	file You own file.
-G	file file ’s group ownership matches one of your group memberships.
file1 -nt file2	file1 is newer than file2.
file1 -ot file2	file1 is older than file2 .

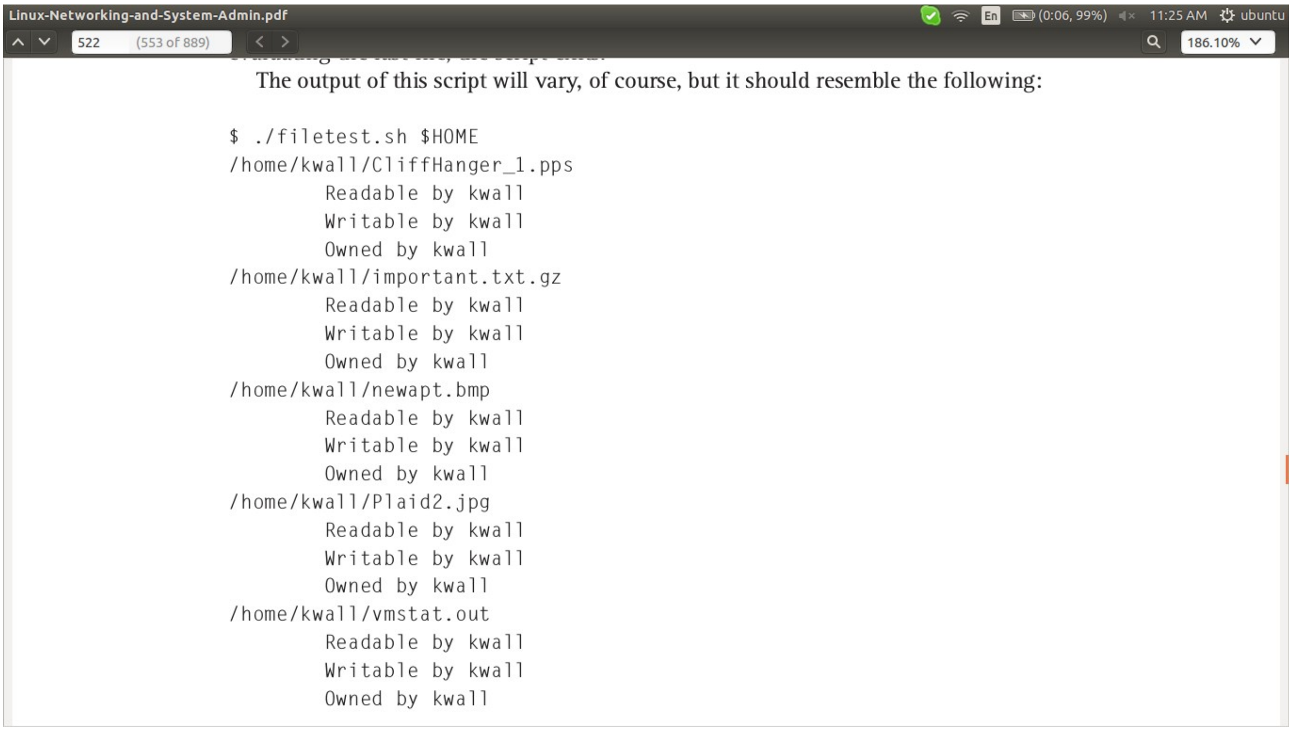
Eg : File Test Operators



Listing 19-8: File Test Operators

```
#!/bin/sh
# filetest.sh - Illustrate file test operators
# Arguments: Name of directory to scan

# Have to provide the argument
if [[ $# != 1 ]]; then
    echo "usage: filetest.sh DIR"
    exit 1
fi

# Only accept directory arguments
if [ ! -d $1 ]; then
    echo "$1 is not a directory"
    exit 1
fi

# Process each file in the directory
for FILE in $1/*
do
    # Ignore directories and special files
    if [ -f $FILE ]; then
        echo $FILE
        if [ -r $FILE ]; then
            echo -e "\tReadable by $USER"
        fi
        if [ -w $FILE ]; then
            echo -e "\tWritable by $USER"
        fi
        if [ -x $FILE ]; then
            echo -e "\t/Executable by $USER"
        fi
        if [ -O $FILE ]; then
            echo -e "\tOwned by $USER"
        fi
    fi
done
```

output :



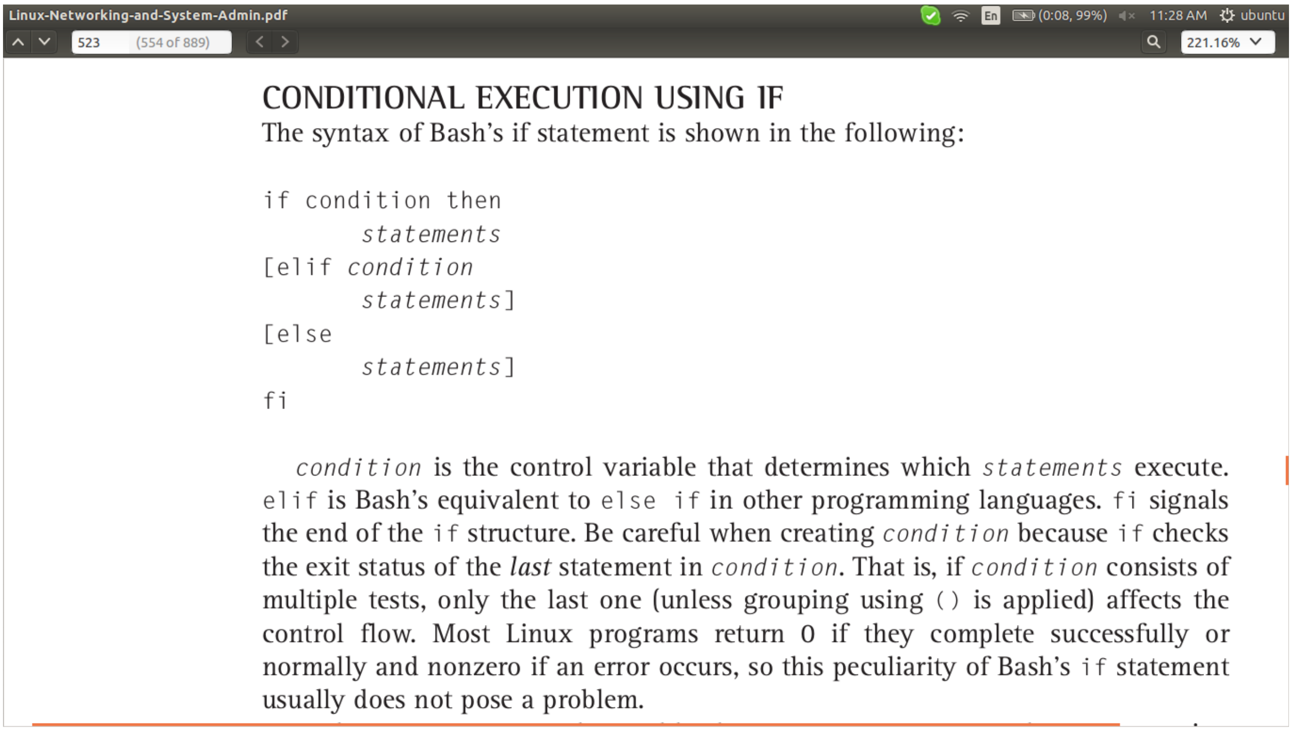
The output of this script will vary, of course, but it should resemble the following:

```
$ ./filetest.sh $HOME
/home/kwall/CliffHanger_1.pps
        Readable by kwall
        Writable by kwall
        Owned by kwall
/home/kwall/important.txt.gz
        Readable by kwall
        Writable by kwall
        Owned by kwall
/home/kwall/newapt.bmp
        Readable by kwall
        Writable by kwall
        Owned by kwall
/home/kwall/Plaid2.jpg
        Readable by kwall
        Writable by kwall
        Owned by kwall
/home/kwall/vmstat.out
        Readable by kwall
        Writable by kwall
        Owned by kwall
```

BASH FLOW CONTROL STATEMENTS

Statement	Type	Description
if	Conditional	Executes code if a condition is true or false
for	Looping	Executes code a fixed number of times
while	Looping	Executes code while a condition is true or false
until	Looping	Executes code until a condition becomes true or false
case	Branching	Executes code specific to the value of a variable
select	Branching	Executes code specific to an option selected by a user

CONDITIONAL EXECUTION USING IF



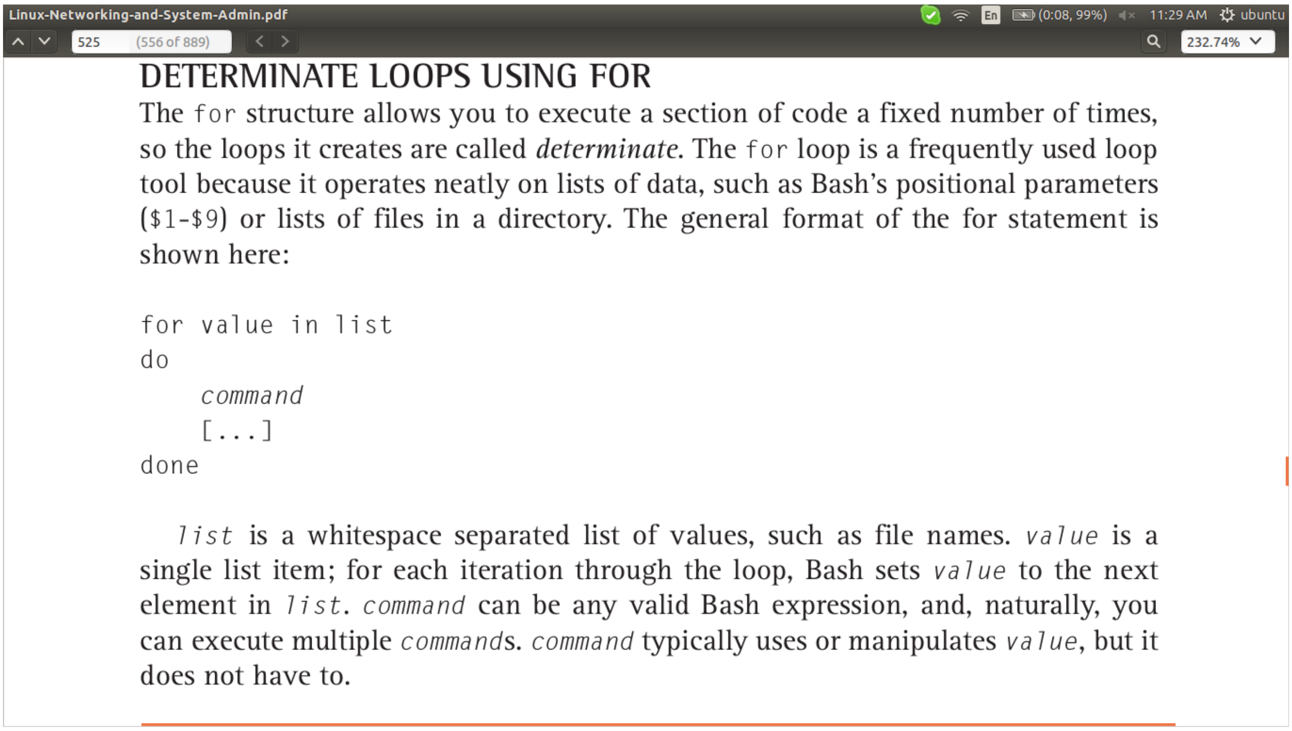
CONDITIONAL EXECUTION USING IF

The syntax of Bash's if statement is shown in the following:

```
if condition then
        statements
[elif condition
        statements]
[else
        statements]
fi
```

`condition` is the control variable that determines which *`statements`* execute. `elif` is Bash's equivalent to `else if` in other programming languages. `fi` signals the end of the `if` structure. Be careful when creating *`condition`* because `if` checks the exit status of the *last* statement in *`condition`*. That is, if *`condition`* consists of multiple tests, only the last one (unless grouping using `()` is applied) affects the control flow. Most Linux programs return 0 if they complete successfully or normally and nonzero if an error occurs, so this peculiarity of Bash's `if` statement usually does not pose a problem.

DETERMINATE LOOPS USING FOR



DETERMINATE LOOPS USING FOR

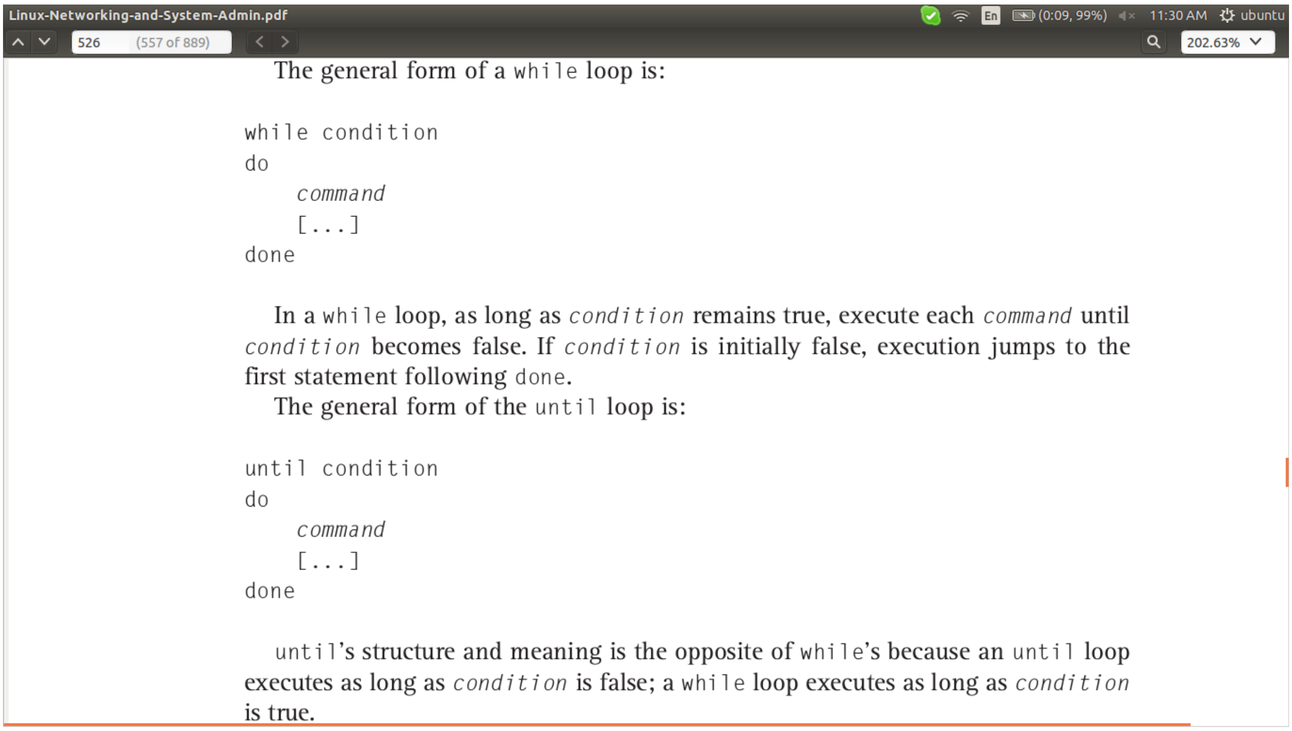
The `for` structure allows you to execute a section of code a fixed number of times, so the loops it creates are called *determinate*. The `for` loop is a frequently used loop tool because it operates neatly on lists of data, such as Bash's positional parameters (`$1-$9`) or lists of files in a directory. The general format of the for statement is shown here:

```
for value in list
do
    command
    [...]
done
```

`list` is a whitespace separated list of values, such as file names. *`value`* is a single list item; for each iteration through the loop, Bash sets *`value`* to the next element in *`list`*. *`command`* can be any valid Bash expression, and, naturally, you can execute multiple *`command`*s. *`command`* typically uses or manipulates *`value`*, but it does not have to.

INDETERMINATE LOOPS USING WHILE AND UNTIL

The for loop is ideal when you know in advance, or can determine before entering the loop, how many times a given section of code must execute. If you do not know in advance or cannot determine at runtime how many times to execute a code block, you need to use one of Bash's indeterminate loop structures, while and until . while and until constructs cause continued code execution as long as or until a particular condition is met. The key here is that your code must ensure that the condition in question is eventually met, or you will get stuck in an infinite loop.



The general form of a `while` loop is:

```
while condition
do
    command
    [...]
done
```

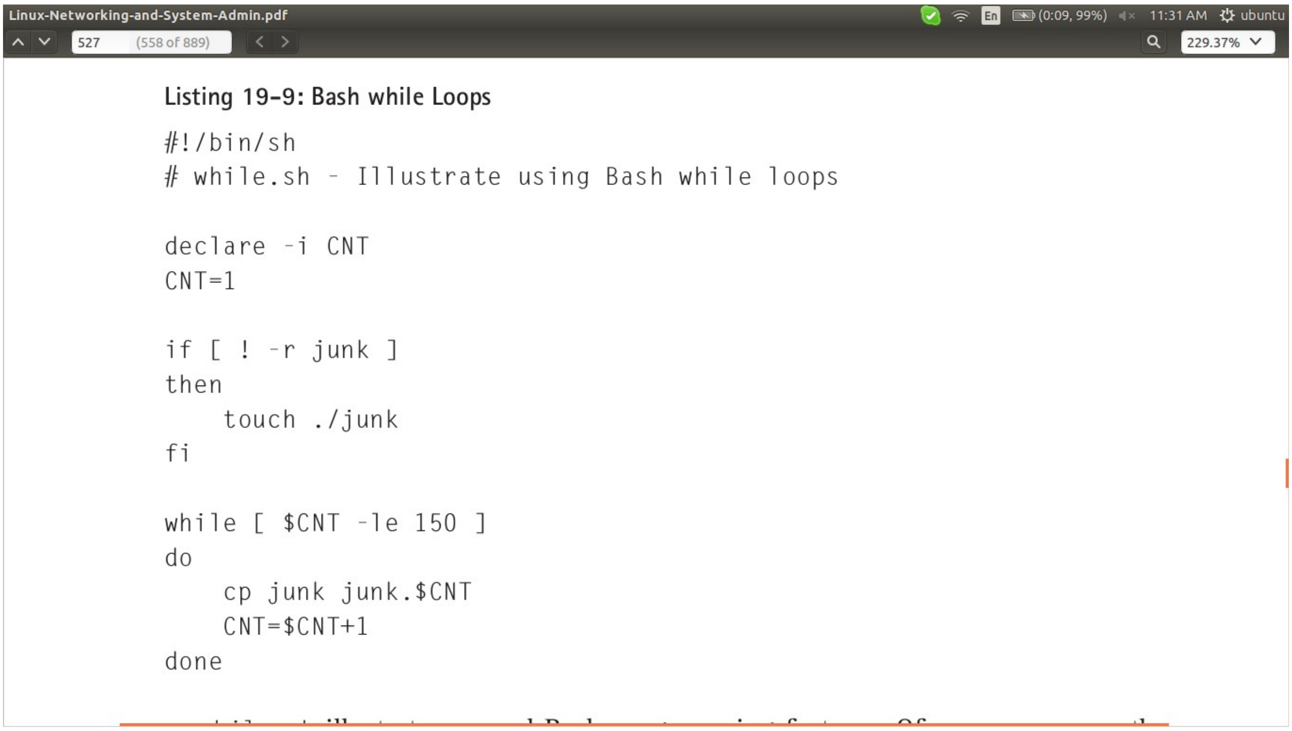
In a `while` loop, as long as *`condition`* remains true, execute each *`command`* until *`condition`* becomes false. If *`condition`* is initially false, execution jumps to the first statement following `done`.

The general form of the `until` loop is:

```
until condition
do
    command
    [...]
done
```

`until`'s structure and meaning is the opposite of `while`'s because an `until` loop executes as long as *`condition`* is false; a `while` loop executes as long as *`condition`* is true.

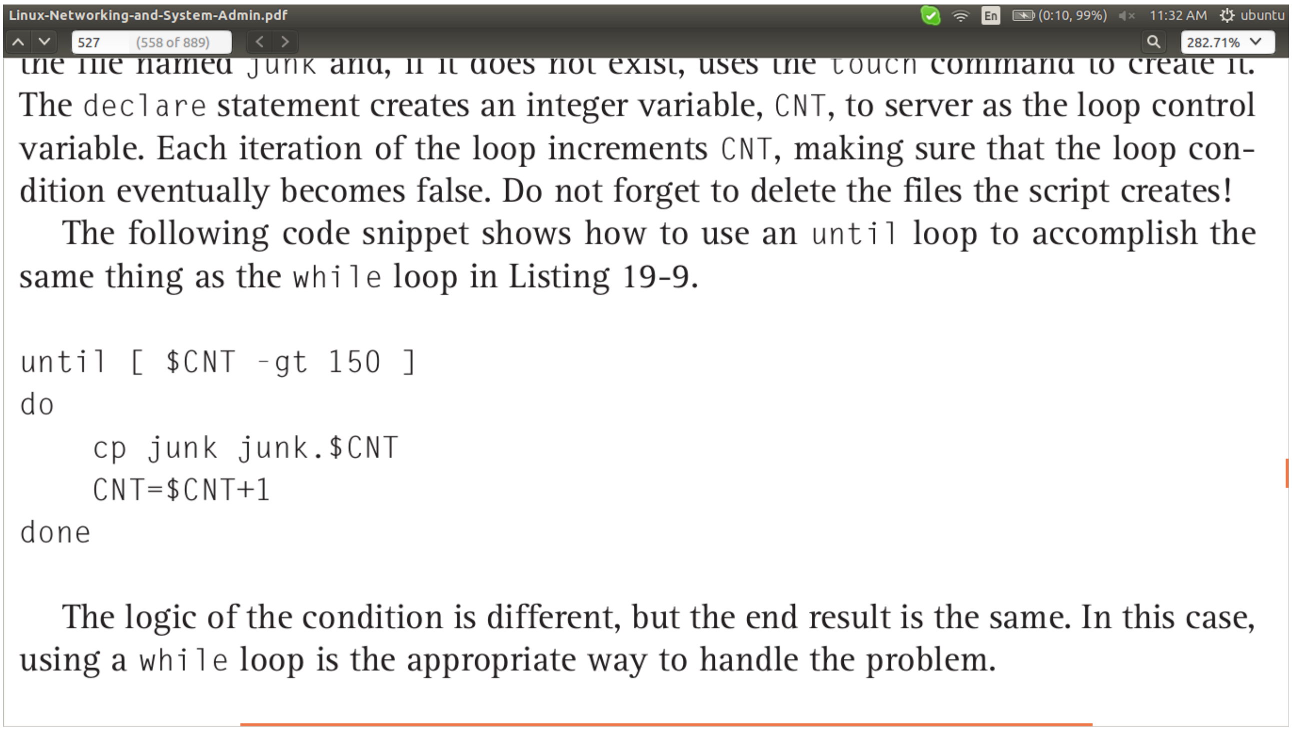
Eg : Bash while Loops



Listing 19-9: Bash while Loops

```
#!/bin/sh
# while.sh - Illustrate using Bash while loops

declare -i CNT
CNT=1

if [ ! -r junk ]
then
    touch ./junk
fi

while [ $CNT -le 150 ]
do
    cp junk junk.$CNT
    CNT=$CNT+1
done
```

Eg : Bash until Loops



the file named junk and, if it does not exist, uses the touch command to create it. The declare statement creates an integer variable, CNT, to server as the loop control variable. Each iteration of the loop increments CNT, making sure that the loop condition eventually becomes false. Do not forget to delete the files the script creates!

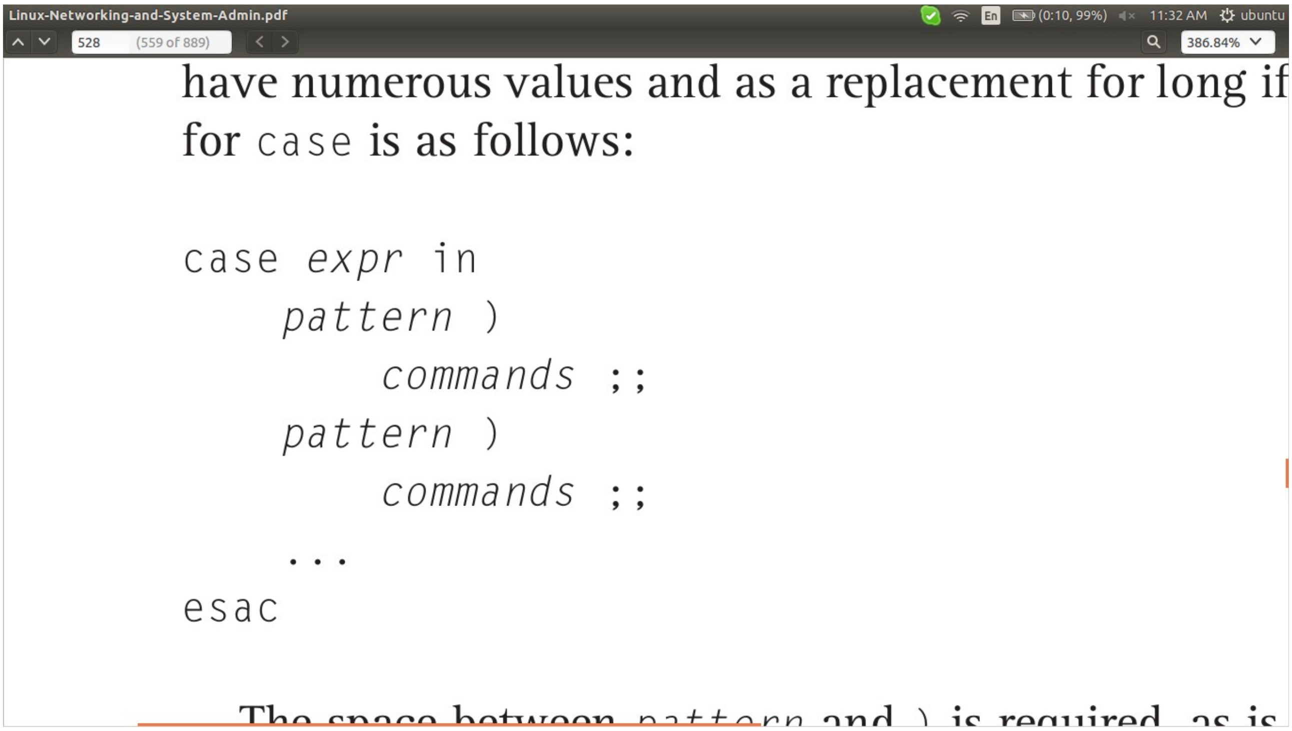
The following code snippet shows how to use an until loop to accomplish the same thing as the while loop in Listing 19-9.

```
until [ $CNT -gt 150 ]
do
    cp junk junk.$CNT
    CNT=$CNT+1
done
```

The logic of the condition is different, but the end result is the same. In this case, using a while loop is the appropriate way to handle the problem.

THE CASE STATEMENT

The case structure, approximately comparable to C’s
switch keyword, is best suited for situations in which a variable or expression can
have numerous values and as a replacement for long if blocks. The complete syntax
for case is as follows:

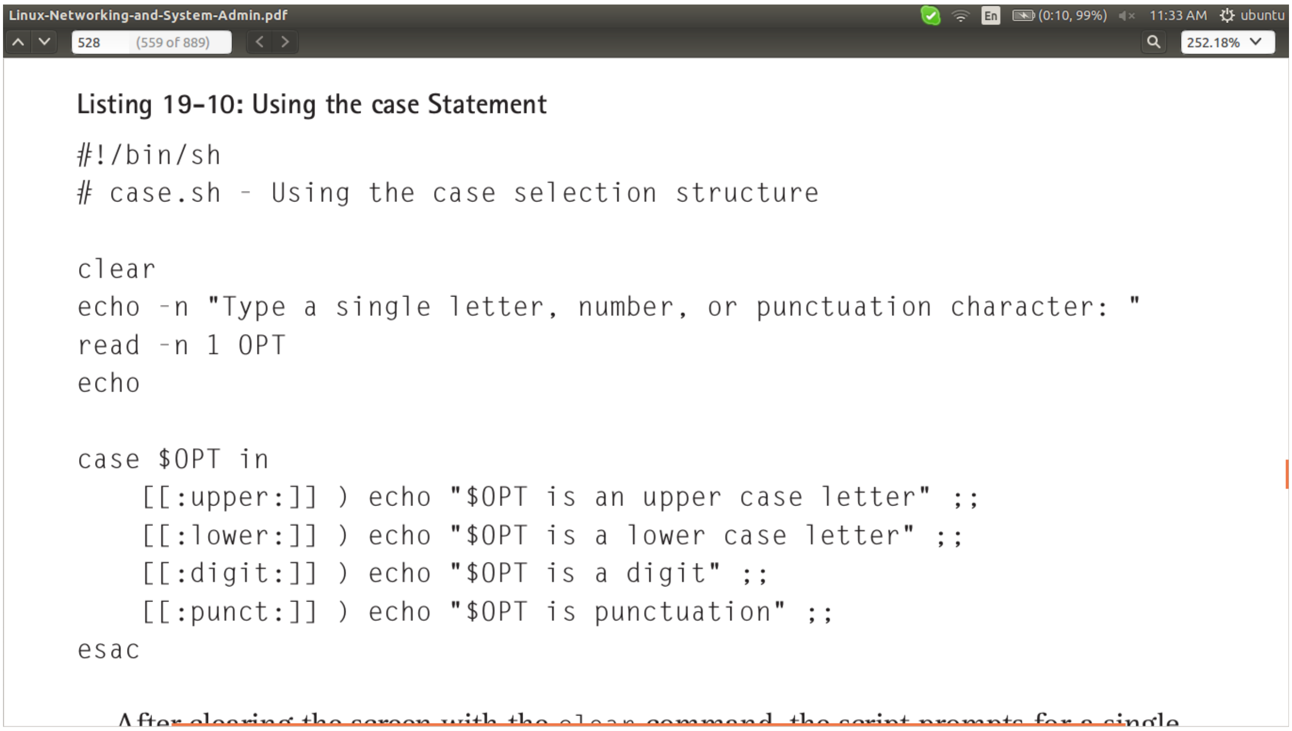


have numerous values and as a replacement for long if
for case is as follows:

```
case expr in
    pattern )
        commands ;;
    pattern )
        commands ;;
    ...
esac
```

The space between pattern and) is required, as is

eg : Using the case Statement



Listing 19-10: Using the case Statement

```
#!/bin/sh
# case.sh - Using the case selection structure

clear
echo -n "Type a single letter, number, or punctuation character: "
read -n 1 OPT
echo

case $OPT in
    [[:upper:]] ) echo "$OPT is an upper case letter" ;;
    [[:lower:]] ) echo "$OPT is a lower case letter" ;;
    [[:digit:]] ) echo "$OPT is a digit" ;;
    [[:punct:]] ) echo "$OPT is punctuation" ;;
esac
```

After clearing the screen with the clear command, the script prompts for a single