

4.2.9 test and []: Companions of If

When you utilize **if** to evaluate expressions, the **test** statement is often used as its control command.

test uses certain operators to evaluate the condition on its right and returns either true or false exit status, which is then used by **if** for making decisions.

test works in three ways:

- Compares two numbers
- Compares two strings or a single one for a null value.
- Checks a file's attributes

These tests can be made by **test** in association with the shell's other statements, but for present we will stick with **if.test** doesn't display any output but simply returns a value that sets the parameter \$?.

In the following sections we will actually check this value

4.2.9.1 Numeric Comparison

The numerical comparison operators (Table 2.8) used by **test** have a form different from what you would have seen anywhere.

They always begin with a - (hyphen), followed by a two-character word and enclosed on either side by whitespace. Here's a typical operator:

-ne

Table 18.2 Numerical Comparison Operators used by **test**

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal to

The operators are quite mnemonic; `-eq` implies equal to, `-gt` implies greater than, and so on.

Numeric comparison in the shell is confined to integer values only; decimal values are simply truncated.

We will assign some values to three variables and check whether they are equal or not.

The last test proves conclusively that numeric comparison is restricted to integers only.

```
$ x=5; y=7; z=7.2
```

```
$ test $x -eq $y; echo $?
```

```
1
```

```
$ test $x -lt $y; echo $?
```

```
0
```

```
$ test $z -gt $y; echo $?
```

```
bash: test: 7.2: integer expression expected
```

```
2
```

```
$ test $z -eq $y; echo $?
```

```
bash: test: 7.2: integer expression expected
```

```
2
```

Having used **test** as a stand-alone feature, you can now use it as **if's** control command.

The next script uses test and the special shell variable

\$#-this time for testing a condition.

The script simply checks whether the right number of arguments have been entered:

Open a file called `argument_checker` and enter the following commands:

```
If test $# -ne 3; then
    echo "You didn't enter three arguments"
else
    echo "You entered the right number of arguments"
fi
```

Now run it twice, first with a single argument and then with three:

```
$ ./argument_checker 1024
```

```
You didn't enter three arguments
```

```
$. /argument_checker /home list.tar 1024
```

```
You entered the right number of arguments
```

4.2.9.2 String Comparison

Test can be used to compare strings with yet another set of operators.

Equality is performed with `==` and inequality with C-type operator `!=`.

Like the other **test** operators, these too should have whitespace on either side.

Open a file called `compile` and type the following line of commands into this file.

```
if test $# -eq 1; then
    if test $1 == "c"; then
        file=`ls -t *.c `
        cc $file
        ./a.out
    elif test $1 == "cpp"; then
        file=`ls -t *.cpp `
        g++ $file
        ./a.out
    else
        echo "Invalid file type"
    fi
else
    echo "Usage: $0 file_type"
    echo "Valid file types are c and cpp"
fi
```

Before running the compile file, create two c and c++ files.

test.c

```
#include<stdio.h>
```

```
int main(int argc, char*argv[])  
{  
printf("Hello, you are learning C!!\n");  
return 0;  
}
```

```
c++.cpp
#include<iostream>
using namespace std;

int main()
{
cout<<"Hello World of C++!\n";
return 0;
}
```

Now, run the compile file with c as the first argument and then cpp as the second argument.

./compile c

Hello, you are learning C.

./compile cpp

Hello World of C++

Now, let us see how we can check for null values of input.

You can use the string comparison features in the next script to check whether the user actually enters a string or simply presses the *[Enter]* key:

Now create a file called **emp4** and enter the following commands into the file.

```
#!/bin/bash
echo "Enter the string to be searched: "
read pname
if test -z "$pname"; then
    echo "You have not entered the string"; exit 1
else
    echo "Enter the file to be used: "
    read flname
    if !(test -n "$flname"); then
        echo "You have not entered the filename"; exit
2
    else
        grep "$pname" "$flname" || echo "Pattern not
found"
    fi
fi
```

All the possible string comparisons are shown below.

Test	True if
<code>s1 != s2</code>	String s1 is not equal to s2
<code>stg</code>	String stg is assigned and not null
<code>-n stg</code>	String stg is not a null string
<code>-z stg</code>	String stg is null string
<code>s1 == s2</code>	String s1=s2 (Korn and bash only)

Test also permits checking more than one condition in the same time, using the -a (AND) and -o (OR) operators.

You can now simplify the earlier script to illustrate this feature.

Create a file called emp5 and type the following commands into the file.

```
#!/bin/bash
echo "Enter the string to be searched: "
read pname
echo "Enter the file to be used: "
read fname
if test -n "$pname" -a -n "$fname"; then
grep "$pname" "$fname" || echo "Pattern not found"
else
echo "At least one input was a null string"; exit 1
fi
```

4.2.9.2 test: File Tests

Test can be used to test various file attributes.

For example, you can test whether a file has the necessary read, write or executable permissions.

So let us test some attributes a newly created file emp6.

```
$ ls -l emp6
```

```
-rw-rw-rw- 1      daniel      group879 Nov  10  
15:52 emp6
```

```
$ test -f emp6; echo $?
```

```
0
```

```
$ test -x emp6; echo $?
```

```
1
```

```
$ test -w emp3 && echo "True the file is writable"
```

```
True that file is writable
```

Here are all the file related Tests with **test**

Test	True if File
-f fname	Fname exists and is a regular file
-r fname	fname exists and is readable
-w fname	Fname exists and is writable
-x fname	Fname exists and is executable
-d fname	Fname exists and is a directory
-s fname	Fname exists and has a size greater than zero
-e fname	Fname exists (Korn and bash only)
-u fname	Fname exists and has SUID bit set
-k fname	Fname exists and has a sticky bit set
-L fname	Fname exists and is a symbolic link
f1 -nt f2	f1 is newer than f2 (Korn and bash only)
f1 -ot f2	f1 is older than f2 (Korn and bash only)
f1 -ef f2	F1 is linked to f2 (Korn and bash only)

The ! Negates a test.

Create a file called filetest and enter the following command into the file.

```
#!/bin/bash

if test ! -f $1; then
    echo "File does not exist"
elif test ! -r $1; then
    echo "File is not readable"
elif test ! -w $1; then
    echo "File is not writable"
else
    echo "File is both readable and writable"
fi
```

```
$ ./filetest test
```

File does not exist

Create a file called test.

```
$ ./filetest test
```

File is both readable and writable

Some people make the mistake of using the shell's wild cards with **test**. For instance, you can't use `test -w index*.html` to test whether the file that matches the pattern is writable.

4.2.10 The case Conditional

The **case** statement is the second conditional offered by the shell.

The statement exactly matches an expression for more than one alternative and uses a compact construct to permit multiway branching

Here is the syntax:

```
case expression in  
    pattern1) commands1;;  
    pattern2) commands2;;  
    pattern3) commands3;;
```

```
esac
```

case first matches *expression* with *pattern1*. If the match succeeds, then it executes *commands1*, which may be one or more commands.

If the match fails, then *pattern2* is matched, and so forth.

Each command list is terminated with a pair of semicolons, and the entire construct is closed with **esac**, the reverse of **case**.

We will now use a simple script called *case* to display

```
#!/bin/bash
tput clear
echo " 1. Find files modified in the last 24 hours"
echo " 2. The free disk space."
echo " 3. Space consumed by this user."
echo " 4. Exit"
echo "Selection"

read choice
case $choice in
1)find $HOME -mtime -1 -print;;
2)df;;
3)du -s $HOME;;
4)exit;;
*)echo "Invalid option"
esac
```

Case matches the value of \$choice with the strings 1, 2, 3, and 4.

The first three options execute the find, df, and du commands.

The last option (a*) matches any option not matched by the previous options.

To find out how much space your files are occupying on your disk, invoke the script and choose option 3.

Now consider another example. Programmers frequently encounter a logic that test a user response for both y and Y (or n or N).

To implement this logic with if, you need to use the compound condition feature.

If test \$choice -eq "y" or test \$choice -eq "Y"

Case takes care of this with the compact express y|Y, which matches both uppercase and lowercase(open a file called case 2 and type the following and execute it.)

```
echo "Do you wish to continue? (y/n): "
```

```
read answer
```

```
case "$answer" in
```

```
    y|Y) echo "continuing...";;
```

```
    n|N) exit;;
```

```
    *)echo "Invalid option";;
```

```
esac
```

4.2.10.2 Wild Cards: case Uses Them

Case has an excellent string matching feature that uses wild cards.

It uses the filename matching metacharacters *, ? – but only to match strings and not the files in the current directory.

The revised **case** construct for a previous example lets the user answer the question in several ways.

```
case $answer in
    [Yy][eE]*) ;;
    [nN][oO]) exit;;
    *)echo "Invalid response"
esac
```

Wild-card usage in the first two options appears simple enough

Note that the * appears in two options but has slightly different meanings.

In the first option, it behaves like a normal wild card.

In the last option, it provides a refuge for all other nonmatched options.

Note that the last case option doesn't need;; but you can provide then if you want.

Now that we have a powerful string matching feature with us, we can make the **compile** script more powerful.

\$cat compile2

```
#!/bin/bash
file=`ls -t *.cpp *.c 2>/dev/null`
case $file in
    *.c) cc $file && ./a.out;;
    *.cpp) g++ $file && ./a.out;;
    *)echo "There's no C++ or C program in the current
directory"
esac
```

The first statement has done most of the work for us.

It selects the last modified .c or .cpp file and stores the value in the variable file.

After that, it is a simple matter for the **case**.

It matches the extensions and calls up the right compiler.

4.2.11: Computation and String Handling

The Bourne shell can check whether an integer is greater than another, but it doesn't have any computing features at all.

It has to rely on the external **expr** command for that purpose.

This command combines two functions in one:

- Performs arithmetic operations on integers
- Manipulates strings

We'll use **expr** to perform both these functions, but with not-very readable code when it comes to string handling.

4.2.11.1 The Arithmetic Functions

Expr can perform the basic four arithmetic operations, as well as the modulus (remainder) function:

\$ x=3; y=5

\$ expr 3 + 5

8

\$ expr \$x - \$y

-2

\$ expr 3 * 5

15

\$ expr \$y / \$x

1

\$ expr 13 % 5

3

The operand, be it +, -, * etc, must be enclosed on either side by whitespace.

Observe that the multiplication operand (*) has to be escaped to prevent the shell from interpreting it as the filename metacharacter.

Since **expr** can only handle integers, division yields only the integral part.

expr is often used with command substitution to assign a variable.

For example, you can set a variable *z* to the sum of two numbers.

```
$x=6; y=2; z=`expr $x + $y`           //no space
```

```
between z, '=', and 'expr..'
```

```
$echo $z
```

```
8
```

Perhaps the most common use of **expr** is in incrementing

```
$x =5  
$x=`expr $x + 1`  
$echo $x  
6
```

4.2.11.2 String Handling

Depending on the nature of the regular expression, **expr** can perform three important string functions:

- Determine the length of the string.
- Extract a substring
- Locate the position of a character in a string

The Length of a String: The length of a string is a relatively simple matter; the regular expression `.*` signifies to **expr** that it has to print the number of characters matching the pattern.

Hence this simplifies the length of the entire string.

```
$ expr "robert kahn" : '.*' #Note the
```

Here, **expr** has counted the number of occurrences of any character (.*)

This feature is useful in validating data entry.

Consider that you want to validate the name of a person accepted through the keyboard so that it doesn't exceed, say, 20 characters in length.

The following **expr** sequence can be quite useful for this task (file name validate):

```
echo "Enter your name. "  
read name  
If test `expr "$name" : '.*'` -gt 20 ; then  
    echo "Name too long"  
fi
```

Extracting a Substring: **expr** can extract a string enclosed by the escaped character \ (and \).

If you wish to extract the 2-digit year from a 4-digit string, you must create a pattern group and extract it this way:

```
$ stg=2001  
$ expr "$stg" : '..\(..\)  
01
```

Note the pattern group `\(..\)`. It signifies the first two characters in the value of `$stg` have to be ignored and two characters have to be extracted (not remembered) from the third character position.

There is no `\1` or `\2` used here.

Locating Position of a Character `expr` can also return the location of the first occurrence of a character inside a string.

To locate the position of the character `b` in the string value `$stg`, you have to count the number of characters which are not `b` (`[^b]*`), followed by a `b`:

```
$stg="paul_baran"  
$expr "$stg" : '[^b]*b'  
6
```

4.2.12 \$0: Calling a Script with Different Names

A file can be called using different names (links) and doing different things depending on the name by which it is called

Now that we know how to use **case** and extract a string with **expr**, it is time we designed a single script called

#Script that is called by different names

```
lastfile=`ls -t *.c`
```

```
case $SHELL in
```

```
*/ksh/*/sh) command=$0;;
```

```
    *bash) command=`expr $0 : '.*\([^/]*\)'\` ;;
```

```
#Removes ./
```

```
esac
```

```
case $command in
```

```
    runj) ./a.out;;
```

```
    vij) vi $lastfile;;
```

```
    comj) cc $lastfile && echo "$lastfile compiled  
successfully";;
```

```
esac
```

The only thing left to do is to create the links for comj with the ln command

\$ln comj runj

./comj

test.c compiled successfully

There are two **case** constructs here.

The first one extracts the name of the script from the \$0.

For the Bourne and Korn shells, there is really nothing to do except assign \$0 to the variable command.

For bash, the ./ is removed by the **expr** command.

expr ignores the group **.***/ which signifies any number of characters followed by a /.

It then extracts a group of non-frontslash characters.

This effectively extracts the command name which it stores in the variable `command`.

The second **case** conditional now check the name by which the program is invoked.

Since the last modified filename is held the *lastfile* variable, the rest of the job becomes a simple matter.

\$/comj
test.c compiled succesfully

4.2.13 Sleep and Wait

You may occasionally need to introduce some delay in a shell script to let the user see some message on the screen before the script starts doing something else.

You may also want to make a check regularly (say, once in two minutes) for an even to occur (say, for a file to spring into existence).

sleep is a UNIX command that introduces this delay.

It is used with an argument that specifies the number of seconds for which the shell will pause of *sleep* before it resumes execution.

```
$ sleep 5 ; echo "5 seconds have elapsed"  
5 seconds have elapsed
```

The message appears 5 seconds after the commands have been invoked.

wait is a shell built-in that checks whether all background processes have been completed.

This can be quite useful when you have run a job in the background and now want to make sure the command is completed so you can run another program.

You can use the **wait** command to wait for the completion of the last background job with or without the process PIDs.

```
wait  
wait 138
```

4.2.14 while and until: Looping

None of the pattern scanning scripts developed so far offers the user another chance to rectify a faulty response.

Loops let you perform a set of instructions repeatedly.

The shell features three types of loops-**while**, **until** and

All of them repeat the instruction set enclosed by certain keywords as often as their control command permits.

The **while** statement **repeatedly** performs a set of instructions till the control command returns a true exit status.

The general syntax of the command is as follows:

```
while condition is true
do
    commands
done
```

The *commands* enclosed by **do** and **done** are executed repeatedly as long as the *condition* remains true.

You can use any UNIX command or **test** as the control command (condition here) as before

We will start with a simple while loop.
The “**while**” script accepts a code and description in the same line and writes out the line to new file.

It then prompts you for more entries

```
#!/bin/bash
answer=y      #Must set it to y first to enter the loop
while test $answer == "y"
do
    echo "Enter the code and description: "
    read code description
    echo "$code|$description" >> newlist
    echo "Enter any more (y/n)?"
    read anymore
    case $anymore in
        y*|Y*) answer=y ;;      #Anything starting with Y
or y
        n*|N*) answer=n ;;      #Anything starting with
N or n
        *) answer=y ;;          #Any other reply means y
    esac
done
```

Let us run it.

Enter the code and description: **03 analgesics**

Enter any more (y/n)? **Y**

Enter the code and description: **04 antibiotics**

Enter any more (y/n)? *[Enter]*

Enter the code and description: **05 OTC drugs**

Enter any more (y/n)? **N**

That was a fairly simple script that didn't feature any validation whatsoever.

Whatever you enter goes to the file newlist.

\$ cat newlist

03|analgesics

04|antibiotics

05|OTC drugs

4.2.14.1 Using while to Wait for a File

Let us now consider an interesting while loop application.

There are situations when a program b needs to read a file created by another program a.

b can run only after the file is created by a.

We will implement this logic in the following example.

The **monitfile** script periodically monitors the disk for the existence of the file (here **invoice**) and then executes the **alloc** script once the file is located.

\$ cat monitfile

```
while test ! -r invoice  
do  
sleep 5  
done  
chmod +x alloc  
./alloc
```

\$cat alloc

```
echo "This is the file that is executed once invoice file is  
created"
```

Now after creating the above two files, run monitfile in the background as follows

\$/monifile &

Notice after assigning a PID as a background job nothing happens. The script is waiting for the creation of the

Now using a vi text editor, create an empty file called invoice and save it.

After five seconds which is the duration we assigned it in the while loop of the monitfile script to check for the existence of the file invoice, the following appears

\$ This is the file that is executed once invoice file is created

4.2.14.2 until: while's Complement

The until statement complements the **while** construct in that the loop body here is executed repeatedly as long as the condition remains false.

It is simply another way of viewing the whole thing.

Either form can be used interchangeably, except for the fact that the expression has to be negated when

The previous **while** control command can be written in this way:

Until test -r invoice.lst.

And this translates into “Until the file invoice.lst is made readable,...”

This form is certainly more intelligible.

4.2.15 A sample script

Have you ever lost an important file by overwriting it with something else?

The script cpback.sh protects your files from accidental overwriting by the cp command.

This program accepts the filename to be copied and a directory name as two arguments

It checks whether a file foo exists at the destination, and if it does, it affixes a numeric extension-beginning with foo.1.

This could exist too, so it increments the number repeatedly to finally copy the file without overwriting.

```
#Program cpback.sh -- Copies a file to a directory
#Makes a backup instead of overwriting the destination file
#Copies foo to foo.1 if foo exists or foo.2 if foo.1 exists....
```

```
if test $# -ne 2; then
    echo "Usage: $0 source destination"; exit
elif test ! -d $2; then
    echo "Directory $2 doesn't exist"
else
    file=$1
    if test ! -f $2/$file; then
        cp $1 $2
    else
        copies=1
        while true; do
            if test ! -f $2/$file.$copies; then
                cp $1 $2/$1.$copies
                echo "File $1 copies to $1.$copies"
                break;
            else
                copies=`expr $copies + 1`
            fi
        done
    fi
fi
```

\$./cpback.sh

Usage: cpback.sh source destination

\$./cpback.sh foo safe

Directory safe doesn't exist

Now create a directory called safe with mkdir safe command

\$cpback.sh foo safe

File foo copied to foo.1

\$cpback.sh foo safe

File foo copied to foo.2

4.2.16 for: Looping with a List

The for loop is different in structure from the ones used in other programming languages.

Unlike **while** and **until**, **for** doesn't test a condition but₄₇ uses a **list** instead

The syntax of this construct is quite simple:

```
for variable in list ; do  
    commands  
done
```

The loop body is the same with the same keywords `do` and `done`.

But the additional parameters here are *variable* and *list*.

The loop body is executed as many times as there are items in the list.

A simple example can help you understand things better (forloop):

```
$ for file in chap1 chap2 chap3; do  
> cp $file ${file}.bak  
> echo $file copies to $file.bak  
> done
```

```
chap1 copies to chap1.bak  
chap2 copies to chap2.bak  
chap3 copies to chap3.bak
```

The list here comprises a series of character string (chap 1 and onwards) separated by whitespace

Each item in the list is assigned to *variable* file.

File is first assigned to value chap1, then chap2, and then chap 3.

Each file is copied with a .bak extension and the completion message is displayed after every file is

You can also use a series of variables in the command line.

They are evaluated by the shell before executing the loop:

```
$ for var in $PATH $HOME $MAIL; do echo "$var";  
done
```

```
/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/  
bin:/home/ateferi/bin  
/home/ateferi  
/var/spool/mail/ateferi
```

You have to provide the semicolons at the right places if you want to enter the entire loop in a single line.

The three lines represent the values of the three environment variables.

4.2.16.1 List From Filenames and Wild Cards

Like **case**, **for** also uses wild cards, *but it actually matches then against files in the current directory.*

In the first example, where we used **for** to copy a group of files, we could have easily used a wild-card pattern:

```
For file in chap[1-3]; do
```

The shell produces an alphabetically stored list of filenames and then instructions in the body act on each file in turn.

4.2.16.2 List from Positional Parameters

One of the most important uses of **for** lies in its ability to process positional parameters that are supplied externally.

The next **emp6.sh** scans the file repeatedly for each

```
for pattern in $*; do
    grep "$pattern" emp2 || echo "$pattern not found"
done
```

```
$ ./emp6.sh director
```

```
1      Abebe  director      Beautiful Mind
2      Daniel director      Lost
```

```
$ ./emp6.sh director actor
```

```
1      Abebe  director      Beautiful Mind
2      Daniel director      Lost
3      Getachew actor        Amen
```

```
[ateferi@localhost ~]$
```