7.1. Introduction to if

7.1.1. General

At times you need to specify different courses of action to be taken in a shell script, depending on the success or failure of a command. The **if** construction allows you to specify such conditions.

The most compact syntax of the **if** command is:

if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi

The **TEST-COMMAND** list is executed, and if its return status is zero, the **CONSEQUENT-COMMANDS** list is executed. The return status is the exit status of the last command executed, or zero if no condition tested true.

The **TEST-COMMAND** often involves numerical or string comparison tests, but it can also be any command that returns a status of zero when it succeeds and some other status when it fails. Unary expressions are often used to examine the status of a file. If the `FILE` argument to one of the primaries is of the form `/dev/fd/N`, then file descriptor "N" is checked. `stdin`, `stdout` and `stderr` and their respective file descriptors may also be used for tests.

7.1.1.1. Expressions used with if

The table below contains an overview of the so-called "primaries" that make up the **TEST-COMMAND** command or list of commands. These primaries are put between square brackets to indicate the test of a conditional expression.

Table 7-1. Primary expressions

Primary	Meaning
[`-a FILE`]	True if `FILE` exists.
[`-b FILE`]	True if `FILE` exists and is a block-special file.
[`-c FILE`]	True if `FILE` exists and is a character-special file.
[`-d FILE`]	True if `FILE` exists and is a directory.
[`-e FILE`]	True if `FILE` exists.
[`-f FILE`]	True if `FILE` exists and is a regular file.
[`-g FILE`]	True if `FILE` exists and its SGID bit is set.
[`-h FILE`]	True if `FILE` exists and is a symbolic link.
[`-k FILE`]	True if `FILE` exists and its sticky bit is set.
[`-p FILE`]	True if `FILE` exists and is a named pipe (FIFO).
[`-r FILE`]	True if `FILE` exists and is readable.
[`-s FILE`]	True if `FILE` exists and has a size greater than zero.
[`-t FD`]	True if file descriptor `FD` is open and refers to a terminal.
[`-u FILE`]	True if `FILE` exists and its SUID (set user ID) bit is set.
[`-w FILE`]	True if `FILE` exists and is writable.
[`-x FILE`]	True if `FILE` exists and is executable.
[`-O FILE`]	True if `FILE` exists and is owned by the effective user ID.
[`-G FILE`]	True if `FILE` exists and is owned by the effective group ID.

Operation	Meaning / Effect
[-L FILE]	True if FILE exists and is a symbolic link.
[-N FILE]	True if FILE exists and has been modified since it was last read.
[-S FILE]	True if FILE exists and is a socket.
[FILE1 -nt FILE2]	True if FILE1 has been changed more recently than FILE2, or if FILE1 exists and FILE2 does not.
[FILE1 -ot FILE2]	True if FILE1 is older than FILE2, or is FILE2 exists and FILE1 does not.
[FILE1 -ef FILE2]	True if FILE1 and FILE2 refer to the same device and inode numbers.
[-o OPTIONNAME]	True if shell option "OPTIONNAME" is enabled.
[-z STRING]	True of the length if "STRING" is zero.
[-n STRING] or [STRING]	True if the length of "STRING" is non-zero.
[STRING1 == STRING2]	True if the strings are equal. "=" may be used instead of "==" for strict POSIX compliance.
[STRING1 != STRING2]	True if the strings are not equal.
[STRING1 < STRING2]	True if "STRING1" sorts before "STRING2" lexicographically in the current locale.
[STRING1 > STRING2]	True if "STRING1" sorts after "STRING2" lexicographically in the current locale.
[ARG1 OP ARG2]	"OP" is one of -eq, -ne, -lt, -le, -gt or -ge. These arithmetic binary operators return true if "ARG1" is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to "ARG2", respectively. "ARG1" and "ARG2" are integers.

Expressions may be combined using the following operators, listed in decreasing order of precedence:

Table 7-2. Combining expressions

Operation	Effect
[! EXPR]	True if **EXPR** is false.
[(EXPR)]	Returns the value of **EXPR**. This may be used to override the normal precedence of operators.
[EXPR1 -a EXPR2]	True if both **EXPR1** and **EXPR2** are true.
[EXPR1 -o EXPR2]	True if either **EXPR1** or **EXPR2** is true.

The **[** (or **test**) built-in evaluates conditional expressions using a set of rules based on the number of arguments. More information about this subject can be found in the Bash documentation. Just like the **if** is closed with **fi**, the opening square bracket should be closed after the conditions have been listed.

7.1.1.2. Commands following the then statement

The **CONSEQUENT-COMMANDS** list that follows the **then** statement can be any valid UNIX command, any executable program, any executable shell script or any shell statement, with the exception of the closing **fi**. It is important to remember that the **then** and **fi** are considered to be separated statements in the shell. Therefore, when issued on the command line, they are separated by a semi-colon.

In a script, the different parts of the **if** statement are usually well-separated. Below, a couple of simple examples.

7.1.1.3. Checking files

The first example checks for the existence of a file:

```
anny ~> cat msgcheck.sh
#!/bin/bash

echo "This scripts checks the existence of the messages file."
echo "Checking..."
if [ -f /var/log/messages ]
  then
    echo "/var/log/messages exists."
fi
echo
echo "...done."

anny ~> ./msgcheck.sh
This scripts checks the existence of the messages file.
Checking...
/var/log/messages exists.

...done.
```

7.1.1.4. Checking shell options

To add in your Bash configuration files:

```
# These lines will print a message if the noclobber option is set:

if [ -o noclobber ]
  then
        echo "Your files are protected against accidental overwriting using redirection."
fi
```


The environment

The above example will work when entered on the command line:

```
anny ~> if [ -o noclobber ] ; then echo ; echo "your files are protected
against overwriting." ; echo ; fi

your files are protected against overwriting.

anny ~>
```

However, if you use testing of conditions that depend on the environment, you might get different results when you enter the same command in a script, because the script will open a new shell, in which expected variables and options might not be set automatically.

7.1.2. Simple applications of if

7.1.2.1. Testing exit status

The `?` variable holds the exit status of the previously executed command (the most recently completed foreground process).

The following example shows a simple test:

```
anny ~> if [ $? -eq 0 ]
More input> then echo 'That was a good job!'
More input> fi
```

```
That was a good job!

anny ~>
```

The following example demonstrates that **TEST-COMMANDS** might be any UNIX command that returns an exit status, and that **if** again returns an exit status of zero:

```
anny ~> if ! grep $USER /etc/passwd
More input> then echo "your user account is not managed locally"; fi
your user account is not managed locally

anny > echo $?
0

anny >
```

The same result can be obtained as follows:

```
anny > grep $USER /etc/passwd

anny > if [ $? -ne 0 ] ; then echo "not a local account" ; fi
not a local account

anny >
```

7.1.2.2. Numeric comparisons

The examples below use numerical comparisons:

```
anny > num=`wc -l work.txt`

anny > echo $num
201

anny > if [ "$num" -gt "150" ]
More input> then echo ; echo "you've worked hard enough for today."
More input> echo ; fi

you've worked hard enough for today.


anny >
```

This script is executed by cron every Sunday. If the week number is even, it reminds you to put out the garbage cans:

```
#!/bin/bash

# Calculate the week number using the date command:

WEEKOFFSET=$[ $(date +"%V") % 2 ]

# Test if we have a remainder.  If not, this is an even week so send a message.
# Else, do nothing.

if [ $WEEKOFFSET -eq "0" ]; then
  echo "Sunday evening, put out the garbage cans." | mail -s "Garbage cans out" your@your_domain.org
fi
```

7.1.2.3. String comparisons

An example of comparing strings for testing the user ID:

```
if [ "$(whoami)" != 'root' ]; then
        echo "You have no permission to run $0 as non-root user."
```

```
        exit 1;
fi
```

With Bash, you can shorten this type of construct. The compact equivalent of the above test is as follows:

```
[ "$(whoami)" != 'root' ] && ( echo you are using a non-privileged account; exit 1 )
```

Similar to the "&&" expression which indicates what to do if the test proves true, "||" specifies what to do if the test is false.

Regular expressions may also be used in comparisons:

```
anny > gender="female"

anny > if [[ "$gender" == f* ]]
More input> then echo "Pleasure to meet you, Madame."; fi
Pleasure to meet you, Madame.

anny >
```

Real Programmers

Most programmers will prefer to use the **test** built-in command, which is equivalent to using square brackets for comparison, like this:

```
test "$(whoami)" != 'root' && (echo you are using a non-privileged account; exit 1)
```

No exit?

If you invoke the **exit** in a subshell, it will not pass variables to the parent. Use { and } instead of (and) if you do not want Bash to fork a subshell.

See the info pages for Bash for more information on pattern matching with the "((EXPRESSION))" and "[[EXPRESSION]]" constructs.

Prev Conditional statements | Home Up | Next More advanced if usage