NAME

if, test – evaluate condition(s) or make execution of actions dependent upon the evaluation of condition(s)

SYNOPSIS

/usr/bin/test [*condition*]

[*condition* **]**

sh

if *condition* **; then** *action* **; fi**

if *condition* **; then** *action* **; else** *action2* **; fi**

if *condition* **; then** *action* **; elif** *condition2* **; then** *action2* **; . . . ; fi**

if *condition* **; then** *action* **; elif** *condition2* **; then** *action2* **; . . . ; else** *action3* **; fi**

test *condition*

[*condition* **]**

csh

if (*condition***) then**
action
else if (*condition2***) then**
action2
else
action3
endif

if (*condition***)** *action*

ksh

if *condition* **; then** *action* **; fi**

if *condition* **; then** *action* **; else** *action2* **; fi**

if *condition* **; then** *action* **; elif** *condition2* **; then** *action2* **; . . . ; fi**

if *condition* **; then** *action* **; elif** *condition2* **; then** *action2* **; . . . ; else** *action3* **; fi**

test *condition*

[*condition* **]**

DESCRIPTION

/usr/bin/test

The **test** utility evaluates the *condition* and indicates the result of the evaluation by its exit status. An exit status of zero indicates that the condition evaluated as true and an exit status of 1 indicates that the condition evaluated as false.

In the second form of the utility, which uses **[ ]** rather than **test**, the square brackets must be separate arguments and *condition* is optional.

sh

The *condition* following **if** is executed and, if it returns a **0** exit status, the *action* following the first **then** is executed. Otherwise, the *condition2* following **elif** is executed and, if its value is **0**, the *action2* following the next **then** is executed. Failing the **if** and **elif** *conditions*, the **else** *action3* is executed. If no **else** *action* or **then** *action* is executed, the **if** command returns a **0** exit status. Any number of **elif . . . then . . .** branching pairs are allowed, but only one **else**.

test evaluates the condition *condition* and, if its value is true, sets exit status to **0**; otherwise, a non-zero (false) exit status is set; **test** also sets a non-zero exit status if there are no arguments. When permissions are tested, the effective user ID of the process is used.

All operators, flags, and brackets (brackets used as shown in the second **SYNOPSIS** line) must be separate arguments to the **test** command; normally these items are separated by spaces.

Primitives:

The following primitives are used to construct *condition*:

–r *filename*	True if *filename* exists and is readable.
–w *filename*	True if *filename* exists and is writable.
–x *filename*	True if *filename* exists and is executable.
–f *filename*	True if *filename* exists and is a regular file. Alternatively, if **/usr/bin/sh** users specify **/usr/ucb** before **/usr/bin** in their **PATH** environment variable, then **test** will return true if *filename* exists and is (**not–a–directory**). This is also the default for **/usr/bin/csh** users.
–d *filename*	True if *filename* exists and is a directory.
–h *filename*	True if *filename* exists and is a symbolic link. With all other primitives (except **–L** *filename*), the symbolic links are followed by default.
–c *filename*	True if *filename* exists and is a character special file.
–b *filename*	True if *filename* exists and is a block special file.
–p *filename*	True if *filename* exists and is a named pipe (fifo).
–u *filename*	True if *filename* exists and its set-user-ID bit is set.
–g *filename*	True if *filename* exists and its set-group-ID bit is set.
–k *filename*	True if *filename* exists and its sticky bit is set.
–s *filename*	True if *filename* exists and has a size greater than zero.
–t [*fildes*]	True if the open file whose file descriptor number is *fildes* (1 by default) is associated with a terminal device.
–z *s1*	True if the length of string *s1* is zero.
–n *s1*	True if the length of the string *s1* is non-zero.
s1 **=** *s2*	True if strings *s1* and *s2* are identical.
s1 **!=** *s2*	True if strings *s1* and *s2* are *not* identical.
s1	True if *s1* is *not* the null string.
n1 **–eq** *n2*	True if the integers *n1* and *n2* are algebraically equal. Any of the comparisons **–ne**, **–gt**, **–ge**, **–lt**, and **–le** may be used in place of **–eq**.
–L *filename*	True if *filename* exists and is a symbolic link. With all other primitives (except **–h** *filename*), the symbolic links are followed by default.

Operators:

These primaries may be combined with the following operators:

!	Unary negation operator.
−**a**	Binary *and* operator.
−**o**	Binary *or* operator (−**a** has higher precedence than −**o**).
(*condition*)	Parentheses for grouping. Notice also that parentheses are meaningful to the shell and, therefore, must be quoted.

The **not**−**a**−**directory** alternative to the −**f** option is a transition aid for BSD applications and may not be supported in future releases.

The −**L** option is a migration aid for users of other shells which have similar options and may not be supported in future releases.

If you test a file you own (the −**r** −**w** or −**x** tests), but the permission tested does not have the *owner* bit set, a non-zero (false) exit status will be returned even though the file may have the **group** or *other* bit set for that permission. The correct exit status will be set if you are super-user.

The **=** and **!=** operators have a higher precedence than the −**r** through −**n** operators, and **=** and **!=** always expect arguments; therefore, **=** and **!=** cannot be used with the −**r** through −**n** operators.

If more than one argument follows the −**r** through −**n** operators, only the first argument is examined; the others are ignored, unless a −**a** or a −**o** is the second argument.

csh With the multi-line form of **if**:

> if *condition* is true, the *action* up to the first **else** or **then** is executed. Otherwise, if **else if** *condition2* is true, the *action2* between the **else if** and the following **else** or **then** is executed. Otherwise, the *action3* between the **else** and the **endif** is executed.
>
> The **if** must appear alone on its input line or after an **else**. Only one **endif** is needed, but it is required. The words **else** and **endif** must be the first nonwhite characters on a line. Any number of **else if . . . then . . .** branching pairs are allowed, but only one **else**.

With the one-line form of **if**, there are no **else**, **then**, or **endif** keywords:

> if the specified *condition* evaluates to true, the single *action* with arguments is executed. Variable substitution on *action* happens early, at the same time it does for the rest of the *if* command. *action* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Note that I/O redirection occurs even if *condition* is false, when *action* is *not* executed (this is a bug).

ksh The *condition* following **if** is executed and, if it returns an exit status of **0**, the *action* following the first **then** is executed. Otherwise, the *condition2* following **elif** is executed and, if its value is **0**, the *action2* following the next **then** is executed. Failing that, the **else** *action3* is executed. If no **else** *action* or **then** *action* is executed, then the **if** command returns an exit status of **0**. Any number of **elif . . . then . . .** branching pairs are allowed, but only one **else**.

For a description of the **test** built-in, see the **ksh**(1) sections "Conditional Expressions" and "Arithmetic Evaluation" as well as the (**sh**) Bourne shell's **test** built-in above.

[*condition* **]** evaluates file attributes, string comparisons, and compound "and" or "or" *condition*s.

OPERANDS

All operators and elements of primaries must be presented as separate arguments to the **test** utility.

The following primaries can be used to construct *condition*:

–**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 set group ID flag is set.

–**n** *string* True if the length of *string* is non-zero.

–**p** *file* True if *file* 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** *file_descriptor*
True if the file whose file descriptor number is *file_descriptor* is open and is associated with a terminal.

–**u** *file* True if *file* exists and its set-user-ID flag is set.

–**w** *file* True if *file* exists and is writable. True will indicate only that the write flag is on. The *file* will not be writable on a read-only file system even if this test indicates true.

–**x** *file* True if *file* exists and is executable. True will indicate only that the execute flag is on. If *file* is a directory, true indicates that *file* can be searched.

–**z** *string* True if the length of string *string* is zero.

string True if the string *string* is not the null string.

s1 = *s2* True if the strings *s1* and *s2* are identical.

s1 **!**= *s2* True if the strings *s1* and *s2* are not identical.

n1 –**eq** *n2* True if the integers *n1* and *n2* are algebraically equal.

n1 –**ne** *n2* True if the integers *n1* and *n2* are not algebraically equal.

n1 –**gt** *n2* True if the integer *n1* is algebraically greater than the integer *n2*.

n1 –**ge** *n2* True if the integer *n1* is algebraically greater than or equal to the integer *n2*.

n1 –**lt** *n2* True if the integer *n1* is algebraically less than the integer *n2*.

n1 **–le** *n2* True if the integer *n1* is algebraically less than or equal to the integer *n2*.

These primaries can be combined with the following operator:

! *condition* True if *condition* is false.

The primaries with two elements of the form:

primary_operand

are known as *unary primaries.* The primaries with three elements in either of the two forms:

primary_operand *primary_operand*

primary_operand *primary_operator* *primary_operand*

are known as *binary primaries.* Additional implementation-dependent operators and *primary_operator s* may be provided by implementations. They will be of the form -*operator* where the first character of *operator* is not a digit.

The algorithm for determining the precedence of the operators and the return value that will be generated is based on the number of arguments presented to **test**. (However, when using the **[. . .]** form, the right-bracket final argument will not be counted in this algorithm.)

In the following list, **$1**, **$2**, **$3** and **$4** represent the arguments presented to **test**.

0 arguments:
Exit false (1).

1 argument:
Exit true (0) if **$1** is not null; otherwise, exit false.

2 arguments:

- If **$1** is **!**, exit true if **$2** is null, false if **$2** is not null.
- If **$1** is a unary primary, exit true if the unary test is true, false if the unary test is false.
- Otherwise, produce unspecified results.

3 arguments:

- If **$2** is a binary primary, perform the binary test of **$1** and **$3**.
- If **$1** is **!**, negate the two-argument test of **$2** and **$3**.
- Otherwise, produce unspecified results.

4 arguments:

- If **$1** is **!**, negate the three-argument test of **$2**, **$3**, and **$4**.
- Otherwise, the results are unspecified.

USAGE

Scripts should be careful when dealing with user-supplied input that could be confused with primaries and operators. Unless the application writer knows all the cases that produce input to the script, invocations like:

test "$1" -a "$2"

should be written as:

test "$1" && test "$2"

to avoid problems if a user supplied values such as **$1** set to **!** and **$2** set to the null string. That is, in cases where maximal portability is of concern, replace:

test expr1 -a expr2

with:

test expr1 && test expr2

and replace:

test expr1 -o expr2

with:

test expr1 | | test expr2

but note that, in **test**, –**a** has higher precedence than –**o** while **&&** and **| |** have equal precedence in the shell.

Parentheses or braces can be used in the shell command language to effect grouping.

Parentheses must be escaped when using **sh**; for example:

test \(expr1 -a expr2 \) -o expr3

This command is not always portable outside XSI-conformant systems. The following form can be used instead:

(test expr1 && test expr2) | | test expr3

The two commands:

test "$1"

test ! "$1"

could not be used reliably on some historical systems. Unexpected results would occur if such a *string* condition were used and **$1** expanded to **!**, **(** or a known unary primary. Better constructs are:

test -n "$1"

test -z "$1"

respectively.

Historical systems have also been unreliable given the common construct:

test "$response" = "expected string"

One of the following is a more reliable form:

test "X$response" = "Xexpected string"

test "expected string" = "$response"

Note that the second form assumes that **expected string** could not be confused with any unary primary. If **expected string** starts with –, **(**, **!** or even **=**, the first form should be used instead. Using the preceding rules without the marked extensions, any of the three comparison forms is reliable, given any input. (However, note that the strings are quoted in all cases.)

Because the string comparison binary primaries, **=** and **!=**, have a higher precedence than any unary primary in the >4 argument case, unexpected results can occur if arguments are not properly prepared. For example, in

test -d $1 -o -d $2

If **$1** evaluates to a possible directory name of **=**, the first three arguments are considered a string comparison, which causes a syntax error when the second –**d** is encountered. is encountered. One of the following forms prevents this; the second is preferred:

test \(-d "$1" \) -o \(-d "$2" \)

test -d "$1" | | test -d "$2"

Also in the >4 argument case,

test "$1" = "bat" -a "$2" = "ball"

Syntax errors will occur if **$1** evaluates to **(** or **!**. One of the following forms prevents this; the third is preferred:

test "X$1" = "Xbat" -a "X$2" = "Xball"

test "$1" = "bat" && test "$2" = "ball"

test "X$1" = "Xbat" && test "X$2" = "Xball"

EXAMPLES

In the **if** command examples, three conditions are tested, and if all three evaluate as true or successful, then their validities are written to the screen.

The 3 tests are:
if a variable set to 1 is greater than 0,
if a variable set to 2 is equal to 2, and
if the word "root" is included in the text file /etc/passwd.

/usr/bin/test

1. Perform a **mkdir** if a directory does not exist:

   test ! -d tempdir && mkdir tempdir

2. Wait for a file to become non-readable:

```
while test -r thefile
do
   sleep 30
done
echo '"thefile" is no longer readable'
```

3. Perform a command if the argument is one of three strings (two variations):

```
if [ "$1" = "pear" ] | | [ "$1" = "grape" ] | | [ "$1" = "apple" ]
then
      command
fi
```

```
case "$1" in
    pear|grape|apple) command ; ;
esac
```

The two forms of the **test** built-in follow the Bourne shell's **if** example.

sh

```
ZERO=0 ONE=1 TWO=2 ROOT=root
if  [ $ONE -gt $ZERO ]
[ $TWO -eq 2 ]
grep $ROOT  /etc/passwd >&1 > /dev/null  # discard output
then
    echo "$ONE is greater than 0, $TWO equals 2, and $ROOT is a user-name
        in the password file"
else
    echo "At least one of the three test conditions is false"
fi
```

Examples of the **test** built-in:

```
test `grep $ROOT /etc/passwd >&1 /dev/null`  # discard output
echo $?    # test for success

[ `grep nosuchname /etc/passwd >&1 /dev/null` ]
echo $?    # test for failure
```

csh

```
@ ZERO = 0; @ ONE = 1; @ TWO = 2;  set ROOT = root
grep $ROOT  /etc/passwd >&1 /dev/null  # discard output
    # $status must be tested for immediately following grep
if ( "$status" == "0" && $ONE > $ZERO && $TWO == 2 ) then
    echo "$ONE is greater than 0, $TWO equals 2, and $ROOT is a user-name
        in the password file"
endif
```

ksh

```
ZERO=0 ONE=1 TWO=$((ONE+ONE)) ROOT=root
if  ((ONE > ZERO))          #  arithmetical comparison
[[ $TWO = 2 ]]              #  string comparison
[ `grep $ROOT  /etc/passwd >&1 /dev/null` ]  # discard output
then
    echo "$ONE is greater than 0, $TWO equals 2, and $ROOT is a user-name
        in the password file"
else
    echo "At least one of the three test conditions is false"
fi
```

The Korn shell will also accept the syntax of both the **if** command and the **test** command of the Bourne shell.

When using the brackets (**[ ]**) within **if** commands, you must separate both inside ends of the brackets from the inside characters with a space.

ENVIRONMENT

See **environ**(5) for descriptions of the following environment variables that affect the execution of **test**: **LC_CTYPE**, **LC_MESSAGES**, and **NLSPATH**.

EXIT STATUS

The following exit values are returned:

0 *condition* evaluated to true.

1 *condition* evaluated to false or *condition* was missing.

>1 An error occurred.

SEE ALSO

csh(1), **ksh**(1), **sh**(1), **test**(1B), **environ**(5)

NOTES

Both the Bourne shell, **sh**, and the Korn shell, **ksh**, can use the semicolon and the carriage return interchangeably in their syntax of the **if**, **for**, and **while** built-in commands.