The conditional expression

Synopsis

```
[[ <EXPRESSION> ]]
```

Description

The conditional expression is meant as the modern variant of the classic test command. Since it is **not** a normal command, Bash doesn't need to apply the normal commandline parsing rules like recognizing `&&` as command list operator.

The testing features basically are the same (see the lists for classic test command), with some additions and extensions.

Operator	Description
(`<EXPRESSION>` **)**	Used to group expressions, to influence precedence of operators
`<EXPRESSION1>` **&&** `<EXPRESSION2>`	`TRUE` if `<EXPRESSION1>` **and** `<EXPRESSION2>` are `TRUE` (do **not** use `-a`!)
`<EXPRESSION1>` **||** `<EXPRESSION2>`	`TRUE` if `<EXPRESSION1>` **or** `<EXPRESSION2>` is `TRUE` (do **not** use `-o`!)
`<STRING>` **==** `<PATTERN>`	`<STRING>` is checked against the pattern `<PATTERN>` - `TRUE` on a match *But note¹, quoting the pattern forces a literal comparison.*
`<STRING>` **=** `<PATTERN>`	equivalent to the `==` operator
`<STRING>` **!=** `<PATTERN>`	`<STRING>` is checked against the pattern `<PATTERN>` - `TRUE` on **no match**
`<STRING>` **=~** `<ERE>`	`<STRING>` is checked against the extended regular expression (https://en.wikipedia.org/wiki/Regular_expression#POSIX_extended) `<ERE>` - `TRUE` on a match

Operator	Description
See the classic test operators	Do **not** use the `test` -typical operators `-a` and `-o` for AND and OR.
See also arithmetic comparisons	Using `(( <EXPRESSION> ))` , the arithmetic expression compound command

When the `==` and `!=` operators are used, the string to the right of the operator is considered a pattern and matched according to the rules of Pattern Matching. If the shell option `nocasematch` is enabled, the match is performed without regard to the case of alphabetic characters.

[1]Any part of the pattern may be quoted to force it to be matched as a literal string.

When the operators `<` and `>` are used (string collation order), the test happens using the current locale when the `compat` level is greater than "40".

Operator precedence (highest ⇒ lowest):

- `( <EXPRESSION> )`
- `! <EXPRESSION>`
- `<EXPRESSION1> && <EXPRESSION2>`
- `<EXPRESSION1> || <EXPRESSION2>`

Do **not** use the `test` -typical operators `-a` and `-o` for AND and OR, they are not known to the conditional expression. Instead, use the operators `&&` and `||` .

Word splitting

Word splitting and pathname expansion are not performed in the expression you give. That means, a variable containing spaces can be used without quoting:

```
sentence="Be liberal in what you accept, and conservative in what you send"
checkme="Be liberal in what you accept, and conservative in what you send"
if [[ $sentence == $checkme ]]; then
  echo "Matched...!"
else
  echo "Sorry, no match :-("
fi
```

Compare that to the classic test command, where word splitting is done (because it's a normal command, not something special):

```
sentence="Be liberal in what you accept, and conservative in what you
send"
checkme="Be liberal in what you accept, and conservative in what you
send"
if [ "$sentence" == "$checkme" ]; then
  echo "Matched...!"
else
  echo "Sorry, no match :-("
fi
```

You need to quote that variable reference in the classic test command, since (due to the spaces) the word splitting will break it otherwise!

Regular Expression Matching

Using the operator `=~` , the left hand side operand is matched against the **extended regular expression (ERE)** on the right hand side.

This is consistent with matching against patterns: Every quoted part of the regular expression is taken literally, even if it contains regular expression special characters.

Best practise is to put the regular expression to match against into a variable. This is to avoid shell parsing errors on otherwise valid regular expressions.

```
REGEX="^[[:upper:]]{2}[[:lower:]]*$"

# Test 1
STRING=Hello
if [[ $STRING =~ $REGEX ]]; then
  echo "Match."
else
  echo "No match."
fi
# ==> "No match."

# Test 2
STRING=HEllo
if [[ $STRING =~ $REGEX ]]; then
  echo "Match."
else
  echo "No match."
fi
# ==> "Match."
```

The interpretation of quoted regular expression special characters can be influenced by setting the `compat31` and `compat32` shell options (`compat*` in general). See List of shell options.

The special BASH_REMATCH array variable

An array variable whose members are assigned by the `=~` binary operator to the `[[` conditional command.

The element with index 0 is the portion of the string matching the entire regular expression. The element with index n is the portion of the string matching the nth parenthesized subexpression.

See BASH_REMATCH.

Example:

```
if [[ "The quick, red fox" =~ ^The\ (.*),\ (.*)\ fox$ ]]; then
    echo "${BASH_REMATCH[0]} is ${BASH_REMATCH[1]} and ${BASH_REMATCH
[2]}.";
fi

==> The quick, red fox is quick and red.
```

Behaviour differences compared to the builtin test command

As of Bash 4.1 alpha, the test primaries '<' and '>' (compare strings lexicographically) use the current locale settings, while the same primitives for the builtin test command don't. This leads to the following situation where they behave differently:

```
$ ./cond.sh
[[ ' 4' < '1' ]]        --> exit 1
[[ 'step+' < 'step-' ]] --> exit 1
[ ' 4' \< '1' ]         --> exit 0
[ 'step+' \< 'step-' ]  --> exit 0
```

It won't be aligned. The conditional expression continues to respect the locate, as introduced with 4.1-alpha, the builtin `test` / `[` command continues to behave differently.

Implicit arithmetic context

When you use a numeric comparison, the arguments are evaluated as an arithmetic expression. The arithmetic expression must be quoted if it both contains whitespace and is not the result of an expansion.

```
[[ 'i=5, i+=2' -eq 3+4 ]] && echo true # prints true.
```

Examples

Portability considerations

- `[[ ... ]]` functionality isn't specified by POSIX(R), though it's a reserved word
- Amongst the major "POSIX-shell superset languages" (for lack of a better term) which do have `[[`, the test expression compound command is one of the very most

portable non-POSIX features. Aside from the `=~` operator, almost every major feature is consistent between Ksh88, Ksh93, mksh, Zsh, and Bash. Ksh93 also adds a large number of unique pattern matching features not supported by other shells including support for several different regex dialects, which are invoked using a different syntax from Bash's `=~`, though `=~` is still supported by ksh and defaults to ERE.

- As an extension to POSIX ERE, most GNU software supports backreferences in ERE, including Bash. According to POSIX, only BRE is supposed to support them. This requires Bash to be linked against glibc, so it won't necessarily work on all platforms. For example,

```
$(m='(abc(def))(\1)(\2)'; [[ abcdefabcdefdef =~ $m ]]; printf
'<%s> ' $? "${BASH_REMATCH[@]}" )
```

  will give `<0> <abcdefabcdefdef> <abcdef> <def> <abcdef> <def>`.
- the `=~` (regex) operator was introduced in Bash 3.0, and its behaviour changed in Bash 3.2: since 3.2, quoted strings and substrings are matched as literals by default.
- the behaviour of the `<` and `>` operators (string collation order) has changed since Bash 4.0

See also

- Internal: pattern matching language
- Internal: the classic test command
- Internal: the if-clause
- What is the difference between test, [and [[? (http://mywiki.wooledge.org/BashFAQ/031) - BashFAQ 31 - Greg's wiki.

1

syntax/ccmd/conditional_expression.txt Last modified: 2022/08/09 03:42 by fgrose

This site is supported by Performing Databases - your experts for database administration

Bash Hackers Wiki

GNU FDL

Except where otherwise noted, content on this wiki is licensed under the following license:
GNU Free Documentation License 1.3