Patterns and pattern matching

A pattern is a **string description**. Bash uses them in various ways:

- Pathname expansion (Globbing - matching filenames)
- Pattern matching in conditional expressions
- Substring removal and search and replace in Parameter Expansion
- Pattern-based branching using the case command

The pattern description language is relatively easy. Any character that's not mentioned below matches itself. The `NUL` character may not occur in a pattern. If special characters are quoted, they're matched literally, i.e., without their special meaning.

Do **not** confuse patterns with ***regular expressions***, because they share some symbols and do similar matching work.

Normal pattern language

Sequence	Description
`*`	Matches **any string**, including the null string (empty string)
`?`	Matches any **single character**
`X`	Matches the character `X` which can be any character that has no special meaning
`\X`	Matches the character `X`, where the character's special meaning is stripped by the backslash
`\\`	Matches a backslash
`[…]`	Defines a pattern **bracket expression** (see below). Matches any of the enclosed characters at this position.

Bracket expressions

The bracket expression `[…]` mentioned above has some useful applications:

Bracket expression	Description

Bracket expression	Description
`[XYZ]`	The "normal" bracket expression, matching either `X`, `Y` or `Z`
`[X-Z]`	A range expression: Matching all the characters from `X` to `Y` (your curr the characters are **sorted**!)
`[[:class:]]`	Matches all the characters defined by a POSIX(r) character class (https://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap(`alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, p `space`, `upper`, `word` and `xdigit`
`[^…]`	A negating expression: It matches all the characters that are **not** in the b
`[!…]`	Equivalent to `[^…]`
`[]...]` or `[-…]`	Used to include the characters `]` and `-` into the set, they need to be th the opening bracket
`[=C=]`	Matches any character that is eqivalent to the collation weight of `C` (curr
`[[.SYMBOL.]]`	Matches the collating symbol `SYMBOL`

Examples

Some simple examples using normal pattern matching:

- Pattern `"Hello world"` matches
  - `Hello world`
- Pattern `[Hh]"ello world"` matches
  - ⇒ `Hello world`
  - ⇒ `hello world`
- Pattern `Hello*` matches (for example)
  - ⇒ `Hello world`
  - ⇒ `Helloworld`
  - ⇒ `HelloWoRlD`
  - ⇒ `Hello`
- Pattern `Hello world[[:punct:]]` matches (for example)
  - ⇒ `Hello world!`
  - ⇒ `Hello world.`
  - ⇒ `Hello world+`
  - ⇒ `Hello world?`
- Pattern `[[.backslash.]]Hello[[.vertical-line.]]world[[.exclamation-mark.]]` matches (using collation symbols (https://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap07.html#tag_
  - ⇒ `\Hello|world!`

Extended pattern language

If you set the shell option `extglob`, Bash understands some powerful patterns. A `<PATTERN-LIST>` is one or more patterns, separated by the pipe-symbol (`PATTERN|PATTERN`).

`?(<PATTERN-LIST>)`	Matches **zero or one** occurrence of the given patterns
`*(<PATTERN-LIST>)`	Matches **zero or more** occurrences of the given patterns
`+(<PATTERN-LIST>)`	Matches **one or more** occurrences of the given patterns
`@(<PATTERN-LIST>)`	Matches **one** of the given patterns
`!(<PATTERN-LIST>)`	Matches anything **except** one of the given patterns

Examples

Delete all but one specific file

```
rm -f !(survivior.txt)
```

Pattern matching configuration

Related shell options

option	classification	description
`dotglob`	globbing	see Pathname expansion customization
`extglob`	global	enable/disable extended pattern matching language, as described above
`failglob`	globbing	see Pathname expansion customization
`nocaseglob`	globbing	see Pathname expansion customization
`nocasematch`	pattern/string matching	perform pattern matching without regarding the case of individual letters
`nullglob`	globbing	see Pathname expansion customization
`globasciiranges`	globbing	see Pathname expansion customization

Bugs and Portability considerations

* Counter-intuitively, only the `[!chars]` syntax for negating a character class is specified by POSIX for shell pattern matching. `[^chars]` is merely a commonly-supported extension. Even dash supports `[^chars]`, but not posh.

* All of the extglob quantifiers supported by bash were supported by ksh88. The set of extglob quantifiers supported by ksh88 are identical to those supported by Bash, mksh, ksh93, and zsh.

* mksh does not support POSIX character classes. Therefore, character ranges like `[0-9]` are somewhat more portable than an equivalent POSIX class like `[:digit:]`.

* Bash uses a custom runtime interpreter for pattern matching. (at least) ksh93 and zsh translate patterns into regexes and then use a regex compiler to emit and cache optimized pattern matching code. This means Bash may be an order of magnitude or more slower in cases that involve complex back-tracking (usually that means extglob quantifier nesting). You may wish to use Bash's regex support (the `=~` operator) if performance is a problem, because Bash will use your C library regex implementation rather than its own pattern matcher.

TODO: describe the pattern escape bug https://gist.github.com/ormaaj/6195070 (https://gist.github.com/ormaaj/6195070)

ksh93 extras

ksh93 supports some very powerful pattern matching features in addition to those described above.

* ksh93 supports arbitrary quantifiers just like ERE using the `{from,to}(pattern-list)` syntax. `{2,4}(foo)bar` matches between 2-4 "foo"'s followed by "bar". `{2,}(foo)bar` matches 2 or more "foo"'s followed by "bar". You can probably figure out the rest. So far, none of the other shells support this syntax.

* In ksh93, a `pattern-list` may be delimited by either `&` or `|`. `&` means "all patterns must be matched" instead of "any pattern". For example,

```
[[ fo0bar == @(fo[0-9]&+([[:alnum:]]))bar ]]
```

would be true while

```
[[ f00bar == @(fo[0-9]&+([[:alnum:]]))bar ]]
```

is false, because all members of the and-list must be satisfied. No other shell supports this so far, but you can simulate some cases in other shells using double extglob negation. The aforementioned ksh93 pattern is equivalent in Bash to:

```
[[ fo0bar == !(!(fo[0-9])|!(+([[:alnum:]])))bar ]]
```

, which is technically more portable, but ugly.

* ksh93's printf builtin can translate from shell patterns to ERE and back again using the `%R` and `%P` format specifiers respectively.

TODO: `~()` (and regex), `.sh.match`, backrefs, special `${var/…/…}` behavior, `%()`

Discussion

Amit Verma, 2010/09/16 12:30 ()

What if we want to match a particular pattern for 'n' number of times??

Jan Schampera, 2010/09/17 04:47 ()

This is not possible with patterns (or extended patterns). Patterns like this are not a substitute for regular expressions (but infact usually patterns can do the most "common" work, sometimes people (ab)use regular expressions for very very easy tasks in shell coding). Maybe Bash will have a new feature like this for extended patterns in future.

Depending on your needs you could either use some external program (grep, awk, sed, ...) or the conditional expression with the `=~` operator. A pathname expansion with regular expressions is not possible.

Tim Jones, 2012/07/11 08:32 ()

Why does bash need extglob enabled to enable those glob 'extensions' when they've been available in ksh for decades, by default?

This is causing many problems as scripts cannot be compatible with both ksh and bash :(

Jan Schampera, 2012/07/12 21:45 ()

I think this is a question you should ask the maintainer. I don't think there's a way to make different shells 100% compatible, not even in theory.

Andi, 2015/09/11 17:53 ()

What if we want to match a +? It should be escaped, like /+, correct?

Bash Hackers Wiki