-- --

IEEE P1003.2 Draft 11.2 – September 1991

The wording about shell scripts is meant to convey the fact that describing ‘‘trap actions’’ can only be understood in the context of the shell command language. Outside this context, such as in a C-language program, signals are the operative condition, not traps.

3.13 Pattern Matching Notation

The pattern matching notation described in this clause is used to specify patterns for matching strings in the shell. Historically, pattern matching notation is related to, but slightly different from, the regular expression notation described in 2.8. For this reason, the description of the rules for this pattern matching notation are based on the description of regular expression notation.

3.13.0.1 Pattern Matching Notation Rationale. *(This subclause is not a part of P1003.2)*

Pattern matching is a simpler concept and has a simpler syntax than regular expressions, as the former is generally used for the manipulation of file names, which are relatively simple collections of characters, while the latter is generally used to manipulate arbitrary text strings of potentially greater complexity. However, some of the basic concepts are the same, so this clause points liberally to the detailed descriptions in 2.8.

3.13.1 Patterns Matching a Single Character

The following *patterns matching a single-character* match a single character: *ordinary characters* , *special pattern characters* , and *pattern bracket expressions* . The pattern bracket expression also shall match a single collating element.

An ordinary character is a pattern that shall match itself. It can be any character
in the supported character set except for NUL, those special shell characters in 3.2 1
that require quoting, and the following three special pattern characters. Match- 1
ing shall be based on the bit pattern used for encoding the character, not on the 1
graphic representation of the character. If any character (ordinary, shell special, 1
or pattern special) is quoted, that pattern shall match the character itself. The 1
shell special characters always require quoting. 1

When unquoted and outside a bracket expression, the following three characters 1
shall have special meaning in the specification of patterns: 1

? A question-mark is a pattern that shall match any character.

* An asterisk is a pattern that shall match multiple characters, as described in 3.13.2.

[The open bracket shall introduce a pattern bracket expression.

The description of basic regular expression bracket expressions in 2.8.3.2 also shall apply to the pattern bracket expression, except that the exclamation-mark character (`!`) shall replace the circumflex character (`^`) in its role in a *nonmatching list* in the regular expression notation. A bracket expression starting with an unquoted circumflex character produces unspecified results.

When pattern matching is used where shell quote removal is not performed [such as in the argument to the `find –name` primary when `find` is being called using an *exec* function, or in the *pattern* argument to the *fnmatch* () function], special characters can be escaped to remove their special meaning by preceding them with a `<backslash>`. This escaping `<backslash>` shall be discarded. The sequence `\\` shall represent one literal backslash. All of the requirements and effects of quoting on ordinary, shell special, and special pattern characters shall apply to escaping in this context.

3.13.1.1 Patterns Matching a Single Character Rationale. *(This subclause is not a part of P1003.2)*

Both ‘‘quoting’’ and ‘‘escaping’’ are described here because pattern matching must work in three separate circumstances:

— Calling directly upon the shell, such as in pathname expansion or in a `case` statement. All of the following will match the string or file `abc`: `abc`, `"abc"`, `a"b"c`, `a\bc`, `a[b]c`, `a["b"]c`, `a[\b]c`, `a?c`, `a*c`. The following will not: `"a?c"`, `a\*c`, `a\[b]c`, `a["\b"]c`.

— Calling a utility or function without going through a shell, as described for `find` and *fnmatch* ().

— Calling utilities such as `find` or `pax` through the shell command line. (Although `find` and `pax` are the only instances of this in the standard utilities, describing it globally here is useful for future utilities that may use pattern matching internally.) In this case, shell quote removal is performed before the utility sees the argument. For example, in

```
find /bin -name "e\c[\h]o" -print
```

after quote removal, the backslashes are presented to `find` and it treats them as escape characters. Both precede ordinary characters, so the `c` and `h` represent themselves and `echo` would be found on many historical systems (that have it in `/bin`). To find a filename that contained shell special characters or pattern characters, both quoting and escaping are required, such as

```
pax -r ... "*a\(\?"
```

to extract a filename ending with ‘`a(?`’.

Conforming applications are required to quote or escape the shell special characters (called ‘‘metacharacters’’ in some historical documentation). If used without this protection, syntax errors can result or implementation extensions can be triggered. For example, the KornShell supports a series of extensions based on parentheses in patterns.

The restriction on circumflex in a bracket expression is to allow implementations that support pattern matching using circumflex as the negation character in addition to the exclamation-mark. 1

3.13.2 Patterns Matching Multiple Characters

The following rules are used to construct *patterns matching multiple characters* from *patterns matching a single character* :

(1) The asterisk (*) is a pattern that shall match any string, including the null string.

(2) The concatenation of *patterns matching a single character* is a valid pattern that shall match the concatenation of the single characters or collating elements matched by each of the concatenated patterns.

(3) The concatenation of one or more *patterns matching a single character* with one or more asterisks is a valid pattern. In such patterns, each asterisk shall match a string of zero or more characters, matching the greatest possible number of characters that still allows the remainder of the pattern to match the string.

3.13.2.1 Patterns Matching Multiple Characters Rationale. *(This subclause is not a part of P1003.2)*

Since each asterisk matches ‘‘zero or more’’ occurrences, the patterns `a*b` and `a**b` have identical functionality.

Examples:

`a[bc]`	matches the strings `ab` and `ac`.
`a*d`	matches the strings `ad`, `abd`, and `abcd`, but not the string `abc`.
`a*d*`	matches the strings `ad`, `abcd`, `abcdef`, `aaaad`, and `adddd`;
`*a*d`	matches the strings `ad`, `abcd`, `efabcd`, `aaaad`, and `adddd`.

3.13.3 Patterns Used for Filename Expansion

The rules described so far in 3.13.1 and 3.13.2 are qualified by the following rules that apply when pattern matching notation is used for filename expansion.

(1) The slash character in a pathname shall be explicitly matched by using one or more slashes in the pattern; it cannot be matched by the asterisk or question-mark special characters or by a bracket expression. Slashes in the pattern are identified before bracket expressions; thus, a slash cannot be included in a pattern bracket expression used for filename expansion.

(2) If a filename begins with a period (`.`), the period shall be explicitly matched by using a period as the first character of the pattern or

immediately following a slash character. The leading period shall not be matched by:

— The asterisk or question-mark special characters, or

— A bracket expression containing a nonmatching list (such as `[!a]`), a range expression (such as `[%-0]`), or a character class expression (such as `[[:punct:]]`).

It is unspecified whether an explicit period in a bracket expression matching list (such as `[.abc]`) can match a leading period in a filename.

(3) Specified patterns are matched against existing filenames and pathnames, as appropriate. Each component that contains a pattern character requires read permission in the directory containing that component. Any component that does not contain a pattern character requires search permission. For example, given the pattern

```
/foo/bar/x*/bam
```

search permission is needed for directory `/foo`, search and read permissions are needed for directory `bar`, and search permission is needed for each `x*` directory. If the pattern matches any existing filenames or pathnames, the pattern shall be replaced with those filenames and pathnames, sorted according to the collating sequence in effect in the current locale. If the pattern contains an invalid bracket expression or does not match any existing filenames or pathnames, the pattern string shall be left unchanged.

3.13.3.1 Patterns Used for File Name Expansion Rationale. *(This subclause is not a part of P1003.2)*

The caveat about a slash within a bracket expression is derived from historical practice. The pattern `a[b/c]d` will not match such pathnames as `abd` or `a/d`. It will only match a pathname of literally `a[b/c]d`.

Filenames beginning with a period historically have been specially protected from view on UNIX systems. A proposal to allow an explicit period in a bracket expression to match a leading period was considered; it is allowed as an implementation extension, but a conforming application cannot make use of it. If this extension becomes popular in the future, it will be considered for a future version of POSIX. 2.

Historical systems have varied in their permissions requirements. To match `f*/bar` has required read permissions on the `f*` directories in the System V shell, but this standard, the C-shell, and KornShell require only search permissions.

3.14 Special Built-in Utilities

The following *special built-in* utilities shall be supported in the shell command language. The output of each command, if any, shall be written to standard output, subject to the normal redirection and piping possible with all commands.

The term *built-in* implies that the shell can execute the utility directly and does not need to search for it. An implementation can choose to make any utility a built-in; however, the special built-in utilities described here differ from regular built-in utilities in two respects:

(1) A syntax error in a special built-in utility may cause a shell executing that utility to abort, while a syntax error in a regular built-in utility shall not cause a shell executing that utility to abort. (See 3.8.1 for the consequences of errors on interactive and noninteractive shells.) If a special built-in utility encountering a syntax error does not abort the shell, its exit value shall be nonzero.

(2) Variable assignments specified with special built-in utilities shall remain in effect after the built-in completes; this shall not be the case with a reg- 1
ular built-in or other utility. 1

As described in 2.3, the special built-in utilities in this clause need not be provided in a manner accessible via the POSIX.1 {8} *exec* family of functions.

Some of the special built-ins are described as conforming to the utility argument syntax guidelines in 2.10.2. For those that are not, the requirement in 2.11.3 that "--" be recognized as a first argument to be discarded does not apply and a conforming application shall not use that argument.

3.14.1 `break` — Exit from `for`, `while`, or `until` loop

`break` [*n*]

Exit from the smallest enclosing `for`, `while`, or `until` loop, if any; or from the *n*th enclosing loop if *n* is specified. The value of *n* is an unsigned decimal integer 1
≥ 1. The default is equivalent to $n=1$. If *n* is greater than the number of enclosing loops, the last enclosing loop shall be exited from. Execution continues with the command immediately following the loop.

Exit Status

0 Successful completion. 2

>0 The *n* value was not an unsigned decimal integer ≥ 1. 2