

NAME

re_format — POSIX 1003.2 regular expressions

DESCRIPTION

Regular expressions (“REs”), as defined in IEEE Std 1003.2 (“POSIX.2”), come in two forms: modern REs (roughly those of `egrep(1)`; 1003.2 calls these “extended” REs) and obsolete REs (roughly those of `ed(1)`; 1003.2 “basic” REs). Obsolete REs mostly exist for backward compatibility in some old programs; they will be discussed at the end. IEEE Std 1003.2 (“POSIX.2”) leaves some aspects of RE syntax and semantics open; ‘‡’ marks decisions on these aspects that may not be fully portable to other IEEE Std 1003.2 (“POSIX.2”) implementations.

A (modern) RE is one‡ or more non-empty‡ *branches*, separated by ‘|’. It matches anything that matches one of the branches.

A branch is one‡ or more *pieces*, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an *atom* possibly followed by a single‡ ‘*’, ‘+’, ‘?’, or *bound*. An atom followed by ‘*’ matches a sequence of 0 or more matches of the atom. An atom followed by ‘+’ matches a sequence of 1 or more matches of the atom. An atom followed by ‘?’ matches a sequence of 0 or 1 matches of the atom.

A *bound* is ‘{’ followed by an unsigned decimal integer, possibly followed by ‘,’ possibly followed by another unsigned decimal integer, always followed by ‘}’. The integers must lie between 0 and `RE_DUP_MAX` (255‡) inclusive, and if there are two of them, the first may not exceed the second. An atom followed by a bound containing one integer *i* and no comma matches a sequence of exactly *i* matches of the atom. An atom followed by a bound containing one integer *i* and a comma matches a sequence of *i* or more matches of the atom. An atom followed by a bound containing two integers *i* and *j* matches a sequence of *i* through *j* (inclusive) matches of the atom.

An atom is a regular expression enclosed in ‘()’ (matching a match for the regular expression), an empty set of ‘()’ (matching the null string)‡, a *bracket expression* (see below), ‘.’ (matching any single character), ‘^’ (matching the null string at the beginning of a line), ‘\$’ (matching the null string at the end of a line), a ‘\’ followed by one of the characters `^ . [$ () | * + ? { \` (matching that character taken as an ordinary character), a ‘\’ followed by any other character‡ (matching that character taken as an ordinary character, as if the ‘\’ had not been present‡), or a single character with no other significance (matching that character). A ‘{’ followed by a character other than a digit is an ordinary character, not the beginning of a bound‡. It is illegal to end an RE with ‘\’.

A *bracket expression* is a list of characters enclosed in ‘[]’. It normally matches any single character from the list (but see below). If the list begins with ‘^’, it matches any single character (but see below) *not* from the rest of the list. If two characters in the list are separated by ‘-’, this is shorthand for the full *range* of characters between those two (inclusive) in the collating sequence, e.g. `[0-9]` in ASCII matches any decimal digit. It is illegal‡ for two ranges to share an endpoint, e.g. `a-c-e`. Ranges are very collating-sequence-dependent, and portable programs should avoid relying on them.

To include a literal ‘]’ in the list, make it the first character (following a possible ‘^’). To include a literal ‘-’, make it the first or last character, or the second endpoint of a range. To use a literal ‘-’ as the first endpoint of a range, enclose it in ‘[.]’ and ‘.]’ to make it a collating element (see below). With the exception of these and some combinations using ‘[’ (see next paragraphs), all other special characters, including ‘\’, lose their special significance within a bracket expression.

Within a bracket expression, a collating element (a character, a multi-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in ‘[.]’ and ‘.]’ stands for the sequence of characters of that collating element. The sequence is a single element of the bracket expression’s list. A bracket expression containing a multi-character collating element can thus match more than one character, e.g. if the collating sequence includes a ‘ch’ collating element, then the RE `[[.ch.]]*c` matches the first five characters of `chchcc`.

Within a bracket expression, a collating element enclosed in ‘[=’ and ‘=]’ is an equivalence class, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were ‘[.]’ and ‘.]’.)

For example, if ‘x’ and ‘y’ are the members of an equivalence class, then `[[=x=]]`, `[[=y=]]`, and `[xy]` are all synonymous. An equivalence class may not‡ be an endpoint of a range.

Within a bracket expression, the name of a *character class* enclosed in ‘`[: ’` and ‘`:]`’ stands for the list of all characters belonging to that class. Standard character class names are:

```

alnum  digit  punct
alpha  graph  space
blank  lower  upper
cntrl  print  xdigit

```

These stand for the character classes defined in `ctype(3)`. A locale may provide others. A character class may not be used as an endpoint of a range.

A bracketed expression like `[[:class:]]` can be used to match a single character that belongs to a character class. The reverse, matching any character that does not belong to a specific class, the negation operator of bracket expressions may be used: `[^[:class:]]`.

There are two special cases‡ of bracket expressions: the bracket expressions `[[:<:]]` and `[[:>:]]` match the null string at the beginning and end of a word respectively. A word is defined as a sequence of word characters which is neither preceded nor followed by word characters. A word character is an *alnum* character (as defined by `ctype(3)`) or an underscore. This is an extension, compatible with but not specified by IEEE Std 1003.2 (“POSIX.2”), and should be used with caution in software intended to be portable to other systems.

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, it matches the longest. Subexpressions also match the longest possible substrings, subject to the constraint that the whole match be as long as possible, with subexpressions starting earlier in the RE taking priority over ones starting later. Note that higher-level subexpressions thus take priority over their lower-level component subexpressions.

Match lengths are measured in characters, not collating elements. A null string is considered longer than no match at all. For example, `bb*` matches the three middle characters of `abbbbc`, `(wee|week)(knights|nights)` matches all ten characters of `weeknights`, when `(.*)*` is matched against `abc` the parenthesized subexpression matches all three characters, and when `(a*)*` is matched against `bc` both the whole RE and the parenthesized subexpression match the null string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g. ‘x’ becomes `[xX]`. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, so that (e.g.) `[x]` becomes `[xX]` and `[^x]` becomes `[^xX]`.

No particular limit is imposed on the length of REs‡. Programs intended to be portable should not employ REs longer than 256 bytes, as an implementation can refuse to accept such REs and remain POSIX-compliant.

Obsolete (“basic”) regular expressions differ in several respects. ‘|’ is an ordinary character and there is no equivalent for its functionality. ‘+’ and ‘?’ are ordinary characters, and their functionality can be expressed using bounds (`{ 1 , }` or `{ 0 , 1 }` respectively). Also note that ‘x+’ in modern REs is equivalent to `xx*`. The delimiters for bounds are ‘\{’ and ‘\}’, with ‘{’ and ‘}’ by themselves ordinary characters. The parentheses for nested subexpressions are ‘\ (’ and ‘\) ’, with ‘(’ and ‘) ’ by themselves ordinary characters. ‘^’ is an ordinary character except at the beginning of the RE or‡ the beginning of a parenthesized subexpression, ‘\$’ is an ordinary character except at the end of the RE or‡ the end of a parenthesized subexpression, and ‘*’ is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading ‘^’). Finally, there is one new type of atom, a *back reference*:

'\' followed by a non-zero decimal digit *d* matches the same sequence of characters matched by the *d*th parenthesized subexpression (numbering subexpressions by the positions of their opening parentheses, left to right), so that (e.g.) `\([bc]\)\1` matches 'bb' or 'cc' but not 'bc'.

ENHANCED FEATURES

When the `REG_ENHANCED` flag is passed to one of the `regcomp()` variants, additional features are activated. Like the enhanced `regex` implementations in scripting languages such as `perl(1)` and `python(1)`, these additional features may conflict with the IEEE Std 1003.2 ("POSIX.2") standards in some ways. Use this with care in situations which require portability (including to past versions of the Mac OS X using the previous `regex` implementation).

For enhanced basic REs, '+', '?' and '|' remain regular characters, but '\+', '\?' and '\|' have the same special meaning as the unescaped characters do for extended REs, i.e., one or more matches, zero or one matches and alteration, respectively. For enhanced extended REs, back references are available. Additional enhanced features are listed below.

Within a bracket expression, most characters lose their magic. This also applies to the additional enhanced features, which don't operate inside a bracket expression.

Assertions (available for both enhanced basic and enhanced extended REs)

In addition to '^' and '\$' (the assertions that match the null string at the beginning and end of line, respectively), the following assertions become available:

- `<` Matches the null string at the beginning of a word. This is equivalent to `[[:<:]]`.
- `>` Matches the null string at the end of a word. This is equivalent to `[[:>:]]`.
- `\b` Matches the null string at a word boundary (either the beginning or end of a word).
- `\B` Matches the null string where there is no word boundary. This is the opposite of '\b'.

Shortcuts (available for both enhanced basic and enhanced extended REs)

The following shortcuts can be used to replace more complicated bracket expressions.

- `\d` Matches a digit character. This is equivalent to `[[:digit:]]`.
- `\D` Matches a non-digit character. This is equivalent to `[^[:digit:]]`.
- `\s` Matches a space character. This is equivalent to `[[:space:]]`.
- `\S` Matches a non-space character. This is equivalent to `[^[:space:]]`.
- `\w` Matches a word character. This is equivalent to `[[:alnum:]]_`.
- `\W` Matches a non-word character. This is equivalent to `[^[:alnum:]]_`.

Literal Sequences (available for both enhanced basic and enhanced extended REs)

Literals are normally just ordinary characters that are matched directly. Under enhanced mode, certain character sequences are converted to specific literals.

- `\a` The "bell" character (ASCII code 7).
- `\e` The "escape" character (ASCII code 27).
- `\f` The "form-feed" character (ASCII code 12).
- `\n` The "new-line/line-feed" character (ASCII code 10).

`\r` The “carriage-return” character (ASCII code 13).

`\t` The “horizontal-tab” character (ASCII code 9).

Literals can also be specified directly, using their wide character values. Note that when matching a multi-byte character string, the string’s bytes are converted to wide character before comparing. This means that a single literal wide character value may match more than one string byte, depending on the locale’s wide character encoding.

`\xx..` An arbitrary eight-bit value. The `x..` sequence represents zero, one or two hexadecimal digits. (Note: if `x..` is less than two hexadecimal digits, and the character following this sequence happens to be a hexadecimal digit, use the (following) brace form to avoid confusion.)

`\x{x..}` An arbitrary, up to 32-bit value. The `x..` sequence is an arbitrary sequence of hexadecimal digits that is long enough to represent the necessary value.

Inline Literal Mode (available for both enhanced basic and enhanced extended REs)

A ‘`\Q`’ sequence causes literal (“quote”) mode to be entered, while ‘`\E`’ ends literal mode, and returns to normal regular expression processing. This is similar to specifying the `REG_NOSPEC` (or `REG_LITERAL`) option to `regcomp()`, except that rather than applying to the whole RE string, it only applies to the part between the ‘`\Q`’ and ‘`\E`’. Note that it is not possible to have a ‘`\E`’ in the middle of an inline literal range, as that would terminate literal mode prematurely.

Minimal Repetitions (available for enhanced extended REs only)

By default, the repetition operators, ‘`*`’, *bound*, ‘`?`’ and ‘`+`’ are *greedy*; they try to match as many times as possible. In enhanced mode, appending a ‘`?`’ to a repetition operator makes it minimal (or *ungreedy*); it tries to match the fewest number of times (including zero times, as appropriate).

For example, against the string `aaa`, the RE ‘`a*`’ would match the entire string, while `a*?` would match the null string at the beginning of the line (matches zero times). Likewise, against the string `ababab`, the RE `.*b`, would also match the entire string, while `.*?b` would only match the first two characters.

The `regcomp()` flag `REG_UNGREEDY` will make the regular (greedy) repetition operators ungreedy by default. Appending ‘`?`’ makes them greedy again.

Note that minimal repetitions are not specified by an official standard, so there may be differences between different implementations. In the current implementation, minimal repetitions have a high precedence, and can cause other standards requirements to be violated. For instance, on the string `aaaaa`, the RE `(aaa??)*` will only match the first four characters, violating the rules that the longest possible match is made and the longest subexpressions are matched. Using `(aaa??)*$` forces the entire string to be matched.

Non-capturing Parenthesized Subexpressions (available for enhanced extended REs only)

Normally, the match offsets to parenthesized subexpressions are recorded in the `pmatch` array (that is, when `REG_NOSUB` is not specified, and `nmatch` is large enough to encompass the parenthesized subexpression in question). In enhanced mode, if the first two characters following the left parenthesis are ‘`?:`’, grouping of the remaining contents is done, but the corresponding offsets are not recorded in the `pmatch` array. For example, against the string `fubar`, the RE `(fu)(bar)` would have two subexpression matches in `pmatch`; the first for ‘`fu`’ and the second for `bar`. But with the RE `(?:fu)(bar)`, there would only be one subexpression match, that of `bar`. Furthermore, against the string `fufubar`, the RE `(?fu)*(bar)` would again match the entire string, but only `bar` would be recorded in `pmatch`.

Inline Options (available for enhanced extended REs only)

Like the inline literal mode mentioned above, other options can be switched on and off for part of a RE. ‘`(?o..)`’ will turn on the options specified in `o..` (one or more options characters; see below), while `(?-o..)`

will turn off the specified options, and ‘(?o1..o2..)’ will turn on the first set of options, and turn off the second set.

The available options are:

- i** Turning on this option will ignore case during matching, while turning off will restore case-sensitive matching. If `REG_ICASE` was specified to `regcomp()`, this option can be used to turn that off.
- n** Turn on or off special handling of the newline character. If `REG_NEWLINE` was specified to `regcomp()`, this option can be used to turn that off.
- U** Turning on this option will make ungreedy repetitions the default, while turning off will make greedy repetitions the default. If `REG_UNGREEDY` was specified to `regcomp()`, this option can be used to turn that off.

The scope of the option change begins immediately following the right parenthesis, but up to the end of the enclosing subexpression (if any). Thus, for example, given the RE `(fu(?i)bar)baz`, the ‘fu’ portion matches case sensitively, `bar` matches case insensitively, and `baz` matches case sensitively again (since it is outside the scope of the subexpression in which the inline option was specified).

The inline options syntax can be combined with the non-capturing parenthesized subexpression to limit the option scope to just that of the subexpression. Then, for example, `fu(?i:bar)baz` is similar to the previous example, except for the parenthesized subexpression around `fu(?i)bar` in the previous example.

Inline Comments (available for enhanced extended REs only)

The syntax `(?#comment)` can be used to embed comments within a RE. Note that *comment* can not contain a right parenthesis. Also note that while syntactically, option characters can be added before the ‘#’ character, they will be ignored.

SEE ALSO

`regex(3)`

Regular Expression Notation, IEEE Std, 1003.2, section 2.8.

BUGS

Having two kinds of REs is a botch.

The current IEEE Std 1003.2 (“POSIX.2”) spec says that ‘)’ is an ordinary character in the absence of an unmatched ‘(’; this was an unintentional result of a wording error, and change is likely. Avoid relying on it.

Back references are a dreadful botch, posing major problems for efficient implementations. They are also somewhat vaguely defined (does `a\((\b\)*\2\)*d` match `abbbd?`). Avoid using them.

IEEE Std 1003.2 (“POSIX.2”) specification of case-independent matching is vague. The “one case implies all cases” definition given above is current consensus among implementors as to the right interpretation.

The bracket syntax for word boundaries is incredibly ugly.