NAME

re_format − POSIX 1003.2 regular expressions

DESCRIPTION

Regular expressions (‘‘RE’’s), as defined in POSIX 1003.2, come in two forms: modern REs (roughly those of **egrep**; 1003.2 calls these ‘‘extended’’ REs) and obsolete REs (roughly those of **ed**; 1003.2 ‘‘basic’’ REs). Obsolete REs mostly exist for backward compatibility in some old programs; they will be discussed at the end. 1003.2 leaves some aspects of RE syntax and semantics open; ‘†’ marks decisions on these aspects that may not be fully portable to other 1003.2 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 o and ô are the members of an equivalence class, then ‘[[=o=]]’, ‘[[=ô=]]’, and ‘[oô]’ 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.

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 POSIX 1003.2, 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 ‘abbbc’, ‘(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. ‘|’, ‘+’, and ‘?’ are ordinary characters and there is no equivalent for their functionality. 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’.

SEE ALSO

regex(3)

POSIX 1003.2, section 2.8 (Regular Expression Notation).

BUGS

Having two kinds of REs is a botch.

The current 1003.2 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.

1003.2’s 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 syntax for word boundaries is incredibly ugly.