<<< Previous Home Next >>>

The Open Group Base Specifications Issue 7, 2018 edition
IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)

9. Regular Expressions

Regular Expressions (REs) provide a mechanism to select specific strings from a set of character strings.

Regular expressions are a context-independent syntax that can represent a wide variety of character sets and character set orderings, where these character sets are interpreted according to the current locale. While many regular expressions can be interpreted differently depending on the current locale, many features, such as character class expressions, provide for contextual invariance across locales.

The Basic Regular Expression (BRE) notation and construction rules in Basic Regular Expressions shall apply to most utilities supporting regular expressions. Some utilities, instead, support the Extended Regular Expressions (ERE) described in Extended Regular Expressions; any exceptions for both cases are noted in the descriptions of the specific utilities using regular expressions. Both BREs and EREs are supported by the Regular Expression Matching interface in the System Interfaces volume of POSIX.1-2017 under *regcomp*(), *regexec*(), and related functions.

9.1 Regular Expression Definitions

For the purposes of this section, the following definitions shall apply:

entire regular expression

The concatenated set of one or more BREs or EREs that make up the pattern specified for string selection.

matched

A sequence of zero or more characters shall be said to be matched by a BRE or ERE when the characters in the sequence correspond to a sequence of characters defined by the pattern.

Matching shall be based on the bit pattern used for encoding the character, not on the graphic representation of the character. This means that if a character set contains two or more encodings for a graphic symbol, or if the strings searched contain text encoded in more than one codeset, no attempt is made to search for any other representation of the encoded symbol. If that is required, the user can specify equivalence classes containing all variations of the desired graphic symbol.

The search for a matching sequence starts at the beginning of a string and stops when the first sequence matching the expression is found, where "first" is defined to mean "begins earliest in the string". If the pattern permits a variable number of matching characters and thus there is more than one such sequence starting at that point, the longest such sequence is matched. For example, the BRE `"bb*"` matches the second to fourth characters of the string `"abbbc"`, and the ERE `"(wee|week)(knights|night)"` matches all ten characters of the string `"weeknights"`.

Consistent with the whole match being the longest of the leftmost matches, each subpattern, from left to right, shall match the longest possible string. For this purpose, a null string shall be considered to be longer than no match at all. For example, matching the BRE `"\(.*\).*"` against `"abcdef"`, the subexpression `"(\1)"` is `"abcdef"`, and matching the BRE `"\(a*\)*"` against `"bc"`, the subexpression `"(\1)"` is the null string.

When a multi-character collating element in a bracket expression (see RE Bracket Expression) is involved, the longest sequence shall be measured in characters consumed from the string to be matched; that is, the collating element counts not as one element, but as the number of characters it matches.

BRE (ERE) matching a single character

A BRE or ERE that shall match either a single character or a single collating element.

Only a BRE or ERE of this type that includes a bracket expression (see RE Bracket Expression) can match a collating element.

BRE (ERE) matching multiple characters

A BRE or ERE that shall match a concatenation of single characters or collating elements.

Such a BRE or ERE is made up from a BRE (ERE) matching a single character and BRE (ERE) special characters.

invalid

This section uses the term "invalid" for certain constructs or conditions. Invalid REs shall cause the utility or function using the RE to generate an error condition. When invalid is not used, violations of the specified syntax or semantics for REs produce undefined results: this may entail an error, enabling an extended syntax for that RE, or using the construct in error as literal characters to be matched. For example, the BRE construct `"\{1,2,3\}"` does not comply with the grammar. A conforming application cannot rely on it producing an error nor matching the literal characters `"\{1,2,3\}"`.

9.2 Regular Expression General Requirements

The requirements in this section shall apply to both basic and extended regular expressions.

The use of regular expressions is generally associated with text processing. REs (BREs and EREs) operate on text strings; that is, zero or more characters followed by an end-of-string delimiter (typically NUL). Some utilities employing regular expressions limit the processing to lines; that is, zero or more characters followed by a <newline>.

In the functions processing regular expressions described in System Interfaces volume of POSIX.1-2017, the <newline> is regarded as an ordinary character and both a <period> and a non-matching list can match one. The Shell and Utilities volume of POSIX.1-2017 specifies within the individual descriptions of those standard utilities employing regular expressions whether they permit matching of <newline> characters; if not stated otherwise, the use of literal <newline> characters or any escape sequence equivalent in either patterns or matched text produces undefined results. Those utilities (like *grep*) that do not allow <newline> characters to match are responsible for eliminating any <newline> from strings before matching against the RE. The *regcomp()* function in the System Interfaces volume of POSIX.1-2017, however, can provide support for such processing without violating the rules of this section.

The interfaces specified in POSIX.1-2017 do not permit the inclusion of a NUL character in an RE or in the string to be matched. If during the operation of a standard utility a NUL is included in the text designated to be matched, that NUL may designate the end of the text string for the purposes of matching.

When a standard utility or function that uses regular expressions specifies that pattern matching shall be performed without regard to the case (uppercase or lowercase) of either data or patterns, then when each character in the string is matched against the pattern, not only the character, but also its case counterpart (if any), shall be matched. This definition of case-insensitive processing is intended to allow matching of multi-character collating elements as well as characters, as each character in the string is matched using both its cases. For example, in a locale where `"Ch"` is a multi-character collating element and where a matching list expression matches such elements, the RE `"[[.Ch.]]"` when matched against the string `"char"` is in reality matched against `"ch"`, `"Ch"`, `"cH"`, and `"CH"`.

The implementation shall support any regular expression that does not exceed 256 bytes in length.

9.3 Basic Regular Expressions

9.3.1 BREs Matching a Single Character or Collating Element

A BRE ordinary character, a special character preceded by a <backslash>, or a <period> shall match a single character. A bracket expression shall match a single character or a single collating element.

9.3.2 BRE Ordinary Characters

An ordinary character is a BRE that matches itself: any character in the supported character set, except for the BRE special characters listed in BRE Special Characters.

The interpretation of an ordinary character preceded by an unescaped <backslash> ('\\') is undefined, except for:

- The characters ')', '(', '{', and '}'
- The digits 1 to 9 inclusive (see BREs Matching Multiple Characters)
- A character inside a bracket expression

9.3.3 BRE Special Characters

A BRE special character has special properties in certain contexts. Outside those contexts, or when preceded by a <backslash>, such a character is a BRE that matches the special character itself. The BRE special characters and the contexts in which they have their special meaning are as follows:

.[\
: The <period>, <left-square-bracket>, and <backslash> shall be special except when used in a bracket expression (see RE Bracket Expression). An expression containing a '[' that is unescaped and is not part of a bracket expression produces undefined results.

*
: The <asterisk> shall be special except when used:
  - In a bracket expression
  - As the first character of an entire BRE (after an initial '^', if any)
  - As the first character of a subexpression (after an initial '^', if any); see BREs Matching Multiple Characters

^
: The <circumflex> shall be special when used as an anchor (see BRE Expression Anchoring). The <circumflex> shall signify a non-matching list expression when it occurs first in a list, immediately following a <left-square-bracket> (see RE Bracket Expression).

$
: The <dollar-sign> shall be special when used as an anchor.

9.3.4 Periods in BREs

A <period> ('.'), when used outside a bracket expression, is a BRE that shall match any character in the supported character set except NUL.

9.3.5 RE Bracket Expression

A bracket expression (an expression enclosed in square brackets, "[]") is an RE that shall match a specific set of single characters, and may match a specific set of multi-character collating elements, based on the non-empty set of list expressions contained in the bracket expression.

The following rules and definitions apply to bracket expressions:

1. A bracket expression is either a matching list expression or a non-matching list expression. It consists of one or more expressions: ordinary characters, collating elements, collating symbols, equivalence classes, character classes, or range expressions. The <right-square-bracket> (`']'`) shall lose its special meaning and represent itself in a bracket expression if it occurs first in the list (after an initial <circumflex> (`'^'`), if any). Otherwise, it shall terminate the bracket expression, unless it appears in a collating symbol (such as `"[.].]"`) or is the ending <right-square-bracket> for a collating symbol, equivalence class, or character class. The special characters `'.'`, `'*'`, `'['`, and `'\\'` (<period>, <asterisk>, <left-square-bracket>, and <backslash>, respectively) shall lose their special meaning within a bracket expression.

   The character sequences `"[."`, `"[="`, and `"[:"` (<left-square-bracket> followed by a <period>, <equals-sign>, or <colon>) shall be special inside a bracket expression and are used to delimit collating symbols, equivalence class expressions, and character class expressions. These symbols shall be followed by a valid expression and the matching terminating sequence `".]"`, `"=]"`, or `":]"`, as described in the following items.

2. A matching list expression specifies a list that shall match any single character that is matched by one of the expressions represented in the list. The first character in the list cannot be the <circumflex>. An ordinary character in the list should only match that character, but may match any single character that collates equally with that character; for example, `"[abc]"` is an RE that should only match one of the characters `'a'`, `'b'`, or `'c'`.

   Note:
   A future version of this standard may require that an ordinary character in the list only matches that character.

   It is unspecified whether a matching list expression matches a multi-character collating element that is matched by one of the expressions.

3. A non-matching list expression begins with a <circumflex> (`'^'`), and the matching behavior shall be the logical inverse of the corresponding matching list expression (the same bracket expression but without the leading <circumflex>). For example, if the RE `"[abc]"` only matches `'a'`, `'b'`, or `'c'`, then `"[^abc]"` is an RE that matches any character except `'a'`, `'b'`, or `'c'`. It is unspecified whether a non-matching list expression matches a multi-character collating element that is not matched by any of the expressions. The <circumflex> shall have this special meaning only when it occurs first in the list, immediately following the <left-square-bracket>.

4. A collating symbol is a collating element enclosed within bracket-period (`"[."` and `".]"`) delimiters. Collating elements are defined as described in *Collation Order*. Conforming applications shall represent multi-character collating elements as collating symbols when it is necessary to distinguish them from a list of the individual characters that make up the multi-character collating element. For example, if the string `"ch"` is a collating element defined using the line:

   ```
   collating-element <ch-digraph> from "<c><h>"
   ```

   in the locale definition, the expression `"[[.ch.]]"` shall be treated as an RE containing the collating symbol `'ch'`, while `"[ch]"` shall be treated as an RE matching `'c'` or `'h'`. Collating symbols are recognized only inside bracket expressions. If the string is not a collating element in the current locale, the expression is invalid.

5. An equivalence class expression shall represent the set of collating elements belonging to an equivalence class, as described in *Collation Order*. Only primary equivalence classes shall be recognized. The class shall be expressed by enclosing any one of the collating elements in the equivalence class within bracket-equal (`"[="` and `"=]"`) delimiters. For example, if `'a'`, `'à'`, and `'â'` belong to the same equivalence class, then `"[[=a=]b]"`, `"[[=à=]b]"`, and `"[[=â=]b]"` are each equivalent to `"[aàâb]"`. If the collating element does not belong to an equivalence class, the equivalence class expression shall be treated as a collating symbol.

6. A character class expression shall represent the union of two sets:

a. The set of single characters that belong to the character class, as defined in the *LC_CTYPE* category in the current locale.

b. An unspecified set of multi-character collating elements.

All character classes specified in the current locale shall be recognized. A character class expression is expressed as a character class name enclosed within bracket- <colon> ("[:" and ":]") delimiters.

The following character class expressions shall be supported in all locales:

```
[:alnum:]   [:cntrl:]   [:lower:]   [:space:]
[:alpha:]   [:digit:]   [:print:]   [:upper:]
[:blank:]   [:graph:]   [:punct:]   [:xdigit:]
```

In addition, character class expressions of the form:

[:*name*:]

are recognized in those locales where the *name* keyword has been given a **charclass** definition in the *LC_CTYPE* category.

7. In the POSIX locale, a range expression represents the set of collating elements that fall between two elements in the collation sequence, inclusive. In other locales, a range expression has unspecified behavior: strictly conforming applications shall not rely on whether the range expression is valid, or on the set of collating elements matched. A range expression shall be expressed as the starting point and the ending point separated by a <hyphen-minus> ('-').

   In the following, all examples assume the POSIX locale.

   The starting range point and the ending range point shall be a collating element or collating symbol. An equivalence class expression used as a starting or ending point of a range expression produces unspecified results. An equivalence class can be used portably within a bracket expression, but only outside the range. If the represented set of collating elements is empty, it is unspecified whether the expression matches nothing, or is treated as invalid.

   The interpretation of range expressions where the ending range point is also the starting range point of a subsequent range expression (for example, "[a-m-o]") is undefined.

   The <hyphen-minus> character shall be treated as itself if it occurs first (after an initial '^', if any) or last in the list, or as an ending range point in a range expression. As examples, the expressions "[-ac]" and "[ac-]" are equivalent and match any of the characters 'a', 'c', or '-'; "[^-ac]" and "[^ac-]" are equivalent and match any characters except 'a', 'c', or '-'; the expression "[%--]" matches any of the characters between '%' and '-' inclusive; the expression "[--@]" matches any of the characters between '-' and '@' inclusive; and the expression "[a--@]" is either invalid or equivalent to '@', because the letter 'a' follows the symbol '-' in the POSIX locale. To use a <hyphen-minus> as the starting range point, it shall either come first in the bracket expression or be specified as a collating symbol; for example, "[][.-.]-0]", which matches either a <right-square-bracket> or any character or collating element that collates between <hyphen-minus> and 0, inclusive.

   If a bracket expression specifies both '-' and ']', the ']' shall be placed first (after the '^', if any) and the '-' last within the bracket expression.

8. If a bracket expression contains at least three list elements, where the first and last list elements are the same single-character element of <period>, <equals-sign>, or <colon>, then it is unspecified whether the bracket expression will be treated as a collating symbol, equivalence class, or character class, respectively; treated as a matching list expression; or rejected as an error.

9.3.6 BREs Matching Multiple Characters

The following rules can be used to construct BREs matching multiple characters from BREs matching a single character:

1. The concatenation of BREs shall match the concatenation of the strings matched by each component of the BRE.

2. A subexpression can be defined within a BRE by enclosing it between the character pairs `"\("` and `"\)"`. Such a subexpression shall match whatever it would have matched without the `"\("` and `"\)"`, except that anchoring within subexpressions is optional behavior; see BRE Expression Anchoring. Subexpressions can be arbitrarily nested.

3. The back-reference expression `'\n'` shall match the same (possibly empty) string of characters as was matched by a subexpression enclosed between `"\("` and `"\)"` preceding the `'\n'`. The character `'n'` shall be a digit from 1 through 9, specifying the *n*th subexpression (the one that begins with the *n*th `"\("` from the beginning of the pattern and ends with the corresponding paired `"\)"`). The expression is invalid if less than *n* subexpressions precede the `'\n'`. The string matched by a contained subexpression shall be within the string matched by the containing subexpression. If the containing subexpression does not match, or if there is no match for the contained subexpression within the string matched by the containing subexpression, then back-reference expressions corresponding to the contained subexpression shall not match. When a subexpression matches more than one string, a back-reference expression corresponding to the subexpression shall refer to the last matched string. For example, the expression `"^\(.*\)\1$"` matches strings consisting of two adjacent appearances of the same substring, and the expression `"\(a\)*\1"` fails to match `'a'`, the expression `"\(a\(b\)*\)*\2"` fails to match `'abab'`, and the expression `"^\(ab*\)*\1$"` matches `'ababbabb'`, but fails to match `'ababbab'`.

4. When a BRE matching a single character, a subexpression, or a back-reference is followed by the special character <asterisk> (`'*'`), together with that <asterisk> it shall match what zero or more consecutive occurrences of the BRE would match. For example, `"[ab]*"` and `"[ab][ab]"` are equivalent when matching the string `"ab"` .

5. When a BRE matching a single character, a subexpression, or a back-reference is followed by an interval expression of the format `"\{m\}"`, `"\{m,\}"`, or `"\{m,n\}"`, together with that interval expression it shall match what repeated consecutive occurrences of the BRE would match. The values of *m* and *n* are decimal integers in the range 0 <= *m*<= *n*<= {RE_DUP_MAX}, where *m* specifies the exact or minimum number of occurrences and *n* specifies the maximum number of occurrences. The expression `"\{m\}"` shall match exactly *m* occurrences of the preceding BRE, `"\{m,\}"` shall match at least *m* occurrences, and `"\{m,n\}"` shall match any number of occurrences between *m* and *n*, inclusive.

   For example, in the string `"abababccccccd"` the BRE `"c\{3\}"` is matched by characters seven to nine, the BRE `"\(ab\)\{4,\}"` is not matched at all, and the BRE `"c\{1,3\}d"` is matched by characters ten to thirteen.

The behavior of multiple adjacent duplication symbols (`'*'` and intervals) produces undefined results.

A subexpression repeated by an <asterisk> (`'*'`) or an interval expression shall not match a null expression unless this is the only match for the repetition or it is necessary to satisfy the exact or minimum number of occurrences for the interval expression.

9.3.7 BRE Precedence

The order of precedence shall be as shown in the following table:

BRE Precedence (from high to low)	
Collation-related bracket symbols	[==] [::] [..]

Escaped characters	\<special character>
Bracket expression	[]
Subexpressions/back-references	\(\) \n
Single-character-BRE duplication	* \{m,n\}
Concatenation	
Anchoring	^ $

9.3.8 BRE Expression Anchoring

A BRE can be limited to matching expressions that begin or end a string; this is called "anchoring". The <circumflex> and <dollar-sign> special characters shall be considered BRE anchors in the following contexts:

1. A <circumflex> (`'^'`) shall be an anchor when used as the first character of an entire BRE. The implementation may treat the <circumflex> as an anchor when used as the first character of a subexpression. The <circumflex> shall anchor the expression (or optionally subexpression) to the beginning of a string; only sequences starting at the first character of a string shall be matched by the BRE. For example, the BRE `"^ab"` matches `"ab"` in the string `"abcdef"`, but fails to match in the string `"cdefab"`. The BRE `"\(^ab\)"` may match the former string. A portable BRE shall escape a leading <circumflex> in a subexpression to match a literal circumflex.

2. A <dollar-sign> (`'$'`) shall be an anchor when used as the last character of an entire BRE. The implementation may treat a <dollar-sign> as an anchor when used as the last character of a subexpression. The <dollar-sign> shall anchor the expression (or optionally subexpression) to the end of the string being matched; the <dollar-sign> can be said to match the end-of-string following the last character.

3. A BRE anchored by both `'^'` and `'$'` shall match only an entire string. For example, the BRE `"^abcdef$"` matches strings consisting only of `"abcdef"`.

9.4 Extended Regular Expressions

The extended regular expression (ERE) notation and construction rules shall apply to utilities defined as using extended regular expressions; any exceptions to the following rules are noted in the descriptions of the specific utilities using EREs.

9.4.1 EREs Matching a Single Character or Collating Element

An ERE ordinary character, a special character preceded by a <backslash,> or a <period> shall match a single character. A bracket expression shall match a single character or a single collating element. An ERE matching a single character enclosed in parentheses shall match the same as the ERE without parentheses would have matched.

9.4.2 ERE Ordinary Characters

An ordinary character is an ERE that matches itself. An ordinary character is any character in the supported character set, except for the ERE special characters listed in ERE Special Characters. The interpretation of an ordinary character preceded by an unescaped <backslash> (`'\\'`) is undefined, except in the context of a bracket expression (see ERE Bracket Expression).

9.4.3 ERE Special Characters

An ERE special character has special properties in certain contexts. Outside those contexts, or when preceded by a <backslash>, such a character shall be an ERE that matches the special character itself. The

extended regular expression special characters and the contexts in which they shall have their special meaning are as follows:

`.[\(`
: The <period>, <left-square-bracket>, <backslash>, and <left-parenthesis> shall be special except when used in a bracket expression (see RE Bracket Expression). Outside a bracket expression, a <left-parenthesis> immediately followed by a <right-parenthesis> produces undefined results. A <left-square-bracket> that is unescaped and is not part of a bracket expression also produces undefined results.

`)`
: The <right-parenthesis> shall be special when matched with a preceding <left-parenthesis>, both outside a bracket expression.

`*+?{`
: The <asterisk>, <plus-sign>, <question-mark>, and <left-brace> shall be special except when used in a bracket expression (see RE Bracket Expression). Any of the following uses produce undefined results:

  - If these characters appear first in an ERE, or immediately following an unescaped <vertical-line>, <circumflex>, <dollar-sign>, or <left-parenthesis>
  - If a <left-brace> is not part of a valid interval expression (see EREs Matching Multiple Characters)

`|`
: The <vertical-line> is special except when used in a bracket expression (see RE Bracket Expression). A <vertical-line> appearing first or last in an ERE, or immediately following a <vertical-line> or a <left-parenthesis>, or immediately preceding a <right-parenthesis>, produces undefined results.

`^`
: The <circumflex> shall be special when used as an anchor (see ERE Expression Anchoring). The <circumflex> shall signify a non-matching list expression when it occurs first in a list, immediately following a <left-square-bracket> (see RE Bracket Expression).

`$`
: The <dollar-sign> shall be special when used as an anchor.

9.4.4 Periods in EREs

A <period> (`'.'`), when used outside a bracket expression, is an ERE that shall match any character in the supported character set except NUL.

9.4.5 ERE Bracket Expression

The rules for ERE Bracket Expressions are the same as for Basic Regular Expressions; see RE Bracket Expression.

9.4.6 EREs Matching Multiple Characters

The following rules shall be used to construct EREs matching multiple characters from EREs matching a single character:

1. A concatenation of EREs shall match the concatenation of the character sequences matched by each component of the ERE. A concatenation of EREs enclosed in parentheses shall match whatever the concatenation without the parentheses matches. For example, both the ERE `"cd"` and the ERE `"(cd)"` are matched by the third and fourth character of the string `"abcdefabcdef"`.

2. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character <plus-sign> (`'+'`), together with that <plus-sign> it shall match what one or more consecutive occurrences of the ERE would match. For example, the ERE `"b+(bc)"` matches the fourth to seventh characters in the string `"acabbbcde"`. And, `"[ab]+"` and `"[ab][ab]*"` are equivalent.

3. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character <asterisk> ('*'), together with that <asterisk> it shall match what zero or more consecutive occurrences of the ERE would match. For example, the ERE "b*c" matches the first character in the string "cabbbcde", and the ERE "b*cd" matches the third to seventh characters in the string "cabbbcdebbbbbbcdbc". And, "[ab]*" and "[ab][ab]" are equivalent when matching the string "ab".

4. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character <question-mark> ('?'), together with that <question-mark> it shall match what zero or one consecutive occurrences of the ERE would match. For example, the ERE "b?c" matches the second character in the string "acabbbcde".

5. When an ERE matching a single character or an ERE enclosed in parentheses is followed by an interval expression of the format "{m}", "{m,}", or "{m,n}", together with that interval expression it shall match what repeated consecutive occurrences of the ERE would match. The values of *m* and *n* are decimal integers in the range 0 <= *m*<= *n*<= {RE_DUP_MAX}, where *m* specifies the exact or minimum number of occurrences and *n* specifies the maximum number of occurrences. The expression "{m}" matches exactly *m* occurrences of the preceding ERE, "{m,}" matches at least *m* occurrences, and "{m,n}" matches any number of occurrences between *m* and *n*, inclusive.

   For example, in the string "abababccccccd" the ERE "c{3}" is matched by characters seven to nine and the ERE "(ab){2,}" is matched by characters one to six.

The behavior of multiple adjacent duplication symbols ('+', '*', '?', and intervals) produces undefined results.

An ERE matching a single character repeated by an '*', '?', or an interval expression shall not match a null expression unless this is the only match for the repetition or it is necessary to satisfy the exact or minimum number of occurrences for the interval expression.

9.4.7 ERE Alternation

Two EREs separated by the special character <vertical-line> ('|') shall match a string that is matched by either. For example, the ERE "a((bc)|d)" matches the string "abc" and the string "ad". Single characters, or expressions matching single characters, separated by the <vertical-line> and enclosed in parentheses, shall be treated as an ERE matching a single character.

9.4.8 ERE Precedence

The order of precedence shall be as shown in the following table:

ERE Precedence (from high to low)	
Collation-related bracket symbols	[==] [::] [..]
Escaped characters	\<special character>
Bracket expression	[]
Grouping	()
Single-character-ERE duplication	* + ? {m,n}
Concatenation	
Anchoring	^ $
Alternation	|

For example, the ERE "abba|cde" matches either the string "abba" or the string "cde" (rather than the string "abbade" or "abbcde", because concatenation has a higher order of precedence than alternation).

9.4.9 ERE Expression Anchoring

An ERE can be limited to matching expressions that begin or end a string; this is called "anchoring". The <circumflex> and <dollar-sign> special characters shall be considered ERE anchors when used anywhere outside a bracket expression. This shall have the following effects:

1. A <circumflex> (`'^'`) outside a bracket expression shall anchor the expression or subexpression it begins to the beginning of a string; such an expression or subexpression can match only a sequence starting at the first character of a string. For example, the EREs `"^ab"` and `"(^ab)"` match `"ab"` in the string `"abcdef"`, but fail to match in the string `"cdefab"`, and the ERE `"a^b"` is valid, but can never match because the `'a'` prevents the expression `"^b"` from matching starting at the first character.

2. A <dollar-sign> (`'$'`) outside a bracket expression shall anchor the expression or subexpression it ends to the end of a string; such an expression or subexpression can match only a sequence ending at the last character of a string. For example, the EREs `"ef$"` and `"(ef$)"` match `"ef"` in the string `"abcdef"`, but fail to match in the string `"cdefab"`, and the ERE `"e$f"` is valid, but can never match because the `'f'` prevents the expression `"e$"` from matching ending at the last character.

9.5 Regular Expression Grammar

Grammars describing the syntax of both basic and extended regular expressions are presented in this section. The grammar takes precedence over the text. See XCU *Grammar Conventions*.

9.5.1 BRE/ERE Grammar Lexical Conventions

The lexical conventions for regular expressions are as described in this section.

Except as noted, the longest possible token or delimiter beginning at a given point is recognized.

The following tokens are processed (in addition to those string constants shown in the grammar):

COLL_ELEM_SINGLE
Any single-character collating element, unless it is a **META_CHAR**.

COLL_ELEM_MULTI
Any multi-character collating element.

BACKREF
Applicable only to basic regular expressions. The character string consisting of a <backslash> character followed by a single-digit numeral, `'1'` to `'9'`.

DUP_COUNT
Represents a numeric constant. It shall be an integer in the range 0 <= **DUP_COUNT** <= {RE_DUP_MAX}. This token is only recognized when the context of the grammar requires it. At all other times, digits not preceded by a <backslash> character are treated as **ORD_CHAR**.

META_CHAR
One of the characters:

^
When found first in a bracket expression

-
When found anywhere but first (after an initial `'^'`, if any) or last in a bracket expression, or as the ending range point in a range expression

]
When found anywhere but first (after an initial `'^'`, if any) in a bracket expression

L_ANCHOR
Applicable only to basic regular expressions. The character `'^'` when it appears as the first character of a basic regular expression and when not **QUOTED_CHAR**. The `'^'` may be recognized as an anchor elsewhere; see BRE Expression Anchoring.

ORD_CHAR

A character, other than one of the special characters in **SPEC_CHAR**.

QUOTED_CHAR

In a BRE, one of the character sequences:

```
\^    \.    \*    \[    \$    \\
```

In an ERE, one of the character sequences:

```
\^    \.    \[    \$    \(    \)    \|
\*    \+    \?    \{    \\
```

R_ANCHOR

(Applicable only to basic regular expressions.) The character `'$'` when it appears as the last character of a basic regular expression and when not **QUOTED_CHAR**. The `'$'` may be recognized as an anchor elsewhere; see BRE Expression Anchoring.

SPEC_CHAR

For basic regular expressions, one of the following special characters:

`.`
: Anywhere outside bracket expressions

`\`
: Anywhere outside bracket expressions

`[`
: Anywhere outside bracket expressions

`^`
: When used as an anchor (see BRE Expression Anchoring)

`$`
: When used as an anchor

`*`
: Anywhere except first in an entire RE, anywhere in a bracket expression, directly following `"\(`", directly following an anchoring `'^'`

For extended regular expressions, shall be one of the following special characters found anywhere outside bracket expressions:

```
^    .    [    $    (    )    |
*    +    ?    {    \
```

(The close-parenthesis shall be considered special in this context only if matched with a preceding open-parenthesis.)

9.5.2 RE and Bracket Expression Grammar

This section presents the grammar for basic regular expressions, including the bracket expression grammar that is common to both BREs and EREs.

```
%token    ORD_CHAR QUOTED_CHAR DUP_COUNT


%token    BACKREF L_ANCHOR R_ANCHOR


%token    Back_open_paren  Back_close_paren
```

```
/*          '\('             '\)'        */


%token    Back_open_brace  Back_close_brace
/*          '\{'             '\}'         */


/* The following tokens are for the Bracket Expression
   grammar common to both REs and EREs. */


%token    COLL_ELEM_SINGLE COLL_ELEM_MULTI META_CHAR


%token    Open_equal Equal_close Open_dot Dot_close Open_colon Colon_close
/*           '[='       '=]'        '[.'     '.]'      '[:'       ':]'  */


%token    class_name
/* class_name is a keyword to the LC_CTYPE locale category */
/* (representing a character class) in the current locale */
/* and is only recognized between [: and :] */


%start    basic_reg_exp
%%


/* -----------------------------------------
   Basic Regular Expression
   -----------------------------------------
*/
basic_reg_exp  :          RE_expression
               | L_ANCHOR
               |                          R_ANCHOR
               | L_ANCHOR                 R_ANCHOR
               | L_ANCHOR RE_expression
               |          RE_expression R_ANCHOR
               | L_ANCHOR RE_expression R_ANCHOR
               ;
RE_expression  :               simple_RE
               | RE_expression simple_RE
               ;
simple_RE      : nondupl_RE
               | nondupl_RE RE_dupl_symbol
               ;
nondupl_RE     : one_char_or_coll_elem_RE
               | Back_open_paren RE_expression Back_close_paren
               | BACKREF
```

```
                ;
one_char_or_coll_elem_RE : ORD_CHAR
                | QUOTED_CHAR
                | '.'
                | bracket_expression
                ;
RE_dupl_symbol : '*'
                | Back_open_brace DUP_COUNT                Back_close_brace
                | Back_open_brace DUP_COUNT ','            Back_close_brace
                | Back_open_brace DUP_COUNT ',' DUP_COUNT Back_close_brace
                ;


/* ------------------------------------------
   Bracket Expression
   ------------------------------------------
*/
bracket_expression : '[' matching_list ']'
                | '[' nonmatching_list ']'
                ;
matching_list  : bracket_list
                ;
nonmatching_list : '^' bracket_list
                ;
bracket_list   : follow_list
                | follow_list '-'
                ;
follow_list    :             expression_term
                | follow_list expression_term
                ;
expression_term : single_expression
                | range_expression
                ;
single_expression : end_range
                | character_class
                | equivalence_class
                ;
range_expression : start_range end_range
                | start_range '-'
                ;
start_range    : end_range '-'
                ;
end_range      : COLL_ELEM_SINGLE
                | collating_symbol
                ;
collating_symbol : Open_dot COLL_ELEM_SINGLE Dot_close
                | Open_dot COLL_ELEM_MULTI Dot_close
                | Open_dot META_CHAR Dot_close
                ;
```

```
equivalence_class : Open_equal COLL_ELEM_SINGLE Equal_close
                  | Open_equal COLL_ELEM_MULTI Equal_close
                  ;
character_class : Open_colon class_name Colon_close
                  ;
```

The BRE grammar does not permit **L_ANCHOR** or **R_ANCHOR** inside `"\("` and `"\)"` (which implies that `'^'` and `'$'` are ordinary characters). This reflects the semantic limits on the application, as noted in BRE Expression Anchoring. Implementations are permitted to extend the language to interpret `'^'` and `'$'` as anchors in these locations, and as such, conforming applications cannot use unescaped `'^'` and `'$'` in positions inside `"\("` and `"\)"` that might be interpreted as anchors.

9.5.3 ERE Grammar

This section presents the grammar for extended regular expressions, excluding the bracket expression grammar.

Note:
The bracket expression grammar and the associated **%token** lines are identical between BREs and EREs. It has been omitted from the ERE section to avoid unnecessary editorial duplication.

```
%token  ORD_CHAR QUOTED_CHAR DUP_COUNT
%start  extended_reg_exp
%%


/* -----------------------------------------
   Extended Regular Expression
   -----------------------------------------
*/
extended_reg_exp   :                       ERE_branch
                   | extended_reg_exp '|' ERE_branch
                   ;
ERE_branch         :            ERE_expression
                   | ERE_branch ERE_expression
                   ;
ERE_expression     : one_char_or_coll_elem_ERE
                   | '^'
                   | '$'
                   | '(' extended_reg_exp ')'
                   | ERE_expression ERE_dupl_symbol
                   ;
one_char_or_coll_elem_ERE  : ORD_CHAR
                   | QUOTED_CHAR
                   | '.'
                   | bracket_expression
                   ;
ERE_dupl_symbol    : '*'
                   | '+'
                   | '?'
```

```
| '{' DUP_COUNT               '}'
| '{' DUP_COUNT ','          '}'
| '{' DUP_COUNT ',' DUP_COUNT '}'
;
```

The ERE grammar does not permit several constructs that previous sections specify as having undefined results. Additionally, there are some constructs which the grammar permits but which still give undefined results:

- **ORD_CHAR** preceded by an unescaped <backslash> character
- One or more *ERE_dupl_symbol*s appearing first in an ERE, or immediately following '|', '^', '(', or '$'
- '{' not part of a valid *ERE_dupl_symbol*
- '|' appearing first or last in an ERE, or immediately following '|' or '(', or immediately preceding ')'

Implementations are permitted to extend the language to allow these. Strictly Conforming applications cannot use such constructs.

return to top of page

[Main Index | XBD | XSH | XCU | XRAT]

<<< Previous Home Next >>>