-- --

IEEE P1003.2 Draft 11.2 – September 1991

```
#
# Here follows special characters.
<space>      IGNORE;IGNORE;IGNORE;<space>
# Other special characters follow here.
#
# Here comes the regular characters.
<a>          <a>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<A>          <a>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<a-acute>    <a>;<ACUTE>;<LOWER-CASE>;IGNORE
<A-acute>    <a>;<ACUTE>;<UPPER-CASE>;IGNORE
<a-grave>    <a>;<GRAVE>;<LOWER-CASE>;IGNORE
<A-grave>    <a>;<GRAVE>;<UPPER-CASE>;IGNORE
<ae>         <a><e>;<LIGATURE><LIGATURE>;<LOWER-CASE><LOWER-CASE>;IGNORE
<AE>         <a><e>;<LIGATURE><LIGATURE>;<UPPER-CASE><UPPER-CASE>;IGNORE
<b>          <b>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<B>          <b>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<c>          <c>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<C>          <c>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<ch>         <ch>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<Ch>         <ch>;<NO-ACCENT>;<PECULIAR>;IGNORE
<CH>         <ch>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
#
# As an example, the strings "Bach" and "bach" could be encoded (for
# compare purposes) as:
# "Bach"  <b>;<a>;<ch>;<LOW_VALUE>;<NO_ACCENT>;<NO_ACCENT>;\          2
#              <NO_ACCENT>;<LOW_VALUE>;<UPPER>;<LOWER>;<LOWER>;<NULL>     2
# "bach"  <b>;<a>;<ch>;<LOW_VALUE>;<NO_ACCENT>;<NO_ACCENT>;\          2
#              <NO_ACCENT>;<LOW_VALUE>;<LOWER>;<LOWER>;<LOWER>;<NULL>     2
#
# The two strings are equal in pass 1 and 2, but differ in pass 3.
#
# Further characters follow.
#
UNDEFINED    IGNORE;IGNORE;IGNORE;IGNORE
#
order_end
#
END LC_COLLATE
#
LC_MONETARY
int_curr_symbol     "USD "
currency_symbol     "$"
mon_decimal_point   "."
mon_grouping        3;0
positive_sign       ""
negative_sign       "-"
p_cs_precedes       1
n_sign_posn         0
END LC_MONETARY
#
LC_NUMERIC
copy "US_en.ASCII"                                                     1
```

```
END LC_NUMERIC
#
LC_TIME
abday     "Sun";"Mon";"Tue";"Wed";"Thu";"Fri";"Sat"
#
day       "Sunday";"Monday";"Tuesday";"Wednesday";\
          "Thursday";"Friday";"Saturday"
#
abmon     "Jan";"Feb";"Mar";"Apr";"May";"Jun";\
          "Jul";"Aug";"Sep";"Oct";"Nov";"Dec"
#
mon       "January";"February";"March";"April";\
          "May";"June";"July";"August";"September";\
          "October";"November";"December"
#
d_t_fmt   "%a %b %d %T %Z %Y\n"
END LC_TIME
#
LC_MESSAGES
yesexpr   "^([yY][[:alpha:]]*)|(OK)"                                1
#
noexpr    "^[nN][[:alpha:]]*"                                       1
END LC_MESSAGES
```

2.6 Environment Variables

Environment variables defined in this clause affect the operation of multiple utilities and applications. There are other environment variables that are of interest only to specific utilities. Environment variables that apply to a single utility only are defined as part of the utility description. See the Environment Variables subclause of the utility descriptions for information on environment variable usage.

The value of an environment variable is a string of characters, as described in 2.7 in POSIX.1 {8}.

Environment variable names used by the standard utilities shall consist solely of uppercase letters, digits, and the _ (underscore) from the characters defined in 2.4. The namespace of environment variable names containing lowercase letters shall be reserved for applications. Applications can define any environment variables with names from this namespace without modifying the behavior of the standard utilities.

If the following variables are present in the environment during the execution of an application or utility, they are given the meaning described below. They may be put into the environment, or changed, by either the implementation or the user. If they are defined in the utility's environment, the standard utilities assume they have the specified meaning. Conforming applications shall not set these environment variables to have meanings other than as described. See 7.2 and 3.12 for methods of accessing these variables.

HOME A pathname of the user's home directory.

LANG This variable shall determine the locale category for any 1
category not specifically selected via a variable starting with 1
LC_. **LANG** and the **LC_** variables can be used by applica- 1
tions to determine the language for messages and instructions, collating sequences, date formats, etc. Additional semantics of this variable, if any, are implementation defined.

LC_ALL This variable shall override the value of the **LANG** variable and the value of any of the other variables starting with **LC_**.

LC_COLLATE This variable shall determine the locale category for character collation information within bracketed regular expressions and for sorting. This environment variable determines the behavior of ranges, equivalence classes, and multicharacter collating elements. Additional semantics of this variable, if any, are implementation defined.

LC_CTYPE This variable shall determine the locale category for character handling functions. This environment variable shall determine the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters), the classification of characters (e.g., alpha, digit, graph), and the behavior of character classes. Additional semantics of this variable, if any, are implementation defined.

LC_MESSAGES This variable shall determine the locale category for processing affirmative and negative responses and the language and cultural conventions in which messages should be written. Additional semantics of this variable, if any, are implementation defined. The language and cultural conventions of diagnostic and informative messages whose format is unspecified by this standard should be affected by the setting of **LC_MESSAGES**.

LC_MONETARY This variable shall determine the locale category for monetary-related numeric formatting information. Additional semantics of this variable, if any, are implementation defined.

LC_NUMERIC This variable shall determine the locale category for numeric formatting (for example, thousands separator and radix character) information. Additional semantics of this variable, if any, are implementation defined.

LC_TIME This variable shall determine the locale category for date and time formatting information. Additional semantics of this variable, if any, are implementation defined.

LOGNAME The user's login name.

PATH The sequence of path prefixes that certain functions and utilities apply in searching for an executable file known only by a filename. The prefixes shall be separated by a colon (`:`). When a nonzero-length prefix is applied to this filename, a slash shall be inserted between the prefix and the filename. A zero-length prefix is an obsolescent feature that indicates the current working directory. It appears as two adjacent colons (`::`), as an initial colon preceding the rest of the list, or as a trailing colon following the rest of the list. A Strictly Conforming POSIX.2 Application shall use an actual pathname (such as `'.'`) to represent the current working directory in **PATH**. The list shall be searched from beginning to end, applying the filename to each prefix, until an executable file with the specified name and appropriate execution permissions is found. If the pathname being sought contains a slash, the search through the path prefixes shall not be performed. If the pathname begins with a slash, the specified path shall be resolved as described in 2.2.2.104. If **PATH** is unset or is set to null, the path search is implementation-defined.

SHELL A pathname of the user's preferred command language interpreter. If this interpreter does not conform to the shell command language in Section 3, utilities may behave differently than described in this standard.

TMPDIR A pathname of a directory made available for programs that need a place to create temporary files.

TERM The terminal type for which output is to be prepared. This information is used by utilities and application programs wishing to exploit special capabilities specific to a terminal. The format and allowable values of this environment variable are unspecified.

TZ Time-zone information. The format is described in POSIX.1 {8}8.1.1.

The environment variables **LANG**, **LC_ALL**, **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC**, and **LC_TIME** (**LC_***) provide for the support of internationalized applications. The standard utilities shall make use of these environment variables as described in this clause and the individual Environment Variables subclauses for the utilities. If these variables specify locale categories that are not based upon the same underlying code set, the results are unspecified.

For utilities used in internationalized applications, if the **LC_ALL** is not set in the environment or is set to the empty string, and if any of **LC_*** variables is not set in the environment or is set to the empty string, the operational behavior of the utility for the corresponding locale category shall be determined by the setting of the **LANG** environment variable. If the **LANG** environment variable is not set or is set to the empty string, the implementation-defined default locale shall be used.

If **LANG** (or any of the **LC_*** environment variables) contains the value `"C"`, or the value `"POSIX"`, the POSIX Locale shall be selected and the standard utilities shall behave in accordance with the rules in the 2.5.1 for the associated category.

If **LANG** (or any of the **LC_*** environment variables) begins with a slash, it shall be interpreted as the pathname of a file that was created in the output format used by the `localedef` utility; see 4.35.6.3. Referencing such a pathname shall result in that locale being used for the category indicated.

If **LANG** (or any of the **LC_*** environment variables) contains one of a set of implementation-defined values, the standard utilities shall behave in accordance with the rules in a corresponding implementation-defined locale description for the associated category.

If **LANG** (or any of the **LC_*** environment variables) contains a value that the implementation does not recognize, the behavior is unspecified.

Additional criteria for determining a valid locale name are implementation defined.

2.6.1 Environment Variables Rationale. *(This subclause is not a part of P1003.2)*

The standard is worded so that the specified variables *may* be provided to the application. There is no way that the implementation can guarantee that a utility will ever see an environment variable, as a parent process can change the environment for its children. The `env −i` command in this standard and the POSIX.1 {8} *exec* family both offer ways to remove any of these variables from the environment.

The language about locale implies that any utilities written in Standard C and conforming to POSIX.2 must issue the following call:

```
setlocale(LC_ALL, "")
```

If this were omitted, the C Standard {7} specifies that the C Locale would be used.

If any of the environment variables is invalid, it makes sense to default to an implementation-defined, consistent locale environment. It is more confusing for a user to have partial settings occur in case of a mistake. All utilities would then behave in one language/cultural environment. Furthermore, it provides a way of forcing the whole environment to be the implementation-defined default. Disastrous results could occur if a pipeline of utilities partially use the environment variables in different ways. In this case, it would be appropriate for utilities that use **LANG** and related variables to exit with an error if any of the variables are invalid. For example, users typing individual commands at a terminal might want `date` to work if **LC_MONETARY** is invalid as long as **LC_TIME** is valid. Since these are conflicting reasonable alternatives, POSIX.2 leaves the results unspecified if the locale environment variables would not produce a complete locale matching the user's specification.

The locale settings of individual categories cannot be truly independent and still guarantee correct results. For example, when collating two strings, characters must first be extracted from each string (governed by **LC_CTYPE**) before being mapped to collating elements (governed by **LC_COLLATE**) for comparison. That is, if **LC_CTYPE** is causing parsing according to the rules of a large, multibyte code set (potentially returning 20 000 or more distinct character code set values), but **LC_COLLATE** is set to handle only an 8-bit code set with 256 distinct characters, meaningful results are obviously impossible.

The **LC_MESSAGES** variable affects the language of messages generated by the standard utilities. This standard does not provide a means whereby applications can easily be written to perform similar feats. Future versions of POSIX.1 {8} and POSIX.2 are expected to provide both functions and utilities to accomplish multilanguage messaging (using message catalogs), but such facilities were not ready for standardization at the time the initial versions of the standards were developed.

This clause is not a full list of all environment variables, but only those of importance to multiple utilities. Nevertheless, to satisfy some members of the balloting group, here is a list of the other environment variable symbols mentioned in this standard:

Variable	Utility	Variable	Utility	
CDPATH	`cd`	**MAKEFLAGS**	`make`	
COLUMNS	`ls`	**OPTARG**	`getopts`	
DEAD	`mailx`	**OPTIND**	`getopts`	
IFS	`sh`	**PRINTER**	`lp`	1
LPDEST	`lp`	**PS 1**	`sh`	
MAIL	`sh`	**PS 2**	`sh`	
MAILRC	`mailx`			

The description of **PATH** is similar to that in POSIX.1 {8}, except:

— The behavior of a null prefix is marked obsolescent in favor of using a real pathname. This was done at the behest of some members of the balloting group, who apparently felt it offered a more secure environment, where the current directory would not be selected unintentionally.

— The POSIX.1 {8} *exec* description requires an implementation-defined path search when **PATH** is ''not present.'' POSIX.2 spells out that this means ''unset or set to null.'' Many implementations historically have used a default value of `/bin` and `/usr/bin`. POSIX.2 does not mandate that this default path be identical to that retrieved from `getconf _CS_PATH` because it is likely that a transition to POSIX.2 conformance will see the newly-standardized utilities in another directory that needs to be isolated from some historical applications.

— The POSIX.1 {8} **PATH** description is ambiguous about whether an ''executable file'' means one that has the appropriate permissions for the searching process to execute it. One reading would say that a file with any of the execution bits set on would satisfy the search and that an [EACCES] could be returned at that point. This is not the way historical systems work and POSIX.2 has clarified it to mean that the path search will continue until it finds the name with the execute permissions that would allow the process to execute it. (The case of the [ENOEXEC] error is handled in the text of 3.9.1.1.)

The terminology ''beginning to end'' is used in **PATH** to avoid the noninternationalized ''left to right.'' There is no way to have a colon character embedded within a pathname that is part of the **PATH** variable string. Colon is not a member of the portable filename character set, so this should not be a problem. A portable application can retrieve a default **PATH** value (that will allow access to all the standard utilities) from the system using the command:

```
getconf _CS_PATH
```

See the rationale with `command` for an example of using this.

The **SHELL** variable names the user's preferred shell; it is a guide to applications. There is no direct requirement that that shell conform to this standard—that decision should rest with the user. It is the intention of the developers of this standard that alternative shells be permitted, if the user chooses to develop or acquire one. An operating system that builds its shell into the ''kernel'' in such a manner that alternative shells would be impossible does not conform to the spirit of the standard.

The following environment variables are not currently used by the standard utilities (although they may be by future UPE utilities). Implementations should reserve the names for the following purposes:

EDITOR The name of the user's preferred text file editor. The value of this variable is the name of a utility: either a pathname containing a slash, or a filename to be located using the **PATH** environment variable.

VISUAL The name of the user's preferred ''visual,'' or full-screen, text file editor. The value of this variable is the name of a utility: either a pathname containing a slash, or a filename to be located using the **PATH** environment variable.

The decision to restrict conforming systems to the use of digits, uppercase letters, and underscores for environment variable names allows applications to use lowercase letters in their environment variable names without conflicting with any conforming system.

PROCLANG was added to an earlier draft for internationalized applications, but was removed from the standard because the working group determined that it was not of use.

USER was removed from an earlier draft because it was an unreasonable duplication of **LOGNAME** .

2.7 Required Files

The following directories shall exist on conforming systems and shall be used as described. Strictly Conforming POSIX.2 Applications shall not assume the ability to create files in any of these directories.

`/` The root directory.

`/dev` Contains `/dev/null` and `/dev/tty`, described below.

The following directory shall exist on conforming systems and shall be used as described.

`/tmp` A directory made available for programs that need a place to create temporary files. Applications shall be allowed to create files in this directory, but shall not assume that such files are preserved between invocations of the application.

The following files shall exist on conforming systems and shall be both readable and writable.

`/dev/null` An infinite data source/sink. Data written to `/dev/null` is discarded. Reads from `/dev/null` always return end-of-file (EOF).

`/dev/tty` In each process, a synonym for the controlling terminal associated with the process group of that process, if any. It is useful for programs or shell procedures that wish to be sure of writing messages to or reading data from the terminal no matter how output has been redirected.

2.7.1 Required Files Rationale. *(This subclause is not a part of P1003.2)*

A description of the historical `/usr/tmp` was omitted, removing any concept of differences in emphasis between the `/` and `/usr` versions. The descriptions of `/bin`, `/usr/bin`, `/lib`, and `/usr/lib` were omitted because they are not useful for applications. In an early draft, a distinction was made between *system* and *application* directory usage, but this was not found to be useful.

In Draft 8, `/`, `/dev`, `/local`, `/usr/local`, and `/usr/man` were removed. The directories `/` and `/dev` were restored in Draft 9. It was pointed out by several balloters that the notion of a hierarchical directory structure is key to other information presented in later sections of the standard. (Previously, some had argued that special devices and temporary files could conceivably be handled without a directory structure on some implementations. For example, the system could treat the characters ‘‘`/tmp`’’ as a special token that would store files using some non-POSIX file system structure. This notion was rejected by the working group, which requires that all the files in this clause be implemented via POSIX file systems.)

The `/tmp` directory is retained in the standard to accommodate historical applications that assume its availability. Future implementations are encouraged to provide suitable directory names in **TMPDIR** and future applications are encouraged to use the contents of **TMPDIR** for creating temporary files.

The standard files `/dev/null` and `/dev/tty` are required to be both readable and writable to allow applications to have the intended historical access to these files.

2.8 Regular Expression Notation

Editor's Note: The entire rationale for this clause appears at the end of the clause.

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 2.8.3 shall apply to most utilities supporting regular expressions. Some utilities, instead, support the Extended Regular Expressions (ERE) described in 2.8.4; 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 7.3.

2.8.1 Regular Expression Definitions

For the purposes of this clause, the following definitions apply.

2.8.1.1 entire regular expression: The concatenated set of one or more BREs or EREs that make up the pattern specified for string selection.

2.8.1.2 matched: A sequence of zero or more characters is said to be matched by a BRE or ERE when the characters in the sequence corresponds 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.

The search for a matching sequence shall start at the beginning of a string and stop 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 shall be matched. For example: the BRE `bb*` matches the second through fourth characters of `abbbc`, and the ERE `(wee|week)(knights|night)` matches all ten characters of `weeknights`.

Consistent with the whole match being the longest of the leftmost matches, each 1
subpattern, from left to right, shall match the longest possible string. For this 2
purpose, a null string shall be considered to be longer than no match at all. For 2
example, matching the BRE `\(.*\).*` against `abcdef`, the subexpression (`\1`) is 2
`abcdef`, and matching the BRE `\(a*\)*` against `bc`, the subexpression (`\1`) is the 2
null string. 2

When a multicharacter collating element in a bracket expression (see 2.8.3.2) is 1
involved, the longest sequence shall be measured in characters consumed from 1
the string to be matched; i.e., the collating element counts not as one element, but 1
as the number of characters it matches. 1

2.8.1.3 BRE [ERE] matching a single character: A BRE or ERE that matches either a single character or a single collating element.

Only a BRE or ERE of this type that includes a bracket expression (see 2.8.3.2) can 1
match a collating element. 1

2.8.1.4 BRE [ERE] matching multiple characters: A BRE or ERE that matches 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 1
BRE (ERE) special character s. 1

2.8.2 Regular Expression General Requirements

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

The use of regular expressions is generally associated with text processing; i.e.,
REs (BREs and EREs) operate on text strings; i.e., zero or more characters followed
by an end-of-string delimiter (typically NUL). Some utilities employing regular
expressions limit the processing to lines; i.e., zero or more characters followed by
a `<newline>`. In the regular expression processing described in this standard,
the `<newline>` character is regarded as an ordinary character. This standard 1
specifies within the individual descriptions of those standard utilities employing 1
regular expressions whether they permit matching of `<newline>`s; if not stated 1
otherwise, the use of literal `<newline>`s or any escape sequence equivalent pro- 1
duces undefined results. 1

The interfaces specified in this standard 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 1
may designate the end of the text string for the purposes of matching. 1

When a standard utility or function that uses regular expressions specifies that pattern matching shall be performed without regard to the case (upper- or lower-) 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.

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

This clause 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.

2.8.3 Basic Regular Expressions

2.8.3.1 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.

2.8.3.1.1 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 2.8.3.1.2.

The interpretation of an ordinary character preceded by a backslash (\) is undefined, except for:

(1) The characters), (, {, and }.

(2) The digits 1 through 9 (see 2.8.3.3).

(3) A character inside a bracket expression.

2.8.3.1.2 BRE Special Characters

A *BRE special character* has special properties in certain contexts. Outside of those contexts, or when preceded by a backslash, such a character shall be a BRE that matches the special character itself. The BRE special characters and the contexts in which they have their special meaning are:

. [\ The period, left-bracket, and backslash shall be special except when used in a bracket expression (see 2.8.3.2). An expression containing a [that is not preceded by a backslash and is not part of a bracket expression produces undefined results.

* The asterisk is special except when used

— In a bracket expression,

— As the first character of an entire BRE (after an initial ^, if any), or

— As the first character of a subexpression (after an initial ^, if any); see 2.8.3.3.

^ The circumflex shall be special when used

— As an anchor (see 2.8.3.5) or,

— As the first character of a bracket expression (see 2.8.3.2).

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

2.8.3.1.3 Periods in BREs

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

2.8.3.2 RE Bracket Expression

A bracket expression (an expression enclosed in square brackets, `[]`) is an RE that matches a single collating element contained in the nonempty set of collating elements represented by the bracket expression.

The following rules and definitions apply to bracket expressions:

(1) A *bracket expression* is either a matching list expression or a nonmatching list expression. It consists of one or more expressions: collating elements, collating symbols, equivalence classes, character classes, or range expressions. Strictly Conforming POSIX.2 Applications shall not use range expressions, but conforming implementations shall support regular expressions containing range expressions. The right-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-bracket for a collating symbol, equivalence class, or character class). The special characters

`.   *   [   \`

(period, asterisk, left-bracket, and backslash, respectively) shall lose their special meaning within a bracket expression.

The character sequences

`[.   [=   [:`

(left-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 one of the expressions represented in the list. The first character in the list shall not be the circumflex. For example, `[abc]` is an RE that matches any of `a`, `b`, or `c`.

(3) A *nonmatching list* expression begins with a circumflex (^), and specifies a list that shall match any character or collating element except for the

expressions represented in the list after the leading circumflex. For example, `[^abc]` is an RE that matches any character or collating element except `a`, `b`, or `c`. The circumflex shall have this special meaning only when it occurs first in the list, immediately following the left-bracket.

(4) A *collating symbol* is a collating element enclosed within bracket-period (`[. .]`) delimiters. Collating elements are defined as described in 2.5.2.2.4. Multicharacter collating elements shall be represented as collating symbols when it is necessary to distinguish them from a list of the individual characters that make up the multicharacter collating element. For example, if the string `ch` is a collating element in the current collation sequence with the associated collating symbol `<ch>`, the expression `[[.ch.]]` shall be treated as an RE matching the character sequence `ch`, while `[ch]` shall be treated as an RE matching `c` or `h`. Collating symbols shall be recognized only inside bracket expressions. This implies that the RE `[[.ch.]]*c` shall match the first through fifth character in the string `chchch`. If the string is not a collating element in the current collating sequence definition, or if the collating element has no characters associated with it (e.g., see the symbol `<HIGH>` in the example collation definition shown in 2.5.2.2.4), the symbol shall be treated as an invalid expression.

(5) An *equivalence class expression* shall represent the set of collating elements belonging to an equivalence class, as described in 2.5.2.2.4. 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 (`[= =]`) delimiters. For example, if `a`, `à`, and `â` belong to the same equivalence class, then `[[=a=]b]`, `[[=à=]b]`, and `[[=â=]b]` shall each be 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 set of characters belonging to a character class, as defined in the LC_CTYPE category in the current locale. All character classes specified in the current locale shall be recognized. A character class expression shall be expressed as a character class name enclosed within ‘‘bracket-colon’’ (`[: :]`) delimiters.

Strictly conforming POSIX.2 applications shall only use the following character class expressions, which shall be supported on all conforming implementations:

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

(7) A *range expression* represents the set of collating elements that fall between two elements in the current collation sequence, inclusively. It shall be expressed as the starting point and the ending point separated

by a hyphen (–).

Range expressions shall not be used in Strictly Conforming POSIX.2 Applications because their behavior is dependent on the collating sequence. Range expressions shall be supported by conforming implementations.

In the following, all examples assume the collation sequence specified for the POSIX Locale, unless another collation sequence is specifically defined.

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. The ending range point shall collate equal to or higher than the starting range point; otherwise the expression shall be treated as invalid. The order used is the order in which the collating elements are specified in the current collation definition. One-to-many mappings (see 2.5.2.2) shall not be performed. For example, assuming that the character eszet (β) is placed in the basic collation sequence after `r` and `s`, but before `t`, and that it maps to the sequence `ss` for collation purposes, then the expression `[r-s]` matches only `r` and `s`, but the expression `[s-t]` matches `s`, β, or `t`.

The interpretation of range expressions where the ending range point also is the starting range point of a subsequent range expression is undefined.

The hyphen 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 `-`; the expressions `[^-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 invalid, because the letter `a` follows the symbol `-` in the POSIX Locale. To use a hyphen 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 bracket or any character or collating element that collates between hyphen and 0, inclusive.

2.8.3.3 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 2.8.3.5. Subexpressions can be arbitrarily nested.

(3) The *backreference* 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 `\(` and ends with the corresponding paired `\)`]. The expression is invalid if less than *n* subexpressions precede the *n*. For example, the expression `^\(.*\)\1$` matches a line consisting of two adjacent appearances of the same string, and the expression `\(a\)*\1` fails to match `a`.

(4) When a BRE matching a single character, a subexpression, or a backreference 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 backreference 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* shall be decimal integers in the range $0 \le m \le n \le$ {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 through nine, the BRE `\(ab\)\{4,\}` is not matched at all, and the BRE `c\{1,3\}d` is matched by characters ten through thirteen.

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

2.8.3.4 BRE Precedence

The order of precedence shall be as shown in Table 2-12, from high to low.

2.8.3.5 BRE Expression Anchoring

A BRE can be limited to matching strings that begin or end a line; 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 circumflex as an anchor when

Table 2-12 – BRE Precedence 1

collation-related bracket symbols	`[= =]` `[: :]` `[. .]`	1
escaped characters	`\`<*special character* >	1
bracket expression	`[ ]`	1
subexpressions/backreferences	`\( \)` `\`*n*	1
single-character- BRE duplication	`*` `\{`*m*`,`*n*`\}`	1
concatenation		1
anchoring	`^` `$`	1

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

(2) A dollar-sign (`$`) shall be an anchor when used as the last character of an 1
entire BRE. The implementation may treat a dollar-sign as an anchor 1
when used as the last character of a subexpression. The dollar-sign shall 1
anchor the expression (or optionally subexpression) to the end of the 1
string being matched; the dollar-sign can be said to match the ‘‘end-of- 1
string’’ following the last character. 1

(3) A BRE anchored by both `^` and `$` shall match only an entire string. For 2
example, the BRE `^abcdef$` matches strings consisting only of `abcdef`. 1

2.8.4 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.

2.8.4.1 EREs Matching a Single Character or Collating Element

An ERE ordinary character, a special character preceded by a backslash, or a 1
period shall match a single character. A bracket expression shall match a single 1
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.

2.8.4.1.1 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 2
listed in 2.8.4.1.2. The interpretation of an ordinary character preceded by a
backslash (\) is undefined.

2.8.4.1.2 ERE Special Characters

An *ERE special character* has special properties in certain contexts. Outside of 1
those contexts, or when preceded by a backslash, such a character shall be an ERE 1
that matches the special character itself. The extended regular expression special
characters and the contexts in which they shall have their special meaning are:

`. [ \ (` The period, left-bracket, backslash, and left-parenthesis are spe- 1
cial except when used in a bracket expression (see 2.8.3.2).

`* + ? {` The asterisk, plus-sign, question-mark, and left-brace are special
except when used in a bracket expression (see 2.8.3.2). Any of the 2
following uses produce undefined results: 2

- — If these characters appear first in an ERE , or immediately fol-
lowing a vertical-line, circumflex, or left-parenthesis.
- — If a left-brace is not part of a valid interval expression. 1

`|` The vertical-line is special except when used in a bracket expres-
sion (see 2.8.3.2). A vertical-line appearing first or last in an ERE ,
or immediately following a vertical-line or a left-parentheses, pro- 1
duces undefined results. 1

`^` The circumflex shall be special when used 1

- — As an anchor (see 2.8.4.6) or, 1
- — As the first character of a bracket expression (see 2.8.3.2). 1

`$` The dollar-sign shall be special when used as an anchor. 1

2.8.4.1.3 Periods in ERE s

A period (`.`), when used outside of a bracket expression, is an ERE that shall
match any character in the supported character set except NUL. 1

2.8.4.2 ERE Bracket Expression

The rules for ERE Bracket Expressions are the same as for Basic Regular Expressions; see 2.8.3.2.

2.8.4.3 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 1
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 a concatenation of EREs 1
enclosed in parentheses is followed by the special character plus-sign (+), 1
together with that plus-sign it shall match what one or more consecutive 2
occurrences of the ERE would match. For example, the ERE `b+(bc)` 2
matches the fourth through seventh characters in the string `acabbbcde`. 2
And, `[ab]+` and `[ab][ab]*` are equivalent. 2

(3) When an ERE matching a single character, or a concatenation of EREs 1
enclosed in parentheses is followed by the special character asterisk (*), 1
together with that asterisk it shall match what zero or more consecutive 2
occurrences of the ERE would match. For example, the ERE `b*c` matches 2
the first character in the string `cabbbcde`, and the ERE `b*cd` matches
the third through seventh characters in the string
`cabbbcdebbbbbbcdbc`. And, `[ab]*` and `[ab][ab]` are equivalent when 2
matching the string `ab`. 2

(4) When an ERE matching a single character, or a concatenation of EREs 1
enclosed in parentheses is followed by the special character question- 1
mark (`?`), together with that question-mark it shall match what zero or 2
one consecutive occurrences of the ERE would match. For example, the 2
ERE `b?c` matches the second character in the string `acabbbcde`.

(5) When an ERE matching a single character, or a concatenation of EREs 1
enclosed in parentheses is followed by an *interval expression* of the for- 1
mat `{`m`}`, `{`m`,}`, or `{`m`,`n`}`, together with that interval expression it 1
shall match what repeated consecutive occurrences of the ERE would 2
match. The values of m and n shall be decimal integers in the range $0 \leq$ 1
$m \leq n \leq$ {RE_DUP_MAX}, where m specifies the exact or minimum number 1
of occurrences and n specifies the maximum number of occurrences. The
expression `{`m`}` shall match exactly m occurrences of the preceding ERE,
`{`m`,}` shall match at least m occurrences, and `{`m`,`n`}` shall match any
number of occurrences between m and n, inclusive. 1

For example, in the string `ababababccccccd` the ERE `c{3}` is matched by 1
characters seven through nine, and the ERE `(ab){2,}` is matched by 2
characters one through six. 2

The behavior of multiple adjacent duplication symbols (`+`, `*`, `?`, and intervals) pro- 1
duces undefined results. 1

2.8.4.4 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 bar and enclosed in parentheses, shall be treated as an ERE matching a single character.

2.8.4.5 ERE Precedence

The order of precedence shall be as shown in Table 2-13, from high to low.

Table 2-13 – ERE Precedence

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` (because concatenation has a higher order of precedence than alternation).

2.8.4.6 ERE Expression Anchoring

An ERE can be limited to matching strings that begin or end a line; this is called *anchoring*. The circumflex and dollar-sign special characters shall be considered ERE anchors in the following contexts:

(1) A circumflex (`^`) shall be an anchor when used anywhere outside a bracket expression. The circumflex shall anchor the (sub)expression to the beginning of a string; only sequences starting at the first character of a string shall be matched by the ERE. For example, the EREs `^ab` and `(^ab)` match `ab` in the string `abcdef`, but fail to match in the string `cdefab`.

(2) A dollar-sign (`$`) shall be an anchor when used anywhere outside a bracket expression. It shall anchor the expression 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) An ERE anchored by both `^` and `$` shall match only an entire string. For example, the EREs `^abcdef$` and `(^abcdef$)` match strings consisting only of `abcdef`.