IEEE P1003.2 Draft 11.2 – September 1991

2.8.5 Regular Expression Grammar

Grammars describing the syntax of both basic and extended regular expressions are presented in this subclause. See the grammar conventions in 2.1.2.

2.8.5.1 BRE/ERE Grammar Lexical Conventions

The lexical conventions for regular expressions shall be as described in this subclause.

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

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

`COLL_ELEM` Shall be any single-character collating element, unless it is a `META_CHAR`.

`BACKREF` (Applicable only to basic regular expressions.) Shall be the character string consisting of `'\'` followed by a single-digit numeral, 1 through 9.

`DUP_COUNT` Shall represent a numeric constant. It shall be an integer in the range 0 ≤ `DUP_COUNT` ≤ {RE_DUP_MAX}. This token shall only be recognized when the context of the grammar requires it. At all other times, digits not preceded by `'\'` shall be treated as `ORD_CHAR`.

`META_CHAR` Shall be 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.) Shall be 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 2.8.3.5.

`ORD_CHAR` Shall be a character, other than one of the special characters in `SPEC_CHAR`.

`QUOTED_CHAR` Shall be one of the character sequences:

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

`R_ANCHOR` (Applicable only to basic regular expressions). Shall be 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 2.8.3.5. 1

`SPEC_CHAR` For basic regular expressions, shall be one of the following special characters:

`.`	Anywhere outside bracket expressions
`\`	Anywhere outside bracket expressions
`[`	Anywhere outside bracket expressions
`^`	When an anchor; see 2.8.3.5 2
`$`	When an anchor; see 2.8.3.5 2
`*`	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.) 2

2.8.5.2 RE and Bracket Expression Grammar

This subclause 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 SPEC_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 META_CHAR                                          1

%token   Open_equal Equal_close Open_dot Dot_close Open_colon Colon_close      1
/*         '[='         '=]'      '[.'     '.]'       '[:'        ':]'  */     1

%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                              1
                   ;
nondupl_RE         : one_character_RE
                   | Back_open_paren RE_expression Back_close_paren
                   | Back_open_paren Back_close_paren
                   | BACKREF
                   ;
/*                                                                          1
                     Note:  This grammar does not permit L_ANCHOR or        1
                     R_ANCHOR inside \( and \) (which implies that ^ and $  1
                     are ordinary characters).  This reflects the semantic  1
                     limits on the application, as noted in 2.8.3.5.        1
                     Implementations are permitted to extend the language to 1
                     interpret ^ and $ as anchors in these locations, and as 1
                     such portable applications shall not use unescaped ^   1
                     and $ in positions inside \( and \) that might be      1
                     interpreted as anchors.                                1
*/                                                                          1
one_character_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 '-'                                 1
                   ;

follow_list        :             expression_term
                   | follow_list expression_term
                   ;

expression_term    : single_expression
                   | range_expression
                   ;

single_expression  : end_range
                   | character_class                                 1
                   ;

range_expression   : start_range end_range
                   | start_range '-'
                   ;

start_range        : end_range '-'
                   ;

end_range          : COLL_ELEM
                   | collating_symbol
                                                                     2
                   ;

collating_symbol   : Open_dot COLL_ELEM Dot_close
                   | Open_dot META_CHAR Dot_close
                   ;

equivalence_class  : Open_equal COLL_ELEM Equal_close
                   ;

character_class    : Open_colon class_name Colon_close               1
                   ;
```

2.8.5.3 ERE Grammar

This subclause 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 subclause to avoid unnecessary editorial duplication.

```
%token   ORD_CHAR QUOTED_CHAR SPEC_CHAR DUP_COUNT

%start   extended_reg_exp
```

```
%%

/*                   -------------------------------------------
                     Extended Regular Expression
                     -------------------------------------------
*/

extended_reg_exp  :                       anchored_ERE
                  |                       nonanchored_ERE
                  | extended_reg_exp '|' nonanchored_ERE
                  | extended_reg_exp '|' anchored_ERE
                  ;

anchored_ERE      : '^' nonanchored_ERE
                  | '^' nonanchored_ERE '$'
                  |     nonanchored_ERE '$'
                  | '^'
                  |                      '$'
                  | '^'                  '$'
                  ;

nonanchored_ERE   :                 ERE_expression
                  | nonanchored_ERE ERE_expression
                  ;

ERE_expression    : one_character_ERE
                  | '(' extended_reg_exp ')'
                  | ERE_expression ERE_dupl_symbol
                  ;

one_character_ERE : ORD_CHAR
                  | '\' SPEC_CHAR
                  | '.'
                  | bracket_expression
                  ;

ERE_dupl_symbol   : '*'
                  | '+'
                  | '?'
                  | '{' DUP_COUNT               '}'
                  | '{' DUP_COUNT ','           '}'
                  | '{' DUP_COUNT ',' DUP_COUNT '}'
                  ;
```

2.8.6 Regular Expression Notation Rationale. *(This subclause is not a part of P1003.2)*

Editor's Note: Some of the text and headings of this rationale have been rear- 1
ranged. Moved text has not been diffmarked unless it changed. 1

Rather than repeating the description of regular expressions for each utility supporting REs, the working group preferred a common, comprehensive description of regular expressions in one place. The most common behavior is described here, and exceptions or extensions to this are documented for the respective utilities, if appropriate.

The Basic Regular Expression corresponds to the `ed` or historical `grep` type, and the Extended Regular Expression corresponds to the historical `egrep` type (now `grep -E`).

The text is based on the `ed` description and substantially modified, primarily to aid developers and others in the understanding of the capabilities and limitations of regular expressions. Much of this was influenced by the internationalization requirements.

It should be noted that the definitions in this clause do not cover the `tr` utility (see 4.64); the `tr` syntax does not employ regular expressions.

The specification of regular expressions are particularly important to internationalization, because pattern matching operations are very basic operations in business and other operations. The syntax and rules of regular expressions are intended to be as intuitive as possible, to make them easy to understand and use. The historical rules and behavior do not provide that capability to non-English-language users, and does not provide the necessary support for commonly used characters and language constructs. It was necessary to provide extensions to the historical regular expression syntax and rules, to accommodate other languages. Such modifications were proposed by the UniForum Technical Committee Subcommittee on Internationalization and accepted by the working group. As they are limited to bracket expressions, the rationale for these modifications can be found in 2.8.6.3.2.

2.8.6.1 Regular Expression Definitions Rationale. *(This subclause is not a part of P1003.2)*

The definition of which sequence is matched when several are possible is based on
the leftmost-longest rule historically used by deterministic recognizers. This rule 1
is much easier to define and describe, and arguably more useful, than the first-match rule historically used by nondeterministic recognizers. It is thought that dependencies on the choice of rule are rare; carefully-contrived examples are needed to demonstrate the difference.

A formal expression of the leftmost-longest rule is: 1

> The search is performed as if all possible suffixes of the string were tested for a prefix matching the pattern; the longest suffix containing a matching prefix is chosen, and the longest possible matching prefix of the chosen suffix is identified as the matching sequence.

It is possible to determine what strings correspond to subexpressions by recur- 1
sively applying the leftmost longest rule to each subexpression, but only with the 1
proviso that the overall match is leftmost longest (see 2.8.1.2). For example, 1
matching `\(ac*\)c*d[ac]*\1` against `acdacaaa` should match `acdacaaa` (with 1
`\1=a`); simply matching the longest match for `\(ac*\)` would yield `\1=ac`, but 1
the overall match would be smaller (`acdac`). In principle, the implementation 1
must examine every possible match and among those that yield the leftmost long- 1
est total matches, pick the one that does the longest match for the leftmost subex- 1
pression and so on. Note that this means that matching by subexpressions is con- 1
text dependent: a subexpression within a larger RE may match a different string 1
from the one it would match as an independent RE, and two instances of the same 1

subexpression within the same larger RE may match different lengths even in similar sequences of characters. For example, in the ERE `(a.*b)(a.*b)`, the two identical subexpressions would match four and six characters, respectively, of `accbaccccb`. Thus, it is not possible to hierarchically decompose the matching problem into smaller, independent, matching problems.

Matching is 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 code set, 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 definition of ‘‘single character’’ has been expanded to include also collating elements consisting of two or more characters; this expansion is applicable only when a bracket expression is included in the BRE or ERE. An example of such a collating element may be the Dutch ‘‘ij’’, which collates as a ‘‘y.’’ In some encodings, a ligature ‘‘i with j’’ exists *as a character*, and would represent a single-character collating element. In another encoding, no such ligature exists, and the two-character sequence ‘‘ij’’ is defined as a multicharacter collating element. Outside brackets, the ‘‘ij’’ is treated as a two-character RE and will match the same characters in a string. Historically, a bracket expression only matched a single character. If, however, the bracket expression defines, for example, a range that includes ‘‘ij’’, then this particular bracket expression will also match a sequence of the two characters ‘‘i’’ and ‘‘j’’ in the string.

2.8.6.2 Regular Expression General Requirements Rationale. *(This subclause is not a part of P1003.2)*

Historically, most regular expression implementations only match lines, not strings. However, that is more an effect of the usage than of an inherent feature of regular expressions itself. Consequently, POSIX.2 does not regard `<newline>`s as special; they are ordinary characters, and both a period and a nonmatching list can match them. Those utilities (like `grep`) that do not allow `<newline>`s to match are responsible for eliminating any `<newline>` from strings before matching against the RE. The *regcomp*() function, however, can provide support for such processing without violating the rules of this clause.

The definition of case-insensitive processing is intended to allow matching of multicharacter collating elements as well as characters. For instance, as each character in the string is matched using both its cases, the RE `[[.Ch.]]`, when matched against `char`, is in reality matched against `ch`, `Ch`, `cH`, and `CH`.

Some implementations of `egrep` have had very limited flexibility in handling complex extended regular expressions. POSIX.2 does not attempt to define the complexity of a BRE or ERE, but does place a lower limit on it—any regular expression must be handled, as long as it can be expressed in 256 bytes or less. (Of course, this does not place an upper limit on the implementation.) There are existing programs using a nondeterministic-recognizer implementation that should have no difficulty with this limit. It is possible that a good approach would be to attempt to use the faster, but more limited, deterministic recognizer for simple expressions and to fall back on the nondeterministic recognizer for those expressions requiring it. Nondeterministic implementations must be careful to observe the 2.8.1.2 rules on which match is chosen; the longest match, not the first match, starting at a given character is used.

The term ‘‘invalid’’ highlights a difference between this clause and some others: 1
POSIX.2 frequently avoids mandating of errors for syntax violations because they 1
can be used by implementors to trigger extensions. However, the authors of the 1
internationalization features of regular expressions desired to mandate errors for 1
certain conditions to identify usage problems or nonportable constructs. These 1
are identified within this rationale as appropriate. The remaining syntax viola- 1
tions have been left implicitly or explicitly undefined. For example, the BRE con- 1
struct `\{1,2,3\}` does not comply with the grammar. A conforming application 1
cannot rely on it producing an error nor matching the literal characters 1
`\{1,2,3\}`. The term ‘‘undefined’’ was used in favor of ‘‘unspecified’’ because 1
many of the situations are considered errors on some implementations and it was 1
felt that consistency throughout the clause was preferable to mixing undefined 1
and unspecified. 1

2.8.6.3 Basic Regular Expressions Rationale. *(This subclause is not a part of P1003.2)*

2.8.6.3.1 BREs Matching a Single Character or Collating Element Rationale. *(This subclause is not a part of P1003.2)*

2.8.6.3.2 RE Bracket Expression Rationale. *(This subclause is not a part of P1003.2)*

If a bracket expression must specify both `-` and `]`, then the `]` must be placed first (after the `^`, if any) and the `-` last within the bracket expression.

Range expressions are, historically, an integral part of regular expressions. However, the requirements of ‘‘natural language behavior’’ and portability does conflict: ranges must be treated according to the current collating sequence, and include such characters that fall within the range based on that collating sequence, regardless of character values. This, however, means that the interpretation will differ depending on collating sequence. If, for instance, one collating sequence defines ‘`ä`’’as a variant of ‘`a`’’,while another defines it as a letter following ‘`z`’’,then the expression `[ä-z]` is valid in the first language and invalid in the second. This kind of ambiguity should be avoided in portable applications, and therefore the working group elected to state that ranges must not be used in strictly conforming applications; however, implementations must support them.

Some historical implementations allow range expressions where the ending range point of one range is also the starting point of the next (for instance `[a-m-o]`). This behavior should not be permitted, but to avoid breaking existing implementations, it is now *undefined* whether it is a valid expression, and how it should be interpreted.

Current practice in `awk` and `lex` is to accept escape sequences in bracket expressions as per Table 2-15, while the normal regular expression behavior is to regard such a sequence as consisting of two characters. Allowing the `awk`/`lex` behavior in regular expressions would change the normal behavior in an unacceptable way; it is expected that `awk` and `lex` will decode escape sequences in regular expressions before passing them to *regcomp* () or comparable routines. Each utility describes the escape sequences it accepts as an exception to the rules in this clause; the list is not the same, for historical reasons.

As noted earlier, the new syntax and rules have been added to accommodate other languages than English. These modifications were proposed by the UniForum Subcommittee on Internationalization and accepted by the working group. The remainder of this clause describes the rationale for these modifications.

Internationalization Requirements

The goal of the internationalization effort was to provide functions and capabilities that matched the capabilities of existing implementations, but that adhered to the user's local customs, rules, and environment. This has also been described as ''removing the ASCII (and English language) bias.''

In addition, other requirements also influence the standardization efforts, such as *portability* , *extensibility* , and *compatibility* .

In a worldwide environment *portability* carries much weight. Wherever feasible, users should be given the capability to develop code that can execute independently of character set, code set, or language.

Standards must also be *extensible* ; to support further development, to allow for local or regional extensions, or to accommodate new concepts (such as multibyte characters).

Compatibility does not only refer to support of existing code, but also to making the new syntax, semantics, and functions compatible with existing environments and implementations.

Internationalization Technical Background

The C Standard {7} (and, by implication, also POSIX) recognizes that the ASCII character set used in historical UNIX system implementations is not adequate outside the Anglo-American language area. It is, however, not enough to remove the ASCII bias; the dependency on Anglo-Saxon conventions and rules must also be broadened to accommodate other cultures, including those that require thousands of characters.

Character sets are defined by their *attributes* ; typical attributes are the *encoding* , the *collating sequence* , the *character classification* , and the *case mapping* .

It is also recognized that, even within one language area, several combinations of attributes exist: character set attributes are *mutable* and *combinatory* . So, rather than replacing one straitjacket by another, the proposed standards make character sets *user-definable* and *program-selectable* .

The existence of character set attributes is implicit in regular expressions (REs). This implies that regular expressions must recognize and adapt to the *program-selected* set of attributes.

A program *selects* the appropriate character set (or combination of attributes) using the mechanism described in 2.5. The *definition* of a character set (its attributes) is *external* to an executing program. Many combinations of attributes can exist concurrently. Of particular interest are the following attributes:

(1) *Collating Sequence*. In existing implementations, the *encoded* ASCII ordering matches the *logical* English collating sequence. This correspondence does not exist for all code sets or languages. In addition, many languages employ concepts that have no counterparts in English collation:

(a) In many languages, ordering is based on the concept of *string collation* rather than *character collation* as in English. One of the effects of this is that the ordering is based on *collating elements* rather than on characters. Characters typically map into collating elements:

One-to-one mapping, where a character is also a collating element,

One-to-N mapping, where a single character maps into two or more collating elements (as the German ''ß'' (eszet), which collates as ''ss''),

N-to-one mapping, where two or more characters map into one collating element (as in the Spanish ''ll'', which collates between ''l''and ''m'';i.e., a word beginning with ''ll''collates *after* a word beginning with ''lo'').

(b) A common method for adding characters to an alphabet is to use diacritical marks, such as accents or circumflex (` ´ ^). In some languages, this creates a completely new character, collated differently from the Latin ''base.'' In other languages these accented characters are collated as variants of the Latin base letter; i.e., they have the same relative order; they are *equivalent*.

If the strings (words) being compared are equal except for ''accents,'' the strings can be ordered based on a secondary ordering *within* the ''equivalence class.'' For instance, in French, the words ''*tache*'', ''*tâche*'',and ''*tacheter*''collate in that order.

The C Standard {7} recognizes this; it includes new library functions capable of handling complex collation rules. These functions depend on the setting of the *setlocale*() category LC_COLLATE for a definition of the current collation rules.

(2) *Character Classification*. Character classification and case mapping is another area where each language (or even language area) has its own rules. Although users in different countries can use the same code set,

such as ISO 8859-1 {5}, the definition of what constitutes a letter or an uppercase letter may vary.

The C Standard {7} recognizes this; library functions used to classify characters or perform case mapping depend on the *setlocale* () category LC_CTYPE for a definition of how characters map to character classes.

Internationalization Proposal Areas

Based on the requirements and attribute characteristics defined above, and after reviewing proposals and definitions by X/Open and other organizations, the UniForum Subcommittee on Internationalization decided to concentrate on the following areas: the range expression, character classes, the definition of one-character RE (multicharacter element), and equivalence classes.

Most of these are heavily dependent on the current definition of collation sequence; the Subcommittee felt it natural to couple the capabilities and interpretation of bracket expressions closely to the requirements for extended collation capabilities.

In addition, the Subcommittee felt that the capabilities described in 2.5 formed a suitable basis for runtime control of regular expression behavior.

The Subcommittee realized that the mechanism selected requires changes in the existing syntax. As a rule, the Subcommittee wished to minimize changes and avoid syntactical changes that may cause existing regular expressions to fail.

(1) *Collating Elements and Symbols* . As noted above, many expressions within a bracket expression are closely connected with collation, and the Subcommittee defined many capabilities in terms of collating elements and collating symbols.

A collating element is defined as a sequence of one or more bytes defined in the current collating sequence definition as a unit of collation. In most cases, a collating element is equal to a character, but the collation sequence may exclude some characters, or define two or more characters as a collating element.

A one-character RE is, logically enough, defined as one character or something that translates into one character (the number of bits used to represent the character is not an issue here). The expression within square brackets is a one-character RE; i.e., single characters are matched against the list of single characters defined within the brackets.

In Spanish, the phrase ‘‘a to d’’ means the sequence of collating elements a, á, b, c, ch, and d. Consequently, with a Spanish character set, the range statement [a–d] includes the ch collating element, even though it is expressed with two characters (N-to-1 mapping).

The historical syntax, however, does not allow the user to define either the range from a through ch, or to define ch as a single character rather than as either c or h.

The Subcommittee decided that N-to-1 mappings be recognized (if properly delimited), as *one-character REs* inside, but not outside, square brackets (e.g., a period will never match `ch`).

To be distinguishable from a list of the characters themselves, the multicharacter element must be delimited from the remainder of the characters in the string. The characters `[.` and `.]` are used to delimit a multicharacter collating element from other elements, and can be used to delimit single-character collating elements.

(2) *Equivalence Classes* . As stated previously, many languages extend the Latin alphabet by using diacritical marks. In some cases, the Latin base character (e.g., `a`) and the accented versions of the base (e.g., `à`, `â` in French) constitute a ''subclass'' of characters with some partially equivalent characteristics but different code values. Because these characters are related, they are often processed as a group. The historical syntax, however, does not provide for this in a portable manner.

Although it represents an extension of the historical capabilities, the X/Open group strongly recommended that a properly delimited collating element be recognized as representing an equivalence class, that is as the collating element itself, and all other characters with the same primary order in the collation sequence.

The Subcommittee supported this recommendation, and also selected `[=` and `=]` as delimiters for equivalence classes.

(3) *Range Expressions.* The hyphen historically indicated ''a range of consecutive ASCII characters;'' typically it stands for the word ''to,''as in '`a to z`,'' and implies an ordered *interval* . In ASCII , the *encoded* order matches the *logical* English order; this is not true with other encodings or with other alphabets.

If the ASCII dependency is removed, an alternative could have been to use the encoded sequence of whatever code set is currently used. This, however, would certainly decrease portability, as well as requiring the user to know the ordering of the current code set. It would also most certainly be counter-intuitive; a French user would expect the expression `[a-d]` to match any of the letters `a`, `à` `â`, `b`, `c`, `ç` or `d`. The Subcommittee regards this interpretation of ranges as most compatible with existing capabilities, and one that provides for the desired portability.

As the *logical* ordering need not be inherent in the *encoded* sequence, an external definition was required. Such a definition was already present via the *collating sequence* attribute of the character set. The *setlocale* () function provides for an LC_COLLATE category, which defines the current collating sequence. The Subcommittee selected this as the basis for the interpretation of ranges, as well as of equivalence classes and multicharacter collating symbols.

(4) *Character Classes.* The *range* expression is commonly used to indicate a *character class*; the *ex(au_cmd)* section of the *SVID* states: ''*. . . a pair of*

characters separated by – defines a range (e.g., `a-z` *defines any lowercase letter). . . .*'' In reality, `[a-z]` means ''any lowercase letter between `a` and `z`, inclusive.'' This is *only* equivalent to ''any lowercase letter'' if the *a* is the first and `z` is the last lowercase letter in the collating sequence.

To provide the intended capabilities in a portable way, the Subcommittee introduced a new syntactical element, namely an explicit *character class*. The definition of which characters constitute a specific character class is already present via the LC_CTYPE category of the *setlocale* () function.

The Subcommittee selected the identification of character classes by *name* , bracketed by `[:` and `:]`. A character class cannot be used as an endpoint in a range statement.

Internationalization Syntax

The Subcommittee was careful to propose changes in the regular expression syntax that minimize the impact on existing REs. In evaluating alternatives, the Subcommittee looked at ease of use (terseness, ease to remember, keyboard availability), impact on historical REs (compatibility), implementability, performance and how error-prone the syntax is likely to be (ambiguity).

The Subcommittee made the following evaluation:

(1) Syntax changes must be limited to expressions within square brackets.

(2) Strings or characters with special meaning must be delimited from ordinary strings, to avoid compatibility problems.

(3) Both initial and terminating delimiter should consist of two characters, to minimize compatibility and ambiguity problems.

(4) Outer delimiter character should be bracketing; i.e., naturally indicate initial and terminating side. Examples: `{}` `<>` `()`.

(5) The brackets (`[ ]`) are, due to the special rules for ''brackets within brackets,'' rather unlikely to be used in the intended way (a closing bracket must precede an open bracket in the existing syntax).

(6) To minimize ambiguity, brackets must be paired with another character. Many other symbols are already in use, either within regular expressions, or in the shell. Examples of usable characters are: `=` `.` `:`

(7) Because a multicharacter collating element also can be a member of an equivalence class, different delimiters must be chosen for these two expressions. Also, the character class expression must be distinguishable from, e.g., multicharacter collating symbols; although no historical example is known to the Subcommittee, prudence dictated that character classes be given separate delimiters.

(8) The Subcommittee selected the period as the secondary delimiter for multicharacter collating symbols.

(9) The Subcommittee selected the equals-sign as the secondary delimiter for equivalence classes.

(10) The Subcommittee selected the colon as the secondary delimiter for character classes.

The specific syntax and facilities described in this clause represent a coalescence of proposals and implementations from several vendors. Due to differences in facilities and syntax, it was not possible to take one implementation and codify it. There are now several implementations closely patterned on the existing proposal.

The facilities presented in this clause are described in a manner that does not preclude their use with multibyte character sets. However, no attempt has been made to include facilities specifically intended for such character sets.

The definitions of character classes is tied to the LC_CTYPE definition. The set of character classes defined in the C Standard {7} represents the minimum set of character classes required worldwide, i.e., those required by all implementations. It is the working group's belief that local standards bodies, as well as individual vendors, will provide extensions to the standard in these areas, for instance to provide, for example, Kanji character classes.

In many historical implementations, an *invalid range* is treated as if it consisted of the endpoints only. For example, `[z-a]` is treated as `[za]`. Some implementations treat the above range as `[z]`, and others as `[-az]`. Neither is correct, and the working group decided that this should be treated as an error.

It was proposed that the syntax for bracket expressions be simplified such that the ‘‘extra’’ brackets are not needed if the bracket expression only consists of a character class, an equivalence class, or a collating symbol: ‘‘`[:alpha:]`’’ instead of ‘‘`[[:alpha:]]`’’. To ensure unambiguity, if a bracket expression starts with `:`, `=`, or `.`, then it cannot contain a class expression or a collating symbol (or duplicated characters). In addition, it was also proposed that only valid class or collating symbol expressions be accepted: e.g., `[[:ctrl:]]` is an invalid expression. The working group rejected the proposal. While the syntax `[:alpha:]` may be intuitive to some, the proposal does not allow, e.g., `[:digit:.ch.]`. The alternative, to require additional brackets for the latter case would probably cause more errors than the historical syntax. Requiring erroneous class expressions or collating symbols to make the regular expression invalid may minimize the risks for inadvertent spelling errors. However, at this point it was judged that this would reduce consensus.

Consideration was given to eliminating the `[.ch.]` syntax and providing that collating element should be recognized as such both inside and outside bracket expressions. In addition, consideration was given to defining character classes such that collating elements are included. The working group rejected these proposals. The `[.ch.]` syntax is only required inside bracket expressions due to the fact that a bracket expression historically only matched a single character. If `ch` is a collating element, a range `[a-z]` (if ‘‘`ch`’’ falls within it) matches `ch`. Outside brackets, an expression `ch` is treated as two concatenated characters, matching the string ‘‘`ch`’’. The `[.ch.]` expression is intended to allow the specification of a multicharacter collating element separately from ranges in a bracket expression. Character classes are not intended to include collating elements; there is no requirement that all characters in a multicharacter collating element belong to the same character class (for instance ''`Ch`'' is ''`alpha`'' but neither ''`upper`'' nor ''`lower`''). Introducing collating elements in character classes would be nonintuitive.

It was suggested that, because ranges may or may not be meaningful (or even accepted) based on the current collating sequence, they should be eliminated from the syntax (or at least marked obsolescent). It was suggested that, e.g., `[z-a]` should always be or never be an error, regardless of collating sequence. The working group did not wish to eliminate ranges from the syntax. While it is true that ranges may not be universally portable, they are nevertheless a useful and fundamental construct in regular expressions. The regular expression syntax has consciously been extended to provide both increased portability and extended local capabilities. Where supported, ranges must reflect the current collating sequence. The working group instead elected to include range expressions as an implementation requirement, but state that strictly conforming applications (but not, e.g., National-Body-conforming applications) shall not use range expressions. Treating erroneous ranges as invalid points out that these may not be portable across collating sequences; and is better than (silently) making them behave in a way contrary to the intents of the user.

Earlier drafts allowed the use of an equivalence class expression as the starting or ending point of a range expression, such as `[[=e=]-f]`. This now produces unspecified results because it is possible to define the equivalence class as a disjoint set of characters. This example could produce different results on various systems:

— An error.

— The equivalent of `[[=e=]e-f]` (which is the correct portable way to include equivalence class effects in a bracket expression).

— All of the collating elements from the lowest value found in the equivalence class, including any of the elements found between the disjoint values.

Consideration was given to saying that equivalence classes with disjoint elements produce unspecified results at the start or end of a range, but since the application cannot predict which equivalence classes are disjoint, this is no improvement over the more general statement chosen.

It was suggested that, while reference to nonprintable characters is partially supported by the proposed set of character classes, the specificity is not precise enough, and that additional character classes should be supported, e.g., `[:tab:]` or `[:a:]`. The working group rejected this proposal, because this feature would represent a substantial enhancement to the current regular expression syntax, and one that cannot be based on internationalization requirements. It is judged that its inclusion would reduce consensus. A future revision of regular expressions should study the capability to create temporary character classes for use in regular expressions; a ''character class macro facility.''

2.8.6.3.3 BRE s Matching Multiple Characters Rationale. *(This subclause is not a part of P1003.2)*

The limit of nine backreferences to subexpressions in the RE is based on the use of a single digit identifier; increasing this to multiple digits would break historical applications. This does not imply that only nine subexpressions are allowed in REs. The following is a valid BRE with ten subexpressions:

```
\(\(\(ab\)*c\)*d\)\(ef\)*\(gh\)\{2\}\(ij\)*\(kl\)*\(mn\)*\(op\)*\(qr\)*
```

The working group regards the common current behavior, which supports `\`*n*`*`, but not `\`*n*`\{`*min,max*`\}`, or `\(...\)*`, or `\(...\)\{`*min,max*`\}`, as a nonintentional result of a specific implementation, and supports both duplication and interval expressions following subexpressions and backreferences.

2.8.6.3.4 Expression Anchoring Rationale. *(This subclause is not a part of P1003.2)*

Often, the dollar-sign is viewed as matching the ending `<newline>` in text files. This is not strictly true; the `<newline>` is typically eliminated from the strings to be matched and the dollar-sign matches the terminating null character.

The ability of `^`, `$`, and `*` to be nonspecial in certain circumstances may be confusing to some programmers, but this situation was changed only in a minor way from historical practice to avoid breaking many existing scripts. Some consideration was given to making the use of the anchoring characters undefined if not escaped and not at the beginning or end of strings. This would cause a number of historical BREs, such as `2^10`, `$HOME`, and `$1.35`, which relied on the characters being treated literally, to become invalid.

However, one relatively uncommon case was changed to allow an extension used on some implementations. Historically, the BREs `^foo` and `\(^foo\)` did not match the same string, despite the general rule that subexpressions and entire BREs match the same strings. To achieve balloting consensus, POSIX.2 has allowed an extension on some systems to treat these two cases in the same way by declaring that anchoring *may* occur at the beginning or end of a subexpression. Therefore, portable BREs that require a literal circumflex at the beginning or a dollar-sign at the end of a subexpression must escape them. Note that a BRE such as `a\(^bc\)` will either match `a^bc` or nothing on different systems under the POSIX.2 rules.

ERE anchoring has been different from BRE anchoring in all historical systems. An unescaped anchor character has never matched its literal counterpart outside of a bracket expression. Some systems treated `foo$bar` as a valid expression that never matched anything, others treated it as invalid. POSIX.2 mandates the former, valid unmatched behavior.

Some systems have extended the BRE syntax to add alternation. For example, the subexpression `\(foo$\|bar\)` would match either `foo` at the end of the string or `bar` anywhere. The extension is triggered by the use of the undefined `\|` sequence. Because the BRE is undefined for portable scripts, the extending system is free to make other assumptions, such as that the `$` represents the end-of-line anchor in the middle of a subexpression. If it were not for the extension, the

`$` would match a literal dollar-sign under the POSIX.2 rules. 1

2.8.6.4 Extended Regular Expressions Rationale. *(This subclause is not a part of P1003.2)*

As with basic regular expressions, the working group decided to make the interpretation of escaped ordinary characters undefined.

The right-parenthesis is not listed as an ERE special character because it is only 1
special in the context of a preceding left-parenthesis. If found without a preced- 1
ing left-parenthesis, the right-parenthesis has no special meaning. 1

Based on objections in several ballots, the *interval expression*, {*m,n*}, has been added to extended regular expressions. Historically, the interval expression has only been supported in some extended regular expression implementations. The working group estimated that the addition of interval expressions to extended regular expressions would not decrease consensus, and would also make basic regular expressions more of a subset of extended regular expressions than in many historical implementations.

It was suggested that, in addition to interval expressions, backreferences (*n*) also should be added to extended regular expressions. This was rejected by the working group as likely to decrease consensus.

In historical implementations, multiple duplication symbols are usually interpreted from left to right and treated as additive. As an example, `a+*b` matches zero or more instances of `a` followed by a `b`. In POSIX.2, multiple duplication symbols are undefined; i.e., they cannot be relied upon for portable applications. One reason for this is to provide some scope for future enhancements; the current syntax is very crowded.

The precedence of operations differs between EREs and those in `lex`; in `lex`, for historical reasons, interval expressions have a lower precedence than concatenation.

2.8.6.5 Regular Expression Grammar Rationale. *(This subclause is not a part of P1003.2)*

None.

2.9 Dependencies on Other Standards

2.9.1 Features Inherited from POSIX.1

This subclause describes some of the features provided by POSIX.1 {8} that are assumed to be globally available by all systems conforming to POSIX.2. This subclause does not attempt to detail all of the POSIX.1 {8} features that are required by all of the utilities and functions defined in this standard; the utility and function descriptions point out additional functionality required to provide the corresponding specific features needed by each.

The following subclauses describe frequently used concepts. Utility and function description statements override these defaults when appropriate.

2.9.1.0.1 Features Inherited from POSIX.1 Rationale. *(This subclause is not a part of P1003.2)*

It has been pointed out that POSIX.2 assumes that a lot of POSIX.1 {8} functionality is present, but never states exactly how much. This is an attempt to clarify the assumptions.

This subclause only covers the ‘‘utilities and functions defined by this standard.’’ It does not mandate that the specific POSIX.1 {8} interfaces themselves be available to all application programs. A C language program compiled on a POSIX.2 system is not guaranteed that any of the POSIX.1 {8} functions are accessible. (For example, although UNIX system-based implementations of `ls` will use *stat*() to get file status, a POSIX.2 implementation of `ls` on a ‘LONG_NAME_OS-based’’ implementation might use the *get_file_attributes*() and the *get_file_time_stamps*() system calls.) POSIX.2 only requires equivalent functionality, not equal means of access. In any event, programs requiring the POSIX.1 {8} system interface should specify that they need POSIX.1 {8} conformance and not hope to achieve it by piggybacking on POSIX.2.

2.9.1.1 Process Attributes

The following process attributes, as described in POSIX.1 {8}, are assumed to be supported for all processes in POSIX.2:

controlling terminal	real group ID
current working directory	real user ID
effective group ID	root directory
effective user ID	saved set-group-ID
file descriptors	saved set-user-ID
file mode creation mask	session membership
process ID	supplementary group IDs
process group ID	

A conforming implementation may include additional process attributes.

2.9.1.1.1 Process Attributes Rationale. *(This subclause is not a part of P1003.2)*

The supplementary group IDs requirement is minimal. If {NGROUPS_MAX} is defined to be zero, they are not required. If {NGROUPS_MAX} is greater than zero, the supplementary group IDs are used as described in POSIX.1 {8} in various permission checking operations.

The saved-set-group-ID and saved-set-user-ID requirements are also minimal. If {_POSIX_SAVED_IDS} is defined, they are required; otherwise, they are not.

A controlling terminal is needed to control access to `/dev/tty`.

The file creation semantics of POSIX.2 require the effective group ID, effective user ID, and the file mode creation mask.

Pathname resolution and access permission checks require the current working directory, effective group ID, effective user ID, and root directory.

The `kill` utility requires the effective group ID, effective user ID, process ID, process group ID, real group ID, real user ID, saved set-group-ID, saved set-user-ID, and session membership attributes to perform the various signal addressing and permission checks.

The `id` utility is based on the effective group ID, effective user ID, real group ID, real user ID, and supplementary group IDs.

The following process attributes described in POSIX.1 {8} do not seem to be required by POSIX.2: parent process ID, pending signals, process signal mask, time left until an alarm clock signal, *tms_cstime*, *tms_cutime*, *tms_stime*, and *tms_utime*. There are probably other attributes mentioned in POSIX.1 {8} that are not listed here.

2.9.1.2 Concurrent Execution of Processes

The following functionality of the POSIX.1 {8} *fork*() function shall be available on all POSIX.2 conformant systems:

(1) Independent processes shall be capable of executing independently without either process terminating.

(2) A process shall be able to create a new process with all of the attributes referenced in 2.9.1.1, determined according to the semantics of a call to the POSIX.1 {8} *fork*() function followed by a call in the child process to one of the POSIX.1 {8} *exec* functions.

2.9.1.2.1 Concurrent Execution of Processes Rationale. *(This subclause is not a part of P1003.2)*

The historical functionality of *fork*() is required, which permits the concurrent execution of independent processes. A system with a single thread of process execution is not an appropriate base upon which to build a POSIX.2 system. (This requirement was not explicitly stated in the 1988 POSIX.1, but is included in the current POSIX.1 {8}.)

2.9.1.3 File Access Permissions

The file access control mechanism described by *file access permissions* in 2.2.2.55 applies to all files on a conforming POSIX.2 implementation.

2.9.1.3.1 File Access Permissions Rationale. *(This subclause is not a part of P1003.2)*

The entire concept of file protections and access control is assumed to be handled as in POSIX.1 {8}.

2.9.1.4 File Read, Write, and Creation

When a file is to be read or written, the file shall be opened with an access mode corresponding to the operation to be performed. If file access permissions deny access, the requested operation shall fail.

When a file that does not exist is created, the following POSIX.1 {8} features shall apply unless the utility or function description states otherwise:

(1) The file's user ID is set to the effective user ID of the calling process.

(2) The file's group ID is set to the effective group ID of the calling process or the group ID of the directory in which the file is being created.

(3) The file's permission bits are set to:

S_IROTH | S_IWOTH | S_IRGRP | S_IWGRP | S_IRUSR | S_IWUSR

(see POSIX.1 {8} 5.6.1.2) except that the bits specified by the process's file mode creation mask are cleared.

(4) The *st_atime*, *st_ctime*, and *st_mtime* fields of the file shall be updated as specified in *file times update* in 2.2.2.69.

(5) If the file is a directory, it shall be an empty directory; otherwise the file shall have length zero.

(6) Unless otherwise specified, the file created shall be a regular file.

When an attempt is made to create a file that already exists, the action shall depend on the file type:

(1) For directories and FIFO special files, the attempt shall fail and the utility shall either continue with its operation or exit immediately with a nonzero status, depending on the description of the utility.