IEEE Standards Interpretations for IEEE Std 1003.2™-1992 IEEE Standard for Information Technology--Portable Operating System Interfaces (POSIX®)--Part 2: Shell and Utilities

Copyright © 1996 by the Institute of Electrical and Electronics Engineers, Inc. 3 Park Avenue New York, New York 10016-5997 USA All Rights Reserved.

Interpretations are issued to explain and clarify the intent of a standard and **do not** constitute an alteration to the original standard. In addition, interpretations are not intended to supply consulting information. Permission is hereby granted to download and print one copy of this document. Individuals seeking permission to reproduce and/or distribute this document in its entirety or portions of this document must contact the IEEE Standards Department for the appropriate license. Use of the information contained in this document is at your own risk.

IEEE Standards Department, Copyrights and Permissions, 445 Hoes Lane, Piscataway, New Jersey 08855-1331, USA

Interpretation Request #135
Topic: Regular Expressions **Relevant Sections:** 2.8.2

The fourth paragraph in subclause 2.8.2 (page 77, lines 2791- 2796) says: Consistent with the whole match being the longest of the leftmost matches, each subpattern, from left to right, shall match the longest possible string. For this purpose, a null string shall be considered to be longer than no match at all. For example, matching the BRE \(.*\).* against abcdef, the subexpression (\1) is abcdef, and matching the BRE \(a*\) against bc, the subexpression (\1) is the null string. This description is unfortunately very dependent on the terms "subpattern" and "subexpression" which are not clearly defined. For BREs, the latter term is defined as a BRE enclosed between \(and \), there is only a strong implication that the corresponding construct for EREs is also a "subexpression". Of the term "subpattern" I cannot find any further use.

For example, when matching the BRE a.*\(.*b.*\(.*c.*\).*d.*\).*e.*\(.*f.*\).*g against "aabbccddeeffgg" there are quite a few possible ways to match, and thus what "\1", "\2", and "\3" are supposed to be is muddy. Two other example BREs extend the issue to repeated subexpressions: \(.\(.\)*\)* and \(\(..\)*\)* Both of these will be matched against "xxxx". Let's assume that the only significant matching differences are those distinguishable through the API. (Strictly speaking, back references can use intermediate subexpression matches, but it is possible to reduce each instance of a back reference as if it were the end of a complete RE. Therefore, we can ignore back references as being a special case of our simplifying assumption.) This means, for example, that it doesn't matter how the RE ".*.*.*" is matched against "abcdef" since only the start and end offset of the entire RE is available.

It may be worth noting here why there was no equivalent problem with historic RE im-

plementations. With EREs, there was no historic API that provided an ERE with any match information beyond the start and end of the full RE. Since EREs don't include back references, subexpression grouping solely governed operator binding; there was no remnant of the groups themselves in the automata constructed. With BREs, grouping solely provided "note here" marks; one could not apply repetition operations to a subexpression. This means that there was no ambiguity about what \1 matched, for example, since there was always exactly one match. Given this more limited BRE definition, a simple polynomial implementation could be written that had its first match guaranteed to maximize earlier portions of the RE. (These "portions" of the BRE were also easy to identify since it was possible to enumerate all combinations since the BRE operators could only be applied to one-character REs.)

It is the extensions to historic REs that were added by POSIX.2 that is the origin of this RFI's concerns. I'll describe two different ways to understand the quoted paragraph: the "subexpression" and the "fencepost" models. Both of these models produce the same results for the two examples in the quoted text, so I'll use my more complex BREs above for this paper. Subexpression Model The "subexpression" model interprets the quoted paragraph as reapplying the longest-match rule used for the match of the entire RE with its implied "zeroth subexpression". In other words, given two RE matches, for subexpressions 1 through n (in that order), choose the match with a longest subexpression i (if the two i-th subexpression match have the same length, go on to subexpression i+1).

For the first example RE, these REs, in order, are used to judge the best match: 0. a.*\(.*b.*\(.*c.*\).*d.*\).*e.*\(.*f.*\).*g 1. .*b.*\(.*c.*\).*d.* 2. .*c.* 3. .*f.* which produces the following match against aabbccddeeffgg: a .* \(.* b .* \(.* c .* \) .* d .* \) .* e .* \(.* f .* \) .* g a a b bc c dd e e f f g g The "subexpression" model has the nice property that the best match for all groups are chosen in the same manner, including the entire RE (once the left edge is found). It sometimes produces behavior counter to expectations because the initial part of the RE is given no priority. (This is a potential compatibility issue for us.) However, it produces a best match choice that is easy to determine and understand. For the other two example REs, since the last instance of a repeated subexpression is to be reported and thus maximized, we have: 0. \(.\(.\)*\)* 0. \(\(..\)*\)* 1. \(.\(.\)*\) 1. \(\(..\)*\) 2. \(.\) 2. \(..\) matching xxxx: \(. \(. \)* \)* \(\(.. \)* \)* x [x][x]x[xx]xx \1:[0,4]=xxxx, \2:[3,4]=x \1:[0,4]=xxxx, \2:[2,4]=xx in which [...] represents a previous match of the subexpression. In essence, the "*" on subexpression one has no effect except when the former BRE is matched against an empty string. The best choice is straightforward to determine and is obvious. Fencepost Model The "fencepost" model takes each begin and end grouping mark as the "subpatterns" separators. In this approach, the length matched by the initial portion of the RE outside of any group is first maximized; the length matched by the final portion is immaterial. There are two variants of this model: the "flat fence" scheme in which all fenceposts are at one level, and the "nested fence" scheme in which the fenceposts within an inner subexpression are not "visible" in the containing subexpression.

I believe (but present no proof) that these two variants produce the same result for REs without repeated subexpressions. The former is easier to explain, the latter is easier to implement with regmatch_t rm_so's and rm_eo's. For the first example RE, these two schemes maximize the following patterns and produce this match against the sample string. The parenthesized steps are those that are immaterial, but have been included for completeness. flat fence 0. a.*\(.*b.*\(.*c.*\).*d.*\).*e.*\(.*f.*\).*g 1. a.* 2. .*b.* 3. .*c.* 4. .*d.* 5. .*e.* 6. .*f.* (7.) .*g a .* \(.* b .* \(.* c .* \) .* d .* \) .* e .* \(.* f .* \) .* g a ab b c c d d e e f f g g nested fence 0. a.*\(.*b.*\(.*c.*\).*d.*\).*e.*\(.*f.*\).*g 1. a.* 2..*b.*\(.*c.*\).*d.* 3. .*e.* 4. .*f.* (5.) .*g 6..*b.* 7. .*c.* (8.) .*d.*

Note that order of the nested fence steps 3-5 and 6-8 are independent. This makes it easier to believe the the two schemes produce the same behavior. Of the two models, the fencepost model most closely matches historic implementations (as discussed above), and since the highest priority is given to the initial part of the RE, it more often seems to behave as expected when the REs are the same as those provided in historic BRE implementations. However, for the other two example REs, the fencepost model is muddier: How does one maximize the length of separated portions of REs when the very separators are potentially applied over and over? Moreover, with these two examples, the strict subpatterns as separated by grouping marks ("", ".", and "..") are always a fixed length--what's to maximize here? The only approach that I've come up with is to imagine that the repeated subexpression is sequentially duplicated in an abstract RE without any change of subexpression number. The choice now is which of the possible abstract REs to use. For \(.\(.\)*\)*, the abstract REs would be the following eight choices. (I've used digits to mark the subexpression numbers just below the parenthesis characters and again a square- bracketed string is an unreported previous subexpression match.)

The third line reframes the abstract RE as it would be seen when the "fenceposts" from the reported subexpressions only are visible. 1. \(. \(. \) \(. \) \(. \) \) 1 x 2 [x] 2 2 [x] 2 2 x 2 1 \1:[0,4]=xxxx, \2:[3,4]=x \(. . . \(. \) \) 2. \(. \(. \) \(. \) \) \(. \) 1 [x] 2 [x] 2 2 [x] 2 1 1 x 1 \1:[3,4]=x, \2:[-1,-1] .. . . \(. \) 3. \(. \(. \) \) \(. \(. \) \) 1 [x] 2 [x] 2 1 1 x 2 x 2 1 \1:[2,4]=xx, \2:[3,4]=x .. . \(. \(. \) \) 4. \(. \(. \) \) \(. \) \(. \) 1 [x] 2 [x] 2 1 1 [x] 1 1 x 1 \1:[3,4]=x, \2:[-1,-1] .. . . \(. \) 5. \(. \) \(. \(. \) \(. \) \) 1 [x] 1 1 x 2 [x] 2 2 x 2 1 \1:[1,4]=xxx, \2:[3,4]=x .. \(. . \(. \) \) 6. \(. \) \(. \(. \) \) \(. \) 1 [x] 1 1 [x] 2 [x] 2 1 1 x 1 \1:[3,4]=x, \2:[-1,-1] .. . . \(. \) 7. \(. \) \(. \) \(. \(. \) \) 1 [x] 1 1 [x] 1 1 x 2 x 2 1 \1:[2,4]=xx, \2:[3,4]=x .. . \(. \(. \) \) 8. \(. \) \(. \) \(. \) \(. \) 1 [x] 1 1 [x] 1 1 [x] 1 1 x 1 \1:[3,4]=x, \2:[-1,-1] ... . \(. \)

Of these eight, we see that there are four unique matches as far as would be reported by the API: A. \1:[0,4], \2:[3,4] (choice 1) B. \1:[1,4], \2:[3,4] (choice 5) C. \1:[2,4], \2:[3,4] (choices 3 and 7) D. \1:[3,4], \2:[-1,-1] (choices 2, 4, 6, and 8) When these four are examined, D is the one that maximizes the length of the subpattern between the start of the RE and the first fencepost, even though subexpression 2 fails to match.

I find this counterintuitive. One variation possible would be to give higher precedence to the abstract REs that include more subexpression matches (again ordered from 1 to n). This would make C the best match, which is at least somewhat intuitive, but justification from the POSIX.2 standard for giving precedence to choices that include more subexpression matches is at best a stretch: there is only a statement that a match of an empty string is preferred to a match failure. There are many more abstract RE choices for \(\(..\)*\)* because subexpression 1 can match an empty string. Although there can be an indefinite number of empty string matches, the only place where one is visible is as the final match of a subexpression.

So, we have the following four abstract REs as the significant choices: 1. \(\(.. \) \(.. \) \) 1 2 [xx] 2 2 xx 2 1 \1:[0,4]=xxxx, \2:[2,4]=xx \(.. \(.. \) \) 2. \(\(.. \) \(.. \) \) \(\) 1 2 [xx] 2 2 [xx] 2 1 1 1 \1:[4,4]=, \2:[-1,-1] .. .. \(\) 3. \(\(.. \) \) \(\(.. \) \) 1 2 [xx] 2 1 1 2 xx 2 2 \1:[2,4]=xx, \2:[2,4]=xx .. \(\(.. \) \) 4. \(\(.. \) \) \(\(.. \) \) \(\) 1 2 [xx] 2 1 1 2 [xx] 2 1 1 1 \1:[4,4]=, \2:[-1,-1] ... .. \(\) Which reduce down to three unique matches A. \1:[0,4], \2:[2,4] B. \1:[2,4], \2:[2,4] C. \1:[4,4], \2:[-1,-1] and again the choice that maximizes the initial subpattern is C, which is counterintuitive since it fails to match subexpression 2. If higher precedence is given to one that includes a match for subexpression 2, choice B wins. In essence, for REs with repeated subexpressions, the fencepost model tends to choose the match that minimizes the reported lengths of repeated subexpressions. My guess is that this result most likely does not match the user's expectation. This is unfortunate since it is otherwise the choice compatible with historic RE implementations. Conclusion

The POSIX.2 standard must be amended to clarify the "how to choose the best match from the set of leftmost-longest" paragraph. The choices I can think of are:

1. Specify that the choice is implementation defined (or unspecified); aa portable application cannot rely on any particular match other than that it will be one of the leftmost-longest.

2. Choose one of the models I've discussed above. Of the two, the ssubexpression model is easiest to understand and describe, but it easily makes a choice that differs from existing practice--often when the RE user was not aware that their RE could be matched in more than one way!

3. Make the best match model be a two-way RE compile-time flag and eexpose it to the user via options for those utilities that employ user-provided REs.

4. Choose some other model. If this approach is taken, be sure that tthere it is completely understood. As a minimum, something like what I've put together in the above is A good start. I would also expect that an implementation of the model be explored, too. I've implemented both of the models I described in our POSIX.2 RE library. I can support any of the above, and am willing to help implement some other model if the fourth choice above is to be explored. Without some change in this area we providers remain

swinging at the end of multiple tails--users who want things to work the way they always have, and the test suites that are going to take a particular interpretation of the quoted paragraph and wait for you to prove their tests to be wrong.

Interpretation Response
Interpretations can not amend the standard. This request expands on the issues raised in Interpretation #43, and in that interpretation it is pointed out that the standard is unclear on these issues, and no conformance distinction can be made between alternative implementations based on this. This is being referred to the sponsor.

Rationale for Interpretation
None.