179 Changes to Unicode Regular Expression Guidelines 2011.05.02

Status: Open

Description of Issue:

The Unicode Consortium is considering changes to UTS #18 *Unicode Regular Expressions* (http://unicode.org/reports/tr18/). These proposed changes have arisen in connection with questions about case-insensitive and canonical-equivalent matching.

The proposed changes eliminate some requirements on implementations of Unicode Regular Expressions which have proven to be problematic in implementations, and add clarifications. The consortium is soliciting feedback on these changes.

Background

Part of the issue to be addressed is to define more precisely the connection between matching of regular expressions and equivalence relations among strings. Matching under equivalence relations can be stated more formally as:

> **Matching under Equivalence Relations.** A regular expression R matches according to an equivalence relation E whenever for all strings *S* and *T*, if *S* is equivalent to *T* under E, then R matches *S* if and only if R matches *T*.

In the Unicode Standard, the relevant equivalence relation for *case-insensitivity* is established according to whether two strings case fold to the same value. The case folding can either be **simple** (a 1:1 mapping of code points) or **full** (with some 1:n mappings).

- “ABC” and “Abc” are equivalent under both full and simple case folding.
- “cliﬀ” (with the “ﬀ” ligature) and “CLIFF” are equivalent under full case folding, but not under simple case folding.

The equivalence relation for *canonical equivalence* is established by whether two strings are identical when normalized to NFD. That normalization involves n:m mappings and rearrangements of code points.

- <*o-horn, dotbelow*> and <*o-dotbelow, horn*> are canonically equivalent, since they both have the same NFD form: <*o, dotbelow, horn*>.

1. Full vs. simple case-insensitive matching

It is proposed to withdraw the recommendation for doing full case-insensitive matching in *RL2.4 Default Loose Matches.* The text would instead be modified to focus that section only on issues of case conversion.

The text currently reads:

> ***RL2.4 Default Loose Matches***
>
> *To meet this requirement:*

- *if an implementation provides for case-insensitive matching, then it shall provide at least the full, default Unicode case-insensitive matching.*
- *if an implementation provides for case conversions, then it shall provide at least the full, default Unicode case conversion.*

The new text would read:

RL2.4 Default Case Conversion

To meet this requirement, if an implementation provides for case conversions, then it shall provide at least the full, default Unicode case conversion.

There are two reasons for removing full case-insensitive matching:

1. It is unclear how full case-insensitive matching can be effectively implemented in regular expressions, especially with back references.
2. There are a number of examples where the results would be counter-intuitive for typical users of regular expressions.

It is feasible to describe how to transform text into the fully-case-folded form, and construct regular expressions targeted at such text. So the discussion in UTS #18 would be changed to focus on such guidelines and not state them as requirements.

Note: The obsolete link in the text of UTS #18 “To correctly implement a caseless match and case conversions, see UAX #21: Case Mappings [Case].” would also be corrected to reference current information about casing in the standard.

2. Canonical-equivalent matching

It is proposed to withdraw the recommendation for doing full canonical-equivalence matching in *RL2.1 Canonical Equivalents*. The current text reads:

RL2.1 Canonical Equivalents

To meet this requirement, an implementation shall provide a mechanism for ensuring that all canonically equivalent literal characters match.

The way most regular expression engines work, this requirement cannot be satisfied. The reason that it cannot be satisfied results from the fact that canonical equivalence may involve reordering, splitting, or merging of characters. For example, all of the following sequences are canonically equivalent:

1. **o + horn + dotbelow**
   - U+006F (o) LATIN SMALL LETTER O +
   - U+031B (̛) COMBINING HORN +
   - U+0323 (̣) COMBINING DOT BELOW
2. **o + dotbelow + horn**
   - U+006F (o) LATIN SMALL LETTER O +
   - U+0323 (̣) COMBINING DOT BELOW +
   - U+031B (̛) COMBINING HORN
3. **o-horn + dotbelow**
   - U+01A1 (ơ) LATIN SMALL LETTER O WITH HORN
   - U+0323 (̣) COMBINING DOT BELOW

4. **o-dotbelow + horn**
    - U+1ECD (ọ) LATIN SMALL LETTER O WITH DOT BELOW +
    - U+031B (̛) COMBINING HORN
5. **o-horn-dotbelow**
    - U+1EE3 (ợ) LATIN SMALL LETTER O WITH HORN AND DOT BELOW

The regular expression pattern /o\x{31B}/ matches the first two characters of #1, the first and third characters of #2, the first character of #3, part of the first character together with the third character of #4, and part of the character in #5. Some of these issues are brought out in the text of UTS #18, but implementing RL2.1 is infeasible, because in practice regex APIs are not set up to match parts of characters or handle discontiguous selections.

There are many other edge cases: A combining mark may come from some part of the pattern far removed from where the base character was, or may not explicitly be in the pattern at all. It is also unclear what /./ should match. It is also unclear how regular expression *back references* should work.

It is feasible to describe how to construct patterns that will match against NFD (or NFKD) text, and the description in UTS #18 will be changed to reflect that. That is, it will describe a process whereby:

- The text being matched is put into into a defined normalization form (NFD or NFKD).
- The pattern is not modified in any way from what the user provides.
- Matching proceeds on a code point by code point basis, as usual.

Note that the author of the pattern must know the normalization form of the text, and write the pattern accordingly.

3. Case-insensitive matching with properties

It is proposed to add text to UTS #18 to describe more precisely how to match text case-insensitively. The discussion will outline how a regular expression pattern P can be made to match insensitively, by making the following changes in the interpretation of P:

1. **Each string is matched case-insensitively.** That is, it is *logically* expanded into a sequence of OR expressions, where each OR expression lists all of the characters that have a simple case-folding to the same value.
    - For example, /Dåb/ matches as if it were expanded into /(?:d|D)(?:å|Å|Å)(?:b|B)/
    - Back references are subject to this logical expansion, such as /(?i)(a.c)\1/, where \1 matches what is in the first grouping.
2. **Each character class is closed under case.** That is, it is *logically* expanded into a set of code points, and then closed by adding all simple case equivalents of each of those code points.
    - For example, [\p{Block=Phonetic_Extensions} [A-E]] is a character class that matches 133 code points (under Unicode 6.0). Its case-closure adds 7 more code points: a-e, Ᵽ, and Ᵹ, for a total of 140 code points.

For both property character classes and explicit character classes, closing under simple case-insensitivity means including characters not in the set. For example:

- The case-closure of \p{Block=Phonetic_Extensions} includes two characters not in that set, namely Ᵽ and Ᵹ.
- The case-closure of [A-E] includes five characters not in that set, namely [a-e].

There have been suggestions to restrict case insensitive regex matching so that it would not apply to some or all property-based character classes. One suggestion for an alternative approach, for example, is to close all of the POSIX-Compatible properties listed in *Annex C: Compatibility Properties* under case, but not other Unicode properties. That would require some narrower notion of matching under an equivalence than that presented in *Matching under Equivalence Relations* above in the Background section. For example, under *Matching under Equivalence Relations*, the following is true:

/(?i)[[\x{80}-\x{FF}]-[:Block=Latin_1_Supplement:]]/ = /[]/
(note that Latin_1_Supplement block is identical to U+0080..U+00FF)

Under that alternative approach, in which the block property was *not* case folded, the following would be true:

/(?i)[[\x{80}-\x{FF}]-[:Block=Latin_1_Supplement:]]/ = /[〓 Ÿ]/

Also under that alternative approach, an implementation cannot fully resolve a character class containing properties, and then apply case-closure; instead, it must apply case-closure selectively as the character class is interpreted.