L2/15-226

Re: Deprecating confusablesWholeScript.txt
From: Mark Davis
Date: 2015-09-16
Draft: https://goo.gl/3t1giA

Problem

The current data in WholeScript confusables is insufficient for certain common use cases, causing implementations that exclude or warn on whole-script confusables to be overly restrictive. There is a caveat in the description of how to do Whole_Script_Confusables in UTS#39. That is: "Note that if the implementation needs a set of **allowed** characters that is different from those in *Section 3.1, General Security Profile for Identifiers*, this process needs to be used to generate a different set of data.", but it is time to generalize the data and algorithm to make it easy for people to handle such cases, rather than simply provide that caveat.

Here is an example of such a case. Suppose that we are computing the whole-script confusable for "google". There is one in Armenian, according to the data.

googℯ = U+0581, U+0585, U+0585, U+0581, U+0031, U+212e

This may not seem like a confusable, but there are a class of common fonts where it is. In Noto Sans Armenian, for example, you'd see:

Most implementations will exclude some characters, and there is no whole-script confusable for "google" in Armenian when the character ℯ (= U+212E ESTIMATED SYMBOL) is excluded—as it is when the identifiers exclude symbols, such as in IDNA2008. The confusablesWholeScript.txt file precomputes scripts based on the confusables.txt file. But in that computation, it can't recognize the particular identifier exclusions that the implementation will have set in place. Thus the caveat quoted above.

In the particular case above, the ℯ has script=Common. That kind of case (non-explicit scripts) could be handled by a fix to the algorithm in Whole_Script_Confusables in UTS#39. However, for excluded characters with explicit scripts, that wouldn't be sufficient. (*PS, Thanks to Jungshik for raising this case.)*

Contents

Proposed fixes for Unicode 9.0

The data in confusablesWholeScript.txt would need to be extended, and the algorithm in Whole_Script_Confusables would get very complicated, so it would be better to restate the algorithm at a logical, high level.

1. Deprecate and remove confusablesWholeScript.txt
2. Rewrite Whole_Script_Confusables in UTS#39 to describe the process from a logical point of view, based on what's below.
3. Relegate a discussion of implementation techniques for production code to an implementation section.

Logically, the building of whole script confusables can be described as the following.

A. Transform the source string into nfd: call it *nfd-source*.
    - Example: *nfd-source* is AB
B. Generate the set of all variants of nfd-source, using all of the combinations for each character from the equivalence classes in the confusables.txt file, filtered to remove characters that are disallowed in the identifiers.
    - Assume: A has the equivalence class {A, X, ZW}, and B has the equivalence class {B, C}
    - The result is {AB, AC, XB, XC, ZWB, ZWC}
C. Remove all combinations that have mixed scripts, according to Mixed Script Detection, and remove the nfd-source string.
    - Assume: A is Latin, X is Common, Z is Hiragana, W is common; B is Common, C is Hiragana
    - The remainders are: {XB, XC, ZWB, ZWC}
D. If there are any remaining, then there is a whole-script confusable for the original.
E. If one of the remainder has the same script from nfd-source, then there is a same-script confusable for the original.
    - For this example, there are none.

In addition, some of the algorithms don't handle Script Extensions well, and need to be expanded for that. Without it, a string like "A—" (the dash is a Hiragana/Katakana length mark) counts as a single-script, which percolates through the algorithms and produces false positives.

Note that WholeScriptConfusables really wants to look at strings that are the same "writing system", not just the same script. In particular, it would be better to look at the following as if they are single scripts:

```
Jpan = Han + Hiragana + Katakana
Kore = Hangul + Han
```

State that whenever the Script_Extensions value is used, it is first (logically) transformed so that Inherited → Common, and certain script values are added:

- scx={...Han...} : add Jpan, Kore, eg => {...Han, Jpan, Kore...}
- scx={...Hiragana...} : add Jpan, eg => {...Hiragana, Jpan...}
- scx={...Katakana...} : add Jpan, eg => {...Katakana, Jpan...}
- scx={...Hangul...} : add Kore, eg => {...Han, Kore...}

Note: if Bopomofo needed to be supported and Han+Bopomofo were considered a single writing system, then we'd need an extra extra script ID (eg Hanb), and the rules (first being a modification)

- scx={...Han...} : add Jpan, Kore, Hanb, eg => {...Han, Jpan, Kore, Hanb...}
- scx={...Bopomofo...} : add Hanb, eg => {...Bopomofo,...}

This is in *all processing* using Script_Extensions in UTS #39 (not just Whole-Script Confusable detection).

Implementation Techniques

The combinatorics in the logical description are clearly a problem in production. That can be avoided as follows.

At build time:

1. Process nfd-source character by character
2. Start with a mapping of scripts to samples, where each sample is initially "".
3. Get each successive character's confusable equivalence class as a set.
    a. Filter to remove entries with characters that are disallowed in the identifiers.
    b. For each script in the mapping, find a string in the remaining that can be appended and yet remain in that script.
        i. Avoid the original character from nfd-source, if possible.
    c. If there is no such string, drop the script mapping

4. At the end of this process, drop any <script, nfd-source> entry.
5. The result is a mapping from scripts to whole-script confusables.

The above can be optimized in a few ways:

1. The mapping of characters to confusable equivalence classes can be preprocessed to filter out characters disallowed in identifiers, and filtered to remove strings with conflicting scripts. That makes step 3a faster.
2. A mapping can be produced that replaces each confusable equivalence class set by a map from script to characters. Note that the same string can appear under multiple scripts. That makes step 3b faster.
3. If the implementation does not require explicit string samples for the scripts, the algorithm can be recast to be just in terms of sets of scripts.
    a. There is one tricky bit: an entry that has only one character needs to be marked specially, so that it can be taken into account for step 4 above (removing generated strings that are identical to nfd-source).

Other

1. Remove or revise the Java sample code in Whole_Script_Confusables
2. (Internal) Add test cases to code that matches the general algorithm in Whole_Script_Confusables, and for the sample code (if it is retained and revised).

Background

Current data

The data in confusablesWholeScript.txt has the following form:

```
# Armenian; Latin: [ɯqqhnɲugpo]; [f-hnoqu-wøħơ]
...
0581          ; Armn; Latn; L #       (g)  ARMENIAN SMALL LETTER CO
0584..0585    ; Armn; Latn; L #  [2] (p..o)  ARMENIAN SMALL LETTER KEH..ARMENIAN SMALL LETTER OH
```

There are two sets of data, one with L in field 3 (for lowercase-only identifiers), and one with A (any characters in identifiers). The mapping is from the source characters in field 0, to its script in field 1, and to a target script in field 2, a script in which there are confusables for the source.

Reference Code

The following is code that I'm working on to validate the logical algorithm.

1. http://unicode.org/repos/unicodetools/trunk/unicodetools/org/unicode/test/TestSecurity.java
2. http://unicode.org/repos/unicodetools/trunk/unicodetools/org/unicode/test/CheckWholeScript.java
3. http://unicode.org/repos/unicodetools/trunk/unicodetools/org/unicode/tools/Confusables.java
4. http://unicode.org/repos/unicodetools/trunk/unicodetools/org/unicode/tools/ScriptDetector.java