L2/16-229

Re: **Revising UTS #39 Algorithms for Whole-Script Confusables and Restriction Levels**
From: **Shane Carr**
Date: **2016-08-01**
Drafts: **http://goo.gl/qcc5PF**

UTS #39 “Unicode Security Mechanisms” recommends a framework for detecting the existence of a whole-script confusable. The primary use case for which the test was developed was to detect the existence of confusables between the major Western scripts, including Latin, Cyrillic, and Greek. However, even when using only the Allowed characters from Section 3.1, it was found that up to 19% of English words and 20% of Spanish words have whole-script confusables, using word frequency lists cited in Section 4.1 below. This revision adds these findings to Section 4.1 and shortens the description of those algorithms. The revision also revises the corresponding algorithm for single-script detection and corrects a number of errors elsewhere in the document.

Proposal

The following revisions should be made to document UTS #39. They are sorted by heading number. The original is shown on the left, and the revision is shown on the right. To highlight changes, text has been colored red to indicate a deletion, and green to indicate an addition. Text that is unchanged has been left with a white background.

Section 2: Conformance

Removing the requirement to implement the test for the existence of single-script, mixed-script, and whole-script confusables. See the editorial for Section 4.1 for more information.

1. X and Y are single-script confusables 2. X and Y are mixed-script confusables 3. X and Y are whole-script confusables 4. X has any simple single-script confusable 5. X has any mixed-script confusable 6. X has any whole-script confusable	1. X and Y are single-script confusables 2. X and Y are mixed-script confusables 3. X and Y are whole-script confusables

Section 3.1: General Security Profile for Identifiers

Various minor corrections.

The Restricted characters are characters not in common use, and are can be blocked to further reduce the possibilities for visual confusion. They include the following:	The Restricted characters are characters not in common use, and they can be blocked to further reduce the possibilities for visual confusion. They include the following:
Entry in Table 1 Exceptional allowed characters, including Table 3, Candidate Characters for Inclusion in Identifiers in [UAX31], and some characters for IDNA2008.	Exceptional allowed characters, including Table 3, Candidate Characters for Inclusion in Identifiers in [UAX31], and some characters for IDNA2008, except for those characters that are Restricted above.
Entry in Table 1: Table 5, Recommended Scripts in [UAX31]	Table 5, Recommended Scripts in [UAX31], except for those characters that are Restricted above.

Section 4: Confusable Detection

This section has been re-organized, but the content is mostly the same. To assist with reading the revision, the paragraphs on the left and right are highlighted to show their correspondence.

The top two paragraphs remain the same.

The data in [confusables] provide a mechanism for determining when two strings are visually confusable. The data in these files may be refined and extended over time. For information on handling modifications over time, see Section 2.9.1, Backward Compatibility in Unicode Technical Report #36, "Unicode Security Considerations" [UTR36] and the Migration section of this document. Collection of data for detecting gatekeeper-confusable strings is not currently a goal for the confusable detection mechanism in this document. For more information, see Section 2 Visual Security Issues in [UTR36].	The data in [confusables] provide a mechanism for determining when two strings are visually confusable. The data in these files may be refined and extended over time. For information on handling modifications over time, see Section 2.9.1, Backward Compatibility in Unicode Technical Report #36, "Unicode Security Considerations" [UTR36] and the Migration section of this document. Collection of data for detecting gatekeeper-confusable strings is not currently a goal for the confusable detection mechanism in this document. For more information, see Section 2 Visual Security Issues in [UTR36].

The concept of "target strings" has been replaced with "prototypes" of "exemplar characters", with an explanation and example. The name "target string" made skeletons sound as if they were something other than a set of symbol classes. It is important that users do not misuse skeletons.

The data provides a mapping from source characters to target strings. To see whether two strings X and Y are confusable (abbreviated as X ≅ Y), an implementation uses a transform of X called a skeleton(X) defined by: 1. Converting X to NFD format, as described in [UAX15]. 2. Successively mapping each source character in X to the target string according to the specified data. 3. Reapplying NFD. The resulting strings skeleton(X) and skeleton(Y) are then compared. If they are identical (code point for code point), then X ≅ Y.	The data consist of mappings from input characters to their prototypes. A prototype should be thought of as a sequence of one or more classes of symbols, where each class has an exemplar character. For example, the character U+0153 (œ), LATIN SMALL LIGATURE OE, has a prototype consisting of two symbol classes: the one with exemplar character U+006F (o), and the one with exemplar character U+0065 (e). If an input character does not have a prototype explicitly defined in the data file, the prototype is assumed to consist of the class of symbols with the input character as the exemplar character. For an input string X, define skeleton(X) to be the following transformation on the string: 1. Convert X to NFD format, as described in [UAX15]. 2. Concatenate the prototypes for each character in X according to the specified data, producing a string of exemplar characters. 3. Reapply NFD. With this framework, we can now define strings X and Y to be confusable if and only if skeleton(X) = skeleton(Y), abbreviated as X ≅ Y.

Below, the two paragraphs on the right are taken directly from the current version with minor changes. The first paragraph, beginning with "This mechanism imposes…", and third paragraph, beginning with "To reduce security risks…", have been moved here from their old locations farther down in the section.

Note: The strings skeleton(X) and skeleton(Y) are not intended for display, storage or transmission. They should be thought of as an intermediate processing form, similar to a hashcode. The characters in skeleton(X) and skeleton(Y) are not guaranteed to be identifier characters.	This mechanism imposes transitivity on the data, so if X ≅ Y and Y ≅ Z, then X ≅ Z. It is possible to provide a more sophisticated confusable detection, by providing a metric between given characters, indicating their "closeness." However, that is computationally much more expensive, and requires more sophisticated data, so at this point in time the simpler mechanism has been chosen. That means that in some cases the test may be

	overly inclusive. **Note:** The strings skeleton(X) and skeleton(Y) are not intended for display, storage or transmission. They should be thought of as an intermediate processing form, similar to a hashcode. The exemplar characters are not guaranteed to be safe for use in identifiers. To reduce security risks, it is advised that identifiers use casefolded forms, thus eliminating uppercase variants where possible.

The following notes about *Jpan* and *Kore* have been moved to Section 5.1, where they are more relevant in context.

Many of the processes in this document use the Script_Extensions (scx) property. When that property is used, its values are first (logically) transformed so that Inherited → Common, and certain script values are added:
scx={...Hani...} → {...Hani, Jpan, Kore...}
scx={...Hira...} → {...Hira, Jpan...}
scx={...Kana...} → {...Kana, Jpan...}
scx={...Hang...} → {...Hang, Kore...}	

The definitions of single-script, mixed-script, and whole-script confusable have been reworded to conform to the new concept of "resolved script sets" introduced in Section 5.1. The meaning of the definitions is intended to be the same. The example for single-script confusable has been replaced with an example that has better cross-system rendering support.

Definitions X and Y are *single-script confusables* if they are confusable, and each of them is a single script string according to Section 5, Mixed-Script Detection, and it is the same script for each. Examples: "so̷s" and "søs" in Latin, where the first word has the character "o" followed by the character U+0337 (̷) COMBINING SHORT SOLIDUS OVERLAY. X and Y are *mixed-script confusables* if they are confusable but they are not single-script confusables. Examples: "paypal" and "pаypal", where the second word has the character U+0430 (а) CYRILLIC SMALL LETTER A. X and Y are *whole-script confusables* if they are mixed-script confusables, and each of them is a single script string. Example: "scope" in Latin and "ѕсоре" in Cyrillic. Characters with the Script_Extension property values COMMON or INHERITED are ignored when testing for differences in script.	**Definitions** Confusables are divided into three classes: single-script confusables, mixed-script confusables, and whole-script confusables, defined below. All confusables are either a single-script confusable or a mixed-script confusable, but not both. All whole-script confusables are also mixed-script confusables. X and Y are *single-script confusables* if and only if they are confusable and their resolved script sets have at least one element in common, according to Section 5, Mixed-Script Detection. Example: "ǉeto" and "ljeto" in Latin (the Croatian word for "summer"), where the first word uses only four codepoints, the first of which is U+01C9 (ǉ) LATIN SMALL LETTER LJ. X and Y are *mixed-script confusables* if and only if they are confusable but their resolved script sets have no elements in common. Example: "paypal" and "pаypal", where the second word has the character U+0430 (а) CYRILLIC SMALL LETTER A. X and Y are *whole-script confusables* if and only if they are mixed-script confusables and each of them is a

	single-script string (has a nonempty resolved script set). Example: "scope" in Latin and "ѕсоре" in Cyrillic. As noted in Section 5, the resolved script set ignores characters with Script_Extensions {Common} and {Inherited} and augments characters with CJK scripts with their respective writing systems.

The remainder of this section is largely unchanged, although parts of it have been reorganized. All text in black is present on both the left side and right side, although it may be in a different location. In addition, the third and seventh paragraphs on the left, those starting with "To reduce security risks..." and "This mechanism imposes...", have been moved higher up in the section.

The first paragraph has a change to the language. The fourth paragraph on the left, starting with "The data may change...", has been removed, since the same statement is made in the first paragraph of this section (Section 4). The last sentence of the seventh paragraph was removed since the claim it makes depends on the scripts in question. (Note that the seventh paragraph is one of the two that was moved higher up in the section.)

Each line in the data file has the following format: Field 1 is the source, Field 2 is the target, and Field 3 is obsolete . Field 3 used to contain different types, but now only has the value MA, which stands for "Mixed-Script, Any-Case". For example: 0441 ; 0063 ; MA # (с → c) CYRILLIC SMALL LETTER ES → LATIN SMALL LETTER C # 2CA5 ; 0063 ; MA # (ⲥ → c) COPTIC SMALL LETTER SIMA → LATIN SMALL LETTER C # →ϲ→ Everything after the # is a comment and is purely informative. A asterisk after the comment indicates that the character is not an XID character [UAX31]. The comments provide the character names. If the data was derived via transitivity, there is an extra comment at the end. For instance, in the above example the derivation was: 1. ⲥ (U+2CA5 COPTIC SMALL LETTER SIMA) 2. → ϲ (U+03F2 GREEK LUNATE SIGMA SYMBOL) 3. → c (U+0063 LATIN SMALL LETTER C) To reduce security risks, it is advised that identifiers use casefolded forms, thus eliminating uppercase variants where possible. The data may change between versions. Even where the data is the same, the order of lines in the files may change between versions. For more information, see Migration. Implementations that use the confusable data do not have to recursively apply the mappings, because the transforms are idempotent. That is, skeleton(skeleton(X)) = skeleton(X)	**Data File Format** Each line in the data file has the following format: Field 1 is the source, Field 2 is the prototype, and Field 3 contains the letters "MA", which stands for "Mixed-Script, Any-Case", maintained for backwards compatibility. For example: 0441 ; 0063 ; MA # (с → c) CYRILLIC SMALL LETTER ES → LATIN SMALL LETTER C # 2CA5 ; 0063 ; MA # (ⲥ → c) COPTIC SMALL LETTER SIMA → LATIN SMALL LETTER C # →ϲ→ Everything after the # is a comment and is purely informative. A asterisk after the comment indicates that the character is not an XID character [UAX31]. The comments provide the character names. Implementations that use the confusable data do not have to recursively apply the mappings, because the transforms are idempotent. That is, skeleton(skeleton(X)) = skeleton(X) If an entry was derived via transitivity, there is an extra comment at the end. For instance, in the above example, the derivation was: 1. ⲥ (U+2CA5 COPTIC SMALL LETTER SIMA) 2. → ϲ (U+03F2 GREEK LUNATE SIGMA SYMBOL) 3. → c (U+0063 LATIN SMALL LETTER C) **Note:** due to production problems, versions before 7.0 did not maintain idempotency in all cases. For more information, see Migration.

Note: due to production problems, versions before 7.0 did not maintain idempotency in all cases. For more information, see Migration. This mechanism imposes transitivity on the data, so if X ≅ Y and Y ≅ Z, then X ≅ Z. It is possible to provide a more sophisticated confusable detection, by providing a metric between given characters, indicating their "closeness." However, that is computationally much more expensive, and requires more sophisticated data, so at this point in time the simpler mechanism has been chosen. That means that in some cases the test may be overly inclusive. However the frequency of such cases in real data should be small.	

Section 4.1: Whole-Script Confusables

Due to the nature of characters available in Western scripts like Latin and Cyrillic, there are many characters between the scripts that map to the same prototypes.

We ran experiments where we made the following restrictions:

1. Use the set of allowable characters to those allowed in Section 3.1.
2. Remove any letters that are not present in a language's exemplar character range, according to data in CLDR.
3. Remove the confusable entries for four characters, тцтк, whole mappings were questionable.

We ran the whole-script confusable test in Version 9.0.0 against three datasets:

1. The top 5,000 words in American English according to the Brigham Young University *Corpus of Contemporary American English*, available at http://www.wordfrequency.info/
2. The top 10,000 words in Chilean Spanish according to the *Lista de Frecuencias de Palabras del Castellano de Chile* (Lifcach) version 2.0, compiled by Scott Sadowsky and Ricardo Martínez Gamboa of the Catholic University of Chile, available at http://sadowsky.cl/lifcach.html
3. The top 10,000 words in Japanese according to the corpus of Japanese Wikipedia, available at https://en.wiktionary.org/wiki/Wiktionary:Frequency_lists/Japanese

Our tests found that 19% of the words from the English dataset had a whole-script confusable. Some English examples are shown below in a font that highlights their confusability. Most of the mappings are Cyrillic.

lane: ӏапе
escape: еѕсаре
calm: саӏгп
prior: ргіог
ass: аѕѕ
clip: сӏір
empire: егпріге
cab: саЬ

Our tests found that 20% of the words from the Spanish dataset had a whole-script confusable. Some Spanish examples are shown below in a font that highlights their confusability. Most of the mappings are Cyrillic.

barón: Ьаго́п
yerba: уегЬа
mermar: гпеггпаг
lecho: ӏесһо
chicha: сһісһа
colocolino: соӏосоӏіпо
paco: расо
caribeño: сагіЬеп̃о

These findings suggest that for applications dealing with identifiers written in Western scripts, the whole-script confusable test has limited use cases.

Our tests found that 0.2% of the words from the Japanese dataset had a whole-script confusable. Some Japanese examples are shown below in a font that highlights their confusability. Most of the mappings involve multi-script CJK characters.

パーカー : パーカー
エーカー : エーカー
ハイパー : ハイパー
イエロー : イエロー
カーター : カーター
カート : カート
トニー : トニー
タ : タ

For the reasons stated above, we propose simplifying this section and removing it from the requirements for a Unicode-conformant implementation. For users interested in Eastern scripts, we keep a high-level overview of the algorithm.

This section specifies how test whether a string has a whole-script confusables, such as "scope" in Latin and "scope" in Cyrillic. The results depend on the set of characters that are accepted by the implementation. The following gives the logical process for determing whether a single-script string source string has a whole-script confusable, given the implementation repertoire of characters R. 1. If the source string is mixed script, then return false. Otherwise transform the source string into nfd, called nfd-source. 2. 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 keep only characters in R. 3. Remove all combinations that have mixed scripts, according to Mixed_Script_Detection, and remove the nfd-source string. 4. If that remainder set is not empty, then there is a whole-script confusable for the original. 5. If one of the remainder set has the same script from nfd-source, then there is a same-script confusable for the original. Example: 1. The nfd-source for the source string is AB. 2. Assume A has the equivalence class {A, X, ZW}, and B has the equivalence class {B, C}. Then the result of generating all variants of the nfd-source is {AB, AC, XB, XC, ZWB, ZWC} 3. Assume A is Latin, C and Z are Hiragana, and the others are Common. Then the remainders after removing mixed-script strings are: {XB, XC, ZWB, ZWC} 4. Because that set is not empty, there is a whole-script confusable for the input string. 5. For this example, there are none. The logical description can be used for a reference implementation for testing, but is not particularly	For some applications, it may be useful to determine if a given input string has any whole-script confusable. For example, the identifier "scope" using Cyrillic characters would pass the single-script test described in Section 5.2, Restriction-Level Detection, even though it is likely to be a spoof attempt. It is possible to determine whether a single-script string X has a whole-script confusable: 1. Consider Q, the set of all strings that are confusable with X. 2. If any string in Q has a nonempty resolved script set that does not intersect with the resolved script set of X, return TRUE. 3. Otherwise, return FALSE. The set of all strings that are confusable with X grows exponentially with the length of the string, so in practice, an equivalent but more efficient algorithm should be used. Note that the confusables data include a large number of mappings between Latin and Cyrillic text. For this reason, the above algorithm is likely to flag a large number of legitimate strings written in Latin or Cyrillic as potential whole-script confusables.

efficient. A production implementation can be optimized to incrementally test for mixed scripts as the combinations in step 2 are built up, and remove any initial substring that fails. That avoids adding the set tree of combinations that start with that initial substring without having to compute them in the first place. For example: 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 not in R. If the remaining set is empty, drop the script mapping b. For each script in the mapping, find a string in the remaining set that can be appended and yet remain in that script (avoiding the original character from nfd-source, where possible). 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 the scripts to a sample whole-script confusable for the input in that script. This process can be further optimized by the following techniques. The mapping of characters to confusable equivalence classes can be preprocessed to filter out characters not in R, and filtered to remove strings with conflicting scripts. That makes step 3a faster. 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. If the implementation does not require explicit string samples for the scripts, the algorithm can be recast to operate on sets of scripts instead. There is one complication to this: 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).	

Section 4.2: Mixed-Script Confusables

I also removed the majority of this section and replaced it with a high-level overview of the algorithm for interested readers.

To test for mixed-script confusables, use the following process: 1. Convert the given string to NFD format, as specified in [UAX15]. 2. For each script found in the given string, see if	To determine the existence of a mixed-script confusable, a similar process could be used: 1. Consider Q, the set of all strings that are confusable with X. 2. Remove all strings from Q whose resolved script

all the characters in the string outside of that script have whole-script confusables for that script (according to Section 4.1, Whole-Script Confusables) .

Example 1: "paypal", with Cyrillic "a"s.

There are two scripts, Latin and Cyrillic. The set of Cyrillic characters {a} has a whole-script confusable in Latin. Thus the string is a mixed-script confusable.

Example 2: "toys-я-us", with one Cyrillic character "я".

The set of Cyrillic characters {я} does not have a whole-script confusable in Latin (there is no Latin character that looks like "я", nor does the set of Latin characters {o s t u y} have a whole-script confusable in Cyrillic (there is no Cyrillic character that looks like "t" or "u"). Thus this string is not a mixed-script confusable.

Example 3: "1ive", with a Greek "ν" and Cyrillic "e".

There are three scripts, Latin, Greek, and Cyrillic. The set of Cyrillic characters {e} and the set of Greek characters {ν} each have a whole-script confusable in Latin. Thus the string is a mixed-script confusable.

set intersects with the resolved script set of X.

3. If Q is nonempty, return TRUE.
4. Otherwise, return FALSE.

Note that due to the number of mappings provided by the confusables data, the above algorithm is likely to flag a large number of legitimate strings as potential mixed-script confusables.

Section 5.1: Mixed-Script Detection

I re-wrote the algorithm in this section to make it easier to understand and more straightforward to implement and apply to the definitions in Section 4 and Section 5.2. The concept of SOSS is restricted to the implementation detail, and instead the concepts of a character's *augmented script set* and a script's *resolved script set* are used.

The mappings to *Jpan*, *Kore*, and *Hanb* have been moved here from Section 4. *Hanb* was not previously listed in Section 4, but it has been listed under Highly Restrictive in Section 5.2.

The Unicode Standard supplies information that can be used for determining the script of characters and detecting mixed-script text. The determination of script is according to the Unicode Standard Annex #24, "Unicode Script Property" [UAX24] , using data from the Unicode Character Database [UCD]. For a given input string, the logical process is the following:

Define a set of sets of scripts SOSS.

For each character in the string:

1. Use the Script_Extensions property to find the set of scripts that the character has.
2. Remove Common and Inherited from that set of scripts.
3. If the result is not empty, add that set to SOSS.

If no single script is common to all of the sets in SOSS, then the string contains mixed scripts.

Define a character's *augmented script set* to be a character's Script_Extensions with the following two modifications.

1. Entries for the writing systems *Hanb* (Han with Bopomofo), *Jpan* (Japanese), and *Kore* (Korean) are added according to the following rules.
    a. If Script_Extensions contains *Hani* (Han), add *Hanb*, *Jpan*, and *Kore*.
    b. If Script_Extensions contains *Hira* (Hiragana), add *Jpan*.
    c. If Script_Extensions contains *Kata* (Katakana), add *Jpan*.
    d. If Script_Extensions contains *Hang* (Hangul), add *Kore*.
    e. If Script_Extensions contains *Bopo* (Bopomofo), add *Hanb*.
2. Sets containing *Zyyy* (Common) or *Zinh* (Inherited) are treated as *Σ*, the set of all script values.

Characters with the script values Common and Inherited are ignored, because they are used with more than one script. For example, "abc-def" counts as a single script Latin because the script of "-" is ignored.

A set of scripts S is said to cover a SOSS if S intersects each element of SOSS. For example, {Latin, Greek} covers {{Latin, Georgian}, {Greek, Cyrillic}}, because:

1. {Latin, Greek} intersects {Latin, Georgian} (the intersection being {Latin}).
2. {Latin, Greek} intersects {Greek, Cyrillic} (the intersection being {Greek}).

The actual implementation of this algorithm can be optimized; as usual, the specification only depends on the results. The following Java sample using [ICU] shows how the above process can be implemented:

```
  public static boolean isSingleScript(String identifier) {
    // Non-optimized code, for simplicity
    Set<BitSet> setOfScriptSets = new
HashSet<BitSet>();
    BitSet temp = new BitSet();
    int cp;
    for (int i = 0; i < identifier.length(); i +=
Character.charCount(i)) {
      cp = Character.codePointAt(identifier, i);
      UScript.getScriptExtensions(cp, temp);
      if (temp.cardinality() == 0) {
        // HACK for older version of ICU
        final int script = UScript.getScript(cp);
        temp.set(script);
      }
      temp.andNot(COMMON_AND_INHERITED);
      if (temp.cardinality() != 0 &&
setOfScriptSets.add(temp)) {
        // If the set hasn't been added already,
        // add it and create new temporary for the next
pass,
        // so we don't rewrite what's already in the set.
        temp = new BitSet();
      }
    }
    if (setOfScriptSets.size() == 0) {
      return true; // trivially true
    }
    temp.clear();
    // check to see that there is at least one script
common to all the sets
    boolean first = true;
    for (BitSet other : setOfScriptSets) {
      if (first) {
        temp.or(other);
        first = false;
      } else {
        temp.and(other);
      }
    }
    return temp.cardinality() != 0;
  }
```

The Script_Extensions data is from the Unicode Character Database [UCD]. For more information on the Script_Extensions property, see Unicode Standard Annex #24, "Unicode Script Property" [UAX24]. For more information on the classes *Jpan*, *Kore*, and *Hanb*, see ISO 15924.

Define a string's *resolved script set* to be the intersection of the augmented script sets over all characters in the string.

A string is said to be *mixed-script* if its resolved script set is empty and *single-script* if its resolved script set is nonempty. Table X gives examples for various strings.

See APPENDIX A of this proposal for the table. The table should be embedded here within the document upon publication.

A set of scripts is said to *cover* a string if the intersection of that set with the augmented script sets of all characters in the string is nonempty; in other words, if every character in the string shares at least one script with the cover set. For example, {Latn, Cyrl} covers "Circle", the third example in Table X. A cover set is said to be *minimal* if a smaller cover set cannot be constructed. For example, {Hira, Hani} covers "〆切", the seventh example in Table X, but it is not minimal, since {Hira} also covers the string, and {Hira} is smaller than {Hira, Hani}. Note that a string may have multiple multiple cover sets that are minimal.

For computational efficiency, a set of script sets, SOSS, can be computed, where the augmented script sets for each character in the string map to one entry in the SOSS. For example, { {Latn}, {Cyrl} } would be the SOSS for "Circle". A set of scripts that covers the SOSS also covers the input string. Likewise, the intersection of all entries of the SOSS will be the input string's resolved script set.

This formulation ignores Common and Inherited scripts, and returns an error when a string contains mixed scripts.	

Section 5.2: Restriction-Level Detection

The content of this section is mostly the same, but I reworded the logic flow within the list of restriction levels.

Cherokee was added because, like Cyrillic and Greek, it shares many glyphs with Latin. Also note that previously, the requirement to conform to the identifier profile was not stated for ASCII-Only.

The statement under Highly Restrictive stating that "this level will satisfy the vast majority of users" was removed.

Restriction Levels 1-5 are defined here for use in implementations. These place restrictions on the use of identifiers according to the appropriate Identifier Profile as specified in Section 3, Identifier Characters. The lists of Recommended and Aspirational scripts are taken from Table 5, Recommended Scripts and Table 6, Aspirational Use Scripts of [UAX31]. For more information on the use of Restriction Levels, see Section 2.9 Restriction Levels and Alerts in [UTR36]. Whenever scripts are tested for in the following definitions, characters with Script_Extension=Common and Script_Extension=Inherited are ignored. 1. ASCII-Only ○ All characters in each identifier must be ASCII 2. Single Script ○ All characters in each identifier must be from a single script. 3. Highly Restrictive ○ All characters in each identifier must be from a single script, or from any of the following combinations: i. Latin + Han + Hiragana + Katakana; or equivalently: Latn + Jpan ii. Latin + Han + Bopomofo; or equivalently: Latn + Hanb iii. Latin + Han + Hangul; or equivalently: Latn + Kore ○ No characters in the identifier can be outside of the Identifier Profile ○ Note that this level will satisfy the vast majority of users. 4. Moderately Restrictive ○ Allow Latin with other Recommended or Aspirational scripts except Cyrillic and Greek ○ Otherwise, the same as Highly Restrictive 5. Minimally Restrictive ○ Allow arbitrary mixtures of scripts, such as Ωmega, Teχ, HλLF-LIFE, Toys-Я-Us. ○ Otherwise, the same as Moderately Restrictive 6. Unrestricted ○ Any valid identifiers, including	Restriction Levels 1-5 are defined here for use in implementations. These place restrictions on the use of identifiers according to the appropriate Identifier Profile as specified in Section 3, Identifier Characters. The lists of Recommended and Aspirational scripts are taken from Table 5, Recommended Scripts and Table 6, Aspirational Use Scripts of [UAX31]. For more information on the use of Restriction Levels, see Section 2.9 Restriction Levels and Alerts in [UTR36]. 1. ASCII-Only a. All characters in the string are in the identifier profile ***and*** all characters in the string are in the ASCII range. 2. Single Script a. The string classifies as ASCII-Only, ***or*** b. All characters in the string are in the identifier profile ***and*** the string is single-script, according to the definition in Section 5.1. 3. Highly Restrictive a. The string classifies as Single Script, ***or*** b. All characters in the string are in the identifier profile ***and*** the string is covered by any of the following sets of scripts, according to the definition in Section 5.1: i. Latin + Han + Bopomofo (or equivalently: Latn + Hanb) ii. Latin + Han + Hiragana + Katakana (or equivalently: Latn + Jpan) iii. Latin + Han + Hangul (or equivalently: Latn +Kore) 4. Moderately Restrictive a. The string classifies as Highly Restrictive, ***or*** b. All characters in the string are in the identifier profile ***and*** the string is covered by Latin and any one other Recommended or Aspirational script, except Cyrillic, Greek, and Cherokee. 5. Minimally Restrictive a. The string classifies as Moderately Restrictive, ***or*** b. All characters in the string are in the identifier profile. (Allow arbitrary mixtures of scripts, such as Ωmega, Teχ,

characters outside of the Identifier Profile, such as I♥NY.org	HλLF-LIFE, Toys-Я-Us.) 6. Unrestricted a. Any valid identifiers, including characters outside of the Identifier Profile, such as I♥NY.org Note that in all levels except ASCII-Only, any character having Script_Extensions {Common} or {Inherited} are allowed in the identifier, as long as those characters meet the Identifier Profile requirements for that Restriction Level.

The following algorithm has been revised in order to be more conformant to the specification above. Consider a string having an SOSS of { {Latn}, {Latn, Mymr}, {Arab} }. This string is covered by {Latn, Arab}, so it falls under Moderately Restrictive. In step 6 of the current algorithm, the algorithm would remove Latn from all entries of the SOSS, leaving us with { {Mymr}, {Arab} }, and since there is no single script covering that set, it would incorrectly return Minimally Restrictive.

These levels can be detected by reusing some of the mechanisms of Section 5.1. For a given input string, the Restriction Level is determined by the following logical process: 1. If the string contains any characters outside of the identifer profile, return Unrestricted. 2. If no character in the string is above 0x7F, return ASCII. 3. Compute SOSS as in Mixed Script Detection. 4. If a single script covers SOSS, return Single Script. 5. If any of the following sets cover SOSS, return Highly Restrictive. a. {Latin, Han, Hiragana, Katakana} b. {Latin, Han, Bopomofo} c. {Latin, Han, Hangul} 6. Remove Latin from each element of SOSS. Then if SOSS contains any single Recommended or Aspirational script except Cyrillic or Greek, return Moderately Restrictive. 7. Otherwise, return Minimally Restrictive. The actual implementation of this algorithm can be optimized; as usual, the specification only depends on the results.	To detect the strictest Restriction Level for a given input string, the following logical process can be followed: 1. If the string contains any characters outside of the identifer profile, return Unrestricted. 2. If no character in the string is above 0x7F, return ASCII-Only. 3. Compute the string's SOSS according to Section 5.1. 4. If the SOSS is empty or if the intersection of all entries in the SOSS is nonempty, return Single Script. 5. Remove all entries from SOSS that contain Latin. 6. If any of the following sets cover SOSS, return Highly Restrictive. a. {Hani, Hira, Kata}, or equivalently {Kore} b. {Hani, Bopo}, or equivalently {Hanb} c. {Hani, Hang}, or equivalently {Jpan} 7. If the intersection of the entries in SOSS contains any single Recommended or Aspirational script except Cyrillic, Greek, or Cherokee, return Moderately Restrictive. 8. Otherwise, return Minimally Restrictive. The actual implementation of this algorithm can be optimized; as usual, the specification only depends on the results.

Appendix A: Table of Resolved Script Sets (referenced in the new Section 5.1)

This table should be embedded directly into Section 5.1. It is in an appendix in this proposal only because it would not fit with the other inline changes.

Caption: This table uses the four-letter script codes from ISO 15924. The symbol *Σ* denotes the set of all script values.

String	Code Points	Script_Extensions	Augmented Script Sets	Resolved Script Set	Is single-script string?
Circle	U+0043 U+0069 U+0072 U+0063 U+006C U+0065	{Latn} {Latn} {Latn} {Latn} {Latn} {Latn}	{Latn} {Latn} {Latn} {Latn} {Latn} {Latn}	{Latn}	Yes
СігсӀе	U+0421 U+0456 U+0433 U+0441 U+04C0 U+0435	{Cyrl} {Cyrl} {Cyrl} {Cyrl} {Cyrl} {Cyrl}	{Cyrl} {Cyrl} {Cyrl} {Cyrl} {Cyrl} {Cyrl}	{Cyrl}	Yes
Сirсlе	U+0421 U+0069 U+0072 U+0441 U+006C U+0435	{Cyrl} {Latn} {Latn} {Cyrl} {Latn} {Cyrl}	{Cyrl} {Latn} {Latn} {Cyrl} {Latn} {Cyrl}	{}	No
Circ1e	U+0043 U+0069 U+0072 U+0063 U+0031 U+0065	{Latn} {Latn} {Latn} {Latn} {Zyyy} {Latn}	{Latn} {Latn} {Latn} {Latn} *Σ* {Latn}	{Latn}	Yes
C𝗂𝗋𝖼𝗅𝖾	U+0043 U+1D5C2 U+1D5CB U+1D5BC U+1D5C5 U+1D5BE	{Latn} {Zyyy} {Zyyy} {Zyyy} {Zyyy} {Zyyy}	{Latn} *Σ* *Σ* *Σ* *Σ* *Σ*	{Latn}	Yes
𝖢𝗂𝗋𝖼𝗅𝖾	U+1D5A2 U+1D5C2 U+1D5CB U+1D5BC U+1D5C5 U+1D5BE	{Zyyy} {Zyyy} {Zyyy} {Zyyy} {Zyyy} {Zyyy}	*Σ* *Σ* *Σ* *Σ* *Σ* *Σ*	*Σ*	Yes
〆切	U+3006 U+5207	{Hani, Hira, Kata} {Hani}	{Hani, Hira, Kata, Hanb, Jpan, Kore} {Hani, Hanb, Jpan, Kore}	{Hani, Hanb, Jpan, Kore}	Yes
ねガ	U+306D U+30AC	{Hira} {Kata}	{Hira, Jpan} {Kata, Jpan}	{Jpan}	Yes