L2/03-171

Proposal to enhance the Unicode normalization algorithm

Date: June 2, 2003
Author: Jonathan Kew, SIL International
Address: Horsleys Green
High Wycombe
Bucks HP14 3XL
England
Tel: +44 (1494) 682306
Email: jonathan_kew@sil.org

I. Summary

It is proposed that the definition of canonical equivalence and normalization be extended to recognize a new normative property of Unicode characters, “required composition”. This property specifies an equivalence between a precomposed character and a sequence of *base+mark(s),* similarly to existing canonical decompositions. The difference is that Unicode normalization will always *compose* sequences that correspond to a “required composition”, even in NFD. This permits code sequences involving newly-encoded combining characters to be treated as canonically equivalent to existing precomposed characters *without* compromising the guaranteed stability of normalization forms.

II. Purpose

There is a desire among significant parts of the Arabic-script user community to encode the dots and other marks used to differentiate Arabic letters based on the same “skeletal” forms as combining marks (see L2/03-154). This would allow more flexible usage for scholarly and pedagogical purposes, as well as supporting the use of otherwise unencoded combinations in minority languages.

An obstacle to this desire is the existence of precomposed Arabic letters that would then have alternate representations as sequences, in addition to their individual precomposed forms. Making the existing precomposed letters canonically equivalent to their decomposed counterparts would allow standard normalization processes to deal with the existence of these alternate representations, just as happens with Latin-script accented letters. However, stability requirements prohibit the addition of canonical decompositions to the existing Arabic letters, or their replacement with decomposed representations in normalization.

Failing to make the precomposed letters canonically equivalent to the corresponding sequences of *base+mark(s),* however, means that there is the potential for confusion among code sequences that are visually indistinguishable, and yet not treated as equivalent by processes such as normalization, collation, and searching. This may be confusing for users, who will not easily understand why processes sometimes appear to fail, and leads to potential security concerns associated with “character spoofing”.

This proposal aims to overcome this obstacle by allowing canonical equivalence to be defined for these characters in terms of a new “required composition” property, maintaining the stability of normalized data.

It is possible that other scripts besides Arabic might also wish to make use of such a facility. For example, it would be possible to encode characters supporting a “decomposed” phonological representation of Ethiopic, with “required compositions” for the existing syllabic Ethiopic characters. This could provide greater flexibility to encode extensions of Ethiopic script used for minority languages.

III. Proposal

It is proposed that a new property, *Required Composition*, be defined in the UCD.

The Required Composition property of a character is a sequence of two or more USVs which, as a sequence, are considered canonically equivalent to that character.

In determining canonical equivalence of code sequences, Required Composition is used similarly to Canonical Decomposition, being applied recursively to generate code sequences equivalent to the original character code.

There are two key differences between Required Composition and Canonical Decomposition. First, the Required Composition property may be a sequence of more than two codes, whereas Canonical Decomposition is only ever a singleton or pair. Second, during normalization to either NFC *or NFD* the Required Composition property is used to re-compose characters where possible. This contrasts with Canonical Decomposition, which is used to re-compose *only* in NFC and not in NFD.

No character will have both a Decomposition (either canonical or compatibility) and a Required Composition.

No code sequence corresponding to a Required Composition can ever occur in normalized data, because both C and D normalization forms will compose the sequence, replacing it with its precomposed equivalent.

IV. Algorithm changes

The changes required to the normalization algorithms defined in UAX #15 to support the Required Composition property can be summarized as follows:

- In both Canonical and Compatibility Decomposition steps, the mappings from the Required Composition property are used *in addition to* the existing decomposition mappings.
- In normalization forms C and KC, the final composition step uses the Required Composition mappings of the latest version of Unicode supported by the implementation *in addition to* the canonical mappings of the composition version of the Unicode Character Database.
- In normalization forms D and KD, after performing Canonical or Compatibility Decomposition respectively, compose the resulting string using the Required Composition mappings of the latest version of Unicode supported by the implementation.

1. Definitions

We revise the definition of a *primary composite* in UAX #15 §5 to take account of the Required Decomposition property:

D3. A *primary composite* is a character that has a canonical decomposition mapping in the Unicode Character Database (or has a canonical Hangul decomposition) but is not in the §6 Composition Exclusion Table; or has a Required Composition mapping.

Because the Required Composition mapping may include sequences of more than two codes, it is necessary to revise the composition process so as to allow longer sequences to be composed in a single step. To do this, we add a new definition:

D5. A character X can be *potentially combined* with a character Y if and only if there is a character Z whose Required Composition is longer than two characters and begins with the sequence <X, Y>. Such a *potential composition* is treated as if it were a character for the purposes of D4 and D5, when looking for the existence of further compositions.

2. Rule changes

Then the revised rule for generating Normalization Form C is as follows:

R1. Normalization Form C

The Normalization Form C for a string S is obtained by applying the following process, or any other process that leads to the same result:

1. Generate the canonical decomposition for the source string S according to the decomposition and required composition mappings in the latest supported version of the Unicode Character Database.
2. Iterate through each character C in that decomposition, from first to last.
   a. If C is not blocked from the last starter L, and it can be primary combined with L, then replace L by the composite L-C, and remove C.
   b. If C is not blocked from the last starter L, and it can be potentially combined with L, then apply *R1.2* recursively starting from the potential composition L-C.

The result of this process is a new string S' which is in Normalization Form C.

An exactly similar extension applies to the rule for NFKC. For NFD and NFKD, an equivalent composition step is added to the algorithm, but uses *only* the Required Composition mappings and *not* the Canonical Decompositions.

(An alternative to revising the composition rules in this way would be to encode additional characters so that each intermediate step in composition always corresponds to a valid character. However, this seems a highly artificial reason to encode new characters, and is not absolutely necessary.)

3. Implementation note

While the composition rules that deal with sequences longer than 2 characters are expressed above in terms of *potential compositions*, implementation may be simplified by rewriting the Required Composition mappings so as to always map to sequences of exactly two codes. To do this, one can allocate noncharacter codes (for internal processing only) to take the place of the intermediate compositions that are not present as Unicode characters. The composition step then works just as it always did. If any such noncharacter codes are in the resulting string, they must then be re-decomposed, and composition performed again but without using potential compositions.

Although this sounds like two extra steps, and therefore an inefficient approach, in practice it is easy to check during the initial composition whether any such internal noncharacter codes are used, and only perform the final decomposition and canonical composition steps if at least one such character occurred. This is the approach taken in the example implementation in Perl, and appears to work well.

V. Stability and compatibility concerns

The Unicode Stability Policy includes a section specifically addressing normalization, and any changes to canonical equivalence and normalization must clearly take account of the stability guarantees given.

In particular, the policy states:

> *Decomposition Mapping*
>
> a. no character will be given a decomposition mapping when it did not previously have one
> b. no decomposition mapping will be removed from a character
> c. decomposition mappings will not change in type (canonical to compatibility or vice versa)
> d. the number of characters in a decomposition mapping will not change

These constraints prevent us adding decompositions to existing characters, even if Required Composition rules ensured that normalized data would always use the composed forms.

The main thrust of the stability policy, however, is intended to ensure the stability over time and versions of normalized data:

> If a string contains only characters from a given version of the Unicode Standard (e.g., Unicode 3.1.1), and it is put into a normalized form in accordance with that version of Unicode, then it will be in normalized form according to any past or future versions of Unicode.

It is to ensure compliance with this guarantee that the concept of Required Compositions is being proposed. This allows new combining characters to be added, such that new character sequences are canonically equivalent to existing composed characters, while continuing to use the existing composed characters as the normalized representation in both C and D normalization forms.

The use of low canonical combining class values for the new marks that take part in Required Compositions ensures that the composition step of the normalization algorithm will find the Required Compositions, where possible, before finding other compositions (in NFC). If it appears necessary or desirable to have marks with high combining class values that participate in Required Compositions, it might be necessary to re-write the specification of NFC so as to have two distinct composition steps, one for Required Compositions only, followed by the existing canonical compositon step. However, we do not anticipate there being any real need for this, and expect that the single-step composition model will be adequate for any such marks that may be added to the Standard.

If new data that contains characters subject to Required Composition is passed through an old normalization process, the composition step will of course not take place, and the result will not be normalized according to the new specification. This is not significantly different than the situation that already exists whenever new characters with non-zero combining classes are encoded; older normalization processes will not put the characters into the proper canonical order. The stability policy recognizes the impossibility of guaranteeing that old processes can correctly normalize newer data:

Note: If an implementation normalizes a string that contains characters that are *not* assigned in the version of Unicode that it supports, that string *might not* be in normalized form according to a future version of Unicode. For example, suppose that a Unicode 3.0 program normalizes a string that contains new Unicode 3.1 characters. That string might not be normalized according to Unicode 3.1.

As a test of backward compatibility for the enhanced algorithm, an implementation of normalization with Required Compositions was written based on the Perl module Unicode::Normalize; see the Appendix. PUA codepoints were allocated to the Arabic Modifier Marks proposed in L2/03-154, combining classes assigned as proposed there, and Required Composition rules added for the hundred or so existing precomposed Arabic letters. The resulting normalization process passes the NormalizationTest suite from Unicode 4.0.0 with no differences, confirming that the result of normalization for existing data is unaffected. In addition, it correctly converts—in all normalization forms—the sequences documented as "discouraged" in L2/03-154 (because they are visually identical to existing precomposed characters) to their composed forms.

VI. References

Davis, Mark and Kamal Mansour. 2002. *Proposal to amend Arabic repertoire.* L2/02-021.

Davis, Mark, Jonathan Kew and Kamal Mansour. 2003. *Proposal to encode productive Arabic-script modifier marks.* L2/03-154.

Kew, Jonathan. 2002. *Proposal for extensions to the Arabic block.* L2/02-274.

———. 2002. *Encoding generative Arabic nuktas: normalization concerns.* (http://www.jfkew.plus.com/unicode/Generative-Arabic.pdf)

———. 2003. *Encoding Arabic extensions: options for the future of Unicode.* L2/03-044.

Appendix: Test implementation of enhanced normalization

As a test implementation, the Perl module Unicode::Normalize (version 0.21) was extended to handle the concept of Required Compositions. A context diff listing showing the changes made is given here; the complete module and the associated data files are available on request.

The module was tested using PUA assignments for the proposed Arabic Modifier Marks from L2/03-154, with corresponding combining class and required composition data files. The modified code still passes all test cases from NormalizationTest.txt (version 4.0.0), and in addition gives the expected results for sequences involving the proposed modifier marks.

Performance of NFC and NFKC is marginally slower than the original module when running the complete normalization test suite (overall running time is increased by approximately 5%). This is due to the added flexibility in the *compose()* routine. It can be at least partially eliminated, at the cost of some duplication of code, by maintaining two versions of *compose()*, but it is not clear that this is worthwhile for the marginal benefit it gives.

The performance of NF(K)D is on a par with NF(K)C; in the original version, without the Required Composition rules, the D normalization forms were of course somewhat faster.

File *normalize.diffs*

```
*** Normalize.pm    Thu May 15 12:19:51 2003
--- NormalizeX.pm   Fri May 16 12:21:26 2003
***************
*** 51,58 ****
--- 51,66 ----
      || croak "$PACKAGE: CombiningClass.pl not found";
+ # Append combining class values for experimental Arabic marks
+ $Combin .= do "CombiningClassX.pl"
+     || croak "$PACKAGE: CombiningClassX.pl not found";
+
  our $Decomp = do "unicore/Decomposition.pl"
      || do "unicode/Decomposition.pl"
      || croak "$PACKAGE: Decomposition.pl not found";
+ # Load Required Composition list
+ our $ReqComb = do "RequiredCompositionX.pl"
+     || croak "$PACKAGE: RequiredCompositionX.pl not found";
+
  our %Combin;  # $codepoint => $number    : combination class
  our %Canon;   # $codepoint => \@codepoints : canonical decomp.
***************
*** 65,68 ****
--- 73,80 ----
  our %Compos;  # $1st,$2nd  => $codepoint : composite
+ # New globals for the Required Composition step
+ our %ReqComp; # $1st,$2nd  => $codepoint      : composite
+ our %ReqDecomp;   # $codepoint => \@codepoints    : required decomposition of internal code
+
  {
      my($f, $fh);
***************
*** 153,156 ****
--- 165,204 ----
  }
+ # Initialize data structures for Required Compositions.
+ # These are added to the existing decomposition and composition tables
+ # as well as being kept in the separate Req structures.
+ my $nextNonCharCode = 0x200000;
+ while ($ReqComb =~ /(.+)/g) {
+     my @tab = split /\t/, $1;
+     my $dec = [ _getHexArray($tab[1]) ]; # decomposition
+     my $ini = hex($tab[0]); # initial decomposable character
+
+   $Compat{ $ini } = $dec;
+
+   $Canon{ $ini } = $dec;
+
+   if (@$dec == 2) {
+       $Compos{ $dec->[0] }{ $dec->[1] } = $ini;
+       $ReqComp{ $dec->[0] }{ $dec->[1] } = $ini;
+       $Comp2nd{ $dec->[1] } = 1;
+   } elsif (@$dec == 3) {  # use a noncharacter as an intermediate code for the initial pair
+       my $internalCode;
+       if (exists $Compos{$dec->[0]} && exists $Compos{$dec->[0]}{$dec->[1]}) {
+           $internalCode = $Compos{$dec->[0]}{$dec->[1]};
+       } else {
+           $internalCode = $nextNonCharCode++;
+           $ReqDecomp{$internalCode} = [@{$dec}[0,1]];
+           $Compos{$dec->[0]}{$dec->[1]} = $internalCode;
+           $ReqComp{$dec->[0]}{$dec->[1]} = $internalCode;
+           $Comp2nd{$dec->[1]} = 1;
+       }
+       $Compos{$internalCode}{$dec->[2]} = $ini;
+       $ReqComp{$internalCode}{$dec->[2]} = $ini;
+       $Comp2nd{$internalCode} = 1;
+   } else {
+       croak("Weird Required Composition of U+$tab[0]");
+   }
+ }
+
  # modern HANGUL JUNGSEONG and HANGUL JONGSEONG jamo
  foreach my $j (0x1161..0x1175, 0x11A8..0x11C2) {
***************
*** 268,271 ****
--- 316,324 ----
  }
+ # Look up Required Composition for given character pair
+ sub getReqComposite ($$) {
+     return $ReqComp{ $_[0] } && $ReqComp{ $_[0] }{ $_[1] };
+ }
+
  sub isExclusion  ($) { exists $Exclus{$_[0]} }
  sub isSingleton  ($) { exists $Single{$_[0]} }
***************
*** 326,330 ****
  ##
! ## string compose(string)
  ##
  ## S : starter; NS : not starter;
--- 379,383 ----
  ##
! ## string compose(string, getCompFn)
  ##
  ## S : starter; NS : not starter;
***************
*** 334,341 ****
  ## NS + NS must not be composed.
  ##
! sub compose ($)
  {
      my @src = unpack_U($_[0]);
      for (my $s = 0; $s+1 < @src; $s++) {
    next unless defined $src[$s] && ! $Combin{ $src[$s] };
--- 387,399 ----
  ## NS + NS must not be composed.
  ##
! ## getCompFn is either getComposite (for NFC) or getReqComposite (for NFD)
! ##
! sub compose ($$)
  {
      my @src = unpack_U($_[0]);
+     my $getCompFn = $_[1];
+     for (my $pass = 0; $pass < 2; $pass++) {
+         my $partialUsed = 0;
          for (my $s = 0; $s+1 < @src; $s++) {
              next unless defined $src[$s] && ! $Combin{ $src[$s] };
***************
*** 353,365 ****
        && $Combin{ $src[$j-1] } == $Combin{ $src[$j] };
!       $c = getComposite($src[$s], $src[$j]);
        # no composite or is exclusion
        next if !$c || $Exclus{$c};
        # replace by composite
        $src[$s] = $c; $src[$j] = undef;
!       if ($blocked) { $blocked = 0 } else { -- $uncomposed_cc }
    }
      }
      return pack_U(grep defined(), @src);
--- 411,437 ----
                      && $Combin{ $src[$j-1] } == $Combin{ $src[$j] };
!                 $c = &$getCompFn($src[$s], $src[$j]);
                  # no composite or is exclusion
                  next if !$c || $Exclus{$c};
+                 if (exists $ReqDecomp{$c}) {
+                     next if $pass > 0;    # don't use intermediate codes on 2nd pass
+                     $partialUsed = 1;
+                 }
+
                  # replace by composite
                  $src[$s] = $c; $src[$j] = undef;
!                 if ($blocked) { $blocked = 0 } else { -- $uncomposed_cc };
!             }
!         }
!         if ($partialUsed) { # used an internal code; may need to decompose it and retry
!           $partialUsed = 0;
!             @src = map { exists $ReqDecomp{$_} ?
!                         ($partialUsed |= 1) && @{$ReqDecomp{$_}} : $_
!                         } grep defined(), @src;
!             next if $partialUsed;
          }
+         last; # no need for second pass unless a partial composition code was used
      }
      return pack_U(grep defined(), @src);
***************
*** 372,379 ****
  use constant COMPAT => 1;
! sub NFD  ($) { reorder(decompose($_[0])) }
! sub NFKD ($) { reorder(decompose($_[0], COMPAT)) }
! sub NFC  ($) { compose(reorder(decompose($_[0]))) }
! sub NFKC ($) { compose(reorder(decompose($_[0], COMPAT))) }
  sub normalize($$)
--- 444,452 ----
  use constant COMPAT => 1;
! # NFK?D forms use the Required Compositions as a final step
! sub NFD  ($) { compose(reorder(decompose($_[0])), \&getReqComposite) }
! sub NFKD ($) { compose(reorder(decompose($_[0], COMPAT)), \&getReqComposite) }
! sub NFC  ($) { compose(reorder(decompose($_[0])), \&getComposite) }
! sub NFKC ($) { compose(reorder(decompose($_[0], COMPAT)), \&getComposite) }
  sub normalize($$)
```