L2/23-140

Setting expectations for grapheme clusters

Norbert Lindenberg, 2023-07-04

The UTC has decided to add conformance clauses to UAX 29, Unicode Text Segmentation, that let implementations "describe their behavior in relation to the default behavior". These clauses are included in the current proposed update for UAX 29.

This makes it more urgent to accurately describe what the segments are that the default behavior creates, and which use cases might require tailoring. In relation to grapheme clusters, the current wording of UAX 29 unfortunately creates unrealistic expectations both for what default grapheme clusters are and what they can be used for.

Proposal

I propose to update UAX 29, Unicode Text Segmentation, to:

- Remove the assumption that grapheme clusters represent "user-perceived characters".
- Clarify that segmentation into grapheme clusters, although it may be useful in a range of use cases, likely requires tailoring to meet the specific requirements of those use cases.

The text below provides the motivation for the proposed changes. The appendix provides proposed changes (in cyan) relative to the current proposed update (yellow) for UAX 29. Similar changes would have to be made to the Core Specification, but those will have to wait until Unicode 16.

Those less familiar with Brahmic scripts can find a good introduction in Richard Ishida's Typographic character units in complex scripts.

Grapheme clusters and user-perceived characters

UAX 29 describes grapheme clusters as "user-perceived characters". For users of many scripts, however, there's not a single definition of "user-perceived character". Instead, they perceive two kinds of units:

- Base-level marks that represent individual consonants, vowels, or other linguistic entities, and that typically are the units of text input. It's quite common for these base-level components to have names.
- Two-dimensional arrangements of these base-level marks that are important units for rendering, line breaking, and editing.

Users of the Korean Hangul script, for example, know *jamo* as base-level marks, which represent individual consonants and vowels, as well as the syllable blocks formed by combining jamo. The jamo ㄱ *giyeog*, ㅣ *i*, and ㅁ *mieum* combine to the syllable block 김 *gim*. Users of Brahmic scripts perceive individual base consonants, conjunct forms, vowel marks, tone marks, and other marks, as well as the orthographic syllables formed by combining the base-level marks. The Khmer characters ស *sa*, ◌្ត *coeng ta*, ◌្រ *coeng ro*, ◌ី *ei* combine to the syllable ស្ត្រី *strei*.

Unicode grapheme clusters tend to be closer to the larger user-perceived units. Hangul text is clearly segmented into syllable blocks. For Brahmic scripts, things are less clear. Grapheme clusters may contain several base-level units, but up to Unicode 15 always broke after virama characters. This broke not only within orthographic syllables, but for a number of scripts also within the encoding of conjunct forms that users perceive as base-level units, such as Khmer *coengs* (see subsection *Subscript Consonant Signs* of section 16.4 Khmer of the Unicode Standard). In Unicode 15.1, this is being corrected for six scripts, while leaving the others broken. So, while it might be argued that for Hangul, grapheme clusters represent the larger user-perceived units, for many Brahmic scripts grapheme clusters don't correspond to anything users would perceive as "characters".

One might consider changing the definition of grapheme clusters such that for Brahmic scripts they correspond to orthographic syllables. In fact, the UTC did consider this in 2017 – see L2/17-167R, L2/17-258, and L2/17-222. Various complications and feedback, however, caused it to give up on a fix in UAX 29 and instead to delegate the issue to CLDR – see the exclusions in L2/17-258, feedback on PRI 355, and L2/18-055. CLDR then implemented the fix for six Brahmic scripts that will now be integrated into UAX 29 for Unicode 15.1.

Given that UAX 29 is a rather low-level specification whose specified behavior becomes visible to end users only through use by higher-level functions, it doesn't really matter whether grapheme clusters correspond to "user-perceived characters". More important is whether the default segmentation behavior leads to appropriate behavior in the use cases where it might get applied. Using the generic term "user-perceived characters" rather than specific use cases lets every feedback provider project the needs of their favorite use case onto the generic term, resulting in unnecessary confusion.

Use cases for grapheme clusters

UAX 29 suggests the use of grapheme clusters for a wide range of use cases:

- Identification of "a basic unit of a writing system for a language".
- Collation.
- Regular expressions.
- UI interactions (without specifying which interactions are meant).
- Segmentation for vertical text.
- Identification of boundaries for first-letter styling.
- Counting "character" positions within text.
- Atomic units for word boundaries, line boundaries, and sentence boundaries.
- Implementation of text selection and arrow key movement.
- Backspacing through text (UAX 29 later proposes a distinction between Backspace and Delete keys, with differing behavior).

Other sources suggest additional uses:

- Section 2.11 *Combining Characters* of the Unicode Standard suggests their use in rendering.
- UAX 14, Unicode Line-Breaking Algorithm, suggests the use for emergency line breaks.
- The CSS Text specification uses extended grapheme clusters as the foundation for its "typographic character unit", and adds tracking and justification as additional uses. However, it warns that "the default rules are not always appropriate or ideal" and advises that user agents are "expected to tailor them differently depending on the operation as needed".

The phrase "differently depending on the operation" seems essential. There's no single set of segmentation rules that works for all the use cases listed above. Here are a few examples:

- Line breaking rules need to ensure that the text before and after the line break can be rendered independent of the segment that ends up on the other line. If a line would start with a (spacing) combining mark that rendering systems draw with a dotted circle when encountered without a base, then no line break can occur before that combining mark.
- When setting an insertion points to correct a typo, users likely want to set it so as to minimize the number of characters they have to delete and retype. As L2/11-114 argues, and UAX 29 acknowledges with its exclusions from the SpacingMark category for some scripts, users should be able to place an insertion point between a base and a spacing combining mark at least where no reordering across the insertion point can occur. This tailoring, however, is in direct conflict with the above requirement for line breaking.
- When deleting characters, it seems reasonable to expect that the base-level units of text are deleted as atomic units, and that, where a base-level unit is encoded using multiple code points, these code points are not exposed. This expectation is, however, contradicted both by the statement in UAX 29 that "on a given system the *backspace key* might delete by code point" and by the current grapheme cluster breaks after viramas, which in numerous scripts break apart two-code point sequences encoding conjunct forms such as Khmer *coengs*.
- For "those relatively rare circumstances where programmers need to supply end users with user-perceived character counts", one would have to find out what units users of each script (or language) want to have counted. The current grapheme clusters are unlikely to give expected results for a number of scripts. Given that counting user-perceived characters is a rare need, it's not clear that the Unicode Standard needs to provide a ready-made solution for it.

At this point, it's not clear which of these use cases is really the design focus for grapheme clusters. With the proposed update, grapheme clusters correspond (mostly) to orthographic syllables in some Brahmic scripts; in others they don't. In some scripts, clusters break before spacing combining marks, in others they don't. It's not clear if there's any use case that's visible to end users for which grapheme clusters today work out of the box, without tailoring, across all of Unicode.

Making grapheme clusters usable out of the box requires at least specifying clearly which use case is the design focus for default grapheme clusters, and making the necessary changes to make them work for that use case. A good candidate might be a "string safe as a rendering unit", taking in an OpenType cluster, but sometimes more, e.g., as needed for Batak reordering.

Beyond that, UAX 29 might specify multiple tailorings for additional important use cases. One tailoring might define the character sequences between which insertion points can be placed, another the base-level units.

Until that happens, though, UAX 29 should advise implementers and users that they have to create their own tailorings, as the CSS Text specification already does.

Acknowledgments

I’d like to thank Martin Hosken for feedback on a draft of this proposal.

PROPOSED CHANGES TO:

Unicode® Standard Annex #29

UNICODE TEXT SEGMENTATION

Review note: Horizontal lines indicate where this document omits sections of UAX 29 for which no changes are proposed.

Summary

This annex describes guidelines for determining default segmentation boundaries between certain significant text elements: grapheme clusters ~~(“user-perceived characters”)~~, words, and sentences. For line boundaries, see [UAX14] .

---

1 Introduction

This annex describes guidelines for determining default boundaries between certain significant text elements: ~~user-perceived characters~~ grapheme clusters, words, and sentences. The process of boundary determination is also called *segmentation*.

A string of Unicode-encoded text often needs to be broken up into text elements programmatically. Common examples of text elements include what users think of as characters, the smallest spans of text suitable for selection or rendering, words, lines (more precisely, where line breaks are allowed), and sentences. The precise determination of text elements may vary according to orthographic conventions for a given script or language. The goal of matching user perceptions cannot always be met exactly because the text alone does not always contain enough information to unambiguously decide boundaries. For example, the *period* (U+002E FULL STOP) is used ambiguously, sometimes for end-of-sentence purposes, sometimes for abbreviations, and sometimes for numbers. In most cases, however, programmatic text boundaries can match user perceptions quite closely, although sometimes the best that can be done is not to surprise the user.

Rather than concentrate on algorithmically searching for text elements (often called *segments*), a simpler and more useful computation instead detects the *boundaries* (or *breaks*) between those text elements. The determination of those boundaries is often critical to performance, so it is important to be able to make such a determination as quickly as possible. (For a general discussion of text elements, see *Chapter 2, General Structure*, of [Unicode].)

The default boundary determination mechanism specified in this annex provides a straightforward and efficient way to determine some of the most significant boundaries in text: ~~user-perceived characters~~ grapheme clusters, words, and sentences. Boundaries used in line breaking (also called *word wrapping*) are defined in [UAX14].

representational power, place requirements on both the specification of text element boundaries and the underlying implementation. The specification needs to allow the designation of large sets of characters sharing the same characteristics (for example, uppercase letters), while the implementation must provide quick access and matches to those large sets. The mechanism also must handle special features of the Unicode Standard, such as nonspacing marks and conjoining jamos.

The default boundary determination builds upon the uniform character representation of the Unicode Standard, while handling the large number of characters and special features such as nonspacing marks and conjoining jamos in an effective manner. As this mechanism lends itself to a completely data-driven implementation, it can be tailored to particular orthographic conventions or user preferences without recoding.

As in other Unicode algorithms, these specifications provide a *logical* description of the processes: implementations can achieve the same results without using code or data that follows these rules step-by-step. In particular, many production-grade implementations will use a state-table approach. In that case, the performance does not depend on the complexity or number of rules. Rather, performance is only affected by the number of characters that may match *after* the boundary position in a rule that applies.

2 Conformance

There are many different ways to divide text elements corresponding to ~~user-perceived characters~~ grapheme clusters, words, and sentences, and the Unicode Standard does not restrict the ways in which implementations can produce these divisions. However, it does provide conformance clauses to enable implementations to clearly describe their behavior in relation to the default behavior.

***UAX29-C1*. *Extended Grapheme Cluster Boundaries:* An implementation shall choose either *UAX29-C1-1* or *UAX29-C1-2* to determine whether an offset within a sequence of characters is an extended grapheme cluster boundary.**

__UAX29-C1-1__. Use the property values defined in the Unicode Character Database [UCD] and the __extended__ rules in Section 3.1 Grapheme Cluster Boundary Rules to determine the boundaries.

The default grapheme clusters are also known as __extended grapheme clusters__.

__UAX29-C1-2__. Declare the use of a __profile__ of __UAX29-C1-1__, and define that profile with a precise specification of any changes in property values or rules and/or provide a description of programmatic overrides to the behavior of __UAX29-C1-1__.

- Legacy grapheme clusters are such a profile.

__UAX29-C2. Word Boundaries:__ An implementation shall choose either UAX29-C2-1 or UAX29-C2-2 to determine whether an offset within a sequence of characters is a word boundary.

[UCD] and the rules in Section 4.1 Default Word Boundary Specification to determine the boundaries.

UAX29-C2-2. *Declare the use of a profile of UAX29-C2-1, and define that profile with a precise specification of any changes in property values or rules and/or provide a description of programmatic overrides to the behavior of UAX29-C2-1.*

UAX29-C3. Sentence Boundaries: *An implementation shall choose either UAX29-C3-1 or UAX29-C3-2 to determine whether an offset within a sequence of characters is a sentence boundary.*

UAX29-C3-1. *Use the property values defined in the Unicode Character Database [UCD] and the rules in Section 5.1 Default Sentence Boundary Specification to determine the boundaries.*

UAX29-C3-2. *Declare the use of a profile of UAX29-C3-1, and define that profile with a precise specification of any changes in property values or rules and/or provide a description of programmatic overrides to the behavior of UAX29-C3-1.*

This specification defines *default* mechanisms; more sophisticated implementations can *and should* tailor them for particular locales or environments and, for the purpose of claiming conformance, document the tailoring in the form of a profile. For example, reliable detection of word boundaries in languages such as Thai, Lao, Chinese, or Japanese requires the use of dictionary lookup or other mechanisms, analogous to English hyphenation. An implementation therefore may need to provide means for a programmatic override of the default mechanisms described in this annex. Note that a profile can both add and remove boundary positions, compared to the results specified by ***UAX29-C1-1***, ***UAX29-C2-1***, or ***UAX29-C3-1***.

Notes:

- Locale-sensitive boundary specifications, including boundary suppressions, can be expressed in LDML [UTS35]. Some profiles are available in the Common Locale Data Repository [CLDR].
- Some changes to rules and data are needed for best segmentation behavior of additional emoji zwj sequences [UTS51]. Implementations are strongly encouraged to use the extended text segmentation rules in the latest version of CLDR.

To maintain canonical equivalence, all of the following specifications are defined on text normalized in form NFD, as defined in Unicode Standard Annex #15, “Unicode Normalization Forms” [UAX15]. Boundaries never occur within a combining character sequence or conjoining sequence, so the boundaries within non-NFD text can be derived from corresponding boundaries in the NFD form of that text. For convenience, the default rules have been written so that they can be applied directly to non-NFD text and and yield equivalent results. (This may not be the case with tailored default rules.) For more information, see Section 6, *Implementation Notes*.

Grapheme clusters were originally, and for many scripts still are, intended to approximate “user-perceived characters”. It is important to recognize that what the user thinks of as a “character”—a basic unit of a writing system for a language—may not be just a single Unicode code point. Instead, that basic unit may be made up of multiple Unicode code points. To avoid ambiguity with the computer use of the term *character,* this is called a *user-perceived character*. For example, “G” + *grave-accent* is a *user-perceived character*: users think of it as a single character, yet is actually represented by two Unicode code points. ~~These user-perceived characters are approximated by what is called a *grapheme cluster*, which can be determined programmatically.~~

However, for a number of scripts users actually perceive two types of units:

- Base-level marks that represent individual consonants, vowels, or other linguistic entities, and that typically are the units of text input. It’s quite common for these base-level components to have names.
- Two-dimensional arrangements of these base-level marks that are important units for rendering, line breaking, and editing.

Users of the Korean Hangul script, for example, know *jamo* as base-level marks, which represent individual consonants and vowels, as well as the syllable blocks formed by combining jamo. The jamo ㄱ *giyeog*, ㅣ *i*, and ㅁ *mieum* combine to the syllable block 김 *gim*. Users of Brahmic scripts perceive individual base consonants, conjunct forms, vowel marks, tone marks, and other marks, as well as the orthographic syllables formed by combining the base-level marks. The Khmer characters ស *sa*, ្ត *coeng ta*, ្រ *coeng ro*, ើ *ei* combine to the syllable ស្ត្រី *strei*.

The default behavior for grapheme clusters leans towards the larger user-perceived units. Hangul text is segmented into syllable blocks, not into jamo. For Brahmic scripts, grapheme clusters may be complete orthographic syllables or fragments thereof, depending on script, but generally not individual base-level units.

Grapheme cluster boundaries are important for collation, regular expressions, UI interactions, segmentation for vertical text, identification of boundaries for first-letter styling, and counting “character” positions within text. It should be understood that these operations differ somewhat in their requirements, and so the default behavior often needs to be tailored, differently depending on the operation.

Word boundaries, line boundaries, and sentence boundaries should not occur within a grapheme cluster: in other words, a grapheme cluster should be an atomic unit with respect to the process of determining these other boundaries.

As far as a user is concerned, the underlying representation of text is not important, but it is important that an editing interface present a uniform implementation of what the user thinks of as characters. Grapheme clusters can be treated as units, by default, for processes such as the formatting of drop caps, as well as the and so forth. For example, when a grapheme cluster is represented internally by a character sequence consisting of base character + accents, then using the right arrow key would skip from the start of the base character to the end of the last accent.

This document defines a default specification for grapheme clusters. It may be customized for particular languages, operations, or other situations. The need to customize to meet the requirements of specific operations has already been mentioned. In addition ~~For example~~, arrow key movement could be tailored by language, or could use knowledge specific to particular fonts to move in a more granular manner, in circumstances where it would be useful to edit individual components. This could apply, for example, to the complex editorial requirements for the Northern Thai script Tai Tham (Lanna). Similarly, editing a grapheme cluster element by element may be preferable in some circumstances. For example, on a given system the *backspace key* might delete by ~~code point~~ base-level mark, while the *delete key* may delete an entire cluster.

Moreover, there is not a one-to-one relationship between grapheme clusters and keys on a keyboard. A single key on a keyboard may correspond to a whole grapheme cluster, a part of a grapheme cluster, or a sequence of more than one grapheme cluster.

Grapheme clusters can only provide an approximation of where to put cursors. Detailed cursor placement depends on the text editing framework. The text editing framework determines where the edges of glyphs are, and how they correspond to the underlying characters, based on information supplied by the lower-level text rendering engine and font. For example, the text editing framework must know if a digraph is represented as a single glyph in the font, and therefore may not be able to position a cursor at the proper position separating its two components. That framework must also be able to determine display representation in cases where two glyphs overlap—this is true generally when a character is displayed together with a subsequent nonspacing mark, but must also be determined in detail for complex script rendering. For cursor placement, grapheme clusters boundaries can only supply an approximate guide for cursor placement using least-common-denominator fonts for the script.

~~In those relatively rare circumstances where programmers need to supply end users with user-perceived character counts, the counts should correspond to the number of segments delimited by grapheme cluster boundaries.~~ Grapheme clusters *may also be* used in searching and matching; for more information, see Unicode Technical Standard #10, “Unicode Collation Algorithm” [UTS10], and Unicode Technical Standard #18, “Unicode Regular Expressions” [UTS18].

The Unicode Standard provides a default algorithm for determining grapheme cluster boundaries; the default grapheme clusters are also known as **extended grapheme clusters**. For backwards compatibility with earlier versions of this specification, the Standard also defines a profile for **legacy grapheme clusters**.

These algorithms can be adapted to produce ***tailored grapheme clusters*** for between these concepts. The tailored examples are only for illustration: what constitutes a grapheme cluster will depend on the customizations used by the particular tailoring in question.

Table 1a. Sample Grapheme Clusters

Review note: No changes are proposed in this table; it is therefore omitted.

*See also: Where is my Character?, and the UCD file **NamedSequences.txt** [Data34].*

A ***legacy grapheme cluster*** is defined as a base (such as A or カ) followed by zero or more continuing characters. One way to think of this is as a sequence of characters that form a “stack”.

The base can be single characters, or be any sequence of Hangul Jamo characters that form a Hangul Syllable, as defined by D133 in The Unicode Standard, or be a pair of Regional_Indicator (RI) characters. For more information about RI characters, see [UTS51].

The continuing characters include nonspacing marks, the Join_Controls (U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER) used in Indic languages, and a few spacing combining marks to ensure canonical equivalence. There are cases in Bangla, Khmer, Malayalam, and Odiya in which a ZWNJ occurs after a consonant and before a *virama* or other combining mark. These cases should not provide an opportunity for a grapheme cluster break. Therefore, ZWNJ has been included in the Extend class. Additional cases need to be added for completeness, so that any string of text can be divided up into a sequence of grapheme clusters. Some of these may be *degenerate* cases, such as a control code, or an isolated combining mark.

An ***extended grapheme cluster*** is the same as a legacy grapheme cluster, with the addition of some other characters. The continuing characters are extended to include all spacing combining marks, such as the spacing (but dependent) vowel signs in Indic scripts. For example, this includes U+093F (ि) DEVANAGARI VOWEL SIGN I. The extended grapheme clusters should be used in implementations in preference to legacy grapheme clusters, because they provide better results for Indic scripts such as Tamil or Devanagari in which editing by orthographic syllable is typically preferred. For scripts such as Thai, Lao, and certain other Southeast Asian scripts, editing by visual unit is typically preferred, so for those scripts the behavior of extended grapheme clusters is similar to (but not identical to) the behavior of legacy grapheme clusters.

For the rules defining the boundaries for grapheme clusters, see *Section 3.1*. For more information on the composition of Hangul syllables, see *Chapter 3, Conformance*, of [Unicode].

> *Note: The boundary between default Unicode grapheme clusters can be determined by just the two adjacent characters. See Section 7, Testing, for a*

A key feature of default Unicode grapheme clusters (both legacy and extended) is that they remain unchanged across all canonically equivalent forms of the underlying text. Thus the boundaries remain unchanged whether the text is in NFC or NFD. Using a grapheme cluster as the fundamental unit of matching thus provides a very clear and easily explained basis for canonically equivalent matching. This is important for applications from searching to regular expressions.

Another key feature is that default Unicode grapheme clusters are atomic units with respect to the process of determining the Unicode default word, and sentence boundaries. They are usually—but not always—atomic units with respect to line boundaries: there are exceptions due to the special handling of spaces. For more information, see *Section 9.2 Legacy Support for Space Character as Base for Combining Marks* in [UAX14].

Grapheme clusters can be tailored to meet further requirements. Such tailoring is permitted, but the possible rules are outside of the scope of this document. One example of such a tailoring would be for the *aksaras*, or *orthographic syllables*, used in many Indic scripts. Aksaras usually consist of a consonant, sometimes with an inherent vowel and sometimes followed by an explicit, dependent vowel whose rendering may end up on any side of the consonant letter base. Extended grapheme clusters include such simple combinations.

However, aksaras may also include one or more additional prefixed consonants, typically with a *virama* (halant) character between each pair of consonants in the sequence. Such consonant cluster aksaras are not incorporated into the default rules for extended grapheme clusters, in part because not all such sequences are considered to be single “characters” by users. Indic scripts vary considerably in how they handle the rendering of such aksaras—in some cases stacking them up into combined forms known as consonant conjuncts, and in other cases stringing them out horizontally, with visible renditions of the halant on each consonant in the sequence. There is even greater variability in how the typical liquid consonants (or “medials”), *ya, ra, la,* and *wa*, are handled for display in combinations in aksaras. So tailorings for aksaras may need to be script-, language-, font-, or context-specific to be useful.

> *Note: Font-based information may be required to determine the appropriate unit to use for UI purposes, such as identification of boundaries for first-letter paragraph styling. For example, such a unit could be a ligature formed of two grapheme clusters, such as* لا *(Arabic* lam + alef*).*

The Unicode definitions of grapheme clusters are defaults: not meant to exclude the use of more sophisticated definitions of tailored grapheme clusters where appropriate. Such definitions may more precisely match the user expectations within individual languages for given processes. For example, “ch” may be considered a grapheme cluster in Slovak, for processes such as collation. The default definitions are, however, designed to provide a much more accurate match to overall user expectations for what the user perceives of as *characters* than is provided by individual Unicode code points.

Note: The default Unicode grapheme clusters were previously referred to as "locale-independent graphemes." The term cluster is used to emphasize that the term grapheme is used differently in linguistics. For simplicity and to align terminology with Unicode Technical Standard #10, "Unicode Collation Algorithm" [UTS10], the terms default and tailored are preferred over locale-independent and locale-dependent, respectively.

Display of Grapheme Clusters. Grapheme clusters are not the same as ligatures. For example, the grapheme cluster "ch" in Slovak is not normally a ligature and, conversely, the ligature "fi" is not a grapheme cluster. Default grapheme clusters do not necessarily reflect text display. For example, the sequence <f, i> may be displayed as a single glyph on the screen, but would still be two grapheme clusters.

For information on the matching of grapheme clusters with regular expressions, see Unicode Technical Standard #18, "Unicode Regular Expressions" [UTS18].

Degenerate Cases. The default specifications are designed to be simple to implement, and provide an algorithmic determination of grapheme clusters. However, they do *not* have to cover edge cases that will not occur in practice. For the purpose of segmentation, they may also include degenerate cases that are not thought of as grapheme clusters, such as an isolated control character or combining mark. In this, they differ from the combining character sequences and extended combining character sequences defined in [Unicode]. In addition, Unassigned (Cn) code points and Private_Use (Co) characters are given property values that anticipate potential usage.

Combining Character Sequences and Grapheme Clusters. For comparison, *Table 1b* shows the relationship between combining character sequences and grapheme clusters, using regex notation. Note that given alternates (X|Y), the first match is taken. The simple identifiers starting with lowercase are variables that are defined in *Table 1c*; those starting with uppercase letters are **Grapheme_Cluster_Break Property Values** defined in *Table 2*.

Table 1b. Combining Character Sequences and Grapheme Clusters

Review note: No changes are proposed in this table; it is therefore omitted.

Table 1b uses several symbols defined in *Table 1c*. Square brackets and \p{...} are used to indicate sets of characters, using the normal UnicodeSet notion.

Table 1c. Regex Definitions

Review note: No changes are proposed in this table; it is therefore omitted.

Review Note: the new rule GB9c has been implemented in CLDR and ICU as a profile for some years. However, the change to the Regex Definitions above are new, so we'd appreciate review to ensure that they end up with the same result.