L2/18-050

Re: **Clean Up Unicode Segmentation Rules**
From: **Mark Davis**
Date: **2018-01-19**

We have long had a problem with a mismatch in the source for break iterator rules in the Unicode Standard (UAX #29, UAX #14) and CLDR, and the kinds of source that high-efficiency implementations need. This document proposes a change to the rule format that reduces the mismatch, and makes it much easier for implementations to use Unicode segmentation.

Contents

Proposal

Assign an action to produce working-draft versions of UAX #29 and UAX #14 and tests, with the following changes:

1. Modify the sequential rule lists to refactor the following (modifying the subsequent rules for no net change in results):
   a. Remove Treat-As rules
   b. Reflect “Division” (÷) rule requirements in subsequent rules. (Division rules are left in place, but are logically fully reflected in subsequent rules).
2. Clean up the rule set
   a. Have a unique ascending number associated with each rule line, as in the charts.
   b. (Optional) For each character range, use the UnicodeSet syntax, thus eliminating “regex” negations.
3. Expand the tests using some of the process for doing #1, to cover more cases.
4. (Optional) Add EBNF (non-normative) formulations that are equivalent to the sequential rules.

Background

A Unicode break iterator rule list is a sequence of rules executed in order, stopping at the first match. Each rule is of the form:

sequence-before relation sequence-after

The relation is either break (÷, aka Division) or no-break (×). The sequences are a simple subset of regex, with *, +, concatenation, and alternation (|), but also negation (¬) — which in the general case requires lookahead in regex notation. There is one further special rule (→) “Treat whatever on the left side as if it were what is on the right side” which is only used to “absorb” characters.

The rules originated in as set of pairs that could be built into pair-tables. That is easily and efficiently implemented, but soon proved to be inadequate for dealing with the real-world requirements of segmentation.

A more useful set of rules defines expressions for ranges of arbitrary length that should not be broken, starting either from the start of text or from a previous boundary. Those rules can be far more easily transformed into a DFA (deterministic finite automaton). That way the break iterator can find the next break point (forward or backward) much more quickly, avoiding needing to traverse a number of rules on every character. It also means that the ordering of the rules is far less important, making the rules easier to understand.

The Treat As and Division rules are the most problematic. Those currently require a manual process of conversion from UAX #29, UAX #14, and CLDR sources, which is inefficient and error-prone. For comparison, see the current ICU rules for word break (see also readme.txt).

A **Clean Rule Set** follows the same syntax as the current rule sets, but contains no Treat As rules, and the Division rules are can be omitted without affecting the results. Because it follows the same syntax (except narrower), converting to a Clean rule set should not negatively impact any implementations.

- The Division rules are left in, even though they are not formally required, because they are useful for some implementations.

Process

As a part of an action for GCB in http://www.unicode.org/draft/reports/tr29/tr29.html, the process of doing #1 and #3 has been done, and the following reflects the experience of doing that, and draws on the work that Andy has been doing in ICU.

Generate Clean Rules

The following is from working on the GCBs. It is not a fully automated process, but I think is pretty complete. The process of converting is somewhat tedious, but has the advantage of also exposing places where the rules should be improved. We can have tests that verify that the BNF passes the tests: I've done that already for GCB.

Step 1. Remove "treat as" as in Replacing Ignore Rules.
1. define a 'macro' P := (Extend | Format)
2. change the "treat as" rule to × (Extend | Format)*
3. change all rules after the "treat as" rule, such as X Y × Z W ⇒ X P* Y P* × Z P* W P*

Step 1.1. Replace any empty sequence-before or sequence-after by "Any".

Examples

```
GB4     (Control | CR | LF)      ÷
=>
GB4     (Control | CR | LF)      ÷        Any

GB9a                       ×      SpacingMark
=>
GB9a                   Any ×      SpacingMark
```

Step 1.2 *Special cases:* The “sot” / “eot” rules require special handling.

Remove the Any rules:

WB1	sot	÷	Any
WB2	Any	÷	eot

Definition: A rule with ÷ is called a **division rule**.

Definition: String S1 and string S2 **overlap** if there are three strings A, B, and C such that:

- S1 = A + B
- S2 = B + C

Definition: Rule R1 and rule R2 **overlap** iff there are two overlapping strings S1 and S2 such that:

- R1 matches S1
- R2 matches S2

Example 1:

GB4 (Control | CR | LF) ÷ Any
GB9 Any × (Extend | ZWJ | Virama)

These two overlap, because of the strings A=<Control>, B=“”, C=<Extend>. GB4 matches A+B, GB9 matches B+C.

Example 2:

WB6 := AHLetter × (MidLetter | MidNumLetQ) AHLetter
WB7a := Hebrew_Letter × Single_Quote

The overlap here isn’t obvious, but exists because of the “macro” definitions:

AHLetter = (ALetter | Hebrew_Letter)
MidNumLetQ = (MidNumLet | Single_Quote)

Detect Overlapping Rules

While it is possible to analyse the structure of the rules to find overlapping rules, it is simpler and less error-prone to just brute-force it with the following process

Generate lists of strings for each rule with the following process.

1. Partition the characters for the rule set. The main properties are all partitioned, but cases like \p{Extended_Pictographic} need to be accounted for. Easy to do mechanically: for Grapheme_Cluster_Break those are:
    a. Extended_Pictographic_And_E_Base
    b. Extended_Pictographic_And_E_Modifier
    c. Extended_Pictographic_And_Other
2. Pick one *exemplar* character from each partition.
3. Generate a *exemplar list* of strings ELx for each rule Rx, where
    a. the properties are all replaced by their exemplar characters
    b. all the alternatives (a|b|c) are taken

c. each x+ is turned into (x | xx | xxx)
d. each x* into (x | xx | xxx)?.
e. The way the rules are structured, 3 instances is plenty to check all cases we care about.

Rule R1 and rule R2 overlap when there is a string S1 from EL1 and S2 from EL2 that overlap.

Step 2. Resolve each division rule, with the following process.
1. Cycle through all the subsequent rules that overlap.
2. Modify each such rule so that it doesn't overlap.
3. In processing all subsequent rules, that division rule is ignored.

Example

GB4	:=	(Control | CR | LF) ÷
GB9	:=	Any × (Extend | ZWJ | Virama)
⇒		
GB9	:=	¬(Control | CR | LF) × (Extend | ZWJ | Virama)

Step 2.1. Remove regex negations
We do not need the generality of regex negations or in many cases, alternation. Reflect that by converting the syntax for each such case into UnicodeSet notation. This is optional: if we want to keep the "backwards compatible" format we can.

Example

GB9	¬(Control | CR | LF) × (Extend | ZWJ | Virama)
⇒	
GB9	[^Control CR LF] × [Extend ZWJ Virama]
SB8	ATerm Close* Sp* × (¬(OLetter | Upper | Lower | ParaSep | SATerm))* Lower
⇒	
SB8	ATerm Close* Sp* × [^OLetter Upper Lower ParaSep SATerm]* Lower

Add Tests

The process of checking the overlap of rules also lets us general more test cases, ones that "tickle" implementation code more than the current tests.

Step 3.1 Test all the strings in the union U1 of the original rules' exemplar strings.

Step 3.2 Create U2 = U1 × U1, and test it.
U2 is the set of all concatenations of 2 strings from U1. These are not, however, just the simple concatenations. If U1 has exemplar strings "abc" and "bcd", U2 has "abcbcd", and the other direction "bcdabc" *and* the overlap "abcd" (there may be multiple overlaps for strings).

Step 3.3 Create U3 = U1 × U2, and test it.

U3 should be sufficient to test for conformance.

Add EBNF Definitions

The goal is to restate each sequential rule set as an *extended right regular grammar* (ERRG, with

EBNF syntax: Because of the way we have structured the rules, it should be possible to avoid non-regular grammar features like lookahead or lookback.) There are two advantages of this construction.

1. An EBNF is often far easier to understand than a sequential rule list.
2. An ERRG can be turned into a NFA mechanically, which can be converted to a DFA mechanically. There are libraries to do this, so implementations don't have to reinvent the wheel.

This is a draft; we'll undoubtedly modify it as we apply the process to more than GBC.

In the examples below, the results of previously-done steps may not be shown, so that the examples focus on the core change. For example, we'll omit the P's just for clarity.

Since we have done Step 1..Step 2, there are no division rules. Each rule just specifies a continuation.

Step 4. Combine rules with the same right or left sides.
1. Cycle through all the subsequent rules.
2. Add a starred right/left side to beginning or end.
3. If the subsequent rule has an empty other side, add a new rule with both.
4. Combine with other optional elements (see example)
5. After processing all subsequent rules, remove that rule with empty right or left side

    Example:

```
GB9             Any × [Extend ZWJ Virama]
GB9a    :=      Any × SpacingMark
⇒
GB9b    :=      Any ×  [Extend ZWJ Virama SpacingMark]
```

Step 4.1. Special case for sot/eot. Simplify the following

Example:

```
WB15    :=      sot (RI RI)* RI   ×         RI
WB16    :=      [^RI] (RI RI)* RI         ×         RI
to
WB15    :=      RI        ×         RI
```

Step 4.1. Discard the × signs, leaving the rules as assignments to simple regular expressions:

Example:

```
GB9     :=       [Control CR LF] [Extend ZWJ Virama]
WB7b    :=      Hebrew_Letter Double_Quote Hebrew_Letter
```

Step 5. Drop duplicates (where later rule is completely covered by earlier).

Example:

WB7b := Hebrew_Letter Double_Quote Hebrew_Letter
~~WB7c := Hebrew_Letter Double_Quote Hebrew_Letter~~

Many cases are duplicated because they were triples, and needed two rules just to show the placement of ×.

Step 6. Revise remaining rules so there are no overlaps. These will use Kleene algebra simplifications. Rules may disappear.

Example:

```
GB6   :=   L          ×    (L | V | LV | LVT)
GB7   :=   (LV | V)×       (V | T)
GB8   :=   (LVT | T)       ×    T
⇒
GB6   :=   L* (V+ | LV V* | LVT) T* | L+ | T+
```

Step 6.1. Produce EBNF by adding a root, which is the alternation of all remaining rules.

Example:

Root := GB2 | ... | GB6 | ...

Step 7. Retitle and restructure the rules to have a more understandable breakdown:

Example:

```
extended grapheme cluster := crlf | Control | precore* core postcore*
core :=  hangul-syllable | ri-sequence | xpicto-sequence | ...
...
```

Step 8. Test

Make sure that the resulting EBNF is equivalent to the original rules, using the expanded tests.