L2/10-200

Tailoring the Unicode Bidi Algorithm

Murray Sargent III

The Unicode Bidi Algorithm is a very useful, general, and standard approach for displaying text that contains right-to-left scripts, such as Arabic and Hebrew. But there are situations in which it is awkward to use and/or is visually confusing. This note considers four such situations: math zones, International Resource Identifiers (IRIs), HTML spans with a directionality attribute, and parenthesized text in a paragraph. In each of these cases, tailoring the Unicode Bidi Algorithm improves the result. The note starts with a summary of the original case of tailoring, namely the popular keyboard “higher-level protocol” used in rich-text applications.

Keyboard-Driven Bidi Algorithm

For the first example of tailoring the Unicode bidi algorithm, consider keyboard input into rich-text editors, such as Word and RichEdit rich-text edit controls. For this kind of input, the directionality of neutral characters, such as space and parentheses, is determined by the keyboard language. Specifically if the keyboard language is Arabic, Hebrew, or some other right-to-left (RTL) language, the neutrals are treated as RTL characters. If the keyboard language is left-to-right (LTR), the neutrals are treated as LTR characters. Spans of digits, possibly including embedded commas and periods, are always displayed LTR even when entered with an RTL keyboard, but the directionality given to the number they comprise has the directionality of the keyboard language.

Users generally like this approach to bidi text, since it’s simple and predictable. If runs of text could always be reliably stamped by the input language, the Unicode Bidi Algorithm would only have to handle some esoteric text embeddings and ensuring that numbers are displayed LTR. But plain text is commonplace and hence there has to be a way to display bidi text correctly without language attributes. Unfortunately in order to handle the wide variety of special character combinations, the resulting algorithm is very complicated, involving the set of special override codes LRM, RLM, LRO, RLO, LRE, RLE, and PDF. In a real sense, these override codes are used to encode a rich-text directionality attribute in plain text. The justification for this is the Unicode principle that “plain text must contain enough information to permit the text to be rendered legibly and nothing more” (see Plain Text subsection of Sec. 2.2 in Chapter 2 of the Unicode Standard).

Note that plain text pasted into a rich-text instance is treated using the Unicode Bidi Algorithm; the keyboard language doesn’t play a role in pastes.

Bidirectional Text in Math Zones

Consider next the directionality of math text in math zones. The guiding principle is that neutrals other than the period and comma are given the directionality of the math zone. This includes math operators and spaces. Alphabetic spans of characters having the same directionality are displayed with that directionality relative to one another, but the span as a whole is treated as object with the directionality of the math zone. As with the Unicode Bidi Algorithm, spans of digits, possibly including embedded commas and periods, are displayed LTR, but analogously to keyboard language tailoring, the number they comprise has the directionality of the math zone.

Normal text embedded in a math zone obeys the directionality rules of normal text. Typically this is given by the Unicode Bidi Algorithm, but the keyboard language algorithm is used if the text is entered by keyboard.

Internationalized Resource Identifiers

Internationalized Resource Identifiers (IRIs) are a generalization of Universal Resource Identifiers (URIs) that can contain many nonASCII characters, such as most alphabetic characters and Chinese characters. Complications occur when bidi characters are used in IRIs especially when displayed in an RTL context. For example according to the Unicode Bidi Algorithm, http://شس.يب.ثق displays in a right-to-left paragraph as

ثق.يب.شسhttp://

Or more confusing yet, http://exchange. شس.ثق displays in an RTL paragraph as

ثق.شس.http://exchange

As the IRI reference discusses, use of the Unicode Bidi Algorithm at least is consistent in the way such IRIs are displayed in plain text, but in some sense the RTL versions are nearly unreadable. The IRI reference does recommend that IRIs always be displayed in LTR display order. But that requires being able to recognize an IRI in the first place.

So assuming an IRI can be recognized, something that RichEdit does on the fly, we can enforce a more readable display order. Namely we force the delimiters '#', '.', '/', ':', '?', '@', '[', ']' to follow the paragraph (or embedding) direction. With this condition, http://exchange. شس.ثق displays in an RTL paragraph as

شس.ثق.exchange//:http

i.e., the alphanumeric spans in between the special delimiters appear in the reverse order from the way they appear in an LTR paragraph and the Unicode Bidi Algorithm is not used to resolve the neutrality of the slash and period. The alphanumeric spans themselves (in effect the “leaves” of the structure) are displayed in the order determined by the Unicode Bidi algorithm.

This approach appears to be ideal. The only problem is that it’s not trivial to identify IRIs using heuristics. Both Word and RichEdit can identify IRIs, but ambiguities can occur when spaces appear at the ends. Word allows the user to overrule the IRI recognition, but RichEdit 7.0 and earlier versions do not. The IRI and URI specifications require that spaces be “escaped”, that is, replaced by their ASCII hexadecimal code %20.

Additional Requirements for Bidi in HTML

The paper Additional Requirements for Bidi in HTML describes a number of situations in HTML when application of the Unicode Bidi Algorithm yields bizarre, or at least unwanted, results. These situations don’t cause problems in Word or RichEdit rich-text controls, since every format run (similar to <span> in HTML) is assigned directionality. But in HTML, a <span> with a particular directionality influences the way the text next to it is displayed unless that text also has an assigned directionality.

Even simple things can display in surprising ways. For example,

```
10 main st.
```

displays in an RTL context as

```
                                        .main st 10
```

This happens because the number at the start of an RTL paragraph is classified as an RTL object in an RTL context (even though the digits are displayed LTR with respect to one another), and hence the number comes first (to the right). It's followed by the LTR text "`main st`", but the period is classified as RTL and appears last, namely to the left of "`main st`".

Let's look at a couple of examples in that paper that can be fixed by adding a new <span> attribute called bdi for "bidirectional isolate". These examples use the notation that upper-case ASCII letters are strong RTL letters and lower-case ASCII letters are strong LTR. The first example consists of the string "`PURPLE PIZZA – 3 reviews`", which is displayed LTR in logical order (RTL characters are displayed LTR). In an LTR context, one would like this string to be displayed as

```
    AZZIP ELPRUP – 3 reviews
```

But because a number is given the directionality of the (in this case RTL) run preceding it, it displays as

```
    3 – AZZIP ELPRUP reviews
```

The second example in logical order is

```
    USE css (<span dir="ltr">position:relative</span>).
```

One wants this to display in an RTL paragraph as

```
                          .(position:relative) css ESU
```

But because the LTR span continues the LTR `css`, it gets displayed as

```
                          .(css (position:relative ESU
```

The proposed fix is to put the embedded text into a <span> with a bdi attribute, which is then "displayed as if it were surrounded with strong-directional characters of the last explicit embedding level within which it appears" (taken from Sec. 2.1 of the paper referenced above). For the "`PURPLE PIZZA`" case, the " - 3" would then be treated as if it were preceded by a strong LTR character. For the "`USE css`" case, the parentheses would resolve to RTL and get mirrored, giving the desired result above. If you're interested in these problems, please read the full article.

Bidi Paragraph with Parenthesized Text

The embedded parentheses problem in HTML is found in ordinary plain text. An algorithm for displaying such text in a reasonable way is given below. This algorithm first shipped in Microsoft Office 2007 and you can see it in action by typing bidi text into Excel 2007/2010 cells. The problem is sort of mathematical in nature, since parenthesized text is like a parenthesized expression. It can be nested and the text should display inside the parentheses.

Nevertheless, according to the UBA, there are cases for which both parentheses of a parenthesized expression have the same glyph. In a rich-text editor like Microsoft Word if you type

(a)b

using an English keyboard and type Ctrl+RightShift to switch to an RtL paragraph you see

(a)b

This is because the parentheses are stamped with LtR directionality by the keyboard language's directionality, thereby overruling the UBA. In NotePad, which follows the UBA, the LtR paragraph version looks the same as in Word, but the RtL version looks like

a)b)

Here the UBA classifies the opening parenthesis as RtL, thereby appearing first (on the far right) and mirrored instead of to the left of the letter a. Similarly a(b) appears as (a(b in an RtL paragraph.

As a more complicated example, consider the nested case a(bف(فc)dc)d. According to the UBA, this displays in an LtR paragraph as a(bف)فc)dc)d and in an RtL paragraph as

c)dc)dف)فa(b

Neither of these renderings preserves the visual nesting of the characters.

We can fix such parenthesized text displays using a bidi-parenthesis-matching algorithm due to Ayman Aldahleh. A basic idea is to ensure that both parentheses of a matched pair have the same directionality and that what's inside the parentheses has bidi level(s) greater than or equal to the parenthesis pair level. The algorithm uses an open-parenthesis stack along with a parenthesis-pair information array. The algorithm can be easily generalized to handle brackets, braces, and other character pairs, but for simplicity we stick with ASCII parentheses and assume that Unicode bidi embeddings aren't present.

1) Run the UBA on a paragraph noting the bidi levels of the characters.
2) Scan the paragraph for parentheses. When an open parenthesis is found (U+0028), record its bidi level and character position in the next information element and push the element index onto the open-parenthesis stack.
3) If a close parenthesis is found (U+0029) and the stack has an entry, a matched pair is found. Use the element index on the top of the stack to find the corresponding pair information element. If both parentheses have the same bidi level, use that for the pair. If they differ, start by setting the pair level equal to the smaller level. If in an RtL/LtR paragraph, the pair level isn't odd/even, increment the pair level. Set both parenthesis levels equal to the resulting pair level and record the character position ending the pair.
4) If an unpaired or improperly nested parenthesis is found, abandon the matching process.
5) Process the pair information elements from first to last. For each element, if any character inside the pair has a level smaller than the pair level, increment the level by 2. This forces the character to display inside the pair and doesn't change its directionality.

This last step is recursive, since outer pair elements precede elements for any pairs they contain and increment the latters' levels. When an inner pair is processed, its level is guaranteed to be greater than or equal to the parent pair, etc.

For the simple paragraph above in an RtL paragraph, the (a)b appears as (a)b, since the opening parenthesis is promoted to level 2. In a case like a(bفc)d in an RtL paragraph, the a(b and c)d all have level 2 and the ف is bumped up to level 3. To see more cases, type various parenthesized expressions into an Excel cell and click on the paragraph direction tool to change the paragraph direction. If you have properly matched parentheses, the results will always look properly nested. It also works with brackets [] and braces {}.

The text here is adapted from the blog posts Tailoring the Unicode Bidi Algorithm and Bidi Paragraph with Parenthesized Text.