L2/22-072R

Proposal for amendments to UAX#9 and UAX#31

To: Properties & algorithms group, UTC
From: Robin Leroy, Mark Davis, Peter Constable, Source code ad hoc working group
Date: 2022-03-31

The source code ad hoc working group recommends amendments to non-normative text in Unicode Standard Annexes #9 and #31.

Consolidated working drafts of the Unicode Standard Annexes incorporating these amendments may be found at the following URLs:

— UAX#9: https://www.unicode.org/reports/tr9/tr9-45.html, document L2/22-089;
— UAX#31: https://www.unicode.org/reports/tr31/tr31-36.html, document L2/22-090.

Proposed amendment to UAX#9

The working group recommends that the note in Section 4 “Bidirectional Conformance”, Clause UAX9-C2, be amended as follows:

Current:

> *Note:* Use of higher-level protocols is discouraged, because it introduces interchange problems and can lead to security problems. For more information, see Unicode Technical Report #36, “Unicode Security Considerations” [UTR36].

Proposed:

> *Note:* The use of higher-level protocols introduces interchange problems, since the text may be displayed differently as plain text; see Section 6.5, *Conversion to Plain Text*. This can have security implications. However, where the semantics of segment order are more significant than those of displayed order, as is the case for source text, higher-level protocols are recommended. For detailed examples for which use of HL4 would be recommended, see Section 4.3.1, *HL4 Example 1 for XML* and Section 4.3.2, *HL4 Example 2 for Program Text*. For more information, see Unicode Technical Report #36, “Unicode Security Considerations” [UTR36].

The working group recommends that the existing example of the application of HL4 in Section 4.3 “Higher-Level Protocols” be given the section number and title “4.3.1 HL4 Example 1 for XML”, and that the following example be added after it:

> 4.3.2 HL4 Example 2 for Program Text
>
> Consider the following two lines:
>
> (1) `x + tav == 1`
> (2) `x + 1 == תו`

Internally, they are the same except that the ASCII identifier `tav` in line (1) is replaced by the Hebrew identifier `תו` in line (2). However, with a plain text display (with left-to-right paragraph direction) the user will be misled, thinking that line (2) is a comparison between `(x + 1)` and `תו`, whereas it is actually a comparison between `(x + תו)` and `1`. The misleading rendering of (2) occurs because the directionality of the identifier `תו` influences subsequent weakly-directional tokens, so that the entire sequence "`1 == תו`" is at a higher resolved level. This is illustrated in the first row of the following table, wherein characters at a resolved level higher than the embedding level are highlighted. Note that while the RTL display of that expression (second row) is not misleading, as the left-to-right directionality of `x` does not influence the subsequent text, a similar issue would arise if the terms were swapped (third row).

Paragraph direction	Underlying representation											Display
LTR	x		+		ת	ו		=	=		1	x + 1 == תו
RTL	x		+		ת	ו		=	=		1	1 == תו + x
RTL	ת	ו		+		x		=	=		1	x == 1 + תו

It is better to apply protocol HL4 when displaying these expressions, treating each identifier as a separate segment, thus isolating it from the rest of the source text, and then ordering the segments in a consistent direction, as shown in the following table.

Segment order	Segments									Display
LTR	x		+		תו		==		1	x + תו == 1
RTL	x		+		תו		==		1	1 == תו + x
RTL	תו		+		x		==		1	1 == x + תו

[Editor's note: the "Display" column in the preceding table ought to be implemented in HTML using `<span>` tags with `dir` attributes around each segment.]

Proposed amendment to UAX#31

The working group recommends that the first sentence of Section 4 "Pattern Syntax" be reworded as follows:

Current:

> There are many circumstances where software interprets patterns that are a mixture of literal characters, whitespace, and syntax characters.

Proposed:

> Most programming languages have a concept of whitespace as part of their lexical structure, as well as some set of characters that are disallowed in identifiers but have syntactic use,

such as arithmetic operators. Beyond general programming languages, there are also many circumstances where software interprets patterns that are a mixture of literal characters, whitespace, and syntax characters.

The working group recommends that the following note and example be added to [Section 4 “Pattern Syntax”, Clause UAX31-R3](), after the existing note:

> Note: This requirement is relevant even for languages that do not use immutable identifiers, or that have lexical structure outside of the categories of syntax and whitespace characters. In particular, the set of Pattern_White_Space characters is chosen to make it possible to correct bidirectional ordering issues that can arise in a wide range of programming languages, visually obfuscating the logic of expressions. In the absence of higher-level protocols (see Section 4.3, *Higher-Level Protocols*, in [UAX9]), tokens may be visually reordered by the Unicode Bidi Algorithm in bidirectional source text, producing a visual result that conveys a different logical intent. To remedy that, two implicit directional marks are among Pattern_White_Space characters; if these can be freely inserted between tokens, implicit directional marks *consistent with the paragraph direction* can be used to ensure that the visual order of tokens matches their logical order.
>
> Since the implicit directional marks are nonspacing, where a syntax requires a sequence of spaces (such as between identifiers), it should require that at least one of those be neither LEFT-TO-RIGHT MARK nor RIGHT-TO-LEFT MARK. The visual appearance would otherwise be too confusing to readers: “`else`⟨LRM⟩`if`” would be seen by the user as “`elseif`” but parsed by the compiler as “`else if`”, whereas “`else`⟨LRM⟩ `if`” would be seen and parsed as “`else  if`” and be harmless.
>
> Example: Consider the following two lines:
>
> (3) `x + tav == 1`
> (4) `x + 1 == תו`
>
> Internally, they are the same except that the ASCII identifier `tav` in line (1) is replaced by the Hebrew identifier `תו` in line (2). However, with a plain text display (with left-to-right paragraph direction) the user will be misled, thinking that line (2) is a comparison between `(x + 1)` and `תו`, whereas it is actually a comparison between `(x + תו)` and `1`. The misleading rendering of (2) occurs because the directionality of the identifier `תו` influences subsequent weakly-directional tokens; inserting a left-to-right mark after the identifier `תו` stops it from influencing the remainder of the line, and thus yields a better rendering in plain text with left-to-right paragraph direction, as demonstrated in the following table, wherein characters whose ordering is affected by that identifier have been highlighted.

Underlying representation												Display (LTR paragraph direction)
x		+		ת	ו			=	=		1	x + 1 == תו
x		+		ת	ו	⟨LRM⟩		=	=		1	x + תו == 1

The simplest automatic mechanism for placement of LRM characters is around every identifier, string literal, and comment that contains RTL characters. However, this can also be reduced in some cases.

Note: Left-to-right marks are used for this purpose when the main direction is left–to-right. Correspondingly, right-to-left marks are used when the main direction is right-to-left.

Rationale

While the working group has yet to define the specifics of the desired rendering of source code, and of the means by which this rendering may be achieved, some broad outlines are already clear. Future guidance will make use of some mechanisms already described in these annexes, some of which have been standardized for very similar purposes; indeed these mechanisms are already in use in existing implementations: Visual Studio implements UAX#9 HL4[1]; multiple syntaxes defined by the Locale Data Markup Language[2], Ada 2012[3] and later, Rust[4] 1.9 and later allow for implicit directional marks wherever whitespace is allowed.

Context for the possible applications of those mechanisms is however lacking in the annexes. The proposed amendments provide that context.

Specifically:

1. The current note in UAX9-C2 discourages what is a very promising mitigation to a security issue—and one that has been in use for a long time in a major editor (Visual Studio).
2. The current example for HL4 (4.3.1 with the proposed numbering) is focused on markup languages, so that it may not be obvious that it applies to general programming languages; it also fails to illustrate the problems with *not* using higher-level protocols.
3. UAX#31 defines requirement UAX31-R3 and the usage of Pattern_White_Space as whitespace in the context of "patterns that are a mixture of literal characters, whitespace, and syntax characters", but, while general programming languages were not the focus when that was defined, the intent was not to limit its applicability so strictly. This is evidenced by the existing note in UAX31-R3, which refers to identifiers: those are not literals, whitespace, nor syntax. Indeed, the editor of UAX#31 went on[5] to make use of Pattern_White_Space in the plural rules syntax in UTS#35, even though the plural rules syntax does not fit that definition of pattern. Other non-pattern languages have made use of Pattern_White_Space, see, *e.g.*, Rust, which even claims conformance to UAX31-R3. However, the wording of Section 4 has led to confusion about the applicability of UAX31-R3 to a wider domain, including general programming languages.

[1] Visual Studio treats each token, including identifiers, string literals and comments, as a separate segment.

[2] See, *e.g.*, the syntaxes for UnicodeSet, plural rules, collation rules.

[3] ISO/IEC 8652:2012; see the *Annotated Ada Reference Manual*, 2.2(7.1/3).

[4] See *The Rust Reference*, 2.5.

[5] See Mark Davis, UTS#35 version 1.7.2, Section C.11 "Language Plural Rules" (2009-12-09).

4. UAX#31 provides no rationale for the set of characters in Pattern_White_Space, in particular for the two nonspacing characters therein, nor guidance for the proper use of implicit directional marks.

The proposed changes are non-normative: we are merely documenting long-standing mechanisms, rather than defining new ones. In particular, this means that an implementation which makes normative references to earlier versions of Unicode should be able to make use of these mechanisms.