

## Proposed changes to Unicode properties and reports for source code handling

To: PAG, ESC, UTC  
From: Robin Leroy, Mark Davis, Source code ad hoc working group  
Date: 2022-10-20

---

The source code ad hoc working group was created by consensus [170-C2](#) of the UTC, on the recommendation of the Properties & Algorithms Group, as described in document [L2/22-007R2](#), section “Proposed Plan”, with Mark Davis as the chair.

Its goals are the following (goals whose completion had already been reported at UTC #172 are struck through).

- |  |
|--|
| <ul style="list-style-type: none"><li><del>A. Engage with MITRE to get more accurate wording into the CVE records.</del></li><li>B. Assemble documentation providing guidance for avoiding spoofing issues. Make that available for review and feedback.</li><li>C. Produce Unicode documentation, such as draft proposed updates of UAX #9 (“Bidi”, aka UBA), UAX #31 (“Identifiers”), UTR #36 (“Security”), and UTS #39 (“Security Mechanisms”) using the information in B, and post for comment.</li><li>D. In ICU, <del>respond to tickets filed, and</del> provide code snippets and/or APIs to implement utility functions that could be used directly to help avoid problems. (The implementations could also be ported to other languages.)</li><li>E. Examine whether new properties and/or property values, or changes to values, would be useful.</li></ul> |
|--|

This document consists of proposed changes to properties, proposed changes to Unicode reports, and a proposed new Unicode Technical Standard. It fulfills goals B, C, and E. Goal D will require the involvement of ICU-TC; we note however that work is in progress to provide reference implementations independent from ICU for some of the algorithms; this is alluded to in *Section 5, Reference Implementations*, in the proposed UTS #55.

**Contents**

<b>Contents</b>	<b>2</b>
<b>P. Proposed changes to properties</b>	<b>5</b>
<b>Proposed changes to existing technical reports</b>	<b>6</b>
<b>#9. Proposed changes to Unicode Standard Annex #9</b>	<b>7</b>
4 Bidirectional Conformance	7
6.5 Conversion to Plain Text	7
<b>#14. Proposed changes to Unicode Standard Annex #14</b>	<b>8</b>
BK: Mandatory Break (A) (Non-tailorable)	8
<b>#31. Proposed changes to Unicode Standard Annex #31</b>	<b>9</b>
1 Introduction	9
1.4 Conformance	10
2 Default Identifiers	10
2.3 Layout and Format Control Characters	13
2.3.1 Limited Contexts for Joining Controls	14
2.3.2 Limitations	14
3 Immutable Identifiers	15
4 Pattern Whitespace and Syntax	16
4.1 Whitespace	17
4.1.1 Bidirectional Ordering	18
4.1.2 Required Spaces	19
4.1.3 Contexts for Ignorable Format Controls	19
4.2 Syntax	20
4.2.1 User-Defined Operators	21
4.3 Pattern syntax	22
7 Standard Profiles	22
7.1 Mathematical notation profile	23
7.2 Emoji profile	23
7.3 Default ignorable exclusion profile	25
<b>#39. Proposed changes to Unicode Technical Standard #39</b>	<b>26</b>
3.1.1 Joining Controls	26
3.1.1.1 Limited Contexts for Joining Controls	27
3.1.1.2 Limitations	27
4 Confusable Detection	27
<b>#51. Proposed changes to Unicode Technical Standard #51</b>	<b>31</b>
4 Presentation Style	31
<b>#55. Proposed Unicode Technical Standard #55</b>	<b>33</b>
1. Introduction	33

---

1.1 Source code spoofing	34
1.1.1 Line break spoofing	34
1.1.2 Spoofing using lookalike glyphs	35
1.1.3 Spoofing using bidirectional reordering	36
1.2 Usability issues	36
1.2.1 Usability issues arising from lookalike glyphs	37
1.2.2 Usability issues arising from bidirectional reordering	37
1.3 Conformance	38
2. Computer Language Specifications	38
2.1 Identifiers	38
2.1.1 Normalization and Case	39
2.1.2 Semantics Based on Case	40
2.2 Whitespace and Syntax	41
2.3 Language Evolution	43
2.2.1 Changing Identifier Definitions	43
2.2.2 Changing Normalization and Case	44
3. Source code display	45
3.1 Bidirectional Ordering	45
3.1.1 Atoms	45
3.1.2 Basic ordering	47
3.1.2.1 Equivalent Isolate Insertion for the Basic Ordering	49
3.1.3 Embedded languages	50
3.1.4 Ordering for Literal Text with Interspersed Syntax	50
3.1.4.1 Equivalent Isolate Insertion for the Ordering for Literal Text with Interspersed Syntax	53
3.2 Blank and Invisible Characters	53
3.2.1 Suggested representations for joiner controls and variation selectors	54
3.2.2 Suggested representations for directional formatting characters	55
3.3 Confusables	57
3.4 Syntax Highlighting	57
4. Tooling and diagnostics	58
4.1 Confusability Mitigation Diagnostics	58
4.1.1 Confusable Detection	58
4.1.2 Mixed-Script Detection	60
4.1.2.1 Identifier chunks	60
4.1.2.2 Mixed-script detection in identifier chunks	61
4.1.3 General Security Profile	62
4.1.4 Multiple visual forms	62
4.1.5 Extent of block comments	63
4.1.6 Directional formatting characters	64

4.2 Conversion to Plain Text	64
4.2.1 Unpaired brackets	66
4.3 Identifier Styles	67
5. Reference Implementations	68



## Proposed changes to existing technical reports

In the following sections:

Text from the reports is indented; proposed text additions are shown with yellow background, and text removals are struck through on a yellow background.

## #9. Proposed changes to Unicode Standard Annex #9

### 4 [Bidirectional Conformance](#)

*Editor's note: 4 unchanged paragraphs omitted.*

**UAX9-C2.** *The only permissible higher-level protocols are those listed in Section 4.3, [Higher-Level Protocols](#). They are [HL1](#), [HL2](#), [HL3](#), [HL4](#), [HL5](#), and [HL6](#).*

**#9/4** In Section 4, Bidirectional Conformance, amend the note under clause UAX9-C2 to refer to the new UTS, as follows.

**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. Higher-level protocols are recommended wherever the semantics of segment order are more significant than those of displayed order, as is the case for source text. For detailed examples for which use of HL4 would be recommended, see Section 4.3.1, [HL4 Example1 for XML](#) and Section 4.3.2, [HL4 Example2 for Program Text](#). For more information, see [Section 3.1, Bidirectional Ordering, in Unicode Technical Standard #55, "Unicode Source Code Handling" \[UTS55\]](#), as well as [Unicode Technical Report #36, "Unicode Security Considerations" \[UTR36\]](#).

**#9/6.5** Amend Section 6.5, Conversion to Plain Text to refer to the new UTS, as follows.

### 6.5 [Conversion to Plain Text](#)

For consistent appearance, when bidirectional text subject to a higher-level protocol is to be converted to Unicode plain text, formatting characters should be inserted to ensure that the display order resulting from the application of the Unicode Bidirectional Algorithm matches that specified by the higher-level protocol. The same principle should be followed whenever text using a higher-level protocol is converted to marked-up text that is unaware of the higher-level protocol. For example, if a higher-level protocol sets the paragraph direction to 1 (R) based on the number of L versus R/AL characters, when converted to plain text the paragraph would be embedded in a bracketing pair of RLE..PDF formatting characters. If the same text were converted to HTML4.0 the attribute `dir = "rtl"` would be added to the paragraph element.

For program text, whose proper display is subject to higher-level protocols, such a conversion to plain text needs to be performed in a way that does not change the semantics of the program. It is recommended that computer languages allow for the insertion of some formatting characters in appropriate locations without changing the meaning of a program; for computer languages that allow this insertion, a procedure is specified for conversion to plain text. See [Section 4.1, \*Whitespace\*](#), in Unicode Standard Annex #31, [Identifiers and Syntax](#), and [Section 4.2, \*Conversion to Plain Text\*](#), in Unicode Technical Standard #55, [Unicode Source Code Handling](#).

## #14. Proposed changes to Unicode Standard Annex #14

**#31/5.1** In Section 5.1, Description of Line Breaking Properties, at the end of the description of class BK, add a note describing the security implications of line breaking behavior when displaying source code, and recommending that source code editors support all of class BK, as well as class NL.

### BK: Mandatory Break (A) (Non-tailable)

Explicit breaks act independently of the surrounding characters. No characters can be added to the BK class as part of tailoring, but implementations are not required to support the VT character.

000B	LINE TABULATION (VT)
000C	FORM FEED (FF)
000B	LINE TABULATION (VT)

FORM FEED separates pages. The text on the new page starts at the beginning of the line. In some layout modes there may be no visible advance to a new “page”.

2028	LINE SEPARATOR
------	----------------

The text after the LINE SEPARATOR starts at the beginning of the line. This is similar to HTML <BR>.

2029	PARAGRAPH SEPARATOR
------	---------------------

The text of the new paragraph starts at the beginning of the line. This character defines a paragraph break, causing suitable formatting to be applied, for example, inter-paragraph spacing or first line indentation. LINE SEPARATOR, FF, VT as well as CR, LF and NL do not define a paragraph break.

**Note:** When displaying source code, failing to support all forms of the new line function can have security implications; for instance, executable code can appear commented out. It is therefore strongly recommended that source code editors support the VT character within the BK class, and support the NEL character within the NL class. See *Unicode Technical Standard #55, Unicode Source Code Handling* [UTS55].



## #31. Proposed changes to Unicode Standard Annex #31

#31/(A) Rename Unicode Standard Annex #31:

### UNICODE IDENTIFIERS AND PATTERN SYNTAX

**Rationale:** “identifiers and patterns” is an odd combination, and the scope of the annex is broader anyway. As clarified in [L2/22-072R](#), that scope includes restricting the set of characters with syntactic use in any programming language. The UAX also includes hashtags; while it terms these “hashtag identifiers”, “identifiers” is not where most readers would look when trying to find a definition of hashtags.

#31/1 Add a paragraph to the introduction clarifying that the scope extends to lexical analysis of computer languages writ large, not just identifiers.

### 1 Introduction

A common task facing an implementer of the Unicode Standard is the provision of a parsing and/or lexing engine for identifiers, such as programming language variables or domain names. There are also realms where identifiers need to be defined with an extended set of characters to align better with what end users expect, such as in hashtags.

To assist in the standard treatment of identifiers in Unicode character-based parsers and lexical analyzers, a set of specifications is provided here as a basis for parsing identifiers that contain Unicode characters. These specifications include:

- Default Identifiers: a recommended default for the definition of identifiers.
- Immutable Identifiers: for environments that need a definition of identifiers that does not change across versions of Unicode.
- Hashtag Identifiers: for identifiers that need a broader set of characters, principally for hashtags.

These guidelines follow the typical pattern of identifier syntax rules in common programming languages, by defining an ID\_Start class and an ID\_Continue class and using a simple BNF rule for identifiers based on those classes; however, the composition of those classes is more complex and contains additional types of characters, due to the universal scope of the Unicode Standard.

This annex also provides guidelines for the use of normalization and case insensitivity with identifiers, expanding on a section that was originally in Unicode Standard Annex #15, “Unicode Normalization Forms” [UAX15].

Lexical analysis of computer languages is also concerned with lexical elements other than identifiers, and with white space and line breaks that separate them. This annex provides guidelines for the sets of characters that have such lexical significance outside of identifiers.

The specification in this annex provides a definition of identifiers that is guaranteed to be backward compatible with each successive release of Unicode, but also allows any appropriate new Unicode characters to become available in identifiers. In addition, Unicode character properties for stable pattern syntax are provided. The resulting pattern syntax is backward compatible *and* forward

compatible over future versions of the Unicode Standard. These properties can either be used alone or in conjunction with the identifier characters.

**#31/1.4** Update the conformance section as follows.

#### 1.4 Conformance

The following describes the possible ways that an implementation can claim conformance to this specification.

**UAX31-C1.** *An implementation claiming conformance to this specification shall identify the version of this specification.*

**UAX31-C2.** *An implementation claiming conformance to this specification shall describe which of the following requirements it observes:*

- R1. Default Identifiers
- ~~R1a. Restricted Format Characters~~
- R1b. Stable Identifiers
- R2. Immutable Identifiers
- R3. Pattern\_White\_Space and Pattern\_Syntax Characters
- R3a. Pattern\_White\_Space Characters
- R3b. Pattern\_Syntax Characters
- R3c. Operator Identifiers
- R4. Equivalent Normalized Identifiers
- R5. Equivalent Case-Insensitive Identifiers
- R6. Filtered Normalized Identifiers
- R7. Filtered Case-Insensitive Identifiers
- R8. Hashtag Identifiers

**Note:** Meeting requirement R3 is equivalent to meeting requirements R3a and R3b.

#### 2 Default Identifiers

*Editor's note: 11 unchanged paragraphs and Table 2 omitted.*

**#31/2(A)** Explicitly allow for character sequences in the sets used by default identifiers. Refer to standard profiles.

**UAX31-R1. Default Identifiers:** *To meet this requirement, to determine whether a string is an identifier an implementation shall choose either UAX31-R1-1 or UAX31-R1-2.*

**UAX31-R1-1.** *Use definition UAX31-D1, setting Start and Continue to the properties XID\_Start and XID\_Continue, respectively, and leaving Medial empty.*

**UAX31-R1-2.** *Declare that it uses a **profile** of UAX31-R1-1 and define that profile with a precise specification of the characters **and character sequences** that are added to or removed from Start, Continue, and Medial and/or provide a list of additional constraints on identifiers.*

**Note:** Such a specification may incorporate a reference to one or more of the standard profiles described in *Section 7, Standard Profiles*.

One such profile may be to use the contents of ID\_Start and ID\_Continue in place of XID\_Start and XID\_Continue, for backward compatibility.

Another such profile would be to include some set of the optional characters, for example:

- *Start := XID\_Start, plus some characters from Table 3*
- *Continue := Start + XID\_Continue, plus some characters from Table 3b*
- *Medial := some characters from Table 3a*

**Note:** Characters in the Medial class must not overlap with those in either the Start or Continue classes. Thus, any characters added to the Medial class from *Table 3a* must be checked to ensure they do not also occur in either the newly defined Start class or Continue class.

Beyond such minor modifications, profiles could also be used to significantly extend the character set available in identifiers. In so doing, care must be taken not to unintentionally include undesired characters, or to violate important invariants.

An implementation should be careful when adding a property-based set to a profile.

For example, consider a profile that adds subscript and superscript digits and operators in order to support technical notations (for example, identifiers such as the Assyriological  $\text{dun}_3^+$ , the chemical  $\text{Ca}^{2+}$  concentration, the mathematical  $x_{k+1}$  or  $f^{(4)}$ , or the phonetic  $\text{daan}^6$ ). That profile may be described as adding the following set to XID\_Continue:

$$[(\text{)}^+ = - \text{ } 0 \text{ } 1 \text{ } 2 \text{ } 3 \text{ } 4 \text{ } 5 \text{ } 6 \text{ } 7 \text{ } 8 \text{ } 9 \text{ } ] .$$

**Note:** The above list is for illustration only. A standard profile is provided to support the use of mathematical notation in identifiers. See *Section 7.1, Mathematical notation profile*.

If instead of listing these characters explicitly, the profile had chosen to use properties or combinations of properties, that might result in including undesired characters.

For example,  $\backslash\text{p}\{\text{General\_Category}=\text{Other\_Number}\}$  is the general category set containing the subscript and superscript digits. But it also includes the compatibility characters  $[(1)1,1.]$ , which are not needed for technical notations, and are very likely inappropriate for identifiers—on multiple counts.

On the other hand, a language that allows currency symbols in identifiers could have  $\backslash\text{p}\{\text{General\_Category}=\text{Currency\_Symbol}\}$  as a profile, since that property matches the intent.

Similarly, a profile based on adding entire blocks is likely to include unintended characters, or to miss ones that are desired. For the use of blocks see *Annex A, Character Blocks*, in [UTS18].

Defining a profile by use of a property also needs to take account of the fact that unless the property is designed to be stable (such as XID\_Continue), code points could be removed in a future version

of Unicode. If the profile also needs stable identifiers (backwards compatible), then it must take additional measures. See *UAX31-R1b Stable Identifiers*.

**#31/2(B)** Qualify the need for closure under normalization.

Implementations that require identifier closure under normalization should ensure that any custom profile preserves identifier closure under the chosen normalization form. See *Section 5.1.3, Identifier Closure Under Normalization*. ~~When defining a profile, it is also critical to ensure that it is compatible with the normalization form chosen for the identifiers.~~ The example cited above regarding subscripts and superscripts preserves identifier closure under Normalization Forms C and D, but *not* under Forms KC and KD. Under NFKC and NFKD, the subscript and superscript parentheses and operators normalize to their ASCII counterparts. ~~If an implementation that uses this profile relies on identifier closure under normalization, it~~ ~~A language using that profile~~ should conform to UAX31-R4 using NFC, not NFKC.

**#31/2(C)** Remove requirement UAX31-R1a. Restricted Format Characters, and the reference to it in the discussion of profiles of UAX31-R1. Add a note about spoofing.

**Note:** While default identifiers are less open-ended than immutable identifiers, they are still subject to spoofing issues arising from invisible characters, visually identical characters, or bidirectional reordering causing distinct sequences to appear in the same order. Where spoofing concerns are relevant, the mechanisms described in Unicode Technical Standard #39, *Unicode Security Mechanisms* [UTS39], should be used. For the specific case of programming languages, recommendations are provided in Unicode Technical Standard #55, *Unicode Source Code Handling* [UTS55].

Implementations defining a profile that includes the ZERO WIDTH JOINER or ZERO WIDTH NON JOINER characters should implement the requirement **UAX31-R1a**:

**UAX31-R1a. Restricted Format Characters:** *This clause has been removed. This requirement remains as part of the more comprehensive General Security Profile defined in Unicode Technical Specification #39, Unicode Security Mechanisms. To meet this requirement, an implementation shall choose either UAX31-R1a-1 or UAX31-R1a-2.*

**UAX31-R1a-1.** *Define a profile for UAX31-R1 which allows format characters, but restricts their use to the contexts **A1**, **A2**, and **B** defined in Section 2.3.1, Limited Contexts for Joining Controls Layout and Format Control Characters.*

**UAX31-R1a-2.** *Define a profile for UAX31-R1 which allows format characters, but imposes further restrictions on the context for ZWJ or ZWNJ in addition to those required by UAX31-R1a-1, such as by limiting the scripts allowed or limiting the occurrence of ZWJ or ZWNJ to specific character combinations, supplying a clear specification for such further restrictions.*

**Note:** The ZWJ and ZWNJ characters in **UAX31-R1a** are not in `XID_Continue`; as a result, meeting the requirement **UAX31-R1 Default Identifiers** does not require supporting **UAX31-R1a Restricted Format Characters**.

The ZWJ and ZWNJ characters are invisible in most contexts, and are only added to Default Identifiers in a declared profile. They have security and usability

implications that make them inappropriate for implementations that do not carefully consider those implications. For example, they should not be added via a profile where spoofing concerns are paramount, such as top-level domain names.

*Review note: Requirement R1a was unusually complex compared to the other requirements and rules usually expressed in lexical analysis. At the same time, it did not meaningfully address security concerns unless it was paired with other mechanisms, such as the exclusion of default ignorable code points (default ignorables include the variation selectors, which are part of default identifiers). The recommendation is to make use of the mechanisms defined in UTS #39, which include the restrictions from UAX31-R1a.*

**#31/2.3(A)** Adapt the discussion of joining controls to reflect the changes to properties and the removal of requirement R1a. This change resolves action item [165-A42](#).

### 2.3 Layout and Format Control Characters

Certain Unicode characters are known as Default\_Ignorable\_Code\_Points. These include variation selectors and characters used to control joining behavior, bidirectional ordering control, and alternative formats for display (having the General\_Category value of Cf). ~~The recommendation is to permit them in identifiers only in special cases, listed below.~~ The use of default-ignorable characters in identifiers is problematic, first because the effects they represent are stylistic or otherwise out of scope for identifiers, and second because the characters themselves often have no visible display. It is also possible to misapply these characters such that users can create strings that look the same but actually contain different characters, which can create security problems. In such environments where spoofing concerns are paramount, such as top-level domain names, identifiers should also be limited to characters that are case-folded and normalized with the NFKC\_Casefold operation. For more information, see *Section 5, Normalization and Case* and *UTR #36: Unicode Security Considerations* [UTR36].

While not all Default\_Ignorable\_Code\_Points are in `XID_Continue`, the variation selectors and joining controls are included in `XID_Continue`. These variation selectors are used in standardized variation sequences, sequences from the Ideographic Variation Database, and emoji variation sequences. The joining controls are used in the orthographies of some languages, as well as in emoji ZWJ sequences. However, these characters they are subject to the same considerations as for other Default\_Ignorable\_Code\_Points listed above. Because variation selectors and joining controls request a difference in display but do not guarantee it, they do not work well in general-purpose identifiers. A profile should be used to remove them from general-purpose identifiers (along with other Default\_Ignorable\_Code\_Points), unless their use is required in a particular domain, such as in a profile that includes emoji. For such a profile it may be useful to explicitly retain or even add certain Default\_Ignorable\_Code\_Points in the identifier syntax.

For programming language identifiers, spoofing issues are more comprehensively addressed by higher-level diagnostics rather than at the syntactic level. See *Unicode Technical Standard #55, Source Code Handling* [UTS55].

**Comparison.** In any environment where the display form for identifiers differs from the form used to compare them, Default\_Ignorable\_Code\_Points should be ignored for comparison. For example, this applies to case-insensitive identifiers, and in particular, for any For more information, see *Section 1.3, Display Format*.

*Review note: The last clause of the penultimate sentence of the paragraph above was turned into the note below.*

**Note:** An implementation of UAX31-R4 and UAX31-R5 (Equivalent Case and Compatibility-Insensitive Identifiers) that uses the NFKC\_Casefold operation, which for comparison ignores Default\_Ignorable\_Code\_Points.

The General Security Profile defined in Section 3.1, General Security Profile for Identifiers, in *UTS #39, Unicode Security Mechanisms* [UTS39], excludes all Default\_Ignorable\_Code\_Points by default, including variation selectors.

In addition, a standard profile is provided to exclude all Default\_Ignorable\_Code\_Points; see *Section 7, Standard Profiles*. Note however that, even if Default\_Ignorable\_Code\_Points are excluded, spoofing issues remain unless the mechanisms in UTS #39 are utilized.

**#31/2.3(B)** Remove the transition to Section 2.3.1, as well as Section 2.3.1 and Section 2.3.2, which are moved to UTS #39.

For the above reasons, default-ignorable characters are normally excluded from Unicode identifiers. However, visible distinctions created by certain format characters (particularly the *Join\_Control* characters) are necessary in certain languages. A blanket exclusion of these characters makes it impossible to create identifiers with the correct visual appearance for common words or phrases in those languages:

Identifier systems that attempt to provide more natural representations of terms in "modern, customary usage" should allow these characters in input and display, but limit them to contexts in which they are necessary. The term *modern customary usage* includes characters that are in common use in newspapers, journals, lay publications, on street signs, in commercial signage, and as part of common geographic names and company names, and so on. It does not include technical or academic usage such as in mathematical expressions, using archaic scripts or words, or pedagogical use (such as illustration of half-forms or joining forms in isolation), or liturgical use:

The goals for such a restriction of format characters to particular contexts are to:

- Allow the use of these characters where required in normal text
- Exclude as many cases as possible where no visible distinction results
- Be simple enough to be easily implemented with standard mechanisms such as regular expressions

### 2.3.1 Limited Contexts for Joining Controls

*Editor's note: Remove the entirety of Section 2.3.1, omitted here for the sake of brevity.*

### 2.3.2 Limitations

While the restrictions in **A1**, **A2**, and **B** greatly limit visual confusability, they do not prevent it. For example, because Tamil only uses a *Join\_Control* character in one specific case, most of the sequences these rules allow in Tamil are, in fact, visually confusable. Therefore based on their knowledge of the script concerned, implementations may choose to have tighter restrictions than

specified in *Section 2.3.1, Limited Contexts for Joining Controls*. There are also cases where a joiner preceding a virama makes a visual distinction in some scripts. It is currently unclear whether this distinction is important enough in identifiers to warrant retention of a joiner. For more information, see UFR #36, *Unicode Security Considerations* [UTR36].

**Performance.** Parsing identifiers can be a performance-sensitive task. However, these characters are quite rare in practice, thus the regular expressions (or equivalent processing) only rarely would need to be invoked. Thus these tests should not add any significant performance cost overall.

**Comparison.** Typically the identifiers with and without these characters should compare as equivalent, to prevent security issues. See *Section 2.4, Specific Character Adjustments*.

**#31/2.4** Remove references to the Join\_Control characters from Section 2.4.

**Rationale:** These characters are now part of default identifiers.

**#31/3(A)** Change the definition of “profile” in UAX31-R2 to allow for arbitrary customization, as in UAX31-R1.

### 3 Immutable Identifiers

*Editor’s note: six unchanged paragraphs omitted.*

**UAX31-R2. Immutable Identifiers:** *To meet this requirement, an implementation shall choose either UAX31-R2-1 or UAX31-R2-2.*

**UAX31-R2-1.** *Define identifiers to be any non-empty string of characters that contains no character having any of the following property values:*

- Pattern\_White\_Space=True
- Pattern\_Syntax=True
- General\_Category=Private\_Use, Surrogate, or Control
- Noncharacter\_Code\_Point=True

**UAX31-R2-2.** *Declare that it uses a **profile** of UAX31-R2-1 and define that profile with a precise specification of the characters and character sequences that are added to or removed from the sets of code points defined by these properties and/or provide a list of additional constraints on identifiers.*

**#31/3(B)** Add a note to UAX31-R2, recommending care when migrating from immutable identifiers to default identifiers.

**Note:** The expectation from an implementation meeting requirement UAX31-R2 Immutable Identifiers is that it will never change its definition of identifiers; in particular, that it will not switch to UAX31-R1 Default Identifiers. However, the downsides of normalization issues and the inapplicability of measures guarding against spoofing attacks may warrant such a change in definition. In such circumstances, a profile should be used to extend XID\_Start and XID\_Continue to cover likely existing usages. See *Section 2.3, Language Evolution*, in *Unicode Technical Standard #55, Unicode Source Code Handling* [UTS55].



#31/4(A) Rename section 4:

#### 4 ~~Pattern~~ Whitespace and Syntax

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. Examples include regular expressions, Java collation rules, Excel or ICU number formats, and many others. In the past, regular expressions and other formal languages have been forced to use clumsy combinations of ASCII characters for their syntax. As Unicode becomes ubiquitous, some of these will start to use non-ASCII characters for their syntax: first as more readable optional alternatives, then eventually as the standard syntax.

#31/4(B) Change its second paragraph as follows:

For forward and backward compatibility, it is advantageous to have a fixed set of whitespace and syntax code points ~~for use in patterns~~. This follows the recommendations that the Unicode Consortium has made regarding completely stable identifiers, and the practice that is seen in XML 1.0, 5th Edition or later [XML]. (In particular, the Unicode Consortium is committed to not allocating characters suitable for identifiers in the range U+2190..U+2BFF, which is being used by XML 1.0, 5th Edition.)

#31/4(C) Move the following two paragraphs from further down in the section to after the second paragraph, and amend the resulting fifth paragraph, as follows:

*Review note: the following two paragraphs have been moved up from what is now section 4.3.*

As of Unicode 4.1, two Unicode character properties are defined to provide for stable syntax: `Pattern_White_Space` and `Pattern_Syntax`. Particular ~~pattern~~ languages may, of course, override these recommendations, for example, by adding or removing other characters for compatibility with ASCII usage.

For stability, the values of these properties are absolutely invariant, not changing with successive versions of Unicode. Of course, this does not limit the ability of the Unicode Standard to encode more symbol or whitespace characters, but the syntax and whitespace code points recommended for use in ~~formal languages~~ ~~patterns~~ will not change.

#31/4(D) Split UAX31-R3 into a whitespace and a syntax part, each treated in its own section, 4.1 and 4.2 respectively.

**UAX31-R3. *Pattern\_White\_Space and Pattern\_Syntax Characters:*** *To meet this requirement, an implementation shall meet both UAX31-R3a and UAX31-R3b.*

**Note:** When meeting ~~this~~ requirement `UAX31-R3 with no profile`, all characters except those that have the `Pattern_White_Space` or `Pattern_Syntax` properties are available for use ~~as in the definition of~~ identifiers or literals.



*Review note: the baseline shown for each of the UAX31-R3a and UAX31-R3b requirements is the relevant part of the former UAX31-R3; as there is some overlap, some text that was replicated is marked as unmodified.*

#### 4.1 Whitespace

**#31/4.1(A)** Clarify the meaning of “interpreted as whitespace”, as follows.

Many formal languages treat two categories of whitespace differently: horizontal space (such as the ASCII horizontal tabulation and space), and ends of line.

When a syntax supports non-ASCII characters, it is useful to consider a third category: ignorable format controls. Ignorable format controls may be inserted between lexical elements in order to resolve bidirectional ordering issues, as described in *Section 4.1.1, Bidirectional Ordering*. The insertion of these characters does not change the meaning of the program; in particular, they are not spacing characters. See *Section 4.1.2, Required Spaces*.

**Note:** Allowing for the insertion of ignorable format controls does not prevent spoofing based on bidirectional reordering. In order to guard against such spoofing, implementations should make use of the higher-level protocols and conversion to plain text described in Unicode Standard Annex #9 *Unicode Bidirectional Algorithm* [UAX9]. See Unicode Technical Standard #55, *Unicode Source Code Handling* [UTS55].

**Note:** Since these characters are allowed only where a boundary would, in their absence, exist between lexical elements, an implementation could ignore them when lexing, and then consider as illegal any lexical element that contains them. An exception must be made for comments and strings, which should be able to freely contain these characters.

Implementations should also allow these characters in other contexts where reordering issues could arise. See *Unicode Technical Standard #55, Unicode Source Code Handling* [UTS55].

**UAX31-R3a. Pattern\_White\_Space Characters:** *To meet this requirement, an implementation shall choose either UAX31-R3a-1 or UAX31-R3a-2.*

**UAX31-R3a-1.** *Use Pattern\_White\_Space characters as ~~all and only those~~ the set of characters interpreted as whitespace in parsing, as follows:*

1. *A sequence of one or more of any of the following characters shall be interpreted as a sequence of one or more end of line:*
  - a. *U+000A (line feed);*
  - b. *U+000B (vertical tabulation);*
  - c. *U+000C (form feed);*
  - d. *U+000D (carriage return);*
  - e. *U+0085 (next line);*
  - f. *U+2028 LINE SEPARATOR;*
  - g. *U+2029 PARAGRAPH SEPARATOR.*
2. *The Pattern\_White\_Space characters with the property Default\_Ignorable\_Code\_Point shall be treated as ignorable format controls; they shall be allowed in the contexts **I1**, **I2**, and **I3** defined in Section 4.1.3, Contexts for Ignorable Format Controls, where their insertion shall have no effect on the meaning of the program.*

3. All other characters in `Pattern_White_Space` shall be interpreted as horizontal space.

**UAX31-R3a-2.** Declare that it uses a **profile** of UAX31-R3a-1 and define that profile with a precise specification of the characters that are added to or removed from the sets of code points defined by these properties the `Pattern_White_Space` property, and of any changes to the criteria under which a character or sequence of characters is interpreted as an end of line, as ignorable format controls, or as horizontal space.

**Note:** The characters to be treated as ignorable format controls under item 2 of UAX31-R3a-1 are U+200E LEFT-TO-RIGHT MARK and U+200F RIGHT-TO-LEFT MARK. The characters to be treated as horizontal space under item 4 of UAX31-R3a-1 are U+0020 SPACE and U+0009 (horizontal tabulation).

**Note:** The characters LEFT-TO-RIGHT MARK and RIGHT-TO-LEFT MARK are two of the Implicit Directional Marks defined by Section 2.6, *Implicit Directional Marks*, in *Unicode Standard Annex #9, Unicode Bidirectional Algorithm* [UAX9]. The third one, ARABIC LETTER MARK, is used far less frequently than the others, even in Arabic text; its behavior differs subtly from RIGHT-TO-LEFT MARK in ways that are not usually relevant to the ordering of source code. If it is added to the set of whitespace characters by a profile, it is interpreted as an ignorable format control.

**Note:** Failing to interpret all characters listed in item 1 of UAX31-R3a-1 as line terminators would lead to spoofing issues; see Unicode Technical Standard #55, *Unicode Source Code Handling* [UTS55].

#31/4.1(B) Promote the long note about bidirectional ordering into two subsections of 4.1.

#### 4.1.1 Bidirectional Ordering

**Note:** This requirement UAX31-R3a 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.

*Review note: one paragraph moved to 4.1.2.*

**Example:** 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  $(x + \text{ת})$  and 1. The misleading rendering of (2) occurs because the directionality of the identifier ת influences subsequent weakly-directional tokens; inserting a left-to-right mark after the identifier ת 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 + ת == 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. See Section 4.2, *Conversion to Plain Text*, in Unicode Technical Standard #55, *Unicode Source Code Handling*.

**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.

#### 4.1.2 Required Spaces

*Review note: This paragraph was moved from 4.1.1.*

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.

#### 4.1.3 Contexts for Ignorable Format Controls

Implementations should at least allow for the insertion of ignorable format controls in the following contexts, illustrated by examples wherein the ignorable format control is represented by <LRM>.

**II.** Adjacent to lexical horizontal space.

**Example:** Between the following keywords separated by a space:

```
else <LRM>if
```

**Note:** The phrase “lexical horizontal space” refers to characters that are not merely in the set of horizontal space characters, but are also in a context where they are lexically spaces. For instance, it does not include horizontal space characters in string literals. Implementations should permit these characters in string literals, but in such a literal, their insertion has an effect on the meaning of the program, as they are then present in the string represented by that literal.

**I2.** As optional space, that is, wherever horizontal space could be inserted without changing the meaning of the program.

**Example:** Before the plus sign in the following arithmetic expression:

```
x⟨LRM⟩+1
```

**I3.** At the start and end of a lexical line.

**Example:** Before the word `import` in the following line of Python:

```
⟨LRM⟩import unicodedata
```

**Note:** As is the case for I1, the start and end of a “lexical line” in I3 does not include the start and end of a line in a multiline string literal, respectively. This context is distinct from I2 in languages where leading or trailing spaces are meaningful.

**#31/4.2(A)** Clarify the meaning of “syntactic use”, as follows.

#### 4.2 Syntax

The lexical structure of formal languages involves characters that are not allowed in identifiers and are not whitespace, but that have some special lexical significance other than being literal characters (such as in string literals) or ignored (such as in comments). These are referred to in this document as *characters with syntactic use*.

Examples of characters with syntactic use include:

- decimal marks in numeric literals;
- arithmetic operators, such as +, -, \*, /;
- parentheses and other brackets;
- characters in comment delimiters, such as #, /\*, --, or  $\textcircled{!}$ ;
- quotation marks delimiting strings;
- characters such as `\` introducing escape sequences.

It is useful to bound the set of characters with syntactic use. In particular, this allows for backward compatibility of literals (including patterns), as described in *Section 4.3, Pattern Syntax*. It also provides a stable set of characters that can be used for user-defined operators.

**UAX31-R3b.** **Pattern\_Syntax Characters:** To meet this requirement, an implementation shall choose either UAX31-R3b-1 or UAX31-R3a-2.

**UAX31-R3b-1.** ~~and use~~ Use *Pattern\_Syntax* characters as **all and only those the set of** characters with syntactic use. The following sets shall be disjoint:

1. characters allowed in identifiers;
2. characters treated as whitespace;
3. characters with syntactic use;

**UAX31-R3b-2.** Declare that it uses a **profile** of UAX31-R3-1 and define that profile with a precise specification of the characters that are added to or removed from the sets of code points defined by the *Pattern\_Syntax* property ~~these properties~~.

**Note:** When meeting requirement UAX31-R3b, characters allowed in identifiers may be given special significance in the syntax even when they are not part of identifiers.

For instance, in a language which uses the C syntax for hexadecimal literals and meets requirement UAX31-R1, the literal `0xDEADBEEF` consists entirely of identifier characters, yet the `0x` has special significance in the syntax, and the characters after that prefix are subject to special restrictions (only 0 through 9 and A through F are allowed).

However, characters outside of those allowed in identifiers, those treated as whitespace, and the set `[:Pattern_Syntax:]` cannot be given special significance in the syntax. For instance, if a language meets requirements UAX31-R1 and UAX31-R3 with no profile and allows for user-defined operators, that language cannot allow the user to define an operator 🦄.

Characters outside of those allowed in identifiers, those treated as whitespace, and the set `[:Pattern_Syntax:]` can still be allowed in a program, for instance, as part of string literals or comments.

#### 4.2.1 User-Defined Operators

Some programming languages allow for user-defined operators. When meeting requirement UAX31-R3b, the set of characters that can be allowed in operators is limited; however, that leaves open the exact definition of operators. In order to avoid ambiguities in lexical analysis, operators should not be allowed to contain characters that may be found at the beginning of an identifier or literal; for instance, `+1` or `-x` should not be operators.

The following definition avoids such interactions with default identifiers and with numbers.

**UAX31-R3c. Operator Identifiers:** *To meet this requirement, an implementation shall meet requirement UAX31-R3b Pattern\_Syntax Characters, and, to determine whether a string is an operator, it shall choose either UAX31-R3c-1 or UAX31-R3c-2.*

**UAX31-R3c-1.** *Use definition UAX31-D1, setting Start to be the set of characters with syntactic use, setting Continue to be the union of the set of characters with syntactic use and the set of characters with General\_Category Mn, and leaving Medial empty.*

**UAX31-R3c-2.** *Declare that it uses a profile of UAX31-R3c-1 and define that profile with a precise specification of the characters and character sequences that are added to or removed from Start, Continue, and Medial and/or provide a list of additional constraints on operators.*

**Note:** The set of Pattern\_Syntax characters, which is the default for characters with syntactic use, contains some emoji. Implementations may wish to remove them, either to allow for their use in identifiers, or to reduce potential confusion arising from 🏠 being an operator but 🍷 not being one. This may be done using the standard profile for UAX31-R3b Pattern\_Syntax Characters defined in Section 7.2, Emoji Profile.

Nonspacing marks are included in Continue because they are part of the representation for many operators, such as some of the negated operators.

When meeting this requirement, a profile is likely to be needed depending on the specifics of the syntax. For instance, a programming language wherein string literals start with `"` should remove that character from the characters allowed in operators.

#31/4.3 Move the discussion specific to pattern languages to a new subsection titled accordingly.

### 4.3 Pattern syntax

With a fixed set of whitespace and syntax code points, a pattern language can **then** have a policy requiring all possible syntax characters (even ones currently unused) to be quoted if they are literals. Using this policy preserves the freedom to extend the syntax in the future by using those characters. Past patterns on future systems will always work; future patterns on past systems will signal an error instead of silently producing the wrong results. Consider the following scenario, for example.

In version 1.0 of program X, '`≈`' is a reserved syntax character; that is, it does not perform an operation, and it needs to be quoted. In this example, '`\`' *quotes* the next character; that is, it causes it to be treated as a literal instead of a syntax character. In version 2.0 of program X, '`≈`' is given a real meaning—for example, “uppercase the subsequent characters”.

- The pattern `abc...\≈...xyz` works on both versions 1.0 and 2.0, and refers to the literal character because it is quoted in both cases.
- The pattern `abc...≈...xyz` works on version 2.0 and uppercases the following characters. On version 1.0, the engine (rightfully) has no idea what to do with `≈`. Rather than silently fail (by ignoring `≈` or turning it into a literal), it has the opportunity to signal an error.

*Review note: The two paragraphs starting with “As of Unicode 4.1...” and “For stability...” have been moved from here.*

When *generating* rules or patterns, all whitespace and syntax code points that are to be literals require quoting, using whatever quoting mechanism is available. For readability, it is recommended practice to quote or escape all literal whitespace and default ignorable code points as well.

Consider the following example, where the items in angle brackets indicate literal characters:

$$a\langle\text{SPACE}\rangle b \rightarrow x\langle\text{ZERO WIDTH SPACE}\rangle y + z;$$

Because `<SPACE>` is a `Pattern_White_Space` character, it requires quoting. Because `<ZERO WIDTH SPACE>` is a default ignorable character, it should also be quoted for readability. So in this example, if `\uXXXX` is used for a code point literal, but is resolved before quoting, and if single quotes are used for quoting, this example might be expressed as:

$$'a\u0020b' \rightarrow 'x\u200By' + z;$$

#31/7 Add a Section 7.

## 7 Standard Profiles

Two standard profiles for default identifiers are provided to cater to common patterns of use observed in programming languages with less restrictive identifier syntaxes, including those that use



1. Removing the characters in the following set PSEP from the set of characters with syntactic use:

$PSEP := [[:Pattern_Syntax:]]\&[[:Emoji_Presentation:]]$

2. For all C in PSEP, adding the sequence consisting of C followed by variation selector-15 (the Text Presentation Selector) to the set of characters with syntactic use.

**Note:** These are the characters removed from the set of characters with syntactic use, separated by spaces:



These are the sequences added to that set, separated by spaces:



*Review note: some of these sequences are currently nonstandard. See P/(B) in this document which proposes standardizing them.*

This change means that if some of the Pattern\_Syntax characters with the Emoji\_Presentation property were in syntactic use (e.g., in operators) prior to adopting the emoji profile, they become identifiers once the profile is adopted, but can be turned back into operators by adding variation-selector-15, allowing for a migration path.

Of course, if a programming language only uses a subset of the Pattern\_Syntax characters that doesn't include these characters, no action needs to be taken.

Some other characters in Pattern\_Syntax (such as ↔) are used in emoji (such as 🐱↔️), but they are not emoji on their own, so that they do not need to be removed from the set of characters with syntactic use as long as lexical analysis properly takes sequences into account.

The emoji sequences require 98 default-ignorable characters:

- U+200D ZERO WIDTH JOINER (aka ZWJ)
- U+FE0F VARIATION SELECTOR-16 (aka emoji presentation selector = EPS)
- U+E0020..U+E007F 98 TAG characters

Thus if this profile is combined with any profile that removes default-ignorable characters, such as the default-ignorable exclusion profile, those characters need to be retained in the context of emoji sequences.

Consider the following examples:

Sequence	Appear.	Status	Reason
A+ZWJ+B	AB	Illegal	ZWJ is not part of an emoji sequence
U+1F408 + ZWJ + U+2B1B	🐱	Legal	ZWJ is part of an emoji sequence
BIG + U+1F408 + ZWJ + U+2B1B	BIG 🐱	Legal	(for <i>black cat</i> )



### 7.3 Default ignorable exclusion profile

The default ignorable exclusion profile for default identifiers consists in the exclusion of the code points with property `Default_Ignorable_Code_Point` from the sets `Start` and `Continue` in definition [UAX31-D1](#).

**Note:** While it reduces the attack surface, excluding default ignorable code points does not prevent spoofing issues. More comprehensive mechanisms are described in *Unicode Technical Standard #39, Unicode Security Mechanisms* [UTS39]; in particular, the exclusion of default ignorable code points is part of the General for Profile for Identifiers.

**Note:** Where higher level diagnostics are available, such as in programming environments, more targeted measures can be taken in order to still allow for the legitimate use of these characters. See *Unicode Technical Standard #55, Source Code Handling* [UTS55].

## #39. Proposed changes to Unicode Technical Standard #39

**#31/3.1.1** Move some text from Section 2.3 from UAX #31 into Section 3.1.1 of UTS #39, as follows.

### 3.1.1 Joining Controls

However, visible distinctions created by certain format characters excluded by the General Security Profile because their Identifier\_Type is Default\_Ignorable (particularly the *Join\_Control* characters) are necessary in certain languages. A blanket exclusion of these characters makes it impossible to create identifiers with the correct visual appearance for common words or phrases in those languages.

Identifier systems that attempt to provide more natural representations of terms in "modern, customary usage" should allow these characters in input and display, but limit them to contexts in which they are necessary. The term modern customary usage includes characters that are in common use in newspapers, journals, lay publications; on street signs; in commercial signage; and as part of common geographic names and company names, and so on. It does not include technical or academic usage such as in mathematical expressions, using archaic scripts or words, or pedagogical use (such as illustration of half-forms or joining forms in isolation), or liturgical use.

The goals for such a restriction of format characters to particular contexts are to:

- Allow the use of these characters where required in normal text
- Exclude as many cases as possible where no visible distinction results
- Be simple enough to be easily implemented with standard mechanisms such as regular expressions

*Review note: The above paragraphs have been moved from Section 2.3 of UAX #31.*

An implementation following the General Security Profile that allows the additional characters ZWJ and ZWNJ shall only permit them where they satisfy the conditions A1, A2, and B in *Section 3.1.1.12.3.1, Limited Contexts for Joiner Controls of [UAX31]*, unless it documents the additional contexts where it allows them.

More advanced implementations may use script-specific information for more detailed testing. In particular, they can:

1. *Disallow joining controls* in sequences that meet the conditions of A1, A2, and B, where in common fonts the resulting appearance of the sequence is normally not distinct from appearance in the same sequences with the joining controls removed.
2. *Allow joining controls* in sequences that don't meet the conditions of A1, A2, and B (such as the following), where in common fonts the resulting appearance of the sequence is normally distinct from the appearance in the same sequences with the joining controls removed.

```
/$L ZWNJ $V $L/
```

```
/$L ZWJ $V $L/
```

The notation is from [UAX31].

#31/3.1.1.1 Move Section 2.3.1 of UAX #31 as Section 3.1.1.1 of UTS #39.

### 3.1.1.1 Limited Contexts for Joining Controls

*Review note: This section has been moved from Section 2.3.1 of UAX #31.*

An implementation that attempts to provide more natural representations of terms in "modern, customary usage" should allow the following Join\_Control characters in the limited contexts specified in [A1](#), [A2](#), and [B](#) below.

U+200C ZERO WIDTH NON-JOINER (ZWNJ)

U+200D ZERO WIDTH JOINER (ZWJ)

*Editor's note: Remainder omitted for brevity.*

#31/3.1.1.2 Move Section 2.3.2 of UAX #31 as Section 3.1.1.2 of UTS #39, except for the paragraph titled *Comparison*.

### 3.1.1.2 Limitations

*Review note: This section has been moved from Section 2.3.2 of UAX #31.*

While the restrictions in **A1**, **A2**, and **B** greatly limit visual confusability, they do not prevent it. For example, because Tamil only uses a Join\_Control character in one specific case, most of the sequences these rules allow in Tamil are, in fact, visually confusable. Therefore based on their knowledge of the script concerned, implementations may choose to have tighter restrictions than specified in [Section 3.1.1.2.3.1, Limited Contexts for Joining Controls](#). There are also cases where a joiner preceding a virama makes a visual distinction in some scripts. It is currently unclear whether this distinction is important enough in identifiers to warrant retention of a joiner. For more information, see UTR #36: *Unicode Security Considerations* [UTR36].

**Performance.** Parsing identifiers can be a performance-sensitive task. However, these characters are quite rare in practice, thus the regular expressions (or equivalent processing) only rarely would need to be invoked. Thus these tests should not add any significant performance cost overall.

## 4 Confusable Detection

*Editor's note: 3 unchanged paragraphs omitted.*

#31/4(A) Change the definition of the *skeleton* operation to exclude default ignorables, as follows.

For an input string X, define `skeleton(X)` to be the following transformation on the string:

1. Convert X to NFD format, as described in [UAX15].
2. Remove any characters in X that have the property `Default_Ignorable_Code_Point`.
3. Concatenate the prototypes for each character in X according to the specified data, producing a string of exemplar characters.

#### 4. Reapply NFD.

The strings  $X$  and  $Y$  are defined to be **confusable** if and only if  $\text{skeleton}(X) = \text{skeleton}(Y)$ . This is abbreviated as  $X \cong Y$ .

This mechanism imposes transitivity on the data, so if  $X \cong Y$  and  $Y \cong Z$ , then  $X \cong Z$ . It is possible to provide a more sophisticated confusable detection, by providing a metric between given characters, indicating their "closeness." However, that is computationally much more expensive, and requires more sophisticated data, so at this point in time the simpler mechanism has been chosen. That means that in some cases the test may be overly inclusive.

**Note:** The strings  $\text{skeleton}(X)$  and  $\text{skeleton}(Y)$  are **not** intended for display, storage or transmission. They should be thought of as an intermediate processing form, similar to a hashcode. The exemplar characters are **not** guaranteed to be identifier characters.

#31/4(B) Define the *bidSkeleton* operation and the bidirectional confusability relation as follows.

For an input string  $X$  and a direction  $d \in \{\text{RTL}, \text{LTR}, \text{FS}\}$ , define  $\text{bidSkeleton}(d, X)$  to be the following transformation on the string:

1. Reorder the code points in  $X$  for display by applying the rules of the Unicode Bidirectional Algorithm [UAX9] up to and including L2, treating  $X$  as a single paragraph; if  $d \neq \text{FS}$ , apply protocol HL1 to set the paragraph level to 1 if  $d = \text{RTL}$ , and to 0 if  $d = \text{LTR}$ ; this yields the reordered sequence of characters  $R$ .
2. Apply rule L3 of the UBA: move combining marks after their base in  $Z$ ; this yields the sequence  $R'$ .
3. Replace any character whose glyph would be mirrored by rule L4 of the UBA by the value of its `Bidi_Mirroring_Glyph` property, yielding  $R''$ .
4.  $\text{bidSkeleton}(d, X)$  is then  $\text{skeleton}(R'')$ .

The strings  $X$  and  $Y$  are defined to be  $d$ -confusable if and only if  $\text{bidSkeleton}(d, X) = \text{bidSkeleton}(d, Y)$ . This is abbreviated as  $X \cong Y(d)$ .

Like confusability,  $d$ -confusability is an equivalence relation; in particular, it is transitive: if  $X \cong Y(d)$  and  $Y \cong Z(d)$ , then  $X \cong Z(d)$ .

**Note:** The operation *skeleton* may change the `Bidi_Class` of characters, so it does not commute with the reordering and mirroring steps, and needs to be performed after them.

**Example:** The sequences of code points  $S_1$  and  $S_2$  are LTR-confusable:

$S_1 := "A1<w" = (\text{LATIN CAPITAL LETTER A, DIGIT ONE, LESS-THAN SIGN, HEBREW LETTER SHIN, HEBREW POINT SIN DOT})$

$S_2 := "A1<w" = (\text{GREEK CAPITAL LETTER ALPHA, HEBREW LETTER SHIN, HEBREW POINT HOLAM HASER FOR VAV, GREATER-THAN SIGN, DIGIT ONE})$

Computation of  $\text{bidSkeleton}(\text{LTR}, S_1)$ :

1.  $R_1 = (\text{LATIN CAPITAL LETTER A, DIGIT ONE, LESS-THAN SIGN, HEBREW POINT SIN DOT, HEBREW LETTER SHIN})$
2.  $R'_1 = (\text{LATIN CAPITAL LETTER A, DIGIT ONE, LESS-THAN SIGN, HEBREW LETTER SHIN, HEBREW POINT SIN DOT})$
3.  $R''_1 = (\text{LATIN CAPITAL LETTER A, DIGIT ONE, LESS-THAN SIGN, HEBREW LETTER SHIN, HEBREW POINT SIN DOT})$
4.  $\text{bidiskeleton}(\text{LTR}, S_1) = \text{skeleton}(R''_1) = (\text{LATIN CAPITAL LETTER A, LATIN SMALL LETTER L, LESS-THAN SIGN, HEBREW LETTER SHIN, COMBINING DOT ABOVE})$

Computation of  $\text{bidiSkeleton}(\text{LTR}, S_2)$ :

1.  $R_2 = (\text{GREEK CAPITAL LETTER ALPHA, DIGIT ONE, GREATER-THAN SIGN, HEBREW POINT HOLAM HASER FOR VAV, HEBREW LETTER SHIN})$
2.  $R'_2 = (\text{GREEK CAPITAL LETTER ALPHA, DIGIT ONE, GREATER-THAN SIGN, HEBREW LETTER SHIN, HEBREW POINT HOLAM HASER FOR VAV})$
3.  $R''_2 = (\text{GREEK CAPITAL LETTER ALPHA, DIGIT ONE, LESS-THAN SIGN, HEBREW LETTER SHIN, HEBREW POINT HOLAM HASER FOR VAV})$
4.  $\text{bidiskeleton}(\text{LTR}, S_2) = \text{skeleton}(R''_2) = (\text{LATIN CAPITAL LETTER A, LATIN SMALL LETTER L, LESS-THAN SIGN, HEBREW LETTER SHIN, COMBINING DOT ABOVE})$

*Review note: Consider moving the details of the computation (but not the basic example) to an appendix.*

Note that these sequences are not RTL-confusable; indeed in a right-to-left paragraph, the strings look distinct:

$$S_1 = "\text{w}>\text{A}1"$$

$$S_2 = "1<\text{wA}"$$

LTR, and RTL, and FS confusability should be used when it is inappropriate to enforce that strings be single-script, or at least single-directionality; this is the case in programming language identifiers. See Section 4.1, *Confusability Mitigations*, in *Unicode Technical Standard #55, Unicode Source Code Handling* [UTS55].

Bidirectional confusability is costlier to check than confusability, as the bidirectional algorithm must be applied. However, a fast path can be used: if  $d=\text{LTR}$  and  $X$  has no characters with bidi classes R or AL,  $\text{bidiSkeleton}(X) = \text{skeleton}(X)$ .

Further, if the strings are known not to contain explicit directional formatting characters (as is the case for UAX31-R1 Default Identifiers defined in *Unicode Standard Annex #31, Identifiers and Syntax* [UAX31]), the algorithm can be drastically simplified, as the X rules are trivial, obviating the need for the directional status stack. The highest possible resolved level is then 2; see Table 5, *Resolving Implicit Levels*, in *Unicode Standard Annex #9, Unicode Bidirectional Algorithm* [UAX9].

**Note:** As is the case for *skeleton*, the strings *bidirectionalSkeleton(d, X)* and *bidirectionalSkeleton(d, Y)* are **not** intended for display, storage or transmission.

## #51. Proposed changes to Unicode Technical Standard #51

#51/4 Add a note in *Section 4, Presentation Style*, as follows.

### 4 Presentation Style

Certain emoji have defined variation sequences, in which an emoji character can be followed by an invisible **emoji presentation selector** or **text presentation selector**.

This capability was added in **Unicode 6.1**. Some systems may also provide this distinction with higher-level markup, rather than variation sequences. For more information on these selectors, see **Emoji Presentation Sequences** [[emoji-charts](#)]. For details regarding the use of emoji or text presentation selectors in emoji sequences specifically, see *Section 2.7, Emoji Implementation Notes*.

Implementations should support both styles of presentation for the characters with emoji and text presentation sequences, if possible. Most of these characters are emoji that were unified with preexisting characters. Because people are now using emoji presentation for a broader set of characters, Unicode 9.0 added emoji and text presentation sequences for all emoji with default text presentation (see discussion below). These are the characters shown in the column labeled “Default Text Style; no VS in U8.0” in the **Text vs Emoji** chart [[emoji-charts](#)].

However, even for cases in which the emoji and text presentation selectors are available, it had not been clear for implementers whether the *default* presentation for pictographs should be emoji or text. That means that a piece of text may show up in a different style than intended when shared across platforms. While this is all perfectly legitimate for Unicode characters—*presentation style is never guaranteed*—a shared sense among developers of when to use emoji presentation by default is important, so that there are fewer unexpected or jarring presentations. Implementations need to know what the generally expected default presentation is, to promote interoperability across platforms and applications.

There had been no clear line for implementers between three categories of Unicode characters:

1. **emoji-default**: those expected to have an emoji presentation by default, but can also have a text presentation
2. **text-default**: those expected to have a text presentation by default, but could also have an emoji presentation
3. **text-only**: those that should only have a text presentation

These categories can be distinguished using properties listed in **Annex A: Emoji Properties and Data Files**. The first category are characters with **Emoji=Yes** and **Emoji\_Presentation=Yes**. The second category are characters with **Emoji=Yes** and **Emoji\_Presentation=No**. The third category are characters with **Emoji=No**.

The presentation of a given emoji character depends on the environment, whether or not there is an emoji or text presentation selector, and the default presentation style (emoji versus text). In informal environments like texting and chats, it is more appropriate for most emoji characters to appear with a colorful emoji presentation, and only get a text presentation with a text presentation selector. Conversely, in formal environments such as word processing, it is generally better for emoji

characters to appear with a text presentation, and only get the colorful emoji presentation with the emoji presentation selector.

Based on those factors, here is typical presentation behavior. However, these guidelines may change with changing user expectations.

*Editor's note: table "[Emoji versus Text Display](#)" omitted.*

There is an additional complication which has to do with computer language syntaxes. Some code points had been reserved for syntactic use in computer languages using the `Pattern_Syntax` property; some of them have been given default emoji presentation. However, all of them have valid text presentation sequences which can be used to unambiguously express that they should be displayed and interpreted as syntactic characters. See *Section 7.2, Emoji Profile*, in Unicode Standard Annex #31, *Unicode Identifiers and Syntax* [UAX31].



## #55. Proposed Unicode Technical Standard #55

*Editor's note: the entirety of the text below forms a new document. For the sake of readability, it has not been written on a yellow background nor indented.*

### UNICODE SOURCE CODE HANDLING

## 1. Introduction

Source code, that is, plain text meant to be interpreted as a computer language, poses special security and usability issues that are absent from ordinary plain text. The reader (who may be the author or a reviewer) should be able to ascertain some properties of the underlying representation of the text by visual inspection, such as:

- the extent of lexical elements within the text;
- the nature of a lexical element (comment, string, or executable text);
- the order in memory of lexical elements;
- the equivalence or inequivalence of identifiers.

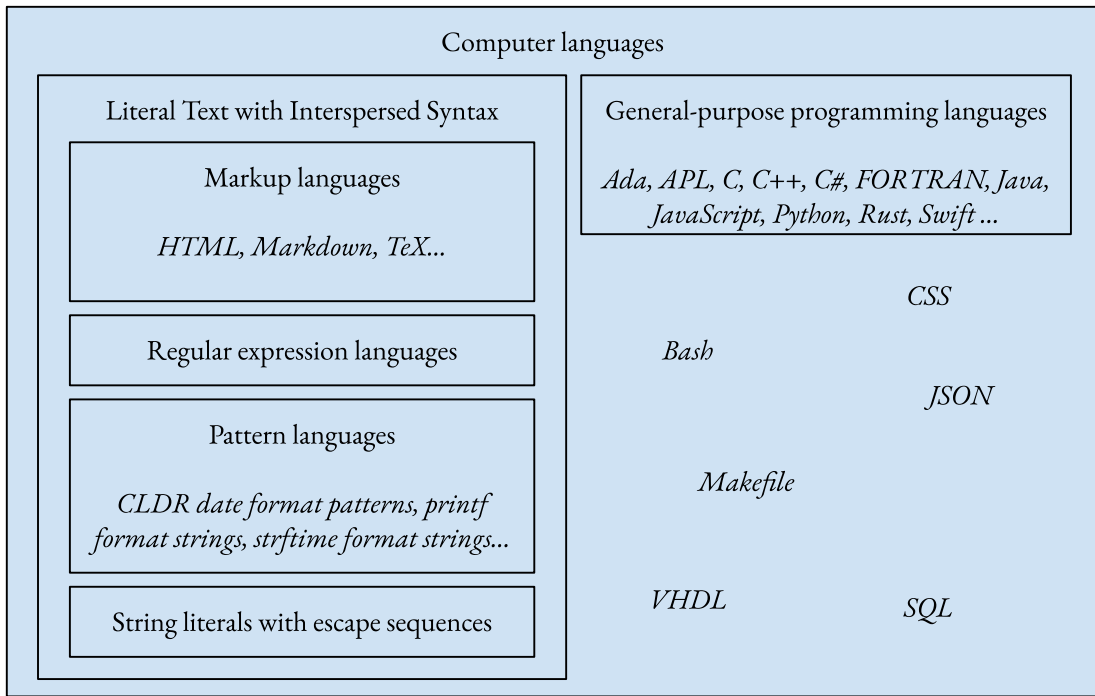
The potential presence in source code of characters from many writing systems, including ones whose writing direction is right-to-left, can make it difficult to ensure these properties are visually recognizable. Further, the reader may not be aware of these sources of confusion. These issues should be remedied at multiple levels: as part of computer language design, by ensuring that editors and review tools display source code in an appropriate manner, and by providing diagnostics that call out likely issues.

Accordingly, this document provides guidance for multiple levels in the ecosystem of tools and specifications surrounding a computer language. *Section 2, Computer Language Specifications*, is aimed at language designers; it provides recommendations on the lexical structure, syntax, and semantics of computer languages. *Section 3, Source Code Display*, is aimed at the developers of source code editors and review tools; it specifies appropriate behavior for source code display. *Section 4, Tooling and diagnostics*, is aimed more broadly at developers in the overall ecosystem around a computer language; it provides guidance for higher-level diagnostics, such as compiler warnings, lint checks, etc., as well as text transformations applicable to pretty-printers and similar tools.

**Note:** While, for the sake of brevity, many of the examples in this document make use of non-ASCII identifiers, most of the issues described here apply even if non-ASCII characters are confined to strings and comments. Further, some of the remedies require allowing specific non-ASCII characters between lexical elements; see *Section 2.2, Whitespace and Syntax*.

Most of the recommendations and specifications in this document are relevant to a broad range of computer languages, from markup languages such as HTML to general-purpose programming languages such as C. Some recommendations are specific to certain classes of languages. In particular, some recommendations in *Section 2, Computer Language Specifications*, apply only to general-purpose programming languages, and the specifications in *Section 3, Source code display*, have special considerations for the broad class of languages consisting of literal text with interspersed syntax (which includes markup languages, but also regular expression languages, etc.). This classification is illustrated in Figure 1.

**Note:** Programming environments such as Wolfram Mathematica where the intended display of source code is rich text (or graphical), rather than plain text highlighted according to its lexical and syntactic structure, are outside the scope of this specification.



**Figure 1.** Classification of computer languages used in this specification.

## 1.1 Source code spoofing

The basic problem occurs when two different lines of code (in memory) can have the same (or confusingly similar) appearance on the screen. That is, the actual text is different from what the reader perceives it to be. This allows a contributor to fool a reviewer into believing that some malicious code is actually innocuous.

Moreover, when a compiler is interpreting the text in a different way than a reader does, inadvertent problems can arise even when there is no malicious intent.

### 1.1.1 Line break spoofing

The Unicode Standard encompasses multiple representations of the New Line Function (NLF). These are described in [Section 5.8, Newline Guidelines](#), of the standard, as well as in Unicode Standard Annex #14, *Line Breaking Algorithm* [UAX14].

An opportunity for spoofing can occur if implementations are not consistent in the supported representations of the newline function: multiple logical lines can be displayed as a single line, or a single logical line can be displayed as multiple lines.

For instance, consider the following snippet of C11, as shown in an editor which conforms to the Unicode Line Breaking Algorithm:

1. `// Check preconditions.`
2. `if (arg == (void*)0) return -1;`

If the line terminator at the end of line 1 is U+2028 Line Separator, which is not recognized as a line terminator by the language, the compiler will interpret this as a single line consisting only of a comment; to a reviewer, the program is visually indistinguishable from one that has a null check, but that check is really absent.

Conversely, consider the following Ada 2005 program, shown in an editor which conforms to the Unicode Line Breaking Algorithm, but does not support the line breaking class NL (whose support is optional for conforming implementations).

1. *-- Here we must not yet `NL`return null;*
2. *-- we need to close the file first.*

While a visible glyph (here `NL`) should still be emitted instead of the unsupported control character (see *Section 5.3, Unknown and Missing Characters*, in *The Unicode Standard*), a reviewer could fail to interpret it as a newline, since line comments are expected to extend to the end of the displayed line. However, Ada 2005 treats U+0085 (next line) as an end of line, so the reviewer would fail to notice that the “comment” is actually executable code that does precisely what it says must not be done.

**Note:** Since syntax highlighting is typically determined by the editor according to its interpretation of line termination—and independently of the compiler’s—it is unlikely to reveal the true extent of the comments in such situations. The examples above have been highlighted accordingly.

The mitigation for this issue includes recommendations for both computer language specifications (see *Section 2.2, Whitespace and Syntax*) and source code editors (see *Unicode Standard Annex #14, Unicode Line Breaking Algorithm [UAX14]*) so that they support the same set of representations of the new line function.

### 1.1.2 Spoofing using lookalike glyphs

The Unicode Standard encodes many characters whose glyphs can be expected to be indistinguishable or hard to distinguish, especially across scripts, but sometimes also within scripts. Examples include Cyrillic, Latin, and Greek Α, Α, and Α, Devanagari कौ (kō) and the “do not use” sequence कौ (\*kâe), etc.

These can be used for spoofing, for instance, by constructing identifiers that look like they are the same, but are actually different.

**Example:** Consider the following C program:

```

1. void zero(double** matrix, int rows, int columns) {
2.     for (int i = 0; i < rows; ++i) {
3.         double* row = matrix[i];
4.         for (int i = 0; i < columns; ++i) {
5.             row[i] = 0.0;
6.         }
7.     }
8. }
```

This program looks like it zeros a `rows` by `columns` rectangle, but it actually only zeros a diagonal, because the identifier `і` on line 4 is a Cyrillic letter, whereas `i` is the Latin letter everywhere else.

The recommended solution for this is twofold: in order to address cases where there are multiple valid representations of a character, computer languages should use equivalent normalized identifiers as described in *Section 2.1.1, Normalization and Case*. In order to address other cases, programming language tools should implement the mitigations described in *Section 4.1, Confusability Mitigation Diagnostics*.

### 1.1.3 Spoofing using bidirectional reordering

The Unicode Bidirectional Algorithm, defined in Unicode Standard Annex #9, is part of the Unicode Standard; it is a necessary part of the display of a number of scripts, such as the Arabic or Hebrew scripts. See [Logical Order](#) in *Chapter 2, General Structure*, and conformance requirement [C12](#) in *Chapter 3, Conformance*, in The Unicode Standard.

Because computer languages have a strong logical structure which differs from that of ordinary plain text, the plain text display of source code may not reflect that logical structure. This can lead to possibilities of spoofing, in particular by using the invisible characters that are used as overrides to the default behavior of the Unicode Bidirectional Algorithm; see [Section 2, Directional Formatting Characters](#), in Unicode Standard Annex #9, *Unicode Bidirectional Algorithm* [UAX9].

#### Examples:

C++98 or later:

```
std::cerr << "encountered " << (errors == 0 ? " 0 " : "")
          << "errors";
```

This program will print “encountered errors” if and only if errors = 0, and “encountered 0 errors” otherwise;

Ada 2005 or later:

```
for Hebrew_Letter in Wide_Character range 'א' .. 'ת' loop
```

While it may seem like it loops over the Hebrew alphabet (from alef to tav), this is actually dead code (looping over the empty range from tav to alef).

Rust 1.9 or later:

```
return x >> 8;
```

While this looks like a right shift by eight bits, it is a left shift by eight bits.

The solution is not to forbid the directional formatting characters; indeed the Ada example above does not use these. The recommendation is instead twofold.

1. Source code editors should display source code according to its lexical structure, as described in [Section 3.1, Bidirectional Ordering](#).
2. In addition, computer languages should allow for the insertion of directional formatting characters as described in [Section 2.2, Whitespace and Syntax](#), and implementers should provide tools that automatically remove spurious directional formatting characters, and insert the correct ones, as described in [Section 4.2, Conversion to Plain Text](#).
  - Maintainers of code bases concerned about spoofing can then enforce the application of this conversion to plain text, so that the code looks as it should wherever it is displayed, even in review tools that fail to apply the recommendations for display of source code.

## 1.2 Usability issues

The same issues described in [Section 1.1, Source code spoofing](#), can affect usability, as one may be misled by the appearance of one’s own code, leading to unexpected behavior, or to compilation errors that cannot be explained by reading the source code. There are however additional usability issues that are not identical to the spoofing issues: the bidirectional display of code treated as plain text can lead to reordering that obscures the logical structure of the computer language, making a program illegible.

### 1.2.1 Usability issues arising from lookalike glyphs

When working with multiple scripts, there is a common usability issue whereby one accidentally types some letters using the wrong keyboard layout. Consider a user trying to type the definition of a class `HTTPOTBET` (*HTTPResponse*). The user would start typing using a Latin keyboard layout:

```
class HTTPotwe
```

Noticing that letters are being typed in the wrong script, the user might then backspace the visibly wrong letters, switch keyboard layout, and type the remainder:

```
class HTTPOtwe
class HTTPOTBET {
```

Trying to refer to `HTTPOTBET` will lead to a compilation error (because it is actually declared as `HTTPOTbet`, with a Latin O). This error can be hard to understand: no amount of time spent looking at the code will reveal it.

A similar issue can occur in a codebase whose identifiers are restricted to the Latin script, if, for instance, comments or string literals are written in a different script; after typing a Cyrillic comment, a user might likewise switch layout midway through an attempt declare an `XMLDocument`, and get a confusing error message, because the resulting identifier has a Cyrillic X and M.

The recommended mitigations for these usability issues are the same as the mitigations for the corresponding spoofing issues described in *Section 1.1.2, Spoofing using lookalike glyphs*.

### 1.2.2 Usability issues arising from bidirectional reordering

The presence of strongly right-to-left characters in source code, including in comments and string literals, can easily mangle source code into unreadability if it is displayed as plain text, even when the result does not look like a valid program, and therefore does not pose a spoofing issue.

#### Examples:

C# 1.0 or later:

```
Console.WriteLine("הודעה", "{1} {0} :השתבש, this);
parses—and is typed—as
Console.WriteLine("Error: {0} ({1})", message, this);
```

Ada 2005 or later:

```
<<שגיאה>> רשם (הודעה); -- משהו השתבש
parses—and is typed—as
<<Error>> Log (Message); -- Something went wrong.
```

Python 3.0 or later:

```
return אינטגרל(lambda 1=ל, 0=מ, 2 ** א :א)
parses—and is typed—as
return integral(lambda a: a ** 2, from_=0, to=1)
```

Rust 1.53.0 or later:

```
fn אינטגרל>פ Fn(f64) -> f64<
    קטע אינטגרנד: פ, std::ops::Range<f64>) -> f64 {
parses—and is typed—as
```

```
fn integral<F: Fn(f64) -> f64>(
    integrand: F, interval: std::ops::Range<f64>) -> f64 {
```

C++11 or later:

```
std::vector<مواء <قطعة>;
```

parses—and is typed—as

```
std::vector<meow> cat;
```

Note that the same can occur if the right-to-left identifiers are limited to string literals and comments; in the same language,

```
return u8"مواء"; // رسالة العنصر النائب
```

parses—and is typed—as

```
return u8"meow"; // Placeholder message.
```

The recommended mitigations for these usability issues are the same as the mitigations for the corresponding spoofing issues described in *Section 1.1.3, Spoofing using bidirectional reordering*.

### 1.3 Conformance

An implementation claiming conformance to this specification must do so in conformance to the following clauses:

**C0** An implementation claiming conformance to this specification shall identify the version of this specification.

**C1** An implementation claiming to implement the Basic Ordering for Source Code shall do so in accordance with the specifications in *Section 3.1.2, Basic Ordering*.

**C2** An implementation claiming to implement the Ordering for Literal Text with Interspersed Syntax shall do so in accordance with the specifications in *Section 3.1.4, Ordering for Literal Text with Interspersed Syntax*.

**C3** An implementation claiming to implement Mixed-Script Detection in Identifier Chunks shall do so in accordance with the specifications in *Section 4.1.2, Mixed-Script Detection in Identifier Chunks*.

**C4** An implementation claiming to implement Conversion to Plain Text for Source Code shall do so in accordance with the specifications in *Section 4.2, Conversion to Plain Text*.

**C5** An implementation claiming to enforce Unicode Identifier Styles shall do so in accordance with the specifications in *Section 4.3, Identifier Styles*.

## 2. Computer Language Specifications

The normative material appropriate for language specifications may be found in Unicode Standard Annex #31, *Identifiers and Syntax* [UAX31]. Since that annex has a broader scope than computer languages—including usernames, hashtags, etc.—specific recommendations for language designers are given here.

### 2.1 Identifiers

Computer languages that require *forward* compatibility in their identifier definitions should use the definition of identifiers given by requirement UAX31-R2 Immutable Identifiers.

Unless they require forward as well as backward compatibility, computer languages should use the definition of identifiers given by requirement UAX31-R1 Default Identifiers.

**Note:** The characters having the General\_Category Mn or Mc (nonspacing spacing combining marks) should not be excluded from default identifiers by a profile; while precomposed characters exist for many common combinations in the Latin script, combining marks are critical to many other scripts. For instance, word-internal vowels in Indic scripts have the General\_Category Mn or Mc.

Profiles may be needed to adjust to the specifics of a language, such as allowing an initial U+005F LOW LINE (\_).

General-purpose programming languages should extend the identifier definition using the mathematical notation profile defined in *Section 7.1, Mathematical Notation* of Unicode Standard Annex #31, *Unicode Identifiers and Syntax* [UAX31]. This is because these languages are used in scientific computing, which benefits from the greater legibility and disambiguation afforded by allowing these additional characters in identifiers.

### 2.1.1 Normalization and Case

It is recommended that all languages that use default identifiers meet requirement UAX31-R4 *Equivalent Normalized Identifiers*, with the normalization described in this section.

**Note:** Alternatively, languages can meet requirement *UAX31-R6 Filtered Normalized Identifiers*. However, some input methods produce non-normalized text, which can make it difficult to use a language implementing this requirement; in the case of NFKC, filtered normalized identifiers can impose unnatural restrictions on the visual representation of source code.

When implementing equivalent normalized identifiers, implementations should treat identifiers as their normalized forms; for instance, linker symbols should be based on the normalized form. This is similar to the situation for case-insensitive languages.

Case-sensitive languages should meet requirement UAX31-R4 with normalization form C. They should not ignore default ignorable code points in identifier comparison.

Case-insensitive languages should meet requirement UAX31-R4 with normalization form KC, and requirement UAX31-R5 with full case folding. They should ignore default ignorable code points in comparison. Conformance with these requirements and ignoring of default ignorable code points may be achieved by comparing identifiers after applying the transformation toNFKC\_Casefold.

**Note:** Full case folding is preferable to simple case folding, as it better matches expectations of case-insensitive equivalence. For compatibility, some implementations may wish to use simple case folding; alternatively, they can migrate to full case folding using the processes described in *Section 2.3, Language Evolution*.

*Review note: While toCasefold toNFKC is stable, toNFKC\_Casefold is not, because Default\_Ignorable\_Code\_Point is not. The Default\_Ignorable\_Code\_Point has changed over time for already-encoded characters, so we may not want to stabilize it; but it may be possible to stabilize it on the set XID\_Continue: “once a character is XID\_Continue, the value of the Default\_Ignorable\_Code\_Point property will never change for that character”.*

The reason for these recommendations is that failing to support normalization creates interchange problems, as canonically equivalent strings are expected to be interpreted in the same way, and distinctions between canonically equivalent sequences are not guaranteed to be preserved in interchange; see [Section 2.12, Equivalent Sequences](#), and conformance clause C6 in [Section 3.2, Conformance Requirements](#), [subsection “Interpretation”](#) in *The Unicode Standard*.

If a language supports non-ASCII identifiers and does not take normalization into account, and implements equivalent normalized identifiers with a normalization other than the recommended one, special compatibility considerations apply when switching to the recommended behavior. See [Section 2.3, Language Evolution](#).

The choice between Normalization Form C and Normalization Form KC should match expectations of identifier equivalence for the language.

In a case-sensitive language, identifiers are the same if and only if they look the same, so Normalization Form C (canonical equivalence) is appropriate, as canonical equivalent sequences should display the same way.

In a case-insensitive language, the equivalence relation between identifiers is one of abstract characters; for instance, `e` and `E` are the same abstract letter. Normalization Form KC (compatibility equivalence) is an equivalence between abstract characters.

**Example:** In a case-insensitive language, `SO` and `so` are the same identifier; if that language uses Normalization Form KC, the identifiers `so` and `so` are likewise identical.

### 2.1.2 Semantics Based on Case

Computer languages should not solely depend on case for semantics; that is, if case indicates a semantic distinction in a language, it should be possible to express that distinction in some other way that does not involve case, such with a symbol or a dedicated syntax. This is because many writing systems are unicameral (that is, they do not have separate lowercase and uppercase letters), so that users of those writing systems would have no way of specifying that distinction. See [Section 5.18, Case Mappings](#), in *The Unicode Standard*.

**Note:** In general, when placing requirements on case, implementations should disallow the unwanted case (*e.g.*, disallow lowercase), rather than requiring the desired case (*e.g.*, requiring uppercase). See also [Section 4.3, Identifier Styles](#).

**Example:** Consider a programming language that meets requirement UAX31-R1, Default Identifiers, with a profile that adds `_` to the set *Start*.

That language should not require identifiers to start with an uppercase letter (General\_Category Lu) or a titlecase letter (General\_Category Lt) in order to be public (so that `Example` is public, and `example` or `_Example` are private), as it would be impossible to create a public identifier using CJKV ideographs.

That language could, however, achieve a similar effect for bicameral scripts by treating identifiers that start with a lowercase letter (General\_Category Ll) or a non-letter (General\_Category other than L, such as `_`) as private. The identifier `Example` would still be public, and `example` or `_Example` would still be private. However, this definition allows the users of unicameral scripts to prefix identifiers with `_` in order to make them private: `例` would be public, and `_例` would be private.



Alternatively, that language could have a syntax to explicitly declare an identifier as public, which would then enforce the case convention in bicameral scripts, but not require it in unicameral scripts:

```
public Example // OK.
public example // Error.
public 例     // OK.
private 例    // OK.
```

Languages that enforce a specific case convention should do so according to the specification in *Section 4.3, Identifier styles*.

## 2.2 Whitespace and Syntax

It is recommended that all computer languages meet requirement UAX31-R3a *Pattern\_White\_Space Characters*, which specifies the characters to be interpreted as end of line and horizontal space, as well as ignorable characters to be allowed between lexical elements, but not treated as spaces.

Using the specified end of line characters prevents spoofing issues; see *Section 1.1.1, Line break spoofing*. Note that the line terminators listed in UAX31-R3a will be interpreted as line terminators by any editor that implements the Unicode Line Breaking Algorithm. See *Unicode Standard Annex #14, Unicode Line Breaking Algorithm* [UAX14].

**Note:** Alternatively, a language could forbid those of the specified line terminators which it does not recognize. Care must be taken to forbid the unrecognized ends of line even in line comments, in order to prevent the issues described in *Section 1.1.1, Line break spoofing*.

Allowing the specified ignorable format controls between lexical elements allows the author of the program to correct its plain-text display by inserting characters where needed, to use a tool to perform these insertions as described in *Section 4.2, Conversion to plain text*. Correct display in plain text is useful, because even if all source code editors and review tools were to implement the recommendations for display in *Section 3.1, Bidirectional Ordering*, source code is often cited verbatim in environments that are not aware of its lexical structure, such as compiler diagnostics or version control diffs written to the console, patches or other code snippets sent via email, etc.

**Industry examples:** Ada 2012 has a concept of ignorable format controls, as characters with General\_Category Cf “are allowed anywhere that a [space] is [and have] no effect on the meaning of an Ada program”; see the Ada Reference Manual, [2.2\(7.1\)](#). It recognizes the specified line terminators.

Rust also allows the left-to-right and right-to-left marks wherever space is allowed; however it treats those as spaces, and it only recognizes the line feed as a line terminator.

**Example:** Consider the following line of C++11, displayed according to the recommendations in *Section 3.1, Bidirectional Ordering*, and assume the identifier תו is undeclared:

```
if (x + תו == 1) {
```

A compiler might emit the following message:

```
<source>:<line>:11: error: use of undeclared identifier 'תו'
if (x + 1 == תו) {
           ^
```

As the cited code is being shown by a terminal which is not aware of the lexical structure of the language (note, for instance, the lack of syntax highlighting), it is improperly displayed; the condition looks like a different one (x plus one equals tav, rather than x plus tav equals one), and the caret points to the wrong place.

Consider now a corresponding line of Rust, where, for clarity, we have made the left-to-right mark visible as described in *Section 3.2, Suggested representations for directional formatting characters*.

```
if x + תו▶ == 1 {
```

A compiler might emit the following message:

```
error[E0425]: cannot find value `תו` in this scope
--> <source>:<line>:12
3 |     if x + תו == 1 {
  |             ^^ not found in this scope
```

The presence of the left-to-right mark causes the code to be displayed correctly even in the language-unaware terminal. Programmers should not be expected to enter these characters themselves; instead tools should be provided that implement the mechanism described in *Section 4.2, Conversion to Plain Text*.

*Editor's note: See [/z/YcrzTMGnP](https://ericniebler.com/2019/01/24/gcc-godbolt/) on [gcc.godbolt.org](https://gcc.godbolt.org) for the C++, and [/z/W3YK8jvM](https://github.com/rust-lang/rust/pull/78844) for the Rust (and lament the display of LRM as `[U+200E]`).*

It is further recommended that languages allow the ignorable format controls between atoms, as defined in *Section 3.1, Bidirectional Ordering*, to the extent possible, even if the atom boundary occurs within a single lexical element.

**Example:** In C++11 and later, the following is a user-defined string literal, which consists of a single token:

```
"text" _מחרוזת
```

It is syntactic sugar for the following function call:

```
operator"" _מחרוזת("text")
```

If the text ends with a strongly right-to-left character, the plain text display of the token with left-to-right paragraph direction is misleading:

```
"מחרוזת_ "א
```

Inserting a left-to-right mark after the closing quotation mark fixes the issue:

```
"א" _מחרוזת
```

However, this requires allowing this character within what is technically a single token. A similar issue occurs with Rust suffixes.

It is recommended that programming languages that allow for user-defined operators, as well as languages that consist of a mixture of literal characters and syntax, such as pattern or regular expression languages, meet requirements UAX31-R3b *Pattern\_Syntax Characters*. It is further recommended that programming

languages that allow for user-defined operators meet requirement UAX31-R3c *Operator Identifiers*. As in the case of programming language identifiers, operators should be treated as equivalent under normalization, that is, these languages should meet requirement UAX31-R4 *Equivalent Normalized Identifiers* for their operators as well as their identifiers. Normalization Form C, rather than KC should always be used for operators rather than. This is because sequences that are equivalent under Normalization Form KC may have different appearances, but programming language operators are not expected to have diverse appearances. For instance, it would be confusing for an operator  $\text{ϕϕ}$  to be the same as an operator  $\text{ϕϕ}$ , but these are equivalent under Normalization Form KC.

**Industry example:** The Swift programming language uses a definition of [operators](#) which corresponds to UAX31-R3c with a small profile.

Languages that do not allow for user-defined operators should nevertheless claim conformance to UAX31-R3b, thereby reserving the classes of characters which may be assigned to syntax or identifiers in future versions. This ensures compatibility should they add additional operators or allow for user-defined operators in future versions. It also allows for better forward compatibility of tools that operate on source code but do not need to validate its lexical correctness, such as syntax highlighters, or some linters or pretty-printers; unidentified runs of characters neither reserved for whitespace nor syntax can be treated as identifiers, which they might become when the language moves to a newer version of the Unicode Standard. See the implementation note in *Section 4.2, Conversion to Plain Text*.

Languages that declare a profile for identifiers may need to declare a corresponding profile for requirement UAX31-R3b. For the standard profiles defined in *Section 7, Standard Profiles* in Unicode Standard Annex #31, *Identifiers and Syntax* [UAX31], the corresponding profile for UAX31-R3b is described if needed.

## 2.3 Language Evolution

The recommendations in the preceding sections apply directly when adding Unicode support to a previously ASCII-only language, or when creating a new language. However, when changing a language that already supports Unicode identifiers to align with these recommendations, special compatibility considerations come into play.

### 2.2.1 Changing Identifier Definitions

As requirements change or become clearer, implementations may need to switch from one definition of identifiers to another; for instance, from immutable identifiers to default identifiers, if normalization or spoofing concerns arise with the use immutable identifiers, and forward compatibility is unneeded; or from default identifiers to immutable identifiers, if forward compatibility turns out to be needed.

Switching from default identifiers to immutable identifiers does not pose backward compatibility issues. However, when switching from immutable to default identifiers, it is likely that existing programs will be affected.

In particular, two likely patterns of use of characters outside of XID\_Continue are mathematical notation and emoji. Standard profiles are provided for both of these in *Section 7, Standard Profiles*, in Unicode Standard Annex #31, *Unicode Identifiers and Syntax* [UAX31]. When switching from immutable to default identifiers, it is recommended to extend the identifier definition using these profiles if these patterns of use are attested. Note that the mathematical notation profile is also recommended on its own merits, regardless of compatibility concerns; See *Section 2.1, Identifiers*.

### 2.2.2 Changing Normalization and Case

Some languages have introduced support for Unicode identifiers without taking normalization into account. A lack of support of normalization leads to interoperability problems, as canonically equivalent strings are expected to be interpreted in the same way, and distinctions between canonically equivalent sequences are not guaranteed to be preserved in interchange; see [Section 2.12, Equivalent Sequences](#), and conformance clause C6 in [Section 3.2, Conformance Requirements](#), [subsection “Interpretation”](#) in *The Unicode Standard*.

As a result, if a language does not meet requirement UAX31-R4 *Equivalent Normalized Identifiers*, its designers may wish to change its definition of identifier equivalence to meet that requirement.

Similarly, a language designer may wish to switch between Normalization Form KC and Normalization Form C to align with the recommendations in [Section 2.1.1, Normalization](#); or a language designer may wish to switch between case-sensitive and case-insensitive identifier definitions.

These changes are all subject to backward compatibility issues. In particular, there is a risk that a previously-legal program would remain legal, but change behavior, as in the following example:

```

1. // Prints documents (consisting of a sequence of lines) to file f,
2. // and prints the number of lines of each document, as well as the
3. // total number of lines, to standard output.
4. // A counter for the total number of lines printed.
5. int lignes_imprimées = 0; // Decomposed e +  .
6. for (std::vector<std::string> const& document: documents) {
7.     // A counter for the number of lines in the document.
8.     int lignes_imprimées = document.size(); // Precomposed é.
9.     // Print each line of the document.
10.    for (std::string const& ligne: document.lignes()) {
11.        std::fputs(ligne.c_str(), f);
12.        ++lignes_imprimées; // Decomposed e +  .
13.    }
14.    std::printf("%s : %d lignes imprimées",
15.                document.front().c_str(),
16.                lignes_imprimées); // Precomposed é.
17. }
18. // Report the total number of lines printed.
19. std::printf("total : %d lignes imprimées",
20.             lignes_imprimées); // Decomposed e +  .

```

If normalization is not taken into account, the above program works as commented. If the implementation uses UAX31-R4 equivalent normalized identifiers, the program always reports that 0 lines were printed in total, and reports double the actual number of lines for each document: the counter declared on line 8 shadows the one declared on line 5, so that line 12 increments the counter declared inside the loop over documents, rather than the outer counter.

In order to safely transition from one identifier equivalence to another, implementations should warn if identifiers exist that are equivalent under the new rules but not under the old rules, or vice-versa. This check for coexistence could be limited to scopes, depending on the rules of the language and the capabilities of the tool issuing the diagnostic; see the discussion in [Section 4.1, Confusability Mitigation Diagnostics](#).

**Note:** Confusable detection as described in Section 4.1 encompasses such a warning, as canonically equivalent sequences are always confusable. The reverse is however not true: the Latin A, Greek Α, and Cyrillic А are confusable but not equivalent.

Note that this is not necessary if only one of the newly equivalent forms was permitted: no special backward compatibility considerations are required when switching from UAX31-R6 Filtered Normalized Identifiers to UAX31-R4 Equivalent Normalized Identifiers, or from UAX31-R7 Filtered case-insensitive identifiers (lowercase-only identifiers) to UAX31-R5 Equivalent case-insensitive identifiers.

### 3. Source code display

Most of the issues described in Section 1 are difficult to usefully address as part of the lexical structure of a language, such as in the definition of identifiers. Language specifications, which usually evolve more slowly than Unicode, are also ill-equipped to alleviate these issues. At the same time, since they are issues that arise from a discrepancy between the visual interpretation of code and its interpretation by a compiler, these issues only affect source code that is shown to a human; a compiler interpreting generated code should not have to implement complex legality rules inspired by visual spoofing concerns.

Instead, diagnostics for these issues are best mitigated by tools in the broader ecosystem of the language; this may include compiler warnings, but also linters, pretty-printers, editor highlighting, etc.

In some cases, such as most of the ordering issues, the issue simply arises from inappropriate display of source code; in that case the best remedy is to display the code in a way that is consistent with its lexical or syntactic structure. This section provides guidance on the display of source code.

#### 3.1 Bidirectional Ordering

The issues arising from bidirectional reordering described in *Section 1.1.3, Spoofing using bidirectional reordering*, and *Section 1.2.2, Usability issues arising from bidirectional reordering*, are resolved by displaying the source code according to its own lexical structure, in application of higher-level protocol HL4 defined in *Section 4.3, Higher-Level Protocols*, defined in Unicode Standard Annex #9, *Unicode Bidirectional Algorithm*.

This section provides more detailed guidance on the application of that protocol to source code.

##### 3.1.1 Atoms

In order to apply protocol HL4, the text must be partitioned into segments. These segments should be entities whose ordering is part of the syntax of the language. We will refer to these entities as *atoms*, as they must not be split in rendering.

Token boundaries are always atom boundaries; that is, the ordering of tokens is part of the syntax of a computer language. However, there may be atom boundaries inside of tokens. For instance, the lexical structures of many languages include delimited tokens such as the following:

1. `-- Line comments.`
2. `(* Block comments. *)`
3. `"String literals."`

In such tokens, the delimiters are ordered syntactically before and after the contents of the token; each of these tokens therefore comprises multiple atoms, as in the following table, where spaces have been made visible as `·`.

--	·Line comments.	
(*	·Block comments.·	*)
"	String literals.	"

Line boundaries are also atom boundaries, so that the contents of a multiline comment or string consist of multiple atoms.

**Note:** In order to avoid the issues described in *Section 1.1.1, Line break spoofing*, source code editors should support all characters treated as hard line breaks in Unicode Standard Annex #14, *Unicode Line Breaking Algorithm* [UAX14], including U+000B VT and U+0085 NEL whose support is optional for general use.

While an editor may warn about unexpected line terminator conventions, it should nevertheless interpret them as line breaks for display purposes. Under no circumstances should an editor remove or ignore unexpected line breaks; see conformance clause C7 in *Section 3.2, Conformance Requirements*, [subsection “Modification”](#) in *The Unicode Standard*. On the other hand, an editor could provide a function to transform all line terminators to a consistent convention.

All hard line breaks should be interpreted as atom boundaries and as line boundaries in algorithms that use atoms, even in languages that do not support them. In such languages, line comments should be processed as block comments whose termination marker happens to be one of the supported line terminators.

Atoms that are part of a comment, but are not comment delimiters, are called *comment content atoms*.

**Example:** The following three-line C-style block comment consists of five atoms:

1. /\*·Author:·Mark·Davis
2. ·\*·Date:···2022-09-13
3. \*/

The atoms are as follows, where atoms (2), (3), and (4) are comment content atoms.

- (1) /\*
- (2) ·Author:·Mark·Davis
- (3) ·\*·Date:···2022-09-13
- (4) ·
- (5) \*/

Runs of whitespace between tokens constitute atoms; these are called *whitespace atoms*. Ignorable format controls, as defined in *Section 4.1, Whitespace*, in Unicode Standard Annex #31, are part of any adjacent whitespace atom; if they are not adjacent to whitespace, they form their own whitespace atoms.

**Example:** The following line of Rust consists of thirteen atoms.

1	2	3	4	5	6	7	8	9	10	11	12	13
if	·	x	·	+	·	ת	▶▶	==	·	1	·	{

As a special exception, numeric literals that use the digits 0 through 9, and, for higher bases, the letters from the Basic\_Latin block, should be treated as single atoms, even if they have inner lexical structure. These atoms are called *numeric atoms*. For instance, the hexadecimal numeric literal `0xDEAD'BEEF` should be treated as a single atom, not as the sequence of five atoms (`0`, `x`, `DEAD`, `'`, `BEEF`). Likewise `3.14159_26E0` is a single atom, not seven. This is because ASCII numbers are left-to-right even when used with a right-to-left writing system.

**Note:** It is not recommended for general-purpose languages to support numbering systems other than the digits 0 through 9 in numeric literals (as well as the ASCII letters for hexadecimal). This is because the ASCII digits are generally acceptable in technical contexts, and numbering systems introduce unique confusability issues (for instance, ৪ is a Bengali digit four, but looks like the digit 8). At the same time, supporting these numbering systems may be very complex, especially in the case of systems that are not positional, such as Chinese or Roman numerals: 1729 = 一千七百二十九 = CDCCXXIX.

An identifier, the contents of a single-line string literal, and the contents of a single-line comment should each form a single atom.

**Note:** In particular, it is not appropriate to treat each character as an atom (which would lead to displaying characters left-to-right as if the Unicode Bidirectional Algorithm were not applied), as this would render any right-to-left text illegible, or even misleading; for instance, a string rendered by forcibly ordering the characters left-to-right as "مرح با" looks like it says “welcome” with broken shaping, it would actually be printed as “اب حرم”, “forbidden father”.

### 3.1.2 Basic ordering

The basic ordering is applicable to computer languages other than marked-up text and pattern languages.

In the basic ordering, the atoms on each line of a source code document are ordered either left-to-right or right-to-left; this order remains the same throughout the document. This *atom order* may be determined as an editor setting, or from the properties or contents of the document; a language specification could also define a default order, or a mechanism to specify the order. The determination of atom order is outside the scope of this specification.

Editors are encouraged to support both atom orders, but should at a minimum support the display described in this section using left-to-right atom order.

Each atom should be displayed using the Unicode Bidirectional Algorithm, using protocol HL1 to set the paragraph direction consistently with the atom direction, with the following exceptions:

1. numeric atoms should always have left-to-right paragraph direction, even with right-to-left atom order;
2. comment content atoms should have their directionality set according to rule P2 of the Unicode Bidirectional Algorithm (that is, they should have “first strong” direction);
3. when an atom has inner structure, that structure should be taken into account when displaying it, as described in *Sections 3.1.3, Embedded languages*.

*Review note: It is unclear whether using the atom direction is the right choice for the contents of string literals.*

*The following alternatives were discussed by the SCWG:*



1. *First-strong. Problems:*
  - a. *The standard technique of inserting a mark to override a first-strong heuristic doesn't work, as that changes the string.*
  - b. *Doesn't work when converting to plain text (an FSI/PDI pair would need to be inserted, which, again, changes the string).*
2. *Recommend that language specifications allow stateful characters preceding and following a string literal. Problem:*
  - a. *Requires special handling when implementing HL4, so that the effect of these characters persists into the string literal.*
  - b. *Manipulating the stateful characters is fiddly, even more so than the stateless characters.*
3. *Recommend that language specifications ignore a stateful character immediately after the opening quotation mark (and a corresponding pop immediately before the closing quotation mark). Problems:*
  - a. *This would be an incompatible change to existing language specifications.*
  - b. *Manipulating the stateful characters is fiddly, even more so than the stateless characters.*
4. *First-strong, and recommend that language specifications ignore an LRM or RLM after the opening quote. Problems:*
  - a. *This would be an incompatible change to existing language specifications.*
  - b. *Doesn't work when converting to plain text (an FSI/PDI pair would need to be inserted).*
5. *Recommend that language specifications introduce a dedicated RTL" syntax, which could allow for an ignorable RLE or RLI so that it supports conversion to plain text. This is appealing, because it avoids all of the issues above, and uses a visible mechanism. However, some questions of language semantics need to be investigated more deeply; for instance, would it be useful for such literals to have a dedicated type that carries the information "this string wants to be in an RTL span in order to display properly"?*

**Implementation note:** When source code is displayed using HTML, the basic ordering may be achieved as follows, where *d* is atom order (ltr or rtl),

1. Enclose each atom in a `<span>`, and apply the attribute `dir=d` to it, except that:
  - a. numeric atoms have the attribute `dir="ltr"` regardless of the value of *d*;
  - b. comment content atoms have the attribute `dir="auto"` regardless of the value of *d*.
2. Apply the attribute `dir=d` to the element containing each line.

Note that if code is displayed using syntax highlighting, the `<span>` elements from step 1. are likely to already exist; they only need to have their `dir` attribute set appropriately.

**Example:** The Python example from *Section 1.2.2, Usability issues arising from bidirectional reordering*, should be displayed as follows with left-to-right atom order:

```
return אינטגרל(lambda א: א ** 2, מ=0, ל=1)
```

and as follows with right-to-left atom order:

```
(1=ל ,0=מ ,2 ** א :א lambda) אינטגרל return
```

*Editor's note: in the HTML document, use spans with the dir attribute, as in the implementation note.*



**Industry Example:** Microsoft Visual Studio, and Microsoft Visual Studio Code since Version 1.66, implement the basic ordering, except that they use LTR rather than first-strong paragraph direction for comments.

### 3.1.2.1 Equivalent Isolate Insertion for the Basic Ordering

This section describes how one would insert isolates into source code in order to have it appear in the right direction. The purpose of this is to establish a formal logical description of how text should be ordered. This does not mean that isolates should be inserted into a copy of the document for display. Instead higher-level protocols should be used to achieve the same display.

The basic ordering can be formally described in terms of an equivalent insertion of explicit directional formatting characters, as in higher-level protocol HL3 of the Unicode Bidirectional Algorithm. That is, a document displayed according to the basic ordering must display in the same order as a document that is modified according to the procedure below and displayed according to the Basic Display Algorithm of Unicode Standard Annex #9, *Unicode Bidirectional Algorithm*, where each line of source code is treated as a paragraph.

**Note:** Actually inserting explicit directional formatting is not necessary to implement the basic ordering. In particular, when code is displayed using HTML, it is better to make use of the features of that language, as described in the implementation note in *Section 3.1.2, Basic Ordering*. In particular, this avoids the need to terminate unmatched isolates.

This formal specification refers to definitions from Unicode Standard Annex #9, *Unicode Bidirectional Algorithm* [UAX9], and makes use of the abbreviations defined in [Section 2, Directional Formatting Characters](#), in that annex.

- A. Let  $d$  be the atom direction, either LTR or RTL;
- B. Define Atom\_Isolate to be the code point LRI if  $d$ =LTR, or RLI if  $d$ =RTL;
- C. Separate the text into lines and each line into atoms, as described in Section 3.1.1, in a language-dependent manner;
- D. For each line:
  - a. For each atom on the line:
    - i. If the atom is known to consist of literal text with interspersed syntax:
      1. Apply the Equivalent Isolate Insertion for the Ordering for Literal Text with Interspersed Syntax to the text of the atom, using  $d$  as the atom direction.
    - ii. Otherwise, if the atom is known to consist of text in some other computer language:
      1. Gather the text of all consecutive atoms that are in that computer language, including intervening line breaks.
      2. Apply the Equivalent Isolate Insertion for the Basic Ordering to that text, using  $d$  as the atom direction.
      3. Continue the loop with the next atom not yet processed.
    - iii. Compute  $u$ , the number of isolate initiators in the text of the atom that do not have a matching PDI within the text of the atom, as defined by [BD9](#).
    - iv. If the atom is a comment content atom:
      1. Insert an FSI character before the atom;
    - v. Otherwise, if the atom is a numeric atom:
      1. Insert an LRI character before the atom;
    - vi. Otherwise:
      1. Insert Atom\_Isolate before the atom;

- vii. Insert  $u+1$  PDI characters at the end of the atom;
- b. Insert Atom\_Isolate at the beginning of the line;
- c. Insert a PDI character at the end of the line;

### 3.1.3 Embedded languages

If an atom consists of text written in a computer language, and the editor is aware of that structure, the internal display of that atom should itself follow either the basic ordering described in *Section 3.1.2, Basic ordering*, or the ordering described in *Section 3.1.4, Ordering for Literal Text with Interspersed Syntax*, as appropriate.

**Example 1:** Consider the following C# statement, as displayed by an editor which is unaware of the internal structure of the string passed to `Parse`:

```
1. var decomposition_mapping =
2.     System.Text.Json.JsonDocument.Parse(
3.         "'1': 'Ⓢ', 'صلى الله عليه وسلم': 'ﷺ', '4田': '4田'}");
```

The JSON is reordered without regard to its logical structure, misleading the reader as to the identity of keys and values. If the editor is capable of recognizing that the string contains JSON, it should instead display it in the following order, applying the basic ordering to the JSON:

```
1. var decomposition_mapping =
2.     System.Text.Json.JsonDocument.Parse(
3.         '{"ﷺ': 'صلى الله عليه وسلم', 'Ⓢ': '1', '4田': '4田'}");
```

**Example 2:** Consider the following lines of Python, as displayed by an editor which is unaware of the internal structure of the string passed to `re.sub`:

```
1. # Replace translations of "Google, Ltd.".
2. terms = re.sub(ר'גוגל', ?\s+מ[""]בע', "Google LLC", terms)
```

The reordering obscures the structure of the regular expression, so that it looks like it replaces the text `בע"מ גוגל` “Ltd. Google”. If the editor is capable of recognizing the string as a regular expression, it should instead display it in the following order; see also Example 1 of Section 3.1.4.

```
1. # Replace translations of "Google, Ltd.".
2. terms = re.sub(ר'מ[""]בע\s+, ?גוגל', "Google LLC", terms)
```

### 3.1.4 Ordering for Literal Text with Interspersed Syntax

Some languages, such as regular expression or markup languages, consist of literal text interspersed with language syntax. The same can be said of interpolated strings and strings containing escape sequences.

In that case, the syntax should not interfere with the displayed order of the literal text; instead, any syntactic elements should appear at the appropriate position within the text, without being influenced by it nor influencing it.

**Note:** Whereas the basic ordering requires only a lexical analysis, this requires a syntactic analysis: for instance, a group in a regular expression must be isolated as a whole. In many languages, this ordering must then be applied recursively: each alternative in a regular expression group is itself a regular expression.

Thus, the ordering of the regular expression `a[bc]d(e|f[g]h)i|j` proceeds as follows:

1. Apply the basic ordering to order the following atoms:
  - a. `a[bc]d(e|f[g]h)i`
  - b. `|`
  - c. `j`
2. Apply the ordering for literal text with interspersed syntax to `a[bc]d(e|f[g]h)i`, displaying it in the same order as `adi` with isolated syntactic elements `[bc]` and `(e|f[g]h)?`.
3. Apply the basic ordering to both `[bc]` and `(e|f[g]h)?`, whose atoms are respectively `[, b, c, ]` and `(, e, |, f[g]h, ), ?`.
4. Apply the ordering for literal text with interspersed syntax to `f[g]h`, displaying it in the same order as `fh` with an isolated `[g]`.
5. Apply the basic ordering to `[g]`.

In Examples 1 through 4, left-to-right atom direction is used.

**Example 1:** Consider the regular expression from Example 2 of Section 3.1.3:

```
בע" ]\s+מ" ? , גוגל
```

It matches strings such as `גוגל, בע"מ` (“Google, Ltd.”); however, its plain text appearance is misleading: it looks like it matches the reversed `גוגל, בע"מ` “Ltd. ,Google”. In this case, treating the syntax characters as atoms makes things worse:

```
מ" ]\s+בע" ? , גוגל
```

That rendering looks like it matches a nonsensical `גוגל, בע"מ` “d.Lt ,Google”. Instead, the subexpressions `, ? , \s+`, and `[""]` should be isolated, so that they display at the appropriate locations within the resulting text:

```
גוגל ? , \s+ בע" ]\s+ מ
```

**Example 2:** Consider the following C statement, prints a string that reads “kilobyte (kB)”, where an escaped ASCII quotation mark is used instead of a gershayim.

```
puts("ק" \ ב" ) ;
```

That rendering is misleading, because the escape sequence is reversed, so that it looks like there is an unescaped quotation mark. However, treating the escape sequence as an atom would be even worse:

```
puts(" ( \ ב" ) ק" ) ;
```

The string then reads “(Bk) kilobyte”. Instead the escape sequence should be isolated, so that it displays normally, but at the correct location:

```
puts(" ( \ ב" ) ק" ) ;
```

**Example 3:** Consider the following lines of JavaScript, which both cause a pop-up window to appear with a Persian translation of “Version 15,1 was released yesterday”:

1. `alert(`دیروز منتشر شد { [15, 1] } نسخه`);`
2. `n=[15, 1]; alert(`دیروز منتشر شد {n} نسخه`);`

That rendering is problematic; in the first case, the executable Javascript `[15, 1]` is displayed with right-to-left atom order even though the program uses left-to-right atom order; likewise the

placeholder syntax `{}` uses right-to-left atom order, so a reviewer could fail to identify it as a placeholder. In the second case, the placeholder is fine, but the Persian text is broken by it, so that it looks like it reads “Was released yesterday *n* version”. Instead, the placeholders should be isolated so that the statements display as follows:

1. `alert(`نسخه` ${[15, 1]} دیروز منتشر شد`);`
2. `n=[15, 1]; alert(`نسخه` ${n} دیروز منتشر شد`);`

**Note:** Isolating interspersed syntax as neutral generally works well for escaped neutral characters (such as escaped spaces or quotation marks), or for escaped line breaks. However, it can lead to unwanted display when the escaped characters have a strong or explicit bidirectional class. For instance, the following string literal would display as “Google تابعة لشركة YouTube” (“YouTube is a subsidiary of Google”), but, in the source code, it looks like “Google is a subsidiary of YouTube”.

```
"\u{202B}YouTube تابعة لشركة Google\u{202C}"
```

On the other hand, treating the escapes as if they had the bidirectional class of the characters for which they stand is technically difficult, and can lead to unexpected results when escapes are meant to represent a string in memory order; for instance, the escapes in the following string literal should not be displayed in right-to-left order, even though the text represented by it would be displayed with characters right-to-left.

```
"\N{ARABIC LETTER MEEM}\N{ARABIC LETTER SAD}\N{ARABIC LETTER REH}"
```

The use of the literal characters is a more reliable way to ensure that the source code display matches the display of the text. This can be combined with a “show invisibles” mode, as described in *Section 3.2.2, Suggested representations for directional formatting characters*.

Editors may wish to provide a way to see what a string looks like when all escape sequences therein are replaced by the characters for which they stand; such a feature would show the contents of the above two strings as “Google تابعة لشركة YouTube” and “مصر”, respectively.

Where a markup language specifies the paragraph direction for the bidirectional algorithm in some span of the text, that direction should be taken into account when displaying the text.

**Example 5:** Consider the HTML source for the following two paragraphs (note: the second paragraph translates to “YouTube is a subsidiary of Google”).

ETCO (إتكو) is a company in Oman

Google تابعة لشركة YouTube

If it is displayed using the basic ordering with left-to-right atom direction, that HTML would look as follows, which is misleading: the second paragraph looks like it reads “Google is a subsidiary of YouTube”.

1. `<p dir="ltr">ETCO (إتكو) is a company in Oman</p>`
2. `<p dir="rtl">YouTube تابعة لشركة Google</p>`

If instead right-to-left atom direction is used, it is the first paragraph whose display is misleading.

```
<p/>is a company in Oman (إتكو) ETCO<"ltr"=dir p> .1
<p/>Google تابعة لشركة YouTube<"rtl"=dir p> .2
```

Instead, the text within each element should be displayed according to the `dir` attribute, thus, with left-to-right atom direction,

3. `<p dir="ltr">ETCO (إتكو) is a company in Oman</p>`
4. `<p dir="rtl">Google تابعة لشركة YouTube</p>`

#### 3.1.4.1 Equivalent Isolate Insertion for the Ordering for Literal Text with Interspersed Syntax

This section describes how one would insert isolates into source code in order to have it appear in the right direction. The purpose of this is to establish a formal logical description of how text should be ordered. This does not mean that isolates should be inserted into a copy of the document for display. Instead higher-level protocols should be used to achieve the same display.

That is, the ordering for literal text with interspersed syntax can be formally described in terms of an equivalent insertion of explicit directional formatting characters, as in higher-level protocol HL3 of the Unicode Bidirectional Algorithm. That is, a document displayed according to this ordering must display in the same order as a document that is modified according to the procedure below and displayed according to the Basic Display Algorithm of Unicode Standard Annex #9, *Unicode Bidirectional Algorithm*, where each line of source code is treated as a paragraph.

**Note:** Actually inserting explicit directional formatting is not necessary to implement the basic ordering. In particular, when code is displayed using HTML, it is better to make use of the features of that language, as described in the implementation note in *Section 3.1.2, Basic Ordering*.

This formal specification refers to definitions from Unicode Standard Annex #9, *Unicode Bidirectional Algorithm* [UAX9], and makes use of the abbreviations defined in [Section 2, Directional Formatting Characters](#), in that annex.

- A. Let  $d$  be the atom direction, either LTR or RTL;
- B. Let  $d'$  be the paragraph direction, LTR, RTL, or FS (First Strong), if specified by the language, otherwise  $d'=d$ ;
- C. Define Text\_Isolate to be the code point LRI if  $d'=LTR$ , RLI if  $d'=RTL$ , or FSI if  $d'=FS$ ;
- D. Separate the text into lines and each line into runs of syntactic elements and literal text, in a language-dependent manner;
- E. For each line:
  - a. For each syntactic element on the line:
    - i. Apply the equivalent isolate insertion for the basic ordering to the syntactic element, using  $d$  as the atom direction;
  - b. Compute  $u$ , the number of isolate initiators in the text of the line that do not have a matching PDI within the text of the atom, as defined by [BD9](#);
  - c. Insert Text\_Isolate at the beginning of the line;
  - d. Insert  $u+1$  PDI characters at the end of the line;

## 3.2 Blank and Invisible Characters

Many source code editors provide options to make blank characters visible, such as representing horizontal tabulations by `→`, spaces by `·`, etc.

It is recommended that editors also provide an option to make visible any default ignorable code points (that is, code points with the `Default_Ignorable_Code_Point` property). These are invisible characters, which, while necessary and commonly used in text, can lead to confusion. However, even if these characters are

made visible, their normal effect on the text should be retained, as this can otherwise lead to misleading rendering.

### Example:

The following string literal, which reads “YouTube is a subsidiary of Google”, contains two invisible characters, as well as three spaces:

(1) `"Google تابعة لشركة YouTube"`

These blank and invisible characters could be made visible as follows:

(2) `"[RLE]Google.تابعة.شركة.YouTube[PDF]"`

However, when adding the markers that make them visible, their effect on the text should be retained; otherwise the string literal would appear as follows, which looks like “Google is a subsidiary of YouTube”, even though that literal represents a string which reads “YouTube is a subsidiary of Google”, as in (1).

(3) `"[RLE]YouTube.تابعة.شركة.Google[PDF]" # Misleading.`

Some of these invisible characters can be expected to occur frequently, or are part of the orthography of some languages. As a result, when they are made visible, their visual representation should be unobtrusive (similar to the use of `.` for space, rather than `[U+0020]`).

### 3.2.1 Suggested representations for joiner controls and variation selectors

The joiner controls (U+200C zero width non-joiner, U+200D zero width joiner) and the variation selectors (U+200C, U+200D, U+FE00..U+FE0F, U+E0100..U+E01EF) can occur within a word, and in particular within an identifier. Further, they can affect shaping; the insertion of a marker into the text stream would likewise affect shaping, possibly obscuring the effect of the character. When these characters are made visible, a nonspacing visual indication should be used.

Two examples are given for each suggested representation in this section; a first one where the invisible character is unexpected, and should therefore be made visible when showing invisible characters; and a second one where it is expected and has an effect, illustrating how the suggested representation preserves its effect on the text. If possible, the visual indication should be suppressed when the character is expected to have a visible effect; that is, in the second example, it is preferable to not indicate the presence of the invisible character at all.

*Review note: the “If possible...” is ICU homework.*

For the zero width non-joiner, the suggested representation is an overlaid vertical bar at the position of the non-joiner:

1. `procedure Up|date (version : Positive);`
2. `procedure به|روز (نسخه : Positive);`

*Editor’s note: in the HTML document, enclose the ZWNJ in a span with a border; here a spacing | was used.*

For the zero width joiner, the suggested representation is an outline around the extended grapheme cluster enclosing the joiner, or, if the joiner lies at an extended grapheme cluster boundary, around the extended grapheme clusters either side of the joiner:

1. `finland = Locale.IsoCountryCode.valueOf("FI");`
2. `ශ්‍රී ලංකා = Locale.IsoCountryCode.valueOf("LK");`

**Note:** In some cases, such as ශ්‍රී above, the zero width joiner merges the glyphs of the grapheme clusters either side of it (ශ්‍රී=ශ්+ZWJ+ඊ), so that there is no position at which a marker could meaningfully be inserted to indicate its presence; styling the two grapheme clusters differently, or attempting to insert a mark, would likely inhibit correct shaping, and mislead the users as to the actual text being displayed.

For variation selectors, the suggested representation is an outline around the extended grapheme cluster containing the variation selector.

1. `infix operator +`: AdditionPrecedence // VS-1 has no effect on +.
2. `infix operator ≍`: ComparisonPrecedence // VS-1 slants the = in ≍.

*Editor's note: in the HTML document, put a border around that instead of highlighting it with a solid color.*

**Industry example:** Visual Studio Code displays variation selectors as suggested.

### 3.2.2 Suggested representations for directional formatting characters

The implicit directional marks (U+061C, U+200E, U+200F) are nonspacing characters which can be inserted between tokens, either manually, or automatically by the procedure described in *Section 4.2, Conversion to Plain Text*. A lightweight representation should therefore be used; for instance, ▶ for left-to-right mark and ◀ for right-to-left and Arabic letter marks:

- `· · · עֵדוּן▶(15); · · // · Update · to · version · 15 · .`

In contrast, the explicit directional formatting characters are more rarely used, and need to be manipulated with care, as they are operations on a stack. A more prominent visual representation is therefore appropriate. However, the code point number is not a very readable representation. It is instead recommended to use the abbreviations defined in *Section 2, Directional Formatting Characters*, in Unicode Standard Annex #9, *Unicode Bidirectional Algorithm* [UAX9].

The markers indicating the presence of these characters should be directionally isolated, and should be inserted before the characters in the case of LRE, RLE, FSI, LRI, RLI, but after in the case of PDF and PDI.

**Example:** "[RLE]Google · تابعة · لشركة · YouTube[PDF]"

Further, it should be possible to selectively turn off these visual indications; in particular, the following levels are recommended:

- S1. Unconditionally show spaces/tabs and default ignorable code points everywhere;
- S2. Do not show spaces, but show default ignorable code points everywhere but in comments;
- S3. Do not show spaces, and only show default ignorable code points in strings;
- S4. Do not show spaces nor default ignorable code points anywhere.

The reason for level S3 is that some default ignorable code points may be inserted between lexical elements throughout the source code in order to preserve the basic ordering in editors that do not implement it. Such



default ignorable code points are not part of the semantics of the program, whereas those in strings are. See *Section 4.2, Conversion to Plain Text*.

**Example:** When writing comments in right-to-left scripts that refer to technical terms in left-to-right scripts, if the comment is displayed using left-to-right paragraph direction, such as when the code is displayed as left-to-right plain text (see *Section 4.2, Conversion to Plain Text*), it is necessary to use the explicit directional formatting characters in order for the text to be readable, as in the following Persian comment, which reads “The variable `message` is not null.”.

1. `// [RLE] متغیر message خالی نیست.`
2. `*message = "[RLE]Google تابعة لشركة YouTube[PDF]";`

The RLE is necessary; in its absence, the comment would appear as the mangled “.is not null / changeable `message`”:

```
// خالی نیست message متغیر.
```

Since mentioning technical terms is a frequent occurrence in programming language comments, and since comments, by virtue of not being executable, are not subject to spoofing concerns as long as their extent made recognizable by correct display, a programmer who writes comments in right-to-left scripts may want to suppress the `[RLE]` marker in line comments while retaining them elsewhere the source code, such as in the string, in order to be able to check that the message is well-formed (levels S2 or S3 above):

1. `// متغیر message خالی نیست.`
2. `*message = "[RLE]Google تابعة لشركة YouTube[PDF]";`

**Implementation note:** with rendering based on HTML and CSS, the suggested representations can be implemented as in the examples above, which use the following CSS:

1. `span.zwj-cluster {`
2. `border-top-style: solid;`
3. `border-bottom: 1px solid skyblue;`
4. `}`
5. `span.zwj-cluster::before {`
6. `content: "\A0";`
7. `position: absolute;`
8. `border-left: 1px solid skyblue;`
9. `}`
10. `span.zwj-cluster::after {`
11. `content: "\A0";`
12. `position: absolute;`
13. `border-left: 1px solid skyblue;`
14. `}`
- 15.
16. `span.zwnj::before {`
17. `content: "\A0";`
18. `position: absolute;`
19. `border-left: 1px solid skyblue;`
20. `}`
21. `span.lrm::before {`
22. `content: "‣";`
23. `color: skyblue;`
24. `unicode-bidi: isolate;`



```

25.  }
26.  span.rle::before {
27.    content: "[RLE]";
28.    color: skyblue;
29.    unicode-bidi: isolate;
30.  }
31.  span.pdf::after {
32.    content: "[PDF]";
33.    color: skyblue;
34.    unicode-bidi: isolate;
35.  }

```

The relevant characters are wrapped in spans with the corresponding classes, except that the class `zwj-cluster` is used to enclose grapheme clusters either side of the ZWJ:

```

1. <div><span class="zwj-cluster">f&zwj;i</span>nland</div>
2. <div>up<span class="zwnj">&zwnj;</span>date</div>
3. <div>עדכון<span class="lrm">&lrm;</span>(15);</div>
4. <div>"<span class="rle">&#x202B;</span>YouTube
5. تابعة لشركة
6. Google<span class="pdf">&#x202C;</span>"</div>

```

### 3.3 Confusables

Issues of confusability, whereby, for instance, two different identifiers look identical, cannot directly be addressed by fixing the display or by syntax highlighting. This is because contrary to display order and to the nature and extent of tokens, which can generally be handled with limited context, confusability is global; confusable identifiers may occur arbitrarily far apart in a file, or even in separate files.

Local solutions, such as highlighting individual characters that are confusable with ASCII, are ill-advised; they have unacceptable false positive rates when non-Latin scripts are used. As a result, users of these scripts would need to turn these diagnostics off; however, as described in *Section 1.2.1, Usability issues arising from lookalike glyphs*, these users are precisely the ones who are most likely to experience such issues.

Instead, the mechanisms described in *Section 4.1, Confusability Mitigation Diagnostics*, should be used. These can then be surfaced in display, for instance, using “squiggles” as for other warnings.

### 3.4 Syntax Highlighting

Many spoofing issues involve confusion over the extent of string literals and comments, so that executable text looks non-executable, or vice-versa. Syntax highlighting can mitigate such issues, by making the extent of such tokens visually evident. Note that syntax highlighting based on color can be an accessibility issue; if color is used, it is advisable to change luminosity and saturation as well as hue. However, syntax highlighting need not be solely based on color; throughout this document, code snippets have comments in italics and reserved words in bold, in addition to coloring for various kinds of tokens. However, readable stylistic alternatives such as bold and italics do not exist in all writing systems, and are limited in number even in the Latin script.

**Note:** Key to the syntax highlighting used in this document:

- **reserved words**
- *comment markers*
- *comment contents*

- **string literals**
- **hints** for blank and invisible characters
- **markup, regex character classes**
- **escape sequences, regex operators**
- in some examples: **types**

## 4. Tooling and diagnostics

Not all issues can be addressed by improving the display of source code. For instance, the words KAI (Greek for AND) and KAI (in Latin script) are expected to display identically; however, having both as identifiers in the same scope is a problem. Compiler and linters warnings are a more appropriate tool to address such issues. Note that the resulting diagnostics may then be visually surfaced by an editor, *e.g.*, as squiggles.

Further, source code is sometimes displayed as plain text in environments that are unaware of its lexical structure, such as in compiler diagnostics or diffs shown in a terminal, patches sent by email, etc. These environments cannot be expected to implement the ordering described in *Section 3.1, Bidirectional Ordering*; instead, the source code itself should be modified, *e.g.*, by a pretty-printer, to minimize issues when it is displayed as plain text.

### 4.1 Confusability Mitigation Diagnostics

The diagnostics defined in this section are recommended for use in linters or other sources of editor squiggles. Some of them may need to be turned off in specialized applications, such as scientific computing. However, they are designed to be an unobtrusive default while drastically reducing the possibility of spoofing attacks and the usability issues resulting from visually identical identifiers.

#### 4.1.1 Confusable Detection

The most effective remedy to issues of identifier spoofing is the use of confusable detection. In a source code document with LTR atom order, it is recommended to warn the user when an identifier is LTR-confusable with some other relevant identifier or with a reserved word of the language, where LTR-confusability is defined in *Section 4, Confusable Detection*, in Unicode Technical Standard #39, *Unicode Security Mechanisms [UTS39]*. When RTL atom order is used, RTL-confusability should be used.

**Note:** An implementation should diagnose only *distinct* confusable identifiers; identifiers that are identical, or that are equivalent under any normalization or case equivalence used by the language, should not be flagged.

*Review note: It would make sense to look for the `bidiSkeleta` of syntactic lexical elements (operators, etc.) in the `bidiSkeleta` of identifiers; however, this requires a narrower definition of confusability, lest we prohibit the letters I and l in any languages where | has syntactic meaning. It could also be useful to have such a narrower definition to avoid warning about the confusability of, *e.g.*, I and l, which are typically distinct in fonts used to display source code.*

The set of “relevant identifiers” to look for depends on the language and the capabilities of the tool implementing this diagnostic.

For instance, consider an editor that is only aware of the lexical structure of a programming language, but cannot resolve dependencies nor determine scopes: that editor could warn on the coexistence of distinct confusable identifiers in the same file (type I in the example below). If that editor is also aware of a workspace

of relevant files, it could warn on the coexistence of distinct confusable identifiers anywhere within those files (type II).

On the other hand, a compiler for that programming language, knowing the visibility rules of the language, could warn when an identifier is confusable with a semantically distinct visible name; this would allow it to diagnose confusabilities with names in other libraries, and it would avoid false positive where local variables in unrelated places have confusable names (type III).

**Example:** Consider the following C program split across two files:

bad\_stdlib.c:

```

1.  #include <ctype.h>
2.  #include <math.h>
3.  // The name of both functions is entirely in Cyrillic.
4.  // The argument in the Latin script for both.
5.  bool isspace(char32_t c) {
6.      return c == U' ';
7.  }
8.  double exp(double x) {
9.      return 1 + x;
10. }

```

main.c:

```

1.  #include <ctype.h>
2.  int main(int argc, char* argv[]) {
3.      // The name of this variable is a Cyrillic s.
4.      char* c = argv[1];
5.      if (isspace(*c)) {
6.          puts("argv[1] starts with a space");
7.      }
8.      // This is a Latin c.
9.      char c = getchar();
10.     return c != 'Y';
11. }

```

A type I diagnostic will flag the confusability between the variables `c` (l. 4) and `c` (l. 9) in `main.c`.

A type II diagnostic will also flag that; in addition, it will flag `isspace` in `bad_stdlib.c` and `isspace` in `main.c`, because they are confusable with each other; it will also flag `c` in `bad_stdlib.c`, because it is confusable with `c` in `main.c`.

A type III diagnostic will flag `c` (l. 4) and `c` (l. 9) in `main.c`; it will flag both `isspace` and `exp` in `bad_stdlib.c`, because they are confusable with identifiers included at lines 1 and 2; it will not flag `c` in `bad_stdlib.c` nor `isspace` in `main.c`, because their confusables are not visible.

The type III diagnostic is most complex to implement, but it avoids false positives; on the other hand, such false positives are still likely to be mistakes (or spoofing attempts) in practice. The inability of the type II diagnostic to see other libraries is mitigated by the fact that using a library will cause its identifiers to appear in the code, as for `isspace` in the example, so that this is unlikely to be a problem in a real code base.

**Industry example:** The Rust compiler implements type II confusable detection, by flagging any confusable identifiers within a crate, with the exception that it uses confusability rather than LTR-confusability (so that it would fail to diagnose the confusability of the identifiers `a18` and `a1᠘`).

In order to mitigate usability issues arising from confusability, such as the ones described in *Section 1.2.1, Usability issues arising from lookalike glyphs*, it is important to detect confusables early, for instance, in the editor, or at the latest while compiling. If this check is only performed after successful compilation, such as in continuous integration on pull requests, the usability issues are not mitigated, as the user will be faced with mysterious compilation errors. When confusable checks are not applied prior to successful compilation, implementations should make use of other mechanisms to alleviate usability issues, such as mixed-script detection in identifier chunks; see *Section 4.1.2, Mixed-Script Detection*.

#### 4.1.2 Mixed-Script Detection

*Editor's note: a detailed rationale for this section is provided in document [L2/22-231](#).*

Mixed-script detection, as described in Unicode Technical Note #39, *Unicode Security Mechanisms*, should not directly be applied to computer language identifiers; indeed, it is often expected to mix scripts in these identifiers, because they may refer to technical terms in a different script than the one used for the bulk of the program. For instance, a Russian HTTP server may use the identifier `HTTP3анрoс` (*HTTPRequest*).


Instead, identifiers should be subdivided into visually recognizable *chunks* based either on both case and punctuation; one can then ensure that these chunks are either single-script, or are *visibly mixed-script* (in which case the reader is not misled about the string being single-script).

**Note:** While mixed-script detection reduces the surface for spoofing attacks, it cannot completely prevent them; identifiers such as `isspace` or `exp` are single-script (Cyrillic), but are confusable with ASCII identifiers from the standard libraries of multiple languages, *viz* `isspace` and `exp`.


Confusable detection should be used to more systematically deal with spoofing issues; see *Section 4.1.1, Confusable Detection*.

##### 4.1.2.1 Identifier chunks


An *identifier word boundary* is defined as any of the following:

 a *CamelBoundary*, defined as the position after the group in a sequence matching the following regular expression:

$$( [\p{Ll}] [\p{Lt}]-\p{Grek}] ] [\p{Mn}\p{Me}]^* ) [\p{Lu}\p{Lt}],$$

 a *HATBoundary*, defined as as the position before a sequence matching the following regular expression:

$$[\p{Lu}\p{Lt}] [\p{Mn}\p{Me}]^* \p{Ll} \quad | \quad [\p{Lt}]-\p{Grek}].$$

 a *snake\_boundary*, defined as the positions either side of a Punctuation character which is not an Other\_Punctuation character, *i.e.*, either side of a sequence matching  $[\p{P}]-\p{Po}]$ .

An identifier splits into *identifier chunks* delimited at identifier word boundaries. Note that multiple kinds of boundaries can coincide.

Examples of the separation into identifier chunks are given in the table below; emoji mark the various boundaries.

Identifier	Identifier chunks	Notes
dromedaryCamel	dromedary 🐪 📦 Camel	🐪
snakeELEPHANTSnake	snake 🐍 ELEPHANT 📦 Snake	🌹 📦
TypeII	📦 Type 🐪 II	
OCam1	O 📦 Cam1	The HATBoundary is designed to accommodate the common practice of keeping acronyms in upper case in a CamelCase identifier.
HTTP3anpoc	HTTP 📦 3anpoc	
UAX9ClauseHL4	UAX9 📦 Clause 🐪 HL4	
LOUD_SNAKE	LOUD 🐍 _ 🐍 SNAKE	
Fancy_Snake	📦 Fancy 🐍 _ 🐍 📦 Snake	
snake-kebab	snake 🐍 - 🐍 kebab	Assuming a profile allowing hyphen-minus in identifiers.
Paral·lel	📦 Paral·lel	Other_Punctuation does not separate words; indeed it is used within words in Catalan.
microB	micro 🐪 B	
microb	microb	The sequence $\{L1\}\{Lo\}$ is not a CamelBoundary, and should not be one: this Other_Letter is confusable with a Lowercase Letter.
HTTPसर्वर	HTTPसर्वर	Here a visible word boundary is not detected, but the resulting multi-word chunk is visibly mixed-script.

#### 4.1.2.2 Mixed-script detection in identifier chunks

An identifier chunk  $X$  is *confusing* if both of the following are true:

1.  $X$  has a restriction level greater than Highly Restrictive, as defined in [UTS #39, section 5.2](#);
2. There exists a string  $Y$  such that all of the following are true:
  - a.  $Y$  is confusable with  $X$ ;
  - b. The resolved script set of  $Y$  is neither  $\emptyset$  nor ALL;
  - c. The resolved script set of  $Y$  is a subset of the union of the Script\_Extensions of the characters of  $X$ .
  - d.  $Y$  is in the General Security Profile for Identifiers.

**Note:** Criteria a through c of condition 2 are similar to “ $X$  has a [whole-script confusable](#) in the union of its Script\_Extensions”, but do not require  $X$  to be single-script.

An identifier chunk for which condition 1 holds but condition 2 does not hold is called *visibly mixed-script*.

**Note:** Visibly mixed-script identifier chunks are not confusing.

An implementation implementing mixed-script detection in identifier chunks shall diagnose confusing identifier chunks in identifier tokens.

Examples of confusing and non-confusing mixed-script identifier chunks are given in the following table; all have a restriction level greater than Highly Restrictive.

Строка	Confusing, confusable with all-Cyrillic Строка.
Δτ	Visibly mixed-script, τ is not confusable with a Greek letter, nor is Δ confusable with a Latin letter.
μῆωω	Visibly mixed-script, μ is not confusable with a Cyrillic letter nor with a Latin letter.
ΜΙΚΡΑ	Confusing, confusable with all-Greek ΜΙΚΡΑ and all-Latin ΜΙΚΡΑ.
HTTPसर्वर	Visibly mixed-script, H is not confusable with a Devanagari letter, nor is ऺ confusable with a Latin letter.
microb	Confusing, confusable with all-Latin microb.

#### 4.1.3 General Security Profile

As described in *Section 6.1, Confusables Data Collection*, of Unicode Technical Standard #39, *Unicode Security Mechanisms* [UTS39], the entirety of Unicode is not in scope for thorough confusables data collection. In order to ensure that confusable detection is effective, implementations should provide a mechanism to warn about identifiers that are not in the General Security Profile for identifiers, as defined in [Section 3.1, General Security Profile for Identifiers](#), in the same specification.

It should be possible to turn off this diagnostic independently from confusable detection: while it may be less comprehensive, data on confusables exists for characters outside the General Security Profile, so that confusable detection is still beneficial when using such characters.

*Review note: This is too strict. We should at least include default ignorables as a modification of the profile per conformance clause C1. We can also get rid of the context checks for ZWJ and ZWNJ, since we have confusable detection which is aware of default ignorables.*

*However, what we really want is a property which defines a set on which we have good data about confusables, regardless of usage considerations, as characters whose Identifier\_Type is Technical, Not\_NFKC, etc. can make sense in programming language identifiers.*

#### 4.1.4 Multiple visual forms

In languages where the formats used for displaying and comparing identifiers are different, as described in *Section 1.3, Display Format*, in Unicode Standard Annex #31, *Identifiers and Syntax* [UAX31], this can lead to confusion or potential for spoofing. For instance, consider the following snippet of Ada:

1.       **package** Matrices\_3\_By\_3 **is new** Matrices (3, 3);
2.       **subtype** so3 **is** Matrices\_3\_By\_3.Skew\_Symmetric\_Matrix;
3.       **begin**

```

4.     declare
5.         subtype SO3 is Matrices_3_By_3.Special_Orthogonal_Matrix;
6.         X : so3;

```

It looks like X is being declared as a skew-symmetric matrix, but it is actually a special orthogonal matrix, because the identifiers so3 and SO3 are equivalent.

The situation is similar if identifiers are treated as equivalent under Normalization Form KC, as in the following Python program, which looks like it returns a vector, but actually returns a scalar, as  $r$  and  $r$  are the same variable.

```

1.     def GravitationalAcceleration(self, position):
2.         Gm = self.gravitational_parameter
3.         r = (position - self.position)
4.         r = r.Norm()
5.         return r * Gm / (r ** 3)

```

One possibility is to warn when, within a given code base, different references to the same entity use different forms under Normalization Form C (but that are equivalent as identifiers).

#### 4.1.5 Extent of block comments

Many spoofing issues involve confusion over the extent of string literals and comments, so that executable text looks non-executable, or vice-versa. As discussed in *Section 3.4, Syntax Highlighting*, while syntax highlighting is a very effective way to mitigate such issues, it has limitations, as the number of clearly distinguishable styles is ultimately limited, especially if very few characters are being styled. It is further limited by accessibility considerations.

**Example:** To a colorblind user, the extent of this Java comment may be unclear; as italics are not commonly used in Hebrew, they cannot reliably be used to help with the identification of the extent of the comment.

```
/*...מוקדמים מוקדמים בודק תנאים */ הודעה.requireNonNullNonNull(); /* בסדר. */
```

The issue here is that the comment contains a right-to-left `/*`, which looks like the `*/` that would terminate the comment.

A similar issue can occur with characters that look like the characters in the comment delimiter:

```
/* Check preconditions... */ message.requireNonNullNonNull(); /* OK. */
```

In order to avert such issues, it is useful to issue a warning if, for any comment content atom A in a block comment whose ending delimiter is D, the string *bidiSkeleton*(FS, A) contains *skeleton*(D) as a substring, where *bidiSkeleton* and *skeleton* are defined in *Section 4, Confusable Detection*, in Unicode Technical Standard #39, *Unicode Security Mechanisms* [UTS39].

**Note:** A similar approach is not recommended for string literals; this is because legitimate string literals often contain “smart” quotation marks, which are confusable with the delimiters, so that warning about their presence would lead to unacceptable false positives. In contrast, comments can reasonably be expected not to contain lookalikes of `*/`, `*`), `-->`, `#>`, `-- ]`], etc.

*Review note: We do not want to recommend a diagnostic that would warn about the possibility of confusion regarding "The “extent” of this string".*



*On the other hand, we would like to warn about the following:*

```
s = "Hello " + "world";
```

*Besides strings that look like they terminate, but don't, there is also the inverse problem, as in the following C statement:*

```
printf("Quotes may be \"dumb\", \"smart\", fullwidth, etc.");
```

*A proper solution would likely have to involve a syntactic analysis, rather than merely a lexical one; even then, it is not clear what such a solution would look like.*

#### 4.1.6 Directional formatting characters

Implementations should not prohibit the use of the directional formatting characters; they are useful in ensuring the correct display of bidirectional text, as illustrated in this document. However, in order to avoid disruption when the code is displayed as plain text, it may be useful to warn when the effect of the *explicit* directional formatting character extends across atoms. The algorithm described in *Section 4.2, Conversion to Plain Text*, includes such a diagnostic.

#### 4.2 Conversion to Plain Text

The following algorithm is a conversion to plain text in the sense of [Section 6.5, Conversion to Plain Text](#), in Unicode Standard Annex #9, *Unicode Bidirectional Algorithm* [UAX9]. It is suitable for languages that allow implicit directional marks between lexical elements and in any other appropriate locations, as described in *Section 2.2, Whitespace and Syntax*.

The algorithm is idempotent. It transforms a source code file into one that has the same semantics and the same visual appearance when displayed according to *Section 3.1, Bidirectional ordering*. In addition, if no errors are emitted, the resulting source code file is correctly ordered when displayed as plain text according to the Unicode Bidirectional Algorithm, where each line of code is treated as a left-to-right paragraph, with the following exceptions:

1. The rules from *Section 3.1.3, Embedded languages*, are not applied.
2. The rules from *Section 3.1.4, Ordering for Literal Text with Interspersed Syntax*, are not applied.
3. The display may be incorrect in edge cases involving strings delimited by brackets, as described in *Section 4.2.1, Unpaired Brackets*.

If embedded languages also allow for the insertion of implicit directional marks, the conversion to plain text could be applied to relevant string literals. Markup, escapes, and interpolated strings cannot be handled by conversion to plain text.

**Note:** It is possible to achieve the same effect by inserting fewer implicit directional marks, by looking ahead on the line for strongly directional characters. However, the algorithm defined in this section attempts to minimize such spooky action at a distance, in order to reduce the potential for confusion if the source code is edited as plain text. For instance, no left-to-right mark is needed in the statement A, but one is inserted by the algorithm, producing B:

- A. תו := X;
- B. תו ▶ := X;



However, if the author notices an off-by-one error and edits that statement in a plain text editor, the text is improperly displayed unless that left-to-right mark is present:

- A.  $\underline{1} \cdot = : \cdot \text{תו} \cdot + \cdot X;$
- B.  $\text{תו} \cdot \cdot = : \cdot \underline{1} \cdot + \cdot X;$

The algorithm inserts LRM as soon as possible after an atom whose last strong direction could be right-to-left; this forces the left-to-right ordering of atoms, and prevents earlier atoms from influencing the ordering of subsequent ones. The algorithm also inserts FSI at the beginning of any comment whose first strong direction could be right-to-left, and terminates any isolates and embeddings in comments by inserting PDI and PDF, so that the effect of these explicit formatting characters does not cross atom boundaries.

**Note:** The PDI and PDF characters are not inserted in end of line comments, as their effect stops at the end of the paragraph, and having them in the source code could lead to unexpected editing behavior if text is appended after a trailing PDI or PDF.

The algorithm is as follows. It refers to definitions from Unicode Standard Annex #9, *Unicode Bidirectional Algorithm* [UAX9], and makes use of the abbreviations defined in [Section 2, Directional Formatting Characters](#), in that annex.

- A. Define the boolean variable Needs\_LRM, initialized to False;
- B. Separate the text into lines and each line into atoms, as described in Section 3.1.1, in a language-dependent manner;
- C. For each line:
  - a. For each atom on the line:
    - i. If the atom is a whitespace atom, remove any instances of LRM and RLM from that atom.
    - ii. If Needs\_LRM is True:
      - 1. If the language allows for the insertion of an implicit directional mark before the atom without changing the meaning of the program:
        - a. Insert LRM before the atom;
        - b. Set Needs\_LRM to False;
      - 2. Otherwise (for instance, if the position before the atom is within a string literal):
        - a. Look for the first character in the atom whose Bidi\_Class property is one of L, R, AL, EN, AN, LRE, RLE, LRI, RLI, or FSI, if any;
        - b. If that character exists and its Bidi\_Class property is not L:
          - i. Emit an error: the line cannot be converted to plain text by this algorithm;
    - iii. If the atom is a comment content atom:
      - 1. If the atom does not start with the character FSI:
        - a. Look for the first character in the atom whose Bidi\_Class property is one of L, R, AL, LRE, RLE, LRI, RLI, or FSI, if any;
        - b. If that character exists and its Bidi\_Class property is not L:
          - i. Prepend FSI to the atom;
      - 2. If the atom is followed by a code point that does not have Bidi\_Class B:
        - a. Insert a sequence of PDI characters after the atom as needed to close any isolate initiators in the atom that do not have a matching PDI, as defined in [BD9](#).
        - b. Insert a sequence of PDF characters after those PDI characters as needed to close any embedding initiators in the atom that do not lie between an isolate

- initiator and its matching PDI, and do not have a matching PDF, as defined in [BD11](#).
- iv. If the atom is followed by a code point that does not have Bidi\_Class other than B:
    1. If the atom has any isolate initiators that do not have a matching PDI, as defined in [BD9](#), emit a diagnostic: the line cannot be converted to plain text by this algorithm;
    2. If the atom has any embedding initiators that do not lie between an isolate initiator and its matching PDI, and do not have a matching PDF, as defined in [BD11](#), emit a diagnostic: the line cannot be converted to plain text by this algorithm;
    3. Look for the last character in the atom whose Bidi\_Class property is one of L, R, AL, PDF, or PDI, if any;
    4. If that character exists and its Bidi\_Class property is not L:
      - a. Set Needs\_LRM to True;
  - b. If the line is terminated by a character with Bidi\_Class B:
    - i. Set Needs\_LRM to False;

**Note:** Conversion to plain text is only provided for left-to-right plain text; this is because numeric atoms must have left-to-right embedding direction, which requires the insertion of embeddings or isolates. This algorithm does not insert such characters except in comments.

It is possible to construct a similar conversion to right-to-left plain text in a programming language whose numeric literals satisfy the following regular expression, which uses the syntax of Unicode Technical Standard #18, *Unicode Regular Expressions* [UTS18]:

$$\begin{aligned}
 &(\backslash p\{bc=L\} \\
 &|\backslash p\{bc=EN\} \\
 &|\backslash p\{bc=EN\}\backslash p\{bc=ET\} \\
 &|\backslash p\{bc=ET\}\backslash p\{bc=EN\} \\
 &|\backslash p\{bc=EN\}\backslash p\{bc=CS\}\backslash p\{bc=EN\})+
 \end{aligned}$$

Such a conversion needs to insert RLM rather than LRM, and to look for R or AL, rather than L, in steps C.b.ii.2.b, C.b.iii.1.b, and C.b.iv.4.

**Implementation note:** In step B of this algorithm, an implementation only needs to correctly identify the extent of those atoms that can contain characters with bidi classes R or AL, or explicit directional formatting characters, and to correctly characterize the first opportunity to insert an implicit directional mark after such atoms. In practice, this generally means correctly lexing for comments, string literals, and identifiers, assuming that implicit directional marks may be inserted after these tokens. In a language that conforms to requirement UAX31-R3b, this allows for simpler and more future-proof treatment of identifiers, whereby any sequence of non-syntax, non-whitespace characters that is not part of a comment or string is treated as a possible identifier for the purposes of this algorithm.

#### 4.2.1 Unpaired brackets

If a language uses atoms that contain closing brackets before which implicit delimiters cannot be inserted without changing the meaning of the program, such as string delimiters that use parentheses, the source code converted by the preceding algorithm may be improperly displayed as plain text even though no error is omitted. This occurs when an earlier atom (such as the contents of the string) has an unmatched opening parenthesis in an established right-to-left context which matches the one in the delimiter.

**Example:** Consider the following C++ string literal, which contains an unpaired opening parenthesis in a right-to-left context; the opening delimiter is `R"(",` the closing delimiter is `)"`, the contents are `"ב)א`:

```
R"("ב)א"
```

Its plain text display is as follows, which makes it look unterminated:

```
R"("ב)א"
```

These situations cannot be resolved by inserting left-to-right marks; however, implementations may wish to signal an error in these cases. This can be done by applying the Unicode bidirectional algorithm to each line after conversion to plain text, and by checking that any bracket pairs set to R in step [N0](#) lie in the same atom.

### 4.3 Identifier Styles

Many linters enforce case conventions, such as having compile time constants in upper case with words separated by low line, public names with the first letter of each word capitalized and no word separator (CamelCase), etc. Generalizing these diagnostics outside of ASCII may not always be obvious; in particular, the generalized diagnostics should not prevent the use of unicameral scripts. This section defines a mechanism which may be used to generalize such style checks while avoiding these pitfalls.

The basic idea is to disallow undesired categories of characters, instead of only allowing desired categories. For instance, to check that an identifier is in lowercase with words separated by low line, the uppercase and titlecase letters are forbidden, instead of allowing only lowercase and underscores (see `small_snake` below).

This section uses the regular expression syntax defined in [Unicode Technical Standard #18, Unicode Regular Expressions, version 23](#) [UTS18].

*Editor's note: a detailed rationale for this section is provided in document [L2/22-232](#).*

An implementation claiming to implement Unicode identifier styles shall emit some of the diagnostics defined below.

1. `BactrianCamel`:  
A diagnostic shall be emitted if an identifier matches the following regular expression:  
`^\p{Ll} | \p{LC}[\p{Mn}\p{Me}]* \p{Pc} \p{LC}`
2. `dromedaryCamel`:  
A diagnostic shall be emitted if an identifier matches the following regular expression:  
`^\p{Lu}\p{Lt} | \p{LC}[\p{Mn}\p{Me}]* \p{Pc} \p{LC}`
3. `small_snake`:  
A diagnostic shall be emitted if an identifier matches the following regular expression:  
`[\p{Lu}\p{Lt}]`
4. `Title_Snake`:  
A diagnostic shall be emitted if an identifier matches the following regular expression:  
`( ^ | \p{Pc} ) \p{Ll}`
5. `CAPITAL_SNAKE`: A diagnostic shall be emitted if an identifier, once normalized under Normalization Form C, matches the following regular expression:  
`[\p{Ll} \p{Lt}]`

Alternatively, it shall declare a profile, and define the situations in which the aforementioned diagnostics are suppressed and the additional situations in which they are emitted.

**Examples:**

An implementation could implement the BactrianCamel diagnosis with a profile that additionally prohibits  $(\backslash\mathbf{Lu}[\backslash\mathbf{Mn}\backslash\mathbf{Me}]^*)\{4\}$  (four uppercase letters in a row).

An implementation could implement the Title\_Snake diagnostic with a profile that allows lowercase after a Connector Punctuation (allowing Proud\_snake\_case).

An implementation which meets requirement UAX31-R1 with a profile adding the hyphen-minus (-) to Continue could implement the various diagnostics with a profile that replaces  $\backslash\mathbf{Pc}$  in the above regular expressions by  $[\backslash\mathbf{Pc}\backslash\mathbf{Pd}]$ , treating the hyphen-minus like the low line (allowing “kebab-case”).

## 5. Reference Implementations

*Review Note: As for other Unicode algorithms, such as the Unicode Bidirectional Algorithm, reference implementations will be provided to illustrate some of the algorithms defined in this specification. A brief exposition of these implementations will be provided in this section.*

---

*Editor's note: The members of the SCWG, as well as people acknowledged in separate rationale documents, should be acknowledged in the appropriate section of each relevant technical report.*