L2/23-098

Technical Reports

Draft Unicode® Technical Standard #55

UNICODE SOURCE CODE HANDLING

Version	1 (draft 1)
Editors	Robin Leroy (eggrobin@unicode.org), Mark Davis (mark@unicode.org)
Date	2023-01-27
This Version	https://www.unicode.org/reports/tr55/tr55-2.html
Previous Version	https://www.unicode.org/reports/tr55/tr55-1.html
Latest Version	https://www.unicode.org/reports/tr55/
Latest Proposed Update	https://www.unicode.org/reports/tr55/proposed.html
Revision	2

Summary

Because Unicode contains such a large number of characters and incorporates the varied writing systems of the world, usability and security issues can arise from improper handling of Unicode program text. This document provides guidance for programming language designers and programming environment developers, and specifies mechanisms to alleviate those issues.

Status

*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time. Publication does not imply endorsement by the Unicode Consortium. This is not a stable document; it is inappropriate to cite this document as other than a work in progress.*

> ***A Unicode Technical Standard (UTS)** is an independent specification. Conformance to the Unicode Standard does not imply conformance to any UTS.*

Please submit corrigenda and other comments with the online reporting form [Feedback]. Related information that is useful in understanding this document is found in the References. For the latest version of the Unicode Standard, see [Unicode]. For a list of current Unicode Technical Reports, see [Reports]. For more information about versions of the Unicode Standard, see [Versions].

Contents

1 Introduction
- 1.1 Terminology and Notation
- 1.2 Source Code Spoofing
  - 1.2.1 Line Break Spoofing
  - 1.2.2 Spoofing using lookalike glyphs
  - 1.2.3 Spoofing using bidirectional reordering
- 1.3 Usability Issues
  - 1.3.1 Usability issues arising from lookalike glyphs
  - 1.3.2 Usability issues arising from bidirectional reordering

2 Conformance

3 Computer Language Specifications
- 3.1 Identifiers
  - 3.1.1 Normalization and Case
  - 3.1.2 Semantics Based on Case
- 3.2 Whitespace and Syntax
- 3.3 Language Evolution

3.3.1 Changing Identifier Definitions
3.3.2 Changing Normalization and Case
4 Source Code Display
4.1 Bidirectional Ordering
4.1.1 Atoms
4.1.2 Basic Ordering
4.1.2.1 Equivalent Isolate Insertion for the Basic Ordering
4.1.3 Nested Languages
4.1.4 Ordering for Literal Text with Interspersed Syntax
4.1.4.1 Equivalent Isolate Insertion for the Ordering for Literal Text with Interespersed Syntax
4.2 Blank and Invisible Characters
4.2.1 Suggested representations for joiner controls and variation selectors
4.2.2 Suggested representations for directional formatting characters
4.2.3 Suggested Levels for “Show Hidden” Modes
4.3 Confusables
4.4 Syntax Highlighting
5 Tooling and Diagnostics
5.1 Confusability Mitigation Diagnostics
5.1.1 Confusable Detection
5.1.2 Mixed-Script Detection
5.1.2.1 Identifier Chunks
5.1.2.2 Mixed-Script Detection in Identifier Chunks
5.1.3 General Security Profile
5.1.4 Multiple Visual Forms
5.1.5 Extent of Block Comments
5.1.6 Directional Formatting Characters
5.2 Conversion to Plain Text
5.2.1 Unpaired Brackets
5.3 Identifier Styles
6 Reference Implementations
References
Acknowledgements
Modifications

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 3, Computer Language Specifications*, is aimed at language designers; it provides recommendations on the lexical structure, syntax, and semantics of computer languages. *Section 4, Source Code Display*, is aimed at the developers of source code editors and review tools; it specifies appropriate behavior for source code display. *Section 5, 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.

While the normative material for computer language specifications is part of the Unicode Standard, in *Unicode Standard Annex #31, Unicode Identifiers and Syntax* [UAX31], the algorithms specific to the display of source code or to higher-level diagnostics are specified in this document. *Section 2, Conformance* describes the corresponding conformance requirements.

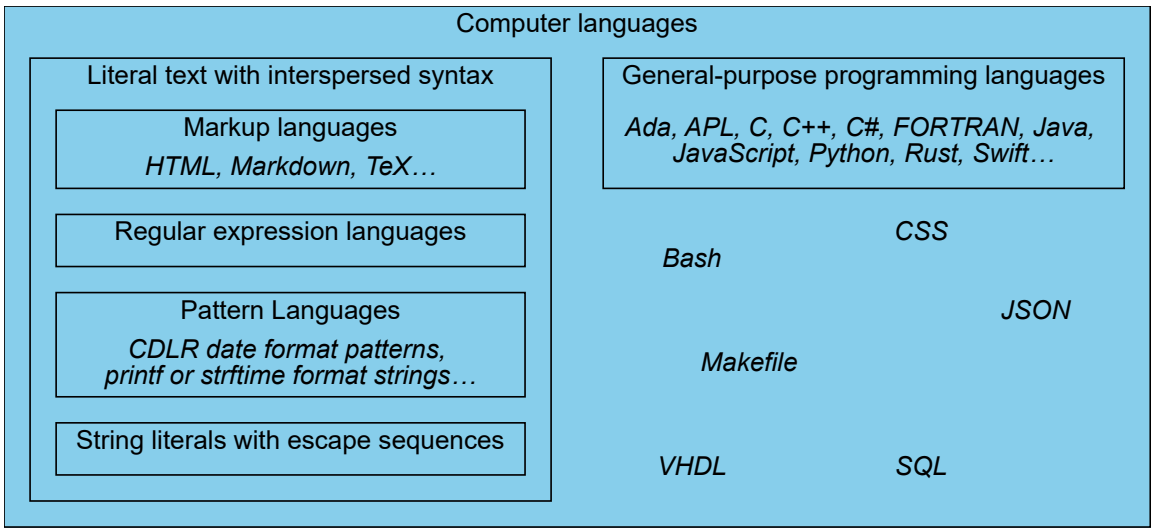
Note: While, for the sake of brevity, many of the examples in this document make use of non-ASCII identifiers, most of the issues described here apply even if non-ASCII characters are confined to strings and comments. Further, some of the remedies require allowing specific non-ASCII characters between lexical elements; see *Section 3.2, Whitespace and Syntax*.

1.1 Terminology and Notation

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 3, Computer Language Specifications*, apply only to general-purpose programming languages, and the specifications in *Section 4, 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 document.

Figure 1. Classification of computer languages used in this document



For the sake of readability, syntax highlighting is used in the code examples throughout this document. The following conventions are used:

Lexical element	Style
Comment	*italic green*
Keyword	**bold blue**
String or character literal	red
Regular expression character class	blue

1.2 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.

1.2.1 Line Break Spoofing

The Unicode Standard encompasses multiple representations of the New Line Function (NLF). These are described in *Section 5.8, Newline Guidelines*, in [Unicode], 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:

```
// Check preconditions.
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).

```
-- Here we must not yet NLreturn null;
-- 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 [Unicode]), 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 3.2, 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.

1.2.2 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 A, A, and A, Devanagari को (kō) and the “do not use” sequence को (*kāe), 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 `i` 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 3.1.1, Normalization and Case*. In order to address other cases, programming language tools should implement the mitigations described in *Section 5.1, Confusability Mitigation Diagnostics*.

1.2.3 Spoofing using bidirectional reordering

The Unicode Bidirectional Algorithm, defined in *Unicode Standard Annex #9, Unicode Bidirectional Algorithm* [UAX9], 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 *Section 2.2, Unicode Design Principles*, and conformance requirement C12 under *Bidirectional Text* in *Section 3.2, Conformance Requirements*, in [Unicode].

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:

The following statement, written in C++98 or later, looks like it prints “encountered 0 errors” if the variable `errors` is equal to 0, and prints “encountered errors” otherwise.

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

In fact, it does the reverse, because the seemingly empty string contains a right-to-left mark.

The following loop, written in Ada 2005 or later, looks like a loop over the Hebrew alphabet, from alef (א) to tav (ת).

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

It is actually dead code (looping over the empty range from tav to alef), because the range appears left-to-right.

The following statement of Rust 1.9 or later looks like a right shift by eight bits.

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

It is a left shift by eight bits: the operator `<<` is surrounded by right-to-left marks.

The solution is not to forbid the directional formatting characters; indeed the Ada example above does not use these. This document instead makes two recommendations.

First, source code editors should display source code according to its lexical structure, as described in *Section 4.1, Bidirectional Ordering*.

Second, computer languages should allow for the insertion of directional formatting characters as described in *Section 3.2, Whitespace and Syntax*, and implementers should provide tools that automatically remove spurious directional formatting characters, and insert the correct ones, as described in *Section 5.2, 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.

1.3 Usability Issues

The same issues described in *Section 1.2, 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.

1.3.1 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 `HTTPОтвет` (*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 HTTPO~~twe~~
```

```
class HTTPOтвет {
```

Trying to refer to `HTTPOтвет` will lead to a compilation error (because it is actually declared as `HTTPOтвет`, 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 1.2.2, Spoofing using lookalike glyphs*.

1.3.2 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.

Example: Consider the following line of C#:

```
Console.WriteLine("Error: {0} {1}", message, this);
```

If the string and the identifier `message` are translated to Hebrew, and the resulting code displayed without taking its structure into account, the text span starting after the opening quotation mark of the string literal and ending with the Hebrew identifier is reordered, so that the string literal is split in rendering, and the arguments of `WriteLine` appear in the wrong order, as shown in Figure 2.

Figure 2. Bidirectional reordering inconsistent with lexical structure

```
Console.WriteLine("Error: {0} {1}", message, this);

Console.WriteLine(" הודעה ,"{1} {0} :השתבש  , this);
```

Additional examples are shown in Table 1, wherein the problematic right-to-left runs are underlined, as well corresponding runs in the left-to-right code.

Note: The issue can occur even if the use of right-to-left characters is limited to string literals and comments, as in the last example in Table 1.

Table 1. Examples of unreadable bidirectional reorderings

Language	Direction	Code Snippet	Issues in the RTL version
C#	LTR	`Console.WriteLine("Error: {0} {1}", message, this);`	The string literal is split and the arguments are reordered; see Figure 2.
	RTL	`Console.WriteLine("הודעה ,"{1} {0} :השתבש, this);`	
Ada	LTR	`<<Error>> Log (Message); -- Something went wrong.`	The label brackets `<<>>` do not visually match. The end-of-line comment is split, and most of it lies in the middle of the line.
	RTL	`<<משהו השתבש -- ;(הודעה) רשם <<שגיאה.`	
Python	LTR	`return integral(lambda a: a ** 2, from =0, to=1)`	The lambda expression is split, and the arguments are reordered.
	RTL	`return אינטגרל(lambda 1=ל ,0=מ ,2 ** א: א)`	
Rust	LTR	`fn integral<F: Fn(f64) -> f64>(` `  integrand: F, interval: std::ops::Range<f64>) -> f64 {`	The generic brackets `<>` do not visually match. The generic parameter is separated from its bound `Fn(f64)`, one of the function parameters is separated from its type `std::ops::Range<f64>`.
	RTL	`fn אינטגרלכפ: Fn(f64) -> f64>(` `  אינטגרנד: פ, קטע: std::ops::Range<f64>) -> f64 {`	

Language	Direction	Code Snippet	Issues in the RTL version
C++	LTR	`std::vector<meow> cat;`	The template brackets `<>` do not visually match; the `std::vector` is separated from its type parameter مواء.
	RTL	`std::vector<مواء> قطة;`	
C++	LTR	`return u8"meow";  // Placeholder message.`	Both the string literal and the comment are split; most of the end-of-line comment lies in the middle of the line.
	RTL	`return u8"مواء";  // رسالة العنصر النائب.`	

The recommended mitigations for these usability issues are the same as the mitigations for the corresponding spoofing issues described in *Section 1.2.3, Spoofing using bidirectional reordering*.

2 Conformance

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

UTS55-C1 *An implementation claiming conformance to this specification shall identify the version of this specification.*

UTS55-C2 *An implementation claiming to implement the Basic Ordering for Source Code shall do so in accordance with the specifications in Section 4.1.2, Basic Ordering.*

UTS55-C3 *An implementation claiming to implement the Ordering for Literal Text with Interspersed Syntax shall do so in accordance with the specifications in Section 4.1.4, Ordering for Literal Text with Interspersed Syntax.*

UTS55-C4 *An implementation claiming to implement Mixed-Script Detection in Identifier Chunks shall do so in accordance with the specifications in Section 5.1.2.2, Mixed-Script Detection in Identifier Chunks.*

UTS55-C5 *An implementation claiming to implement Conversion to Plain Text for Source Code shall do so in accordance with the specifications in Section 5.2, Conversion to Plain Text.*

UTS55-C6 *An implementation claiming to enforce Unicode Identifier Styles shall do so in accordance with the specifications in Section 5.3, Identifier Styles.*

3 Computer Language Specifications

The normative material appropriate for language specifications may be found in *Unicode Standard Annex #31, Unicode 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.

3.1 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 or 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 compatibility notation profile defined in *Section 7.1, Mathematical Compatibility Notation Profile*, of *Unicode Standard Annex #31, Unicode Identifiers and Syntax* [UAX31]. This is because these languages are used in scientific computing, which can benefit from the greater legibility and disambiguation afforded by allowing these additional characters in identifiers.

> **Note:** Like many characters that are part of default identifiers, the additional characters from the mathematical compatibility notation profile can be confusing to most readers; as a result, they are

discouraged in non-specialist use by being excluded from the General Security profile. See *Section 5.1.3, General Security Profile*.

3.1.1 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 3.3, 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 property has changed over time for already-encoded characters, so we may not want to stabilize it completely; but it may be possible to stabilize toNFKC_Casefold on the set XID_Continue, along the lines of “once a character is XID_Continue, the value of the NFKC_Casefold property will never change for that character”.

Simple case folding has no stability guarantees.

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 under *Interpretation*, in *Section 3.2, Conformance Requirements*, in [Unicode].

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 3.3, 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 based on a more abstract sense of character identity; for instance, e and E are treated as the same letter. Normalization Form KC (compatibility equivalence) is an equivalence between characters that share such an abstract identity.

> **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.

3.1.2 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 [Unicode].

> **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 5.3, 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 5.3, Identifier Styles*.

3.2 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 1.2.1, 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 1.2.1, 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 5.2, 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 4.1, 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 paragraph 2.2(7.1) in [ARM12]. 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. See *Section 2.5, Whitespace*, in [Rust]
>
> **Example:** Consider the following line of C++11, displayed according to the recommendations in *Section 4.1, 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) {
           ^
```

Consider now a corresponding line of Rust, where, for clarity, we have made the left-to-right mark visible as described in *Section 4.2.2, 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 5.2, Conversion to Plain Text*.

It is further recommended that languages allow the ignorable format controls between atoms, as defined in *Section 4.1, 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 ∮∮ to be the same as an operator ∮∮, 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 5.2, 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, Unicode Identifiers and Syntax* [UAX31], the corresponding profile for UAX31-R3b is described if needed.

3.3 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.

3.3.1 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 compatibility 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 compatibility notation profile is also recommended on its own merits, regardless of compatibility concerns; See *Section 3.1, Identifiers*.

3.3.2 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 under *Interpretation*, in *Section 3.2, Conformance Requirements*, in [Unicode].

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 3.1.1, Normalization and Case*; 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:

```
// Prints documents (consisting of a sequence of lines) to file f,
// and prints the number of lines of each document, as well as the
// total number of lines, to standard output.
// A counter for the total number of lines printed.
int lignes_imprimées = 0;  // Decomposed e + ◌́
for (std::vector<std::string> const& document: documents) {
  // A counter for the number of lines in the document.
  int lignes_imprimées = document.size();  // Precomposed é.
  // Print each line of the document.
  for (std::string const& ligne: document.lignes()) {
    std::fputs(ligne.c_str(), f);
    ++lignes_imprimées;  // Decomposed e + ◌́
  }
  std::printf("%s : %d lignes imprimées",
              document.front().c_str(),
              lignes_imprimées);  // Precomposed é.
}
// Report the total number of lines printed.
std::printf("total : %d lignes imprimées",
            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 5.1, Confusability Mitigation Diagnostics*.

> **Note:** Confusable detection as described in *Section 5.1.1, Confusable Detection*, encompasses such a warning, as canonically equivalent sequences are always confusable. The reverse is however not true: the Latin A, Greek A, and Cyrillic A 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 in most scripts) to UAX31-R5 Equivalent case-insensitive identifiers.

4 Source Code Display

Most of the issues described in *Section 1, Introduction*, 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.

4.1 Bidirectional Ordering

The issues arising from bidirectional reordering described in *Section 1.2.3, Spoofing using bidirectional reordering*, and *Section 1.3.2, 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*, in *Unicode Standard Annex #9, Unicode Bidirectional Algorithm* [UAX9].

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

4.1.1 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 1.2.1, 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 under *Modification* in *Section 3.2, Conformance Requirements*, in [Unicode]. 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, Unicode Identifiers and Syntax* [UAX31], 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	·	+	·	תו	ּ·	==	·	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”.

4.1.2 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 *Section 4.1.3, Nested 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:
    1. The standard technique of inserting a mark to override a first-strong heuristic doesn’t work, as that changes the string.
    2. 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. Problems:
    1. Requires special handling when implementing HL4, so that the effect of these characters persists into the string literal.
    2. 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:
    1. This would be an incompatible change to existing language specifications.
    2. 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:
    1. This would be an incompatible change to existing language specifications.
    2. 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:
    1. numeric atoms have the attribute `dir="ltr"` regardless of the value of d;
    2. 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 1.3.2, 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
```

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.

4.1.2.1 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* [UAX9], 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 4.1.2, 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.

```
 1. declare
 2.   Atom_Direction := The atom direction, either LTR or RTL;
 3.   Atom_Isolate : Code_Point := (if Atom_Direction = LTR then LRI else RLI);
 4. begin
 5.   Separate the text into lines and each line into atoms,
 6.       as described in Section 4.1.1, Atoms, in a language-dependent manner;
 7.   for each Line loop
 8.     for each Atom of Line loop
 9.       if Atom is known to consist of text in some computer language L then
10.         -- For the purposes of this algorithm, the contents of a string literal with
11.         -- escape sequences is text in some computer language, see Figure 1.
12.         Gather the text of all consecutive atoms that are in that computer language,
13.             including intervening line breaks;
14.         if L is literal text with interspersed syntax then
15.           Apply the algorithm described
16.               in Section 4.1.4.1, Equivalent Isolate Insertion for the Ordering for Literal Text with Interespersed Syntax
17.               to the text of these atoms, using Atom_Direction as the atom direction;
18.         else
19.           Apply the Equivalent Isolate Insertion for the Basic Ordering
20.               to the text of these atoms, using Atom_Direction as the atom direction;
21.         end if;
22.       end if;
23.       -- If the ordering is implemented using higher-level protocols, such as HTML dir,
24.       --  this step is unnecessary: the higher-level protocol closes the isolates.
25.       Compute U, the number of isolate initiators in the text of Atom that
26.           do not have a matching PDI within the text of Atom, as defined in BD9;
27.       if Atom is a comment content atom then
28.         Insert an FSI character before Atom;
29.       elsif Atom is a numeric atom then
30.         Insert an LRI character before Atom;
31.       else
32.         Insert Atom_Isolate before Atom;
33.       end if;
34.       Insert U + 1 PDI characters at the end of Atom;
35.     end loop;
36.     Insert Atom_Isolate at the beginning of Line;
```

```
    Insert a PDI character at the end of Line;
  end loop;
end;
```

4.1.3 Nested 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 4.1.2, Basic Ordering*, or the ordering described in *Section 4.1.4, Ordering for Literal Text with Interspersed Syntax*, as appropriate.

Example 1: Consider the following C# statement, which constructs a `JsonDocument` object from a string.

```
var decomposition_mapping =
    System.Text.Json.JsonDocument.Parse(
        "{'①':'1','4日':'4日'}");
```

If a property with the key `'ﷺ'` and the value `'صلى الله عليه وسلم'` is added to the JSON object, the string is displayed as follows if the JSON structure is not taken into account when displaying it:

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

This display is misleading, as it obscures the JSON structure; for instance, it looks like `'1'` is a key rather than the value corresponding to `'①'`.

An editor which is capable of recognizing that the string contains JSON should instead order the contents as follows, applying the basic ordering to the JSON.

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

Example 2: Consider the following Python statement, which uses regular expressions to replace various orthographies of “Google, Inc.” with “Google, LLC” in the contents of the variable `terms`.

```
terms = re.sub(r'Google,?\s+Inc\.?', 'Google LLC', terms)
```

Consider now a similar regular expression which replaces various orthographies of the corresponding Hebrew “גוגל, בע״מ”, as shown by an editor which is not aware that the string is a regular expression:

```
terms = re.sub(r'גוגל,?\s+מ["״]בע', 'Google LLC', terms)
```

This display is misleading, as it looks like the regular expression matches the nonsensical “גוגל, בע״מ” (“Inc. ,Google”).

An editor which is capable of recognizing the string as a regular expression should instead display it as follows; see *Section 4.1.4, Ordering for Literal Text with Interspersed Syntax*.

```
terms = re.sub(r'מ["״]בע\s+,?גוגל', 'Google LLC', terms)
```

4.1.4 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:
    1. `a[bc]d(e|f[g]h)i`
    2. `|`
    3. `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 3, left-to-right atom direction is used.

Example 1: Consider the regular expression from Example 2 of *Section 4.1.3, Nested Languages*, which matches strings like “גוגל, בע״מ” with one or more spaces, an optional comma, and either an ASCII quotation mark or a gershayim.

As described in that section, displaying that regular expression without taking its structure into account is misleading. Moreover, even displaying it according to the basic ordering, treating the syntax characters as atoms, is also misleading, as shown in the table below. Instead, the subexpressions `\s+` (one or more spaces), `,?` (optional comma) and `["״]` should be directionally isolated.

Ordering	Display	Notes
Plain Text	גוגל,?\s+מ["״]בע	Misleading: looks like it matches the nonsensical “גוגל, בע״מ” (“Inc. ,Google”).
Basic	גוגל,?\s+בע["״]מ	Misleading: looks like it matches the nonsensical “גוגל, מ״בע” (“c.In ,Google”).
Literal Text with Interspersed Syntax	מ["״]בע\s+,?גוגל	This is the recommended display.

Example 2: Consider the following C statement, which prints the text “kilobyte (kB)” to standard output:

```
puts("kilobyte (kB)");
```

If the string is translated to the Hebrew “קילו בית (ק"ב)” (with an ASCII quotation mark instead of the gershayim), the corresponding string literal should be displayed not as plain text, nor according to the basic ordering by treating the escape sequence `\"` as an atom, but instead by directionally isolating the escape sequence, as shown in the following table.

Ordering of the string literal	Display	Notes
Plain Text	puts("קילו בית (ק"\ב");)	The escape sequence is unrecognizable, which makes the extent of the string literal confusing.
Basic	puts("קילו בית (ק"\ב)");)	The string appears to be a nonsensical “קילו בית (ק"ב)” (“(Bkilobyte (k”).
Literal Text with Interspersed Syntax	puts("קילו בית (ק"\ב");)	This is the recommended display.

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

```
1. alert(`Version ${[15,1]} was released yesterday`);
2. n=[15, 1]; alert(`Version ${n} was released yesterday`);
```

If the string is translated to the Persian “نسخهٔ 15,1 دیروز منتشر شد”, the corresponding template literal literal should be displayed not as plain text, nor according to the basic ordering by treating the placeholder `${}` as an atom, but instead by directionally isolating the placeholder, as shown in the following table.

Ordering of the template literal	Display	Notes
Plain Text	1. alert(`دیروز منتشر شد {[15,1]}$ نسخهٔ`); 2. n=[15, 1]; alert(`دیروز منتشر شد {n}$ نسخهٔ`);	On line 1, the placeholder syntax is obscured; on line 2, the string is

Ordering of the template literal	Display	Notes
		reordered compared to what will be shown.
Basic	1. alert(`نسخهٔ ${[15,1]} دیروز منتشر شد`); 2. n=[15, 1]; alert(`نسخهٔ ${n} دیروز منتشر شد`);	In both cases, the string is reordered compared to what will be shown.
Literal Text with Interspersed Syntax	1. alert(`دیروز منتشر شد ${[15,1]} نسخهٔ`); 2. n=[15, 1]; alert(`دیروز منتشر شد ${n} نسخهٔ`);	This is the recommended display.

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 4.2.2, 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 4: 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 its `dir` attribute, thus, with left-to-right atom direction,

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

4.1.4.1 Equivalent Isolate Insertion for the Ordering for Literal Text with Interespersed 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* [UAX9], 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 4.1.2, 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.

```
 1. declare
 2.   Atom_Direction := The atom direction, either LTR or RTL;
 3.   Paragraph_Direction :=
 4.       The text direction, LTR, RTL, or FS (first strong), if specified by the language,
 5.       otherwise Atom_Direction;
 6.   Text_Isolate : Code_Point := (if Paragraph_Direction = LTR then LRI
 7.                                elsif Paragraph_Direction = RTL then RLI
 8.                                else FSI);
 9. begin
10.   Separate the text into lines and each line into runs of syntactic elements and literal text,
11.       in a language-dependent manner;
12.   for each Line loop
13.     for each Syntactic_Element of Line loop
14.       Apply the Equivalent Isolate Insertion for the Basic Ordering
15.           to Syntactic_Element, using Atom_Direction as the atom direction;
16.     end loop;
17.     -- If the ordering is implemented using higher-level protocols, such as HTML dir,
18.     --   this step is unnecessary: the higher-level protocol closes the isolates.
19.     Compute U, the number of isolate initiators in the text of Line that
20.         do not have a matching PDI within the text of Line, as defined in BD9;
21.     Insert Text_Isolate at the beginning of Line;
22.     Insert U + 1 PDI characters at the end of Line;
23.   end loop;
24. end;
```

4.2 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:
>
> ```
> "Google تابعة لشركة YouTube"  # (1)
> ```
>
> These blank and invisible characters could be made visible as follows:
>
> ```
> "[RLE]Google·تابعة·لشركة·YouTube[PDF]"  # (2)
> ```
>
> 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) and (2).
>
> ```
> "[RLE]YouTube·تابعة·لشركة·Google[PDF]"  # Misleading display.
> ```

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]).

4.2.1 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 Update (version : Positive);
2. procedure به‌روز (نسخه : Positive);
```

For the zero width joiner, the suggested representation is a vertical bar surmounded by an x overlaid at the position of the joiner:

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

Industry example: Notepad uses the suggested representations for the joiner controls.

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 ≤.
```

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

4.2.2 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 5.2, Conversion to Plain Text*. A lightweight nonspacing representation should therefore be used; for instance, ↱ for left-to-right mark and ↰ for right-to-left and Arabic letter marks:

```
→       עדכון(15);··//·Update·to·version·15.
```

Industry example: Notepad uses the suggested representations for the implicit directional marks.

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]"

4.2.3 Suggested Levels for “Show Hidden” Modes

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 5.2, 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 5.2, 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.”.

```
// [RLE].متغیر message خالی نیست
*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”:

```
// متغیر message خالی نیست.
```

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):

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

4.3 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 1.3.1, 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 5.1, Confusability Mitigation Diagnostics*, should be used. These can then be surfaced in display, for instance, using “squiggles” as for other warnings.

4.4 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 has 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. 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.

5 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 4.1, 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.

5.1 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.

5.1.1 Confusable Detection

Mitigating identifier spoofing requires detecting when identifiers are confusable. A warning to the user is recommended when an identifier is confusable with some other relevant identifier or with a reserved word of the language. An implementation should only diagnose as confusables identifiers that are *distinct*, yet look identical or confusingly similar; identifiers that are identical, or that are equivalent under any normalization or case equivalence used by the language, should not be flagged.

Section 4, Confusable Detection, in *Unicode Technical Standard #39, Unicode Security Mechanisms* [UTS39], defines both LTR-confusability and RTL-confusability. In a source code document with LTR atom order, LTR-confusability should be used; in a source code document with RTL atom order, RTL-confusability should be used.

> **Note:** Confusability is determined by the *skeleton* operation defined in *Unicode Technical Standard #39, Unicode Security Mechanisms* [UTS39]. It is best to treat the skeleton operation as a black box, and not make unwarranted assumptions about the behavior of the operation. First, it is not determined on a code point by code point basis. For example, skeleton(A) may be identical with skeleton(BC), even though A is shorter than BC. In addition, a likely future enhancement for complex scripts is the addition of contextual skeletons. Then skeleton(XY) may be the same as skeleton(XZ), even though skeleton(Y) ≠ skeleton(Z).

~~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 is 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 `a1ᴎ` and `a1ᴎ`).

In order to mitigate usability issues arising from confusability, such as the ones described in *Section 1.3.1, 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 5.1.2, Mixed-Script Detection*.

5.1.2 Mixed-Script Detection

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 HTTPЗапрос (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, *viз* `isspace` and `exp`.

Confusable detection should be used to more systematically deal with spoofing issues; see *Section 5.1.1, Confusable Detection*.

5.1.2.1 Identifier Chunks

An *identifier word boundary* is defined by the following rules, using the notation from *Section 1.1, Notation*, in *Unicode Standard Annex #29, Unicode Text Segmentation* [UAX29].

Treat a letter followed by a sequence of nonspacing or enclosing marks as that letter.

The regular expressions for the following rules incorporate this one; only the text descriptions rely on it.

🐫 *CamelBoundary. An identifier word boundary exists after a lowercase or non-Greek titlecase letter followed by an uppercase or titlecase letter:*

```
[ \p{Ll} [\p{Lt}-\p{Grek}] ] [\p{Mn}\p{Me}]* ÷ [\p{Lu}\p{Lt}]
```

🎩 *HATBoundary. An identifier word boundary exists before an uppercase or titlecase letter followed by a lowercase letter, or before a non-Greek titlecase letter:*

```
÷ [\p{Lu}\p{Lt}] [\p{Mn}\p{Me}]* \p{Ll}  |  [\p{Lt}-\p{Grek}]
```

🐍 *snake_boundary. An identifier word boundary exists either side either side of a Punctuation character which is not an Other_Punctuation character:*

```
÷ [\p{P}-\p{Po}]

[\p{P}-\p{Po}] ÷
```

No other identifier word boundaries exist.

Any × Any

An identifier splits into *identifier chunks* delimited at identifier word boundaries.

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

Identifier	Identifier chunks	Notes
TypeII	🎩Type🐫II	
OCaml	O🎩Caml	The HATBoundary is designed to accommodate the common practice of keeping acronyms in upper case in a CamelCase identifier.
HTTPЗапрос	HTTP🎩Запрос	
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 \p{Ll}\p{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.
dromedaryCamel	dromedary🐫🎩Camel	🐫
snakeELEPHANTSnake	snake🐫ELEPHANT🎩Snake	🌹📦

5.1.2.2 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 *Section 5.2, Restriction-Level Detection*, in *Unicode Technical Standard #39, Unicode Security Mechanisms* [UTS39];
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 ∅ 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.

Identifier Chunk	Notes
`Строка`	Confusing, confusable with all-Cyrillic Cтpoкa.
`Δt`	Visibly mixed-script, t is not confusable with a Greek letter, nor is Δ confusable with a Latin letter.
`μэow`	Visibly mixed-script, μ is not confusable with a Cyrillic letter nor with a Latin letter.
`MIKPA`	Confusing, confusable with all-Greek MIKPA and all-Latin MIKPA.
`HTTPसर्वर`	Visibly mixed-script, H is not confusable with a Devanagari letter, nor is स confusable with a Latin letter.
`microb`	Confusing, confusable with all-Latin microb.

5.1.3 General Security Profile

As described in *Section 6.1, Confusables Data Collection*, in *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 [UTS39].

For this purpose, implementations should use a modification of the General Security Profile which allows the characters U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER. This is because these characters are necessary in the orthographies of some major languages. The potential issues arising from the use of these default ignorable characters are a small subset of the broader problem of confusability, which must be dealt with using the mechanisms described in *Section 5.1.1, Confusable Detection*.

The contextual restrictions for these characters described in *Section 3.1.1, Joining Controls*, in [UTS39], should be applied as part of the General Security Profile, as they can mitigate usability issues.

Example: Consider بەروز, the Persian word for “update” (which could be romanized be-ruz, where the hyphen roughly corresponds to the linguistic separation indicated by the ZWNJ in Persian). This word contains a zero width non-joiner; removing it yields بهروز, which displays differently, and is a different word (the name Behrooz). An implementation should not warn about the use of the identifier بەروز. However, a programmer who uses Persian identifiers, having a key for that character, could accidentally type a zero width non-joiner in a place where it has no effect, as in the following:

```
if Version > Current_Version then  -- ZWNJ between V and e.
   بەروز (Version);  -- ZWNJ between و and ز.
```

Unless confusable detection is performed prior to successful compilation, the programmer would be presented with baffling compilation errors, as the code would fail to compile even though it looks correct. A diagnostic based on the General Security Profile, which requires only lexical analysis, and can therefore be performed early, would warn about those spurious zero width non-joiners.

Such a diagnostic also guards against obscure characters that may be intrinsically confusing in identifiers, such as U+01C3 LATIN LETTER RETROFLEX CLICK (!), a letter which looks identical to the exclamation mark.

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. Further, a user may wish to make use of some obscure characters while retaining confusable detection.

> **Industry example:** The Rust compiler warns about identifiers that use characters outside the General Security Profile.

5.1.4 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, Unicode 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 $\boldsymbol{r}$ 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).

5.1.5 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 4.4, 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.

```
/*...בדוק תנאים מוקדמים */ הודעה.requireNonNull(); /* 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.requireNonNull(); /* 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.

5.1.6 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 5.2, Conversion to Plain Text*, includes such a diagnostic.

5.2 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 3.2, 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 4.1, 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 4.1.3, Nested Languages*, are not applied.
2. The rules from *Section 4.1.4, 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 5.2.1, Unpaired Brackets*.

If nested 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. `1 =: תו + X;`
B. `תו :=1 + 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 is 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.

```
declare
  Needs_LRM : Boolean := False;
begin
  Separate the text into lines and each line into atoms,
      as described in Section 4.1.1, Atoms, in a language-dependent manner;
  for each Line loop
    for each Atom of Line loop
      if Atom is a whitespace atom then
```

```
 9.         Remove any instances of LRM and RLM from Atom;
10.       end if;
11.       if Needs_LRM then
12.         if inserting LRM before Atom has no effect on the meaning of the program then
13.           Insert LRM before Atom;
14.           Needs_LRM := False;
15.         else  -- For instance, if the position before Atom is within a string literal
16.           Look for the first character C in Atom such that
17.             the Bidi_Class property of C is one of L, R, AL, EN, AN, LRE, RLE, LRI, RLI, or FSI;
18.         if C exists and its Bidi_Class property is not L then
19.             Emit an error: the line cannot be converted to plain text by this algorithm;
20.         end if;
21.       end if;
22.       if Atom is a comment content atom then
23.         if Atom does not start with the character FSI then
24.           Look for the first character C in Atom such that
25.               the Bidi_Class property of C is one of L, R, AL, LRE, RLE, LRI, RLI, or FSI;
26.           if C exists and its Bidi_Class property is not L then
27.             Prepend FSI to Atom;
28.           end if;
29.         end if;
30.         if Atom is followed by a code point that does not have Bidi_Class B then
31.           Insert a sequence of PDI characters after Atom as needed to
32.               close any isolate initiators in Atom that do not have a matching PDI, as defined in BD9;
33.           Insert a sequence of PDF characters after those PDI characters as needed to
34.               close any embedding initiators in Atom that
35.                   do not lie between an isolate initiator and its matching PDI, and
36.                   do not have a matching PDF, as defined in BD11;
37.         end if;
38.       end if;
39.       if Atom is followed by a code point that does not have Bidi_Class B then
40.         if Atom has any isolate initiators that
41.             do not have a matching PDI, as defined in BD9 then
42.           Emit an error: the line cannot be converted to plain text by this algorithm;
43.         end if;
44.         if Atom has any embedding initiators that
45.             do not lie between an isolate initiator and its matching PDI, and
46.             do not have a matching PDF, as defined in BD11 then
47.           Emit an error: the line cannot be converted to plain text by this algorithm;
48.         end if;
49.         Look for the last character C in Atom such that
50.           the Bidi_Class property of C is one of L, R, AL, PDF, or PDI;
51.         if C exists and its Bidi_Class property is not L then
52.           Needs_LRM := True;
53.         end if;
54.       end if;
55.     end loop;
56.     if Line is terminated by a character with Bidi_Class B then
57.       Needs_LRM := False;
58.     end if;
59.   end loop;
60. end;
```

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]:

```
(\p{bc=L}
|\p{bc=EN}
|\p{bc=EN}\p{bc=ET}
|\p{bc=ET}\p{bc=EN}
|\p{bc=EN}\p{bc=CS}\p{bc=EN})+
```

Such a conversion needs to insert RLM rather than LRM, and to look for R or AL, rather than L, in lines 18, 26, and 51.

Implementation note: When separating the source code into atoms on lines 4 and 5, 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.

5.2.1 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 `"אב(`:

```
R"("אב(")"
```

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

```
R"("אב(")"
```

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.

5.3 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*, [UTS18].

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 `(\p{Lu}[\p{Mn}\p{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 `\p{Pc}` in the above regular expressions by `[\p{Pc}\p{Pd}]`, treating the hyphen-minus like the low line (allowing “kebab-case”).

6 Reference Implementations

Review Note: If reference implementations are available in time for publication, they will be mentioned in this section.

References

[ARM12] *Ada Reference Manual, 2012 Edition*
https://www.adaic.org/resources/add_content/standards/12rm/html/RM-TTL.html

[Rust] *The Rust Reference*
https://doc.rust-lang.org/reference/

[Unicode] *The Unicode Standard*
Latest version:
https://www.unicode.org/versions/latest/

[UAX9] *Unicode Standard Annex #9: Unicode Bidirectional Algorithm*
Latest version:
https://www.unicode.org/reports/tr9/

[UAX14] *Unicode Standard Annex #14: Unicode Line Breaking Algorithm*
Latest version:
https://www.unicode.org/reports/tr14/

[UAX15] *Unicode Standard Annex #15: Unicode Normalization Forms*
Latest version:
https://www.unicode.org/reports/tr15/

[UAX29] *Unicode Standard Annex #29: Unicode Text Segmentation*
Latest version:
https://www.unicode.org/reports/tr29/

[UAX31] *Unicode Standard Annex #31: Unicode Identifiers and Syntax*
Latest version:
https://www.unicode.org/reports/tr31/

[UTS39] *Unicode Technical Standard #39: Unicode Security Mechanisms*
Latest version:
https://www.unicode.org/reports/tr39/

Acknowledgements

Robin Leroy and Mark Davis authored the bulk of the text, under direction from the Unicode Technical Committee.

The attendees of the Source Code Working Group meetings were critical to identifying the technical issues and ensuring the document was understandable to its intended audience: Peter Constable, who kept the process on schedule, and repeatedly found elegant solutions to wording issues; Elnar Dakeshov; Mark Davis, who suggested creating the group and chaired it; Barry Dorrans, who pointed out many interesting edge cases among programming languages and programming environments; Steve Dower, who ensured we provided enough room to language designers while giving them clear guidance; Michael Fanning; Asmus Freytag, who provided valuable insights on security-sensitive identifier systems; Dante Gagne; Rich Gillam, who ensured the group kept sight of the bigger picture, and suggested the recommendation for multiple “show hidden” levels; Manish Goregaokar, who helped with many examples and pointed out the need for guidance on identifier styles; Tom Honermann, who relayed valuable feedback from ISO/IEC JTC 1/SC 22/WG 21/SG 16, and ensured that SG 16 was kept informed of Unicode’s progress; Jan Lahoda; Nathan Lawrence, who ensured that descriptions of bidirectional behaviour

remained understandable; Robin Leroy, vice-chair; Chris Ries; Markus Scherer, who ensured the group's recommendation were promptly reviewed by the Properties and Algorithms Group; Richard Smith, who found many interesting edge cases, and coined the term "Proud_snake_case".

Catherine "whitequark" came up with the idea of confusable detection in identifier chunks, which is adapted in this specification to encompass chunks delimited by case.

Thanks also to the following people for their feedback or contributions to this document: Julie Allen, Deborah Anderson, Ned Holbrook, Corentin Jabot, 梁海 Liang Hai, Roozbeh Pournader, Ken Whistler.

Modifications

The following summarizes modifications from the previous revision of this document.

Revision 2

- Advanced from Proposed Draft to Draft Unicode Technical Standard.
- Addressed comments made at UTC #174.

Revision 1

- Initial version following proposal L2/22-229R to the UTC.
- Addressed comments made at UTC #173.
- *Section 5.1.3, General Security Profile*:
  - Added a recommendation to allow ZWJ and ZWNJ, emphasizing the need to make use of confusable detection to deal with confusability issues.