L2/24-033

Proposed Update Unicode® Standard Annex #31

UNICODE IDENTIFIERS AND SYNTAX

Version	Unicode 16.0.0 (draft 1)
Editors	Mark Davis (mark@unicode.org) and Robin Leroy (eggrobin@unicode.org)
Date	2023-12-12
This Version	https://www.unicode.org/reports/tr31/tr31-40.html
Previous Version	https://www.unicode.org/reports/tr31/tr31-39.html
Latest Version	https://www.unicode.org/reports/tr31/
Latest Proposed Update	https://www.unicode.org/reports/tr31/proposed.html
Revision	40

Summary

This annex describes specifications for recommended defaults for the use of Unicode in the definitions of general-purpose identifiers, immutable identifiers, hashtag identifiers, and in pattern-based syntax. It also supplies guidelines for use of normalization with identifiers.

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*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time.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 Standard Annex (UAX)** forms an integral part of the Unicode Standard, but is published online as a separate document. The Unicode Standard may require conformance to normative content in a Unicode Standard Annex, if so specified in the Conformance chapter of that version of the Unicode Standard. The version number of a UAX document corresponds to the version of the Unicode Standard of which it forms a part.*

Please submit corrigenda and other comments with the online reporting form [Feedback]. Related information that is useful in understanding this annex is found in Unicode Standard Annex #41, "Common References for Unicode Standard Annexes." 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]. For any errata which may apply to this annex, see [Errata].

Contents

1 Introduction

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

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

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

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

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

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

The specification in this annex provides a definition of identifiers that is guaranteed to be backward compatible with each successive release of Unicode, but also allows any appropriate new Unicode characters to become available in identifiers. In addition, Unicode character properties for stable pattern syntax are provided. The resulting pattern syntax is backward compatible *and* forward compatible over future versions of the Unicode Standard. These properties can either be used alone or in conjunction with the identifier characters.

Figure 1 shows the disjoint categories of code points defined in this annex. (The sizes of the boxes are not to scale.)

Figure 1. Code Point Categories for Identifier Parsing

ID_Start Characters	Pattern_Syntax Characters	Unassigned Code Points
ID_Nonstart Characters	Pattern_White_Space Characters	
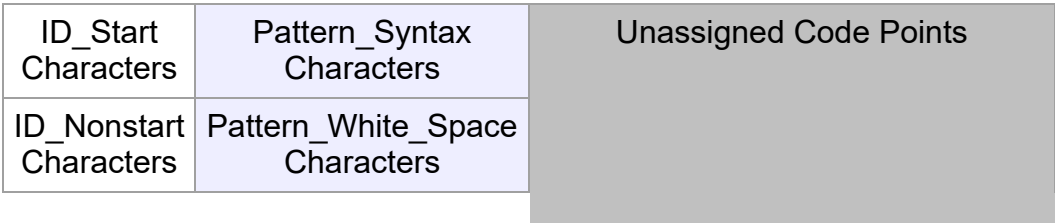
Other Assigned Code Points	

The set consisting of the union of *ID_Start* and *ID_Nonstart* characters is known as *Identifier Characters* and has the property *ID_Continue*. The *ID_Nonstart* set is defined as the set difference *ID_Continue* minus *ID_Start*: it is not a formal Unicode property. While lexical rules are traditionally expressed in terms of the latter, the discussion here is simplified by referring to disjoint categories.

1.1 Stability

There are certain features that developers can depend on for stability:

- Identifier characters, Pattern_Syntax characters, and Pattern_White_Space are disjoint: they will never overlap.
- By definition, the Identifier characters are always a superset of the ID_Start characters.
- The Pattern_Syntax characters and Pattern_White_Space characters are immutable and will not change over successive versions of Unicode.
- The ID_Start and ID_Nonstart characters may grow over time, either by the addition of new characters provided in a future version of Unicode or (in rare cases) by the addition of characters that were in Other.

In successive versions of Unicode, the only allowed changes of characters from one of the above classes to another are those listed with a plus sign (+) in *Table 1.*

Table 1. Permitted Changes in Future Versions

	ID_Start	ID_Nonstart	Other Assigned
Unassigned	+	+	+
Other Assigned	+	+	
ID_Nonstart	+		

The Unicode Consortium has formally adopted a stability policy on identifiers. For more information, see [Stability].

1.2 Customization

Each programming language standard has its own identifier syntax; different programming languages have different conventions for the use of certain characters such as $, @, #, and _ in identifiers. To extend such a syntax to cover the full behavior of a Unicode implementation, implementers may combine those specific rules with the syntax and properties provided here.

Each programming language can define its identifier syntax as *relative* to the Unicode identifier syntax, such as saying that identifiers are defined by the Unicode properties, with the addition of "$". By addition or subtraction of a small set of language specific characters, a programming language standard can easily track a growing repertoire of Unicode characters in a compatible way. See also *Section 2.5, Backward Compatibility.*

Similarly, each programming language can define its own whitespace characters or syntax characters relative to the Unicode Pattern_White_Space or Pattern_Syntax characters, with some specified set of additions or subtractions.

Systems that want to extend identifiers to encompass words used in natural languages, or narrow identifiers for security may do so as described in *Section 2.3, Layout and Format Control Characters*, *Section 2.4, Specific Character Adjustments*, and *Section 5, Normalization and Case*.

To preserve the disjoint nature of the categories illustrated in *Figure 1*, any character *added* to one of the categories must be *subtracted* from the others.

> **Note:** In many cases there are important security implications that may require additional constraints on identifiers. For more information, see [UTR36].

1.3 Display Format

Implementations may use a format for *displaying* identifiers that differs from the internal form used to *compare* identifiers. For example, an implementation might display what the user has entered, but use a normalized format for comparison. Examples of this include:

> **Case.** The display format retains case differences, but the comparison format erases them by using Case_Folding. Thus "A" and its lowercase variant "a" would be treated as the same identifier internally, even though they may have been input differently and may display differently.

> **Variants.** The display format retains variant distinctions, such as halfwidth versus fullwidth forms, or between variation sequences and their base characters, but the comparison format erases them by using NFKC_Case_Folding. Thus "A" and its full-width variant "Ａ" would be treated as the same identifier internally, even though they may have been input differently and may display differently.

For an example of the use of display versus comparison formats see *UTS #46: Unicode IDNA Compatibility Processing* [UTS46]. For more information about normalization and case in identifiers see *Section 5, Normalization and Case*.

1.4 Conformance

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

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

> **Note:** An implementation can make use of the property-based definitions from a specific version of this specification with property assignments from an unversioned reference to the Unicode Character Database. In this case, the implementation should specify a minimum version of Unicode for the properties.

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

- R1. Default Identifiers
- R1b. Stable Identifiers

- R2. Immutable Identifiers
- R3. Pattern_White_Space and Pattern_Syntax Characters
- R3a. Pattern_White_Space Characters
- R3b. Pattern_Syntax Characters
- R3c. Operator Identifiers
- R4. Equivalent Normalized Identifiers
- R5. Equivalent Case-Insensitive Identifiers
- R6. Filtered Normalized Identifiers
- R7. Filtered Case-Insensitive Identifiers
- R8. Hashtag Identifiers

Note: Requirement R1a has been removed. The characters that were added when meeting this requirement are now part of the default; the contextual checks required by this requirement remain as part of the General Security Profile in Unicode Technical Standard #39, "Unicode Security Mechanisms" [UTS39].

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

1.5 Notation

This annex uses *UnicodeSet* notation to illustrate the derivation of some properties or sets of characters. This notation is defined in the "Unicode Sets" section of *UTS #35, Unicode Locale Data Markup Language* [UTS35].

2 Default Identifiers

The formal syntax provided here captures the general intent that an identifier consists of a string of characters beginning with a letter or an ideograph, and followed by any number of letters, ideographs, digits, or underscores. It provides a definition of identifiers that is guaranteed to be backward compatible with each successive release of Unicode, but also adds any appropriate new Unicode characters.

The formulations allow for extensions, also known as *profiles*. That is, the particular set of code points or sequences of code points for each category used by the syntax can be customized according to the requirements of the environment. Profiles are described as additions to or removals from the categories used by the syntax. They can thus be combined, provided that there are no conflicts (whereby one profile adds a character and another removes it), or that the resolution of such conflicts is specified.

If such extensions include characters from Pattern_White_Space or Pattern_Syntax, then such identifiers do not conform to an unmodified *UAX31-R3 Pattern_White_Space and Pattern_Syntax Characters*. However, such extensions may often be necessary. For example, Java and C++ identifiers include '$', which is a Pattern_Syntax character.

UAX31-D1. **Default Identifier Syntax:**

```
<Identifier> := <Start> <Continue>* (<Medial> <Continue>+)*
```

Identifiers are defined by assigning the sets of lexical classes defined as properties in the Unicode Character Database [UAX44]. These properties are shown in *Table 2*. The first column shows the property name, whose values are defined in the UCD. The second column provides a general description of the coverage for the associated class, the

derivational relationship between the ID properties and the XID properties, and an associated UnicodeSet notation for the class.

Table 2. Properties for Lexical Classes for Identifiers

Properties	General Description of Coverage
ID_Start	ID_Start characters are derived from the Unicode General_Category of uppercase letters, lowercase letters, titlecase letters, modifier letters, other letters, letter numbers, plus Other_ID_Start, minus Pattern_Syntax and Pattern_White_Space code points. In UnicodeSet notation: [\p{L}\p{Nl}\p{Other_ID_Start}-\p{Pattern_Syntax}-\p{Pattern_White_Space}]
XID_Start	XID_Start characters are derived from ID_Start as per *Section 5.1, NFKC Modifications*.
ID_Continue	ID_Continue characters include ID_Start characters, plus characters having the Unicode General_Category of nonspacing marks, spacing combining marks, decimal number, connector punctuation, plus Other_ID_Continue, minus Pattern_Syntax and Pattern_White_Space code points. In UnicodeSet notation: [\p{ID_Start}\p{Mn}\p{Mc}\p{Nd}\p{Pc}\p{Other_ID_Continue}-\p{Pattern_Syntax}-\p{Pattern_White_Space}]
XID_Continue	XID_Continue characters are derived from ID_Continue as per *Section 5.1, NFKC Modifications*. XID_Continue characters are also known simply as *Identifier Characters*, because they are a superset of the XID_Start characters.

Note that "other letters" includes ideographs. For more about the stability extensions, see *Section 2.5 Backward Compatibility*.

The innovations in the identifier syntax to cover the Unicode Standard include the following:

- Incorporation of proper handling of combining marks.
- Allowance for layout and format control characters, which should be ignored when parsing identifiers.

The XID_Start and XID_Continue properties are improved lexical classes that incorporate the changes described in *Section 5.1, NFKC Modifications*. They are recommended for most purposes, especially for security, over the original ID_Start and ID_Continue properties.

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

UAX31-R1-1. *Use definition UAX31-D1, setting Start and Continue to the properties XID_Start and XID_Continue, respectively, and leaving Medial empty.*

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

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

One such profile may be to use the contents of ID_Start and ID_Continue in place of XID_Start and XID_Continue, for backward compatibility.

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

- Start := XID_Start, plus some characters from Table 3
- Continue := Start + XID_Continue, plus some characters from Table 3b
- Medial := some characters from Table 3a

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

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

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

For example, consider a profile that adds subscript and superscript digits and operators in order to support technical notations, such as:

Context	Example Identifier
Assyriology	$\text{dun}_3{}^+$
Chemistry	$\text{Ca}^{2+}\_\text{concentration}$
Mathematics	x_{k+1} *or* $f^{(4)}$
Phonetics	daan^6

That profile may be described as adding the following set to XID_Continue:

$$[˹ ˻ ˼ ˺ ⁺ ₊ ⁼ ₌ ⁻ ₋ ⁰ ₀ ¹ ₁ ² ₂ ³ ₃ ⁴ ₄ ⁵ ₅ ⁶ ₆ ⁷ ₇ ⁸ ₈ ⁹ ₉].$$

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

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

For example, `\p{General_Category=Other_Number}` is the general category set containing the subscript and superscript digits. But it also includes the compatibility characters [⑴ ₁ ₁.],

which are not needed for technical notations, and are very likely inappropriate for identifiers —on multiple counts.

On the other hand, a language that allows currency symbols in identifiers could have `\p{General_Category=Currency_Symbol}` as a profile, since that property matches the intent.

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

Defining a profile by use of a property also needs to take account of the fact that unless the property is designed to be stable (such as XID_Continue), code points could be removed in a future version of Unicode. If the profile also needs stable identifiers (backwards compatible), then it must take additional measures. See *UAX31-R1b Stable Identifiers*.

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

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

UAX31-R1a. **Restricted Format Characters:** *This clause has been removed.*

The characters that were added when meeting this requirement are now part of the default; the contextual checks required by this requirement remain as part of the General Security Profile in Unicode Technical Standard #39, "Unicode Security Mechanisms" [UTS39].

UAX31-R1b. **Stable Identifiers:** *To meet this requirement, an implementation shall guarantee that identifiers are stable across versions of the Unicode Standard: that is, once a string qualifies as an identifier, it does so in all future versions of the Unicode Standard.*

> **Note:** The UAX31-R1b requirement is relevant when an identifier definition is based on property assignments from an unversioned reference to the Unicode Standard, as property assignments may change in a future version of the standard. It is typically achieved by using a small list of characters that qualified as identifier characters in some previous version of Unicode. See *Section 2.5, Backward Compatibility*. Where profiles are allowed, management of those profiles may also be required to guarantee backwards compatibility. Typically such management also uses a list of characters that qualified previously. Because of the stability policy [Stability], if an implementation meets either requirement UAX31-R1 or UAX31-R2 without declaring a profile, that implementation also meets requirement UAX31-R1b.

Example: Consider an identifier definition which uses UAX31-R1 default identifiers with a profile that adds digits (characters with General_Category=Nd) to the set *Start*, and uses an unversioned reference to the Unicode Character Database, with a minimum version of 5.2.0.

With property assignments from Unicode Version 5.2.0, both ꧚ (U+19DA) and A꧚ (U+0041, U+19DA) are valid identifiers under this definition: U+19DA has General_Category=Nd.

In Unicode Version 6.0.0, U+19DA has General_Category=No. The identifier A꧚ (U+0041, U+19DA) remains valid, because XID_Continue includes any characters that used to be XID_Continue. However, ꧚ is not a valid identifier, because U+19DA is no longer in the set [:Nd:].

In order to meet requirement UAX31-R1b, the definition would need to be changed to add to the set *Start* all characters that have the property General_Category=Nd in any version of Unicode starting from Unicode 5.2.0 and up to the version used by the implementation.

2.1 Combining Marks

Combining marks are accounted for in identifier syntax: a composed character sequence consisting of a base character followed by any number of combining marks is valid in an identifier. Combining marks are required in the representation of many languages, and the conformance rules in *Chapter 3, Conformance*, of [Unicode] require the interpretation of canonical-equivalent character sequences. The simplest way to do this is to require identifiers in the NFC format (or transform them into that format); see *Section 5, Normalization and Case*.

Enclosing combining marks (such as U+20DD..U+20E0) are excluded from the definition of the lexical class ID_Continue, because the composite characters that result from their composition with letters are themselves not normally considered valid constituents of these identifiers.

2.2 Modifier Letters

Modifier letters (General_Category=Lm) are also included in the definition of the syntax classes for identifiers. Modifier letters are often part of natural language orthographies and are useful for making word-like identifiers in formal languages. On the other hand, modifier symbols (General_Category=Sk), which are seldom a part of language orthographies, are excluded from identifiers. For more discussion of modifier letters and how they function, see [Unicode].

Implementations that tailor identifier syntax for special purposes may wish to take special note of modifier letters, as in some cases modifier letters have appearances, such as raised commas, which may be confused with common syntax characters such as quotation marks.

2.3 Layout and Format Control Characters

Certain Unicode characters are known as Default_Ignorable_Code_Points. These include variation selectors and characters used to control joining behavior, bidirectional ordering control, and alternative formats for display (having the General_Category value of Cf). The use of default-ignorable characters in identifiers is problematic, first because the effects they represent are stylistic or otherwise out of scope for identifiers, and second because

the characters themselves often have no visible display. It is also possible to misapply these characters such that users can create strings that look the same but actually contain different characters, which can create security problems. In environments where spoofing concerns are paramount, such as top-level domain names, identifiers should also be limited to characters that are case-folded and normalized with the NFKC_Casefold operation. For more information, see *Section 5, Normalization and Case* and *UTR #36: Unicode Security Considerations* [UTR36].

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

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

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

Notes:

- An implementation of UAX31-R4 and UAX31-R5 (Equivalent Case and Compatibility-Insensitive Identifiers) that ~~uses~~ compares identifiers under the *identifier caseless match* defined by D147 [Unicode], that is, canonical decomposition (NFD) followed by the ~~to~~NFKC_Casefold operation ~~for comparison~~ ignores Default_Ignorable_Code_Points.
- The Default_Ignorable_Code_Point property values are not guaranteed to be stable. However, the derivation of the NFKC_Casefold property will be changed if necessary to ensure that it remains stable for default identifiers. That means that ~~if the to~~NFKC_Casefold operation applied to a string with only characters in XID_Continue in a version of Unicode will have the same results in any future version of Unicode.

In addition, a standard profile is provided to exclude all Default_Ignorable_Code_Points; see *Section 7, Standard Profiles*. Note however that, even if Default_Ignorable_Code_Points are excluded, spoofing issues remain unless the mechanisms in Unicode Technical Standard #39, "Unicode Security Mechanisms" [UTS39] are utilized.

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

2.4 Specific Character Adjustments

Specific identifier syntaxes can be treated as tailorings (or *profiles*) of the generic syntax based on character properties. For example, SQL identifiers allow an underscore as an identifier continue, but not as an identifier start; C identifiers allow an underscore as either an identifier continue or an identifier start. Specific languages may also want to exclude the characters that have a Decomposition_Type other than Canonical or None, or to exclude some subset of those, such as those with a Decomposition_Type equal to Font.

There are circumstances in which identifiers are expected to more fully encompass words or phrases used in natural languages. For example, it is recommended that U+00B7 (·) MIDDLE DOT be allowed in medial positions in natural-language identifiers such as hashtags or search terms, because it is required for grammatical Catalan. For related issues about MIDDLE DOT, see *Section 5, Normalization and Case*.

For more natural-language identifiers, a profile should allow the characters in *Table 3*, *Table 3a*, and *Table 3b* in identifiers, unless there are compelling reasons not to. Most additions to identifiers are restricted to medial positions, such as U+00B7 (·) MIDDLE DOT, which is not needed as a trailing character in Catalan. These are listed in *Table 3a*. A few characters can also occur in final positions, and are listed in *Table 3b*. The contents of these tables may overlap.

In some environments even spaces and @ are allowed in identifiers, such as in SQL: *SELECT * FROM Employee Pension.*

Table 3. Optional Characters for Start

Code Point	Character	Name
0024	$	DOLLAR SIGN
005F	_	LOW LINE

Table 3a. Optional Characters for Medial

Code Point	Character	Name
0027	'	APOSTROPHE
002D	-	HYPHEN-MINUS
002E	.	FULL STOP
003A	:	COLON
00B7	·	MIDDLE DOT
058A	-	ARMENIAN HYPHEN
05F4	″	HEBREW PUNCTUATION GERSHAYIM
0F0B	·	TIBETAN MARK INTERSYLLABIC TSHEG
2010	-	HYPHEN
2019	'	RIGHT SINGLE QUOTATION MARK
2027	·	HYPHENATION POINT

30A0	=	KATAKANA-HIRAGANA DOUBLE HYPHEN

Table 3b. Optional Characters for Continue

Code Point	Character	Name
05F3	׳	HEBREW PUNCTUATION GERESH

In UnicodeSet notation, the characters in these tables are:

- Table 3: [\\$_]
- Table 3a: ['\\-.\\:··-‟-’· = ・]
- Table 3b: [׳]

In identifiers that allow for unnormalized characters, the compatibility equivalents of the characters listed in *Table 3*, *Table 3a*, and *Table 3b* may also be appropriate.

For more information on characters that may occur in words, and those that may be used in name validation, see Section 4, *Word Boundaries*, in [UAX29].

Some scripts are not in customary modern use, and thus implementations may want to exclude them from identifiers. These include historic and obsolete scripts, scripts used mostly liturgically, and regional scripts used only in very small communities or with very limited current usage. Some scripts also have unresolved architectural issues that make them currently unsuitable for identifiers. The scripts in *Table 4, Excluded Scripts* are recommended for exclusion from identifiers.

Table 4. Excluded Scripts

Property Notation	Description
\p{script=Aghb}	Caucasian Albanian
\p{script=Ahom}	Ahom
\p{script=Armi}	Imperial Aramaic
\p{script=Avst}	Avestan
\p{script=Bass}	Bassa Vah
\p{script=Bhks}	Bhaiksuki
\p{script=Brah}	Brahmi
\p{script=Bugi}	Buginese
\p{script=Buhd}	Buhid
\p{script=Cari}	Carian
\p{script=Chrs}	Chorasmian
\p{script=Copt}	Coptic
\p{script=Cpmn}	Cypro-Minoan
\p{script=Cprt}	Cypriot

`\p{script=Diak}`	Dives Akuru
`\p{script=Dogr}`	Dogra
`\p{script=Dsrt}`	Deseret
`\p{script=Dupl}`	Duployan
`\p{script=Egyp}`	Egyptian Hieroglyphs
`\p{script=Elba}`	Elbasan
`\p{script=Elym}`	Elymaic
`\p{script=Glag}`	Glagolitic
`\p{script=Gong}`	Gunjala Gondi
`\p{script=Gonm}`	Masaram Gondi
`\p{script=Goth}`	Gothic
`\p{script=Gran}`	Grantha
`\p{script=Hano}`	Hanunoo
`\p{script=Hatr}`	Hatran
`\p{script=Hluw}`	Anatolian Hieroglyphs
`\p{script=Hmng}`	Pahawh Hmong
`\p{script=Hung}`	Old Hungarian
`\p{script=Ital}`	Old Italic
`\p{script=Kawi}`	Kawi
`\p{script=Khar}`	Kharoshthi
`\p{script=Khoj}`	Khojki
`\p{script=Kits}`	Khitan Small Script
`\p{script=Kthi}`	Kaithi
`\p{script=Lina}`	Linear A
`\p{script=Linb}`	Linear B
`\p{script=Lyci}`	Lycian
`\p{script=Lydi}`	Lydian
`\p{script=Maka}`	Makasar
`\p{script=Mahj}`	Mahajani
`\p{script=Mani}`	Manichaean
`\p{script=Marc}`	Marchen
`\p{script=Medf}`	Medefaidrin
`\p{script=Mend}`	Mende Kikakui

`\p{script=Merc}`	Meroitic Cursive
`\p{script=Mero}`	Meroitic Hieroglyphs
`\p{script=Modi}`	Modi
`\p{script=Mong}`	Mongolian
`\p{script=Mroo}`	Mro
`\p{script=Mult}`	Multani
`\p{script=Nagm}`	Nag Mundari
`\p{script=Narb}`	Old North Arabian
`\p{script=Nand}`	Nandinagari
`\p{script=Nbat}`	Nabataean
`\p{script=Nshu}`	Nushu
`\p{script=Ogam}`	Ogham
`\p{script=Orkh}`	Old Turkic
`\p{script=Osma}`	Osmanya
`\p{script=Ougr}`	Old Uyghur
`\p{script=Palm}`	Palmyrene
`\p{script=Pauc}`	Pau Cin Hau
`\p{script=Perm}`	Old Permic
`\p{script=Phag}`	Phags-pa
`\p{script=Phli}`	Inscriptional Pahlavi
`\p{script=Phlp}`	Psalter Pahlavi
`\p{script=Phnx}`	Phoenician
`\p{script=Prti}`	Inscriptional Parthian
`\p{script=Rjng}`	Rejang
`\p{script=Runr}`	Runic
`\p{script=Samr}`	Samaritan
`\p{script=Sarb}`	Old South Arabian
`\p{script=Sgnw}`	SignWriting
`\p{script=Shaw}`	Shavian
`\p{script=Shrd}`	Sharada
`\p{script=Sidd}`	Siddham
`\p{script=Sind}`	Khudawadi
`\p{script=Sora}`	Sora Sompeng

\p{script=Sogd}	Sogdian
\p{script=Sogo}	Old Sogdian
\p{script=Soyo}	Soyombo
\p{script=Tagb}	Tagbanwa
\p{script=Takr}	Takri
\p{script=Tang}	Tangut
\p{script=Tglg}	Tagalog
\p{script=Tirh}	Tirhuta
\p{script=Tnsa}	Tangsa
\p{script=Toto}	Toto
\p{script=Ugar}	Ugaritic
\p{script=Vith}	Vithkuqi
\p{script=Wara}	Warang Citi
\p{script=Xpeo}	Old Persian
\p{script=Xsux}	Cuneiform
\p{script=Yezi}	Yezidi
\p{script=Zanb}	Zanabazar Square

Some characters used with recommended scripts may still be problematic for identifiers, for example because they are part of extensions that are not in modern customary use, and thus implementations may want to exclude them from identifiers. These include characters for historic and obsolete orthographies, characters used mostly liturgically, and in orthographies for languages used only in very small communities or with very limited current or declining usage. Some characters also have architectural issues that may make them unsuitable for identifiers. See *UTS #39, Unicode Security Mechanisms* [UTS39] for more information.

The scripts listed in *Table 5, Recommended Scripts* are generally recommended for use in identifiers. These are in widespread modern customary use, or are regional scripts in modern customary use by large communities.

Table 5. Recommended Scripts

Property Notation	Description
\p{script=Zyyy}	Common
\p{script=Zinh}	Inherited
\p{script=Arab}	Arabic
\p{script=Armn}	Armenian
\p{script=Beng}	Bengali
\p{script=Bopo}	Bopomofo

`\p{script=Cyrl}`	Cyrillic
`\p{script=Deva}`	Devanagari
`\p{script=Ethi}`	Ethiopic
`\p{script=Geor}`	Georgian
`\p{script=Grek}`	Greek
`\p{script=Gujr}`	Gujarati
`\p{script=Guru}`	Gurmukhi
`\p{script=Hang}`	Hangul
`\p{script=Hani}`	Han
`\p{script=Hebr}`	Hebrew
`\p{script=Hira}`	Hiragana
`\p{script=Kana}`	Katakana
`\p{script=Knda}`	Kannada
`\p{script=Khmr}`	Khmer
`\p{script=Laoo}`	Lao
`\p{script=Latn}`	Latin
`\p{script=Mlym}`	Malayalam
`\p{script=Mymr}`	Myanmar
`\p{script=Orya}`	Oriya
`\p{script=Sinh}`	Sinhala
`\p{script=Taml}`	Tamil
`\p{script=Telu}`	Telugu
`\p{script=Thaa}`	Thaana
`\p{script=Thai}`	Thai
`\p{script=Tibt}`	Tibetan

As of Unicode 10.0, there is no longer a distinction between aspirational use and limited use scripts, as this has not proven to be productive for the derivation of identifier-related classes used in security profiles. (See *UTS #39, Unicode Security Mechanisms* [UTS39].) Thus the aspirational use scripts in *Table 6, Aspirational Use Scripts* have been recategorized as Limited Use and moved to *Table 7, Limited Use Scripts*.

Table 6. Aspirational Use Scripts (Withdrawn)

Property Notation	Description
intentionally blank	

Modern scripts that are in more limited use are listed in *Table 7, Limited Use Scripts*. To avoid security issues, some implementations may wish to disallow the limited-use scripts in identifiers. For more information on usage, see the Unicode Locale project [CLDR].

Table 7. Limited Use Scripts

Property Notation	Description
\p{script=Adlm}	Adlam
\p{script=Bali}	Balinese
\p{script=Bamu}	Bamum
\p{script=Batk}	Batak
\p{script=Cakm}	Chakma
\p{script=Cans}	Canadian Aboriginal Syllabics
\p{script=Cham}	Cham
\p{script=Cher}	Cherokee
\p{script=Hmnp}	Nyiakeng Puachue Hmong
\p{script=Java}	Javanese
\p{script=Kali}	Kayah Li
\p{script=Lana}	Tai Tham
\p{script=Lepc}	Lepcha
\p{script=Limb}	Limbu
\p{script=Lisu}	Lisu
\p{script=Mand}	Mandaic
\p{script=Mtei}	Meetei Mayek
\p{script=Newa}	Newa
\p{script=Nkoo}	Nko
\p{script=Olck}	Ol Chiki
\p{script=Osge}	Osage
\p{script=Plrd}	Miao
\p{script=Rohg}	Hanifi Rohingya
\p{script=Saur}	Saurashtra
\p{script=Sund}	Sundanese
\p{script=Sylo}	Syloti Nagri
\p{script=Syrc}	Syriac
\p{script=Tale}	Tai Le
\p{script=Talu}	New Tai Lue

`\p{script=Tavt}`	Tai Viet
`\p{script=Tfng}`	Tifinagh
`\p{script=Vaii}`	Vai
`\p{script=Wcho}`	Wancho
`\p{script=Yiii}`	Yi

This is the recommendation as of the current version of Unicode; as new scripts are added to future versions of Unicode, characters and scripts may be added to Tables *4*, *5*, and *7*. Characters may also be moved from one table to another as more information becomes available.

There are a few special cases:

- The Common and Inherited script values [\p{script=Zyyy}\p{script=Zinh}] are used widely with other scripts, rather than being scripts per se. See also the Script_Extensions property in the Unicode Character Database [UAX44].
- The Unknown script \p{script=Zzzz} is used for Unassigned characters.
- Braille \p{script=Brai} consists only of symbols
- Katakana_Or_Hiragana \p{script=Hrkt} is empty. This value was used in earlier versions, but is no longer used.
- With respect to the scripts Balinese, Cham, Ol Chiki, Vai, Kayah Li, and Saurashtra, there may be large communities of people speaking an associated language, but the script itself is not in widespread use. However, there are significant revival efforts.
- Bopomofo is used primarily in education.

For programming language identifiers, normalization and case have a number of important implications. For a discussion of these issues, see *Section 5, Normalization and Case*.

2.5 Backward Compatibility

Unicode General_Category values are kept as stable as possible, but they can change across versions of the Unicode Standard. The bulk of the characters having a given value are determined by other properties, and the coverage expands in the future according to the assignment of those properties. In addition, the Other_ID_Start property provides a small list of characters that qualified as ID_Start characters in some previous version of Unicode solely on the basis of their General_Category properties, but that no longer qualify in the current version.

The Other_ID_Start property includes characters such as the following:

```
U+2118 ( ℘ ) SCRIPT CAPITAL P
U+212E ( ℮ ) ESTIMATED SYMBOL
U+309B ( ゛ ) KATAKANA-HIRAGANA VOICED SOUND MARK
U+309C ( ゜ ) KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK
```

Similarly, the Other_ID_Continue property adds a small list of characters that qualified as ID_Continue characters in some previous version of Unicode solely on the basis of their General_Category properties, but that no longer qualify in the current version.

The Other_ID_Continue property includes characters such as the following:

U+1369 ETHIOPIC DIGIT ONE...U+1371 ETHIOPIC DIGIT NINE
U+00B7 (·) MIDDLE DOT
U+0387 (·) GREEK ANO TELEIA
U+19DA (᧚) NEW TAI LUE THAM DIGIT ONE

The exact list of characters covered by the Other_ID_Start and Other_ID_Continue properties depends on the version of Unicode. For more information, see Unicode Standard Annex #44, "Unicode Character Database" [UAX44].

The Other_ID_Start and Other_ID_Continue properties are thus designed to ensure that the Unicode identifier specification is backward compatible. Any sequence of characters that qualified as an identifier in some version of Unicode will continue to qualify as an identifier in future versions.

If a specification tailors the Unicode recommendations for identifiers, then this technique can also be used to maintain backwards compatibility across versions.

3 Immutable Identifiers

The disadvantage of working with the lexical classes defined previously is the storage space needed for the detailed definitions, plus the fact that with each new version of the Unicode Standard new characters are added, which an existing parser would not be able to recognize. In other words, the recommendations based on that table are not upwardly compatible.

This problem can be addressed by turning the question around. Instead of defining the set of code points that are allowed, define a small, fixed set of code points that are reserved for syntactic use and allow everything else (including unassigned code points) as part of an identifier. All parsers written to this specification would behave the same way for all versions of the Unicode Standard, because the classification of code points is fixed forever.

The drawback of this method is that it allows "nonsense" to be part of identifiers because the concerns of lexical classification and of human intelligibility are separated. Human intelligibility can, however, be addressed by other means, such as usage guidelines that encourage a restriction to meaningful terms for identifiers. For an example of such guidelines, see the XML specification by the W3C, Version 1.0 5th Edition or later [XML].

By increasing the set of disallowed characters, a reasonably intuitive recommendation for identifiers can be achieved. This approach uses the full specification of identifier classes, as of a particular version of the Unicode Standard, and permanently disallows any characters not recommended in that version for inclusion in identifiers. All code points unassigned as of that version would be allowed in identifiers, so that any future additions to the standard would already be accounted for. This approach ensures both upwardly compatible identifier stability and a reasonable division of characters into those that do and do not make human sense as part of identifiers.

With or without such fine-tuning, such a compromise approach still incurs the expense of implementing large lists of code points. While they no longer change over time, it is a matter of choice whether the benefit of enforcing somewhat word-like identifiers justifies their cost.

Alternatively, one can use the properties described below and allow all sequences of characters to be identifiers that are neither Pattern_Syntax nor Pattern_White_Space. This has the advantage of simplicity and small tables, but allows many more "unnatural" identifiers.

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

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

- Pattern_White_Space=True
- Pattern_Syntax=True
- General_Category=Private_Use, Surrogate, or Control
- Noncharacter_Code_Point=True

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

> **Note:** The expectation from an implementation meeting requirement UAX31-R2 Immutable Identifiers is that it will never change its definition of identifiers; in particular, that it will not switch to UAX31-R1 Default Identifiers. However, the downsides of normalization issues and the inapplicability of measures guarding against spoofing attacks may warrant such a change in definition. In such circumstances, a profile should be used to extend XID_Start and XID_Continue to cover likely existing usages. See *Section 3.3, Language Evolution*, in Unicode Technical Standard #55, "Unicode Source Code Handling" [UTS55].

In its profile, a specification can define identifiers to be more in accordance with the Unicode identifier definitions at the time the profile is adopted, while still allowing for strict immutability. For example, an implementation adopting a profile after a particular version of Unicode is released (such as Unicode 5.0) could define the profile as follows:

1. All characters satisfying *UAX31-R1 Default Identifiers* according to Unicode 5.0
2. Plus all code points unassigned in Unicode 5.0 that do not have the property values specified in *UAX31-R2 Immutable Identifiers*.

This technique allows identifiers to have a more natural format—excluding symbols and punctuation already defined—yet also provides absolute code point immutability.

Immutable identifiers are intended for those cases (like XML) that cannot update across versions of Unicode, and do not require information about normalization form, or properties such as General_Category and Script. Immutable identifers that allow unassigned characters cannot provide for normalization forms or these properties, which means that they:

- cannot be compared for NFC, NFKC, or case-insensitive equality
- are unsuitable for restrictions such as those in UTS #39

For best practice, a profile disallowing unassigned characters should be provided where possible.

Specifications should also include guidelines and recommendations for those creating new identifiers. Although *UAX31-R2 Immutable Identifiers* permits a wide range of characters, as a best practice identifiers should be in the format NFKC, without using any unassigned characters. For more information on NFKC, see Unicode Standard Annex #15, "Unicode Normalization Forms" [UAX15].

4 Whitespace and Syntax

Most programming languages have a concept of whitespace as part of their lexical structure, as well as some set of characters that are disallowed in identifiers but have syntactic use, such as arithmetic operators. Beyond general programming languages, there are also many circumstances where software interprets patterns that are a mixture of literal characters, whitespace, and syntax characters. Examples include regular expressions, Java collation rules, Excel or ICU number formats, and many others. In the past, regular expressions and other formal languages have been forced to use clumsy combinations of ASCII characters for their syntax. As Unicode becomes ubiquitous, some of these will start to use non-ASCII characters for their syntax: first as more readable optional alternatives, then eventually as the standard syntax.

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

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

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

UAX31-R3. **Pattern_White_Space and Pattern_Syntax Characters:** *To meet this requirement, an implementation shall meet both UAX31-R3a and UAX31-R3b.*

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

4.1 Whitespace

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

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

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

[UAX9]. See Unicode Technical Standard #55, "Unicode Source Code Handling" [UTS55].

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

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

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

UAX31-R3a-1. *Use Pattern_White_Space characters as the set of characters interpreted as whitespace in parsing, as follows:*

1. *A sequence of one or more of any of the following characters shall be interpreted as a sequence of one or more end of line:*
    a. U+000A (line feed)
    b. U+000B (vertical tabulation)
    c. U+000C (form feed)
    d. U+000D (carriage return)
    e. U+0085 (next line)
    f. U+2028 LINE SEPARATOR
    g. U+2029 PARAGRAPH SEPARATOR
2. *The Pattern_White_Space characters with the property Default_Ignorable_Code_Point shall be treated as ignorable format controls; they shall be allowed in the contexts UAX31-I1, UAX31-I2, and UAX31-I3 defined in Section 4.1.3, Contexts for Ignorable Format Controls, where their insertion shall have no effect on the meaning of the program.*
3. *All other characters in Pattern_White_Space shall be interpreted as horizontal space.*

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

> **Note:** The characters to be treated as ignorable format controls under item 2 of UAX31-R3a-1 are U+200E LEFT-TO-RIGHT MARK and U+200F RIGHT-TO-LEFT MARK. The characters to be treated as horizontal space under item 3 of UAX31-R3a-1 are U+0020 SPACE and U+0009 (horizontal tabulation, TAB).

> **Note:** The characters LEFT-TO-RIGHT MARK and RIGHT-TO-LEFT MARK are two of the Implicit Directional Marks defined by *Section 2.6, Implicit Directional Marks*, in Unicode Standard Annex #9, "Unicode Bidirectional Algorithm" [UAX9]. The third one, ARABIC LETTER MARK, is used far less frequently than the others, even in Arabic text; its behavior differs subtly from RIGHT-TO-LEFT MARK in ways that are not

usually relevant to the ordering of source code. If it is added to the set of whitespace characters by a profile, it is interpreted as an ignorable format control.

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

4.1.1 Bidirectional Ordering

Requirement UAX31-R3a is relevant even for languages that do not use immutable identifiers, or that have lexical structure outside of the categories of syntax and whitespace characters. In particular, the set of Pattern_White_Space characters is chosen to make it possible to correct bidirectional ordering issues that can arise in a wide range of programming languages, visually obfuscating the logic of expressions. In the absence of higher-level protocols (see Section 4.3, *Higher-Level Protocols*, in [UAX9]), tokens may be visually reordered by the Unicode Bidi Algorithm in bidirectional source text, producing a visual result that conveys a different logical intent. To remedy that, two implicit directional marks are among Pattern_White_Space characters; if these can be freely inserted between tokens, implicit directional marks *consistent with the paragraph direction* can be used to ensure that the visual order of tokens matches their logical order.

Example: Consider the following two lines:

(1) `x + tav == 1`

(2) `x + 1 == תו`

Internally, they are the same except that the ASCII identifier `tav` in line (1) is replaced by the Hebrew identifier `תו` in line (2). However, with a plain text display (with left-to-right paragraph direction) the user will be misled, thinking that line (2) is a comparison between `(x + 1)` and `תו`, whereas it is actually a comparison between `(x + תו)` and `1`. The misleading rendering of (2) occurs because the directionality of the identifier `תו` influences subsequent weakly-directional tokens; inserting a left-to-right mark after the identifier `תו` stops it from influencing the remainder of the line, and thus yields a better rendering in plain text with left-to-right paragraph direction, as demonstrated in the following table, wherein characters whose ordering is affected by that identifier have been highlighted.

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

Section 5.2, Conversion to Plain Text, in Unicode Technical Standard #55, "Unicode Source Code Handling" [UTS55], specifies an algorithm for the automatic insertion of LRM characters.

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

4.1.2 Required Spaces

Since the implicit directional marks are nonspacing, where a syntax requires a sequence of spaces (such as between identifiers), it should require that at least one of those be neither LEFT-TO-RIGHT MARK nor RIGHT-TO-LEFT MARK. The visual appearance would otherwise be too confusing to readers: "else⟨LRM⟩if" would be seen by the user as "elseif" but parsed by the compiler as "else if", whereas "else⟨LRM⟩ if" would be seen and parsed as "else if" and be harmless.

4.1.3 Contexts for Ignorable Format Controls

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

UAX31-I1. Adjacent to lexical horizontal space (within a sequence of lexical horizontal spaces, or at the start or end of such a sequence).

> **Example:** Between the following keywords separated by a space:
>
> else ⟨LRM⟩if
>
> **Note:** The phrase "lexical horizontal space" refers to characters that are not merely in the set of horizontal space characters, but are also in a context where they are lexically spaces. For instance, it does not include horizontal space characters in string literals. Implementations should permit these characters in string literals, but in such a literal, their insertion has an effect on the meaning of the program, as they are then present in the string represented by that literal.

UAX31-I2. As optional space, that is, wherever horizontal space could be inserted without changing the meaning of the program.

> **Example:** Before the plus sign in the following arithmetic expression:
>
> x⟨LRM⟩+1

UAX31-I3. At the start and end of a lexical line.

> **Example:** Before the word import in the following line of Python:
>
> ⟨LRM⟩import unicodedata
>
> **Note:** As is the case for UAX31-I1, the start and end of a "lexical line" in UAX31-I3 does not include the start and end of a line in a multiline string literal, respectively. This context is distinct from UAX31-I2 in languages where leading or trailing spaces are meaningful.

4.2 Syntax

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

Examples of characters with syntactic use include:

- decimal marks in numeric literals
- arithmetic operators, such as +, -, *, /
- parentheses and other brackets
- characters in comment delimiters, such as #, /*, --, or ⊙
- quotation marks delimiting strings
- characters such as \ introducing escape sequences

It is useful to bound the set of characters with syntactic use. This makes it possible to build tools that handle source code, but do not validate it, such as syntax highlighters, in a forward-compatible way; see Unicode Technical Standard #55, "Unicode Source Code Handling" [UTS55]. It further provides a stable set of characters that can be used for user-defined operators. In addition, this allows for backward compatibility of literals (including patterns), as described in *Section 4.3, Pattern Syntax*.

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

UAX31-R3b-1. *Use Pattern_Syntax characters as the set of characters with syntactic use. The following sets shall be disjoint:*

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

UAX31-R3b-2. *Declare that it uses a **profile** of UAX31-R3b-1 and define that profile with a precise specification of the characters that are added to or removed from the set of code points defined by the Pattern_Syntax property.*

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

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

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

Characters outside of those allowed in identifiers, those treated as whitespace, and those with syntactic use can still be allowed in a program, for instance, as part of string literals or comments.

4.2.1 User-Defined Operators

Some programming languages allow for user-defined operators. When meeting requirement UAX31-R3b, the set of characters that can be allowed in operators is limited; however, that leaves open the exact definition of operators. In order to avoid ambiguities in

lexical analysis, operators should not be allowed to contain characters that may be found at the beginning of an identifier or literal; for instance, +1 or −x should not be operators.

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

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

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

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

> **Note:** The set of Pattern_Syntax characters, which is the default for characters with syntactic use, contains some emoji. Implementations may wish to remove them, either to allow for their use in identifiers, or to reduce potential confusion arising from ⚽ being an operator but 🏈 not being one. This may be done using the standard profile for UAX31-R3b Pattern_Syntax Characters defined in *Section 7.2, Emoji Profile*.
>
> Nonspacing marks are included in Continue because they are part of the representation for many operators, such as some of the negated operators.
>
> Unassigned code points are not characters; they are therefore excluded by this definition.

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

4.3 Pattern Syntax

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

> In version 1.0 of program X, '≉' is a reserved syntax character; that is, it does not perform an operation, and it needs to be quoted. In this example, '\' *quotes* the next character; that is, it causes it to be treated as a literal instead of a syntax character. In version 2.0 of program X, '≉' is given a real meaning—for example, "uppercase the subsequent characters".
>
> - The pattern abc...\≉...xyz works on both versions 1.0 and 2.0, and refers to the literal character because it is quoted in both cases.
> - The pattern abc...≉...xyz works on version 2.0 and uppercases the following characters. On version 1.0, the engine (rightfully) has no idea what to do with ≉.

Rather than silently fail (by ignoring ≈ or turning it into a literal), it has the opportunity to signal an error.

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

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

a<SPACE>b → x<ZERO WIDTH SPACE>y + z;

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

'a\u0020b' → 'x\u200By' + z;

5 Normalization and Case

This section discusses issues that must be taken into account when considering normalization and case folding of identifiers in programming languages or scripting languages. Using normalization avoids many problems where apparently identical identifiers are not treated equivalently. Such problems can appear both during compilation and during linking—in particular across different programming languages. To avoid such problems, programming languages can normalize identifiers before storing or comparing them. Generally if the programming language has case-sensitive identifiers, then Normalization Form C is appropriate; whereas, if the programming language has case-insensitive identifiers, then Normalization Form KC is more appropriate.

Implementations that take normalization and case into account have two choices: to treat variants as equivalent, or to disallow variants.

UAX31-R4. **Equivalent Normalized Identifiers:** *To meet this requirement, an implementation shall specify the Normalization Form and shall provide a precise specification of the characters that are excluded from normalization, if any. If the Normalization Form is NFKC, the implementation shall apply the modifications in Section 5.1, NFKC Modifications, given by the properties XID_Start and XID_Continue. Except for identifiers containing excluded characters, any two identifiers that have the same Normalization Form shall be treated as equivalent by the implementation.*

UAX31-R5. **Equivalent Case-Insensitive Identifiers:** *To meet this requirement, an implementation shall specify either simple or full case folding, and adhere to the Unicode specification for that folding. Any two identifiers that have the same case-folded form shall be treated as equivalent by the implementation.*

UAX31-R6. **Filtered Normalized Identifiers:** *To meet this requirement, an implementation shall specify the Normalization Form and shall provide a precise specification of the characters that are excluded from normalization, if any. If the Normalization Form is NFKC, the implementation shall apply the modifications in Section 5.1, NFKC Modifications, given*

by the properties XID_Start and XID_Continue. Except for identifiers containing excluded characters, allowed identifiers must be in the specified Normalization Form.

> **Note:** For requirement UAX31-R6, filtering involves disallowing any characters in the set \p{NFKC_QuickCheck=No}, or equivalently, disallowing \P{isNFKC}.

UAX31-R7. **Filtered Case-Insensitive Identifiers:** *To meet this requirement, an implementation shall specify either simple or full case folding, and adhere to the Unicode specification for that folding. Except for identifiers containing excluded characters, allowed identifiers must be in the specified case folded form.*

> **Note:** For requirement UAX31-R7 with full case folding, filtering involves disallowing any characters in the set \p{Changes_When_Casefolded}.

As of Unicode 5.2, an additional string transform is available for use in matching identifiers: toNFKC_Casefold(S). See **UAX31-R5** in *Section 3.13, Default Case Algorithms* in [Unicode]. That operation case folds and normalizes a string, and also removes default-ignorable code points. It can be used to support an implementation of UAX31-R4 and UAX31-R5 *Equivalent Case and Compatibility-Insensitive Identifiers.* In order to implement requirement UAX31-R4, canonical decomposition must be applied prior to the toNFKC_Casefold operation. The resulting equivalence relation between identifiers is an *identifier caseless match*, see definition D147 of [Unicode]. There is a corresponding boolean property, Changes_When_NFKC_Casefolded, which can be used to support an implementation of *Filtered Case and Compatibility-Insensitive Identifiers*. The NFKC_Casefold character mapping property and the Changes_When_NFKC_Casefolded property are described in Unicode Standard Annex #44, "Unicode Character Database" [UAX44].

> **Note:** In mathematically oriented programming languages that make distinctive use of the Mathematical Alphanumeric Symbols, such as U+1D400 MATHEMATICAL BOLD CAPITAL A, an application of NFKC must filter characters to exclude characters with the property value Decomposition_Type=Font.

5.1 NFKC Modifications

Where programming languages are using NFKC to fold differences between characters, they need the following modifications of the identifier syntax from the Unicode Standard to deal with the idiosyncrasies of a small number of characters. These modifications are reflected in the XID_Start and XID_Continue properties.

5.1.1 Modifications for Characters that Behave Like Combining Marks

Certain characters are not formally combining characters, although they behave in most respects as if they were. In most cases, the mismatch does not cause a problem, but when these characters have compatibility decompositions, they can cause identifiers not to be closed under Normalization Form KC. In particular, the following four characters are included in XID_Continue and not XID_Start:

> U+0E33 THAI CHARACTER SARA AM
> U+0EB3 LAO VOWEL SIGN AM
> U+FF9E HALFWIDTH KATAKANA VOICED SOUND MARK
> U+FF9F HALFWIDTH KATAKANA SEMI-VOICED SOUND MARK

5.1.2 Modifications for Irregularly Decomposing Characters

U+037A GREEK YPOGEGRAMMENI and certain Arabic presentation forms have irregular compatibility decompositions and are excluded from both XID_Start and XID_Continue. It is recommended that all Arabic presentation forms be excluded from identifiers in any event, although only a few of them must be excluded for normalization to guarantee identifier closure.

5.1.3 Identifier Closure Under Normalization

With these amendments to the identifier syntax, all identifiers are closed under all four Normalization Forms. This means that for any string S, the implications shown in *Figure 5* hold.

Figure 5. Normalization Closure

```
                        isIdentifier(toNFD(S))
                        isIdentifier(toNFC(S))
isIdentifier(S) →
                        isIdentifier(toNFKD(S))
                        isIdentifier(toNFKC(S))
```

Identifiers are also closed under case operations. For any string S (with exceptions involving a single character), the implications shown in *Figure 6* hold.

Figure 6. Case Closure

```
                        isIdentifier(toLowercase(S))
isIdentifier(S) →       isIdentifier(toUppercase(S))
                        isIdentifier(toFoldedcase(S))
```

The one exception for casing is U+0345 COMBINING GREEK YPOGEGRAMMENI. In the very unusual case that U+0345 is at the start of S, U+0345 is not in XID_Start, but its uppercase and case-folded versions are. In practice, this is not a problem because of the way normalization is used with identifiers.

The reverse implication is true for canonical equivalence but *not* true in the case of compatibility equivalence:

Figure 7. Reverse Normalization Closure

```
isIdentifier(toNFD(S))
                            →  isIdentifier(S)
isIdentifier(toNFC(S))

isIdentifier(toNFKD(S))
                            ↛  isIdentifier(S)
isIdentifier(toNFKC(S))
```

There are many characters for which the reverse implication is not true for compatibility equivalence, because there are many characters counting as symbols or non-decimal numbers—and thus outside of identifiers—whose compatibility equivalents are letters or decimal numbers and thus in identifiers. Some examples are shown in *Table 8*.

Table 8. Compatibility Equivalents to Letters or Decimal Numbers

Code Points	GC	Samples	Names

2070	No	º	SUPERSCRIPT ZERO
20A8	Sc	Rs	RUPEE SIGN
2116	So	№	NUMERO SIGN
2120..2122	So	ˢᴹ..™	SERVICE MARK..TRADE MARK SIGN
2460..2473	No	①..⑳	CIRCLED DIGIT ONE..CIRCLED NUMBER TWENTY
3300..33A6	So	㌀..km̈	SQUARE APAATO..SQUARE KM CUBED

If an implementation needs to ensure both directions for compatibility equivalence of identifiers, then the identifier definition needs to be tailored to add these characters.

For canonical equivalence the implication is true in both directions. `isIdentifier(toNFC(S))` if and only if `isIdentifier(S)`.

There were two exceptions before Unicode 5.1, as shown in *Table 9*. If an implementation needs to ensure full canonical equivalence of identifiers, then the identifier definition must be tailored so that these characters have the same value, so that either both isIdentifier(S) and isIdentifier(toNFC(S)) are true, or so that both values are false.

Table 9. Canonical Equivalence Exceptions Prior to Unicode 5.1

isIdentifier(toNFC(S))=True	isIdentifier(S)=False	Different in
02B9 (ʹ) MODIFIER LETTER PRIME	0374 (ʹ) GREEK NUMERAL SIGN	XID and ID
00B7 (·) MIDDLE DOT	0387 (·) GREEK ANO TELEIA	XID alone

Those programming languages with case-insensitive identifiers should use the case foldings described in *Section 3.13, Default Case Algorithms*, of [Unicode] to produce a case-insensitive normalized form.

When source text is parsed for identifiers, the folding of distinctions (using case mapping or NFKC) must be delayed until after parsing has located the identifiers. Thus such folding of distinctions should not be applied to string literals or to comments in program source text.

The Unicode Standard supports case folding with normalization, with the function toNFKC_Casefold(X). See definition UAX31-R5 in *Section 3.13, Default Case Algorithms* in [Unicode] for the specification of this function and further explanation of its use.

5.2 Case and Stability

The alphabetic case of the initial character of an identifier is used as a mechanism to distinguish syntactic classes in some languages like Prolog, Erlang, Haskell, Clean, and Go. For example, in Prolog and Erlang, variables must begin with capital letters (or underscores) and atoms must not. There are some complications in the use of this mechanism.

For such a casing distinction in a programming language to work with unicameral writing systems (such as Kanji or Devanagari), another mechanism (such as underscores) needs to substitute for the casing distinction.

Casing stability is also an issue for bicameral writing systems. The assignment of General_Category property values, such as gc=Lu, is not guaranteed to be stable, nor is the assignment of characters to the broader properties such as Uppercase. So these property values cannot be used by themselves, without incorporating a mechanism that preserves backward compatibility, such as is done for Unicode identifiers in *Section 2.5 Backward Compatibility*. That is, the implementation would maintain its own list of special inclusions and exclusions that require updating for each new version of Unicode.

Alternatively, a programming language specification can use the operation specified in Case Folding Stability as the basis for its casing distinction. That operation *is* guaranteed to be stable. That is, one can use a casing distinction such as the following:

1. S is a **variable** if S begins with an underscore.
2. Otherwise, produce S' = toCasefold(toNFKC(S))
    a. S is a **variable** if firstCodePoint(S) ≠ firstCodePoint(S'),
    b. otherwise S is an **atom**.

This test can clearly be optimized for the normal cases, such as initial ASCII. It is also recommended that identifiers be in NFKC format, which makes the detection even simpler.

5.2.1 Edge Cases for Folding

In Unicode 8.0, the Cherokee script letters have been changed from gc=Lo to gc=Lu, and corresponding lowercase letters (gc=Ll) have been added. This is an unusual pattern; typically when case pairs are added, existing letters are changed from gc=Lo to gc=Ll, and new corresponding uppercase letters (gc=Lu) are added. In the case of Cherokee, it was felt that this solution provided the most compatibility for existing implementations in terms of font treatment.

The downside of this approach is that the Cherokee characters, when case-folded, will convert as necessary to the pre-8.0 characters, namely to the uppercase versions. This folding is unlike that of any other case-mapped characters in Unicode. Thus the case-folded version of a Cherokee string will contain uppercase letters instead of lowercase letters. Compatibility with fonts for the current user community was felt to be more important than the confusion introduced by this edge case of case folding, because Cherokee programmatic identifiers would be rare.

The upshot is that when it comes to identifiers, implementations should never use the General_Category or Lowercase or Uppercase properties to test for casing conditions, nor use toUppercase(), toLowercase(), or toTitlecase() to fold or test identifiers. Instead, they should instead use Case_Folding or NFKC_CaseFold.

6 Hashtag Identifiers

Hashtag identifiers have become very popular in social media. They consist of a number sign in front of some string of characters, such as #emoji. The actual composition of allowable Unicode hashtag identifiers varies between vendors. It has also become common for hashtags to include emoji characters, without a clear notion of exactly which characters are included.

This section presents a syntax that can be used for parsing Unicode hashtag identifiers for increased interoperability.

UAX31-D2. **Default Hashtag Identifier Syntax:**

```
<Hashtag-Identifier> := <Start> <Continue>* (<Medial> <Continue>+)*
```

When parsing hashtags in flowing text, it is recommended that an extended Hashtag only be recognized when there is no Continue character before a Start character. For example, in "abc#def" there would be no hashtag, while there would be in "abc #def" or "abc.#def".

UAX31-R8. **Extended Hashtag Identifiers:** *To meet this requirement, to determine whether a string is a hashtag identifier an implementation shall choose either UAX31-R8-1 or UAX31-R8-2.*

UAX31-R8-1. *Use definition UAX31-D2, setting:*

1. Start := [# ＃ ♯]
   - U+0023 NUMBER SIGN
   - U+FE5F SMALL NUMBER SIGN
   - U+FF03 FULLWIDTH NUMBER SIGN
   - (These are # and its compatibility equivalents.)
2. Medial is currently empty, but can be used for customization.
3. Continue := XID_Continue, plus Extended_Pictographic, Emoji_Component, and "_", "-", "+", minus Start characters.
   - Note the subtraction of # characters.
   - This is expressed in UnicodeSet notation as:
     [\p{XID_Continue}\p{Extended_Pictographic}\p{Emoji_Component}[-+_]-[# ＃ ♯]]

UAX31-R8-2. *Declare that it uses a **profile** of UAX31-R8-1 as in UAX31-R1.*

The emoji properties are from the corresponding version of [UTS51]. The version of the emoji properties is tied to the version of the Unicode Standard, starting with Version 11.0.

The techniques mentioned in Section 2.5 Backward Compatibility may be used where stability between successive versions is required.

Comparison and matching should be done after converting to NFKC_CF format. Thus #MötleyCrüe should match #MÖTLEYCRÜE and other variants.

Implementations may choose to add characters in *Table 3a, Optional Characters for Medial* to **Medial** and *Table 3b, Optional Characters for Continue* to **Continue** for better identifiers for natural languages.

7 Standard Profiles

Two standard profiles for default identifiers are provided to cater to common patterns of use observed in programming languages with less restrictive identifier syntaxes, including those that use UAX31-R2 default identifiers: the inclusion of characters suitable for mathematical usage in identifiers, and the inclusion of emoji in identifiers.

These profiles are associated with profiles for requirements UAX31-R3b.

Further, a standard profile is provided to exclude default-ignorable code points from identifiers. Having no visible effect in most contexts, these characters can lead to spoofing issues; see *Section 2.3, Layout and Format Control Characters*.

For guidance on the applicability of these profiles to programming languages, see Unicode Technical Standard #55, "Unicode Source Code Handling" [UTS55].

7.1 Mathematical Compatibility Notation Profile

The Mathematical Compatibility Notation Profile for default identifiers consists of the addition of the set [:ID_Compat_Math_Start:] to the set *Start*, and the set [:ID_Compat_Math_Continue:] to the set *Continue*, in definition UAX31-D1.

> **Note:** The set [:ID_Compat_Math_Start:] comprises ∂, ∇, and their mathematical style variants, as well as ∞. The set [:ID_Compat_Math_Continue:] comprises [:ID_Compat_Math_Start:], as well as subscript and superscript digits and signs with mathematical use.

It is associated with a profile for UAX31-R3b, which consists of removing the characters in [[:Pattern_Syntax:] - [:ID_Compat_Math_Continue:]] from the set of characters with syntactic use (these are the characters ∂, ∇, and ∞).

> **Note:** While *supporting* these characters is recommended for some computer languages because they can be beneficial in some applications, these characters, like many others characters that are allowed in default identifiers, are discouraged in general use, as they are confusing to most readers. See Unicode Technical Standard #55, "Unicode Source Code Handling" [UTS55].

7.2 Emoji Profile

The Emoji Profile for default identifiers provides for the inclusion of emoji characters and sequences in identifiers. A large subset of emoji are already supported in some programming languages, but this profile provides a mechanism for treating them consistently as part of the lexical structure of a language.

The Emoji Profile for default identifiers consists of:

1. The addition of the RGI emoji set defined by ED-27 in Unicode Technical Standard #51, "Unicode Emoji" [UTS51] for a given version of Unicode to the sets *Start* and *Continue* in definition UAX31-D1.
2. The removal of the code point U+FE0E VARIATION SELECTOR-15 (the Text Presentation Selector) from the set *Continue*.

> **Note:** The Emoji Profile requires the use of character sequences, rather than individual code points, in the sets *Start* and *Continue* defined by UAX31-D1. When using this profile, U+002A asterisk (*), U+203C double exclamation mark (‼), or U+263A white smiling face (☺) are not legal identifiers, but the sequences (U+002A, U+FE0F, U+20E3) ⃣, (U+203C, U+FE0F) ‼, and (U+263A, U+FE0F) ☺ are allowed in identifiers. This would require some changes to lexers: when they hit a character that starts an emoji sequence they will (logically) switch to a different mechanism for parsing.

The Emoji Profile includes characters that are in Pattern_Syntax; it is therefore associated with a profile for UAX31-R3b, which consists of replacing each emoji character of a certain subset of [:Pattern_Syntax:] by its ***text presentation sequence*** (ED-8a):

1. Remove the characters in the set [[:Pattern_Syntax:]&[:Emoji_Presentation:]] from the set of characters with syntactic use.
2. For all C in [[:Pattern_Syntax:]&[:Emoji_Presentation:]], add the sequence consisting of C followed by U+FE0E VARIATION SELECTOR-15 (the Text Presentation Selector) to the set of characters with syntactic use.

In addition, in order to avoid lexical ambiguities between identifiers and operators, the Emoji Profile includes a profile for UAX31-R3c, which consists of the removal of the character U+FE0F VARIATION SELECTOR-16 (the Emoji Presentation Selector) from the set *Continue*.

Example: Consider a language that meets requirements UAX31-R3b and UAX31-R3c with no profile. U+2615 HOT BEVERAGE (☕) is a character with syntactic use, and therefore it is an operator. When meeting these requirements with the Emoji Profile, U+2615 HOT BEVERAGE (☕) is not a character with syntactic use (which allows it to be an identifier character) and ☕ is not a valid operator. However, the sequence U+2615 U+FE0F (☕) is added to the set of characters with syntactic use, and therefore ☕ is a valid operator.

This change means that if some of the Pattern_Syntax characters with the Emoji_Presentation property were in syntactic use (e.g., in operators) prior to adopting the Emoji Profile, they become identifiers once the profile is adopted, but can be turned back into operators by adding U+FE0E VARIATION SELECTOR-15, allowing for a migration path.

Of course, if a programming language only uses a subset of the Pattern_Syntax characters that does not include these characters, no action needs to be taken.

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

The emoji sequences require 98 default-ignorable characters:

- U+200D ZERO WIDTH JOINER (also known as ZWJ)
- U+FE0F VARIATION SELECTOR-16 (also known as Emoji Presentation Selector)
- U+E0020..U+E007F 96 TAG characters

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

Consider the following examples, in a language that meets requirement UAX31-R1 with both the Emoji Profile and the Default Ignorable Exclusion Profile:

Sequence	Appearance	Legal Identifier?	Reason
A+ZWJ+B	AB	No	ZWJ is not part of an emoji sequence

Sequence	Appearance	Legal Identifier?	Reason
U+1F408 + ZWJ + U+2B1B	🐈⬛	Yes	ZWJ is part of an emoji sequence (for *black cat*)
BIG + U+1F408 + ZWJ + U+2B1B	BIG🐈⬛	Yes	

7.3 Default-Ignorable Exclusion Profile

The default-ignorable exclusion profile for default identifiers consists of the exclusion of the code points with property Default_Ignorable_Code_Point from the sets *Start* and *Continue* in definition UAX31-D1.

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

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

Acknowledgments

Mark Davis is the author of the initial version and has added to and maintained the text of this annex. Robin Leroy has assisted in updating it starting with Version 15.0.

The attendees of the Source Code Working Group meetings assisted with the substantial changes made in Versions 15.0 and 15.1: Peter Constable, Elnar Dakeshov, Mark Davis, Barry Dorrans, Steve Dower, Michael Fanning, Asmus Freytag, Dante Gagne, Rich Gillam, Manish Goregaokar, Tom Honermann, Jan Lahoda, Nathan Lawrence, Robin Leroy, Chris Ries, Markus Scherer, Richard Smith.

Thanks to Eric Muller, Asmus Freytag, Lisa Moore, Julie Allen, Jonathan Warden, Kenneth Whistler, David Corbett, Klaus Hartke, Martin Dürst, Deborah Anderson, Steve Downey, Ned Holbrook, Corentin Jabot, 梁海 Liang Hai, Jens Maurer, and Hubert Tong for feedback on this annex.

References

For references for this annex, see Unicode Standard Annex #41, "Common References for Unicode Standard Annexes."

Migration

Version 15.1

Requirement UAX31-R1a Restricted Format Characters has been withdrawn.

If implementations that claimed conformance to UAX31-R1a wish to retain the contextual checks for ZWJ and ZWNJ, they should refer to the General Security Profile in Unicode

Technical Standard #39, "Unicode Security Mechanisms" [UTS39].

In previous versions, requirement UAX31-R3 Pattern_White_Space and Pattern_Syntax Characters did not require any particular interpretation of whitespace characters. It now specifies which characters are to be treated as line terminators, horizontal space, and ignorable format controls. The meaning of syntactic use has also been clarified.

Implementations that claim conformance to UAX31-R3 should check that they interpret the characters in Pattern_White_Space as described in UAX31-R3a Pattern_White_Space Characters, and that their use of Pattern_Syntax characters is consistent with UAX31-R3b Pattern_Syntax Characters.

Version 15.0

In previous versions, the note explaining how to implement requirement UAX31-R7 Filtered Case-Insensitive Identifiers with full case folding referred to the wrong property, and the requirement itself incorrectly refered to Normalization Form rather than case folded form.

Implementations that claim conformance to UAX31-R7 should check that they use the correct property.

Version 13.0

Version 13.0 changed the structure of Table 4. Excluded Scripts significantly, dropping conditions that were not based on script. Implementations that were based on Table 4 should refer to *UTS #39, Unicode Security Mechanisms* [UTS39] for additional restrictions.

Version 11.0

Version 11.0 refines the use of ZWJ in identifiers (adding some restrictions and relaxing others slightly), and broadens the definition of hashtag identifiers somewhat. For details, see the Modifications.

Version 9.0

In previous versions, the text favored the use of XID_Start and XID_Continue, as in the following paragraph. However, the formal definition used ID_Start and ID_Continue.

> The XID_Start and XID_Continue properties are improved lexical classes that incorporate the changes described in *Section 5.1, NFKC Modifications*. They are recommended for most purposes, especially for security, over the original ID_Start and ID_Continue properties.

In version 9.0, that is swapped and the X versions are stated explicitly in the formal definition. This affects just the following characters.

```
037A ; GREEK YPOGEGRAMMENI
0E33 ; THAI CHARACTER SARA AM
0EB3 ; LAO VOWEL SIGN AM
309B ; KATAKANA-HIRAGANA VOICED SOUND MARK
309C ; KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK
FC5E..FC63 ; ARABIC LIGATURE SHADDA WITH SUPERSCRIPT ALEF ISOLATED FORM
FDFA ; ARABIC LIGATURE SALLALLAHOU ALAYHE WASALLAM
FDFB ; ARABIC LIGATURE JALLAJALALOUHOU
FE70 ; ARABIC FATHATAN ISOLATED FORM
FE72 ; ARABIC DAMMATAN ISOLATED FORM
FE74 ; ARABIC KASRATAN ISOLATED FORM
```

```
FE76 ; ARABIC FATHA ISOLATED FORM
FE78 ; ARABIC DAMMA ISOLATED FORM
FE7A ; ARABIC KASRA ISOLATED FORM
FE7C ; ARABIC SHADDA ISOLATED FORM
FE7E ; ARABIC SUKUN ISOLATED FORM
FF9E ; HALFWIDTH KATAKANA VOICED SOUND MARK
FF9F ; HALFWIDTH KATAKANA SEMI-VOICED SOUND MARK
```

Implementations that wish to maintain conformance to the older recommendation need only declare a profile that uses ID_Start and ID_Continue instead of XID_Start and XID_Continue.

Version 9.0 splits the older Table 3 from Version 8.0 into 3 parts.

Current Tables	Unicode 8.0
Table 3, Optional Characters for Start	*Table 3, Candidate Characters for Inclusion in ID_Continue*
Table 3a, Optional Characters for Medial	
Table 3b, Optional Characters for Continue	*only outlined in text*

Version 6.1

Between Unicode Versions 5.2, 6.0 and 6.1, Table 5 was split in three. In Version 6.1, the resulting tables were renumbered for easier reference. The titles and links remain the same, for stability.

The following shows the correspondences:

Current Tables	Unicode 6.0	Unicode 5.2
Table 5, Recommended Scripts	5a	5
Table 6, Aspirational Use Scripts		
Table 7, Limited Use Scripts	5b	
Table 8, Compatibility Equivalents to Letters or Decimal Numbers	6	6
Table 9, Canonical Equivalence Exceptions Prior to Unicode 5.1	7	7

Modifications

The following summarizes modifications from the previously published version of this annex.

Revision 40

- **Proposed Update** for Unicode 16.0.
- *Section 2.3, Layout and Format Control Characters*, and *Section 5, Normalization and Case*: clarified that NFD must be applied before toNFKC_Casefold in order to

correctly meet requirements UAX31-R4 and UAX-R5 with NFKC and full case folding, and added a reference to definition D147 of the Unicode Standard.

Revision 39

- **Reissued** for Unicode 15.1.
- Renamed from Unicode Identifier and Pattern Syntax to Unicode Identifiers and Syntax.
- Section 1.4, Conformance
  - Added a note clarifying that unversioned references to the UCD are allowed, but should have a minimum.
- Section 2, Default Identifiers
  - Clarified that profiles are described as amendments that can be combined.
  - Clarified that spoofing issues should be dealt with using the mechanisms in UTS #39.
  - Removed requirement UAX31-R1a, which remains as part of the security mechanisms in UTS #39.
  - Emphasized the relevance of UAX31-R1b when using unversioned references to the UCD, and clarified that the defaults meet this requirement.
  - Added an example of a profile that fails to meet UAX31-R1b, describing how it could be changed to meet that requirement.
  - Added a note on the stability of NFKC_Casefold for XID_Continue characters.
- Section 3, Immutable Identifiers
  - Clarified the expectation of immutability for implementations that use immutable identifiers; added a reference to guidance in UTS #55 for the cases where it is necessary to move away from an immutable definition.
- Section 4, Whitespace and Syntax
  - Split UAX31-R3 into UAX31-R3a and UAX31-R3b, and correspondingly split the discussions of whitespace and syntax.
  - Significantly expanded the discussion of whitespace, which now covers the interpretation of whitespace characters as line terminators, horizontal space, or ignorable format controls.
  - Clarified the meaning of *characters with syntactic use*.
  - Added a requirement UAX31-R3c for operator identifiers.
  - Reorganized the section to separate the considerations that are specific to patterns from those that apply to computer languages in general.
- Section 7, Standard Profiles
  - Added standard profiles corresponding to common usages of identifiers outside of the XID_Continue space.
- Minor editorial corrections.

Modifications for previous versions are listed in those respective versions.