Technical Reports

<div align="center">

Proposed Update Unicode® Standard Annex #31

UNICODE IDENTIFIER AND PATTERN SYNTAX

</div>

Version	Unicode 12.0.0
Editors	Mark Davis (markdavis@google.com)
Date	2018–10–18
This Version	http://www.unicode.org/reports/tr31/tr31-30.html
Previous Version	http://www.unicode.org/reports/tr31/tr31-29.html
Latest Version	http://www.unicode.org/reports/tr31/
Latest Proposed Update	http://www.unicode.org/reports/tr31/proposed.html
Revision	30

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

*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 an 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 user of normalization and case insensitivity with identifiers, expanding on a section that was originally in Unicode Standard Annex #15, "Unicode Normalization Forms" [UAX15].

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		
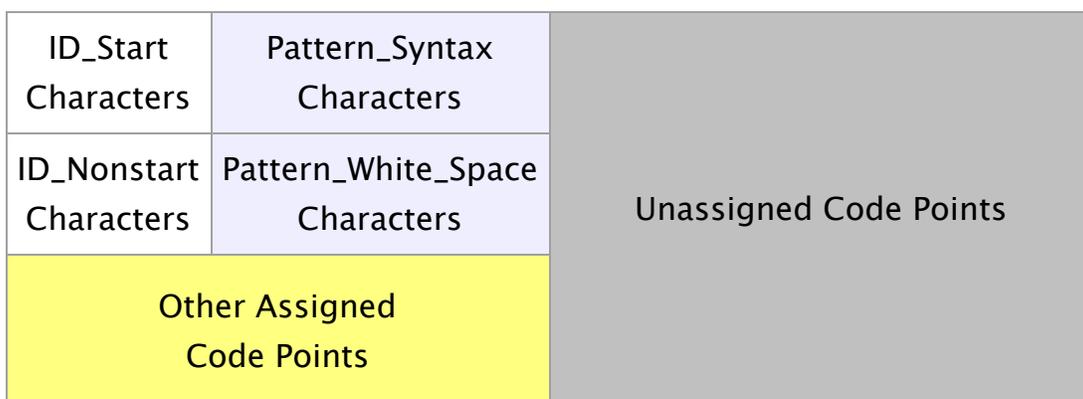
The set consisting of the union of *ID_Start* and *ID_Nonstart* characters is known as *Identifier Characters* and has the property *ID_Continue*. The *ID_Nonstart* set is defined as the set difference *ID_Continue* minus *ID_Start*: it is not a formal Unicode property. While lexical rules are traditionally 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.
- 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 format 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.

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

- R1. Default Identifiers
- R1a. Restricted Format Characters
- R1b. Stable Identifiers
- R2. Immutable Identifiers
- R3. Pattern_White_Space and Pattern_Syntax Characters
- R4. Equivalent Normalized Identifiers
- R5. Equivalent Case-Insensitive Identifiers
- R6. Filtered Normalized Identifiers
- R7. Filtered Case-Insensitive Identifiers

- R8. Hashtag Identifiers

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 for each category used by the syntax can be customized according to the requirements of the environment.

If such extensions include characters from Pattern_White_Space or Pattern_Syntax, then such identifiers do not conform to an unmodified 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 set 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 set 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 set 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 use definition UAX31-D1, setting Start and Continue to the properties XID_Start and XID_Continue, respectively, and leaving Medial empty.*

- *Alternatively, it shall declare that it uses a **profile** and define that profile with a precise specification of the characters that are added to or removed from Start, Continue, and Medial and/or provide a list of additional constraints on identifiers.*

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*

UAX31-R1a. ***Restricted Format Characters:*** *To meet this requirement, an implementation shall define a profile for UAX31-R1 which allows format characters as described in Section 2.3, Layout and Format Control Characters.*

- *An implementation may further restrict the context for ZWJ or ZWNJ, such as by limiting the scripts allowed or limiting the occurrence of ZWJ or ZWNJ to specific*

character combinations, if a clear specification for such a further restriction is
supplied.

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.*

> **Note:** The UAX31-R1b requirement is typically achieved by using grandfathered
> characters. 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 grandfathered characters.

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
recommendation is to permit them in identifiers only in special cases, listed below. The use
of default-ignorable characters in identifiers is problematical, first because the effects they
represent are stylistic or otherwise out of scope for identifiers, and second because the
characters themselves often have no visible display. It is also possible to misapply these
characters such that users can create strings that look the same but actually contain
different characters, which can create security problems. In such environments, 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].

Variation selectors, in particular, including standardized variants and sequences from the Ideographic Variation Database, are not included in the default identifier syntax. These are subject to the same considerations as for other Default_Ignorable_Code_Points listed above. Because variation selectors request a difference in display but do not guarantee it, they do not work well in general-purpose identifiers. The NFKC_Casefold operation can be used to remove them, along with other Default_Ignorable_Code_Points. However, in some environments it may be useful to retain variation sequences in the display form for identifiers. For more information, see *Section 1.3, Display Format* .

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

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

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

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

Thus in such circumstances, an implementation should allow the following Join_Control characters in the limited contexts specified in **A1**, **A2**, and **B** below.

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

There are also two global conditions incorporated in each of **A1**, **A2**, and **B**:

- **Script Restriction.** In each of the following cases, the specified sequence must only consist of characters from a single script (after ignoring *Common* and *Inherited* script characters).
- **Normalization.** In each of the following cases, the specified sequence must be in NFC format. (To test an identifier that is not required to be in NFC, first transform into NFC format and then test the condition.)

Implementations may also impose tighter restrictions than provided below, in order to eliminate some other circumstances where the characters either have no visual effect or the effect has no semantic importance.

A1. Allow ZWNJ in the following context:

Breaking a cursive connection. That is, in the context based on the Joining_Type property, consisting of:

- A Left-Joining or Dual-Joining character, followed by zero or more Transparent characters, followed by a ZWNJ, followed by zero or more Transparent characters, followed by a Right-Joining or Dual-Joining character

This corresponds to the following regular expression (in Perl-style syntax): **/$LJ $T* ZWNJ $T* $RJ/**
where:

    $T = \p{Joining_Type=Transparent}
    $RJ = [\p{Joining_Type=Dual_Joining}\p{Joining_Type=Right_Joining}]
    $LJ = [\p{Joining_Type=Dual_Joining}\p{Joining_Type=Left_Joining}]

For example, consider Farsi <*Noon, Alef, Meem, Heh, Alef, Farsi Yeh*>. Without a ZWNJ, it translates to "names", as shown in the first row; with a ZWNJ between Heh and Alef, it means "a letter", as shown in the second row of *Figure 2*.

Figure 2. Persian Example with ZWNJ

Appearance	Code Points	Abbreviated Names
نامهای	0646 + 0627 + 0645 + 0647 + 0627 + 06CC	NOON + ALEF + MEEM + HEH + ALEF + FARSI YEH
نامهای	0646 + 0627 + 0645 + 0647 + 200C + 0627 + 06CC	NOON + ALEF + MEEM + HEH + ZWNJ + ALEF + FARSI YEH

A2. Allow ZWNJ in the following context:

In a conjunct context. That is, a sequence of the form:

- A Letter, followed by a Virama, followed by a ZWNJ, (optionally preceded or followed by certain nonspacing marks), followed by a Letter

This corresponds to the following regular expression (in Perl-style syntax): **/$L $M* $V $M_1* ZWNJ $M_1* $L2/**
where:

    $L = \p{General_Category=Letter}
    $V = \p{Canonical_Combining_Class=Virama}
    $M = \p{General_Category=Mn}
    $M_1 = [\p{General_Category=Mn}&\p{CCC≠0}]

For example, the Malayalam word for *eyewitness* is shown in *Figure 3*. The form without the ZWNJ in the second row is incorrect in this case.

Figure 3. Malayalam Example with ZWNJ

Appearance	Code Points	Abbreviated Names
	0D26 + 0D43 + 0D15 +	DA + VOWEL SIGN VOCALIC R +

ദക്സാക്ഷി	0D4D + 200C + 0D38 + 0D3E + 0D15 + 0D4D + 0D37 + 0D3F	KA + VIRAMA + ZWNJ + SA + VOWEL SIGN AA + KA + VIRAMA + SSA + VOWEL SIGN I
ദൃക്സാക്ഷി	0D26 + 0D43 + 0D15 + 0D4D + 0D38 + 0D3E + 0D15 + 0D4D + 0D37 + 0D3F	DA + VOWEL SIGN VOCALIC R + KA + VIRAMA + SA + VOWEL SIGN AA + KA + VIRAMA + SSA + VOWEL SIGN I

B. **Allow ZWJ in the following context:**

In a conjunct context. That is, a sequence of the form:

- A Letter, followed by a Virama, followed by a ZWJ (optionally preceded or followed by certain nonspacing marks), and not followed by a character of type Indic_Syllabic_Category=Vowel_Dependent

This corresponds to the following regular expression (in Perl-style syntax): **/$L $M* $V M_1$* ZWJ (?!$D)/**
where:

$L= \p{General_Category=Letter}
$V = \p{Canonical_Combining_Class=Virama}
$M = \p{General_Category=Mn}
M_1$ = [\p{General_Category=Mn}&\p{CCC≠0}]
$D = \p{Indic_Syllabic_Category=Vowel_Dependent}

For example, the Sinhala word for the country 'Sri Lanka' is shown in the first row of *Figure 4*, which uses both a space character and a ZWJ. Removing the space results in the text shown in the second row of *Figure 4*, which is still legible, but removing the ZWJ completely modifies the appearance of the 'Sri' cluster and results in the unacceptable text appearance shown in the third row of *Figure 4*.

Figure 4. Sinhala Example with ZWJ

Appearance	Code Points	Abbreviated Names
ශ්‍රී ලංකා	0DC1 + 0DCA + 200D + 0DBB + 0DD3 + 0020 + 0DBD + 0D82 + 0D9A + 0DCF	SHA + VIRAMA + ZWJ + RA + VOWEL SIGN II + SPACE + LA + ANUSVARA + KA + VOWEL SIGN AA
ශ්‍රීලංකා	0DC1 + 0DCA + 200D + 0DBB + 0DD3 + 0DBD + 0D82 + 0D9A + 0DCF	SHA + VIRAMA + ZWJ + RA + VOWEL SIGN II + LA + ANUSVARA + KA + VOWEL SIGN AA

ශ්‍රී ලංකා	0DC1 + 0DCA + 0DBB + 0DD3 + 0020 + 0DBD + 0D82 + 0D9A + 0DCF	SHA + VIRAMA + RA + VOWEL SIGN II + SPACE + LA + ANUSVARA + KA + VOWEL SIGN AA

Implementations that allow emoji characters in identifiers should also normally allow emoji sequences. These are defined in **ED-17, emoji sequence** in [UTS51]. In particular, that means allowing ZWJ characters, emoji presentation selector (U+FE0F), and TAG characters, but only in the particular defined contexts described in [UTS51].

2.3.1 Limitations

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

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

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

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
200C	\u200C	ZERO WIDTH NON-JOINER*
2010	-	HYPHEN
2019	'	RIGHT SINGLE QUOTATION MARK
2027	‧	HYPHENATION POINT
30A0	=	KATAKANA-HIRAGANA DOUBLE HYPHEN
30FB	・	KATAKANA MIDDLE DOT

Table 3b. Optional Characters for Continue

Code Point	Character	Name
05F3	׳	HEBREW PUNCTUATION GERESH
200D	\u200D	ZERO WIDTH JOINER*

The characters marked with an asterisk in *Table 3a* and *Table 3b* are Join_Control characters, discussed in *Section 2.3, Layout and Format Control Characters*.

In UnicodeSet syntax, the characters in these tables are:

- Table 3: [\$_]
- Table 3a: ['\-.\:·‿-‟"‑'· = ·]
- Table 3b: [\u200D ']

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 characters are not in modern customary use, and thus implementations may want to exclude them from identifiers. These include characters in 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 set of characters in *Table 4, Candidate Characters for Exclusion from Identifiers* provides candidates of these, plus some inappropriate technical blocks.

Table 4. Candidate Characters for Exclusion from Identifiers

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=Copt}	Coptic
\p{script=Cprt}	Cypriot
\p{script=Dogr}	Dogra
\p{script=Dsrt}	

	Deseret
\p{script=Dupl}	Duployan
\p{script=Egyp}	Egyptian Hieroglyphs
\p{script=Elba}	Elbasan
\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=Khar}	Kharoshthi
\p{script=Khoj}	Khojki
\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=Mero}`	Meroitic Hieroglyphs
`\p{script=Merc}`	Meroitic Cursive
`\p{script=Modi}`	Modi
`\p{script=Mong}`	Mongolian
`\p{script=Mroo}`	Mro
`\p{script=Mult}`	Multani
`\p{script=Narb}`	Old North Arabian
`\p{script=Nbat}`	Nabataean
`\p{script=Nshu}`	Nushu
`\p{script=Ogam}`	Ogham
`\p{script=Orkh}`	Old Turkic
`\p{script=Osma}`	Osmanya
`\p{script=Palm}`	Palmyrene
`\p{script=Pauc}`	Pau Cin Hau
`\p{script=Perm}`	Old Permic
`\p{script=Phag}`	Phags Pa
`\p{script=Phlp}`	Psalter Pahlavi
`\p{script=Phli}`	Inscriptional 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=Shrd}`	Sharada

\p{script=Shaw}	Shavian
\p{script=Sidd}	Siddham
\p{script=Sind}	Khudawadi
\p{script=Sogd}	Sogdian
\p{script=Sogo}	Old Sogdian
\p{script=Sora}	Sora Sompeng
\p{script=Soyo}	Soyombo
\p{script=Tagb}	Tagbanwa
\p{script=Tglg}	Tagalog
\p{script=Tang}	Tangut
\p{script=Takr}	Takri
\p{script=Tirh}	Tirhuta
\p{script=Ugar}	Ugaritic
\p{script=Wara}	Warang Citi
\p{script=Xpeo}	Old Persian
\p{script=Xsux}	Cuneiform
\p{script=Zanb}	Zanabazar Square
[\p{Extender=True} & \p{Joining_Type=Join_Causing}]	0640 (-) ARABIC TATWEEL 07FA (ˍ) NKO LAJANYALAN
\p{Default_Ignorable_Code_Point}	Default Ignorable Code Points See *Section 2.3, Layout and Format Control Characters*
\p{block=Combining_Diacritical_Marks_for_Symbols} \p{block=Musical_Symbols} \p{block=Ancient_Greek_Musical_Notation} \p{block=Phaistos_Disc}	

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=Hani}`	Han
`\p{script=Hang}`	Hangul
`\p{script=Hebr}`	Hebrew
`\p{script=Hira}`	Hiragana
`\p{script=Knda}`	Kannada
`\p{script=Kana}`	Katakana
`\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 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
`\p{script=Cham}`	Cham
`\p{script=Cher}`	Cherokee
`\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=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 (used historically in Unicode, but no longer.)
- 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. These are called *grandfathered* characters.

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 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

*Alternatively, it shall declare that it uses a **profile** and define that profile with a precise specification of the characters that are added to or removed from the sets of code points defined by these properties.*

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 Pattern Syntax

There are 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 for use in patterns. This follows the recommendations that the Unicode Consortium has made regarding completely stable identifiers, and the practice that is seen in XML 1.0, 5th Edition or later [XML]. (In particular, the Unicode Consortium is committed to not allocating characters suitable for identifiers in the range U+2190..U+2BFF, which is being used by XML 1.0, 5th Edition.)

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

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

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

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

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

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;

UAX31-R3. *Pattern_White_Space and Pattern_Syntax Characters:* *To meet this requirement, an implementation shall use Pattern_White_Space characters as all and only those characters interpreted as whitespace in parsing, and shall use Pattern_Syntax characters as all and only those characters with syntactic use.*

*Alternatively, it shall declare that it uses a **profile** and define that profile with a precise specification of the characters that are added to or removed from the sets of code points defined by these properties.*

Note: When meeting this requirement, all characters except those that have the Pattern_White_Space or Pattern_Syntax properties are available for use as identifiers or literals.

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 Normalization Form.*

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

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 *Equivalent Case and Compatibility-Insensitive Identifiers*. 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(S) →	isIdentifier(toNFD(S)) isIdentifier(toNFC(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(S) →	isIdentifier(toLowercase(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(toNFC(S))	→ isIdentifier(S)
isIdentifier(toNFKD(S)) isIdentifier(toNFKC(S))	↛ isIdentifier(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	⁰	SUPERSCRIPT ZERO
20A8	Sc	Rs	RUPEE SIGN
2116	So	№	NUMERO SIGN
2120..2122	So	℠..™	SERVICE MARK..TRADE MARK SIGN
2460..2473	No	①..⑳	CIRCLED DIGIT ONE..CIRCLED NUMBER TWENTY
3300..33A6	So	㌀..㎦	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 grandfathering mechanism, 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 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 Emoji, Emoji_Component, and "_", "-", "+", minus Start characters.*
   - *Note the subtraction of # characters.*
   - *This is expressed in set notation as:*
     [\p{XID_Continue}\p{Extended_Pictographic}\p{Emoji_Component}[-+_]-[# ﹟ ＃]]

   - *Alternatively, it shall declare that it uses a **profile** 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 grandfathering 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.

Acknowledgments

Mark Davis is the author of the initial version and has added to and maintained the text of this annex.

Thanks to Eric Muller, Asmus Freytag, Lisa Moore, Julie Allen, Jonathan Warden, Kenneth Whistler, and Martin Duerst for feedback on this annex.

References

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

Migration

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 30

- **Proposed Update** for Unicode 12.0.
- **Section 2.3** Layout and Format Control Characters
  - Tightened up the context for A2 by requiring a $Letter at the end of the sequence.

Revision 29

- **Reissued** for Unicode 11.0.
- Section 2.3 Layout and Format Control Characters
  - Modified B to disallow ZWJ followed by Indic_Syllabic_Category = Vowel_Dependent.
  - Modified A2 and B to allow the Virama to be preceded or followed by certain combining marks.
  - Added clarifying notes.
- Section 2.4 Specific Character Adjustments
  - Added new Version 11.0 scripts to Table 4, Candidate Characters for Exclusion from Identifiers and Table 7, Limited Use Scripts.
  - Mongolian was moved from Table 7 to Table 4, pending the resolution of architectural issues that make it currently unsuitable for identifiers.
- Section 6 Hashtag Identifiers
  - Updated to include the broader Extended_Pictographic instead of Emoji (for future-proofing) and the set of Emoji_Component characters instead of the less precise "plus any other characters that can occur in emoji sequences (such as ZWJ)".
  - Added the common "-" and "+".
  - Changed text to note that the versions of Unicode and Emoji will be synchronized from Version 11.0 onwards.
  - Added recommendation for comparison with NFKC_CF.
- General
  - Changed to update set syntax, eg from [: … :] to the more standard \p{…}

Revision 28 being a proposed update, only changes between revisions 27 and 29 are noted here.

Modifications for previous versions are listed in those respective versions.