Technical Reports

L2/07-360

Proposed Update to

Unicode Standard Annex #31

IDENTIFIER AND PATTERN SYNTAX

Version	Unicode 5.1 draft 3
Authors	Mark Davis (mark.davis@google.com)
Date	2007-09-18
This Version	http://www.unicode.org/reports/tr31/tr31-8.html
Previous Version	http://www.unicode.org/reports/tr31/tr31-7.html
Latest Version	http://www.unicode.org/reports/tr31/
Revision	8

Summary

This annex describes specifications for recommended defaults for the use of Unicode in the definitions of 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*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time.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].

Contents

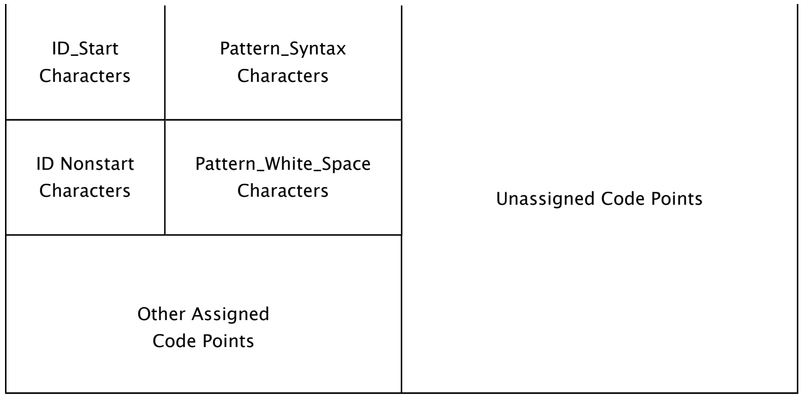
1 Introduction

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. 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 recommended default for the definition of identifier syntax. These guidelines are no more complex than current rules in the common programming languages, except that they include more characters of different types. 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 provide 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 stable 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



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*. While lexical rules are traditionally expressed in terms of the latter, the discussion here is simplified by referring to disjoint categories.

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. However, neither will ever decrease.

In successive versions of Unicode, the only allowed changes of characters from one of the above classes to another are those listed with a + 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].

Programming Languages. 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.

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 so as to encompass words used in natural languages may add characters identified in *Section 4, Word Boundaries*, of [UAX29] with the property values Katakana, ALetter, and MidLetter, plus characters described in the notes at the end of that section.

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].

1.1 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 at any Level shall identify the version of this specification and the version of the Unicode Standard.*

UAX31-C2. *An implementation claiming conformance to Level 1 of this specification shall describe which of the following it observes:*

- *R1 Default Identifiers*
- *R1a Restricted Format Characters*
- *R1b Stable Identifiers*
- *R2 Alternative Identifiers*
- *R3 Pattern_White_Space and Pattern_Syntax Characters*

2 Default Identifier Syntax

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 following with 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.

D1. Default Identifier Syntax

```
<identifier> := <ID_Start> <ID_Continue>*
```

Identifiers are defined by the sets of lexical classes defined as properties in the Unicode Character Database. These properties are shown in *Table 2*.

Table 2. Lexical Classes for Identifiers

Properties	Alternates	General Description of Coverage
`ID_Start`	`XID_Start`	Characters having the Unicode General_Category of uppercase letters (Lu), lowercase letters (Ll), titlecase letters (Lt), modifier letters (Lm), other letters (Lo), letter numbers (Lu), plus stability extensions. Note that "other letters" includes ideographs. In set notation, this is [[:L:][:Nl:]] plus stability extensions.
`ID_Continue`	`XID_Continue`	All of the above, plus characters having the Unicode General_Category of nonspacing marks (Mn), spacing combining marks (Mc), decimal number (Nd), connector punctuations (Pc), plus stability extensions. In set notation, this is [[:L:][:Nl:][:Mn:][:Mc:][:Nd:]] plus stability extensions. These are also known simply as *Identifier Characters*, because they are a superset of the `ID_Start`.

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.

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.

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 Layout and Format Control Characters

Certain Unicode characters are used to control joining behavior, bidirectional ordering control, and alternative formats for display. These have the General_Category value of Cf. Unlike space characters or other delimiters, they do not indicate word, line, or other unit boundaries.

While it is possible to ignore these characters in determining identifiers, the recommendation is to not ignore them and to not permit them in identifiers except in special cases. ~~This is because of the possibility for confusion between two visually identical strings; see [UTR36]. Some possible exceptions are the ZWJ and ZWNJ in certain contexts, such as between certain characters in Indic words.~~ The use of format characters in identifiers is problematical because the formatting effects they represent are normally just stylistic or otherwise out of scope for identifiers. It is also possible to misapply format characters such that users can create strings that look the same but actually contain different characters, which can create security problems; see *UTR# 36: Unicode Security Considerations* [UTR36].

For these reasons format characters are normally excluded from Unicode identifiers. However, visible distinctions created by certain format characters (particularly the *joiner controls*) are necessary and make necessary distinctions in certain languages. A blanket exclusion of format characters makes it impossible to create identifiers based on certain words or phrases in those languages. Identifier systems that attempt to provide more natural representations of terms in modern, customary use should consider allowing these characters, but limited to particular contexts where they are necessary.

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 (eg, illustration of half-forms or joining forms in isolation).

The goal for such a restriction of format characters to particular contexts is to

a. allow the use of these characters where required in normal text
b. exclude as many cases as possible where no visible distinction results
c. be simple enough to be easily implemented with standard mechanisms such as regular expressions

Thus for such circumstances, an implementation may choose to allow the following characters, but only in very limited contexts as specified below.

- Joiner controls (U+200C ZERO WIDTH NON-JOINER [ZWNJ] and U+200D ZERO WIDTH JOINER [ZWJ]) and

- Mongolian separators (U+202F NARROW NO-BREAK SPACE [NNBSP] and U+180E MONGOLIAN VOWEL SEPARATOR in the Unicode recommendations for identifiers.

Implementations may further restrict the contexts in which these characters may be used. For more information, see UTR# 36: **Unicode Security** Considerations [UTR36].

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).

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.

Comparion. Typically the identifiers with and without these characters should not compare as equivalent. However, in certain language-specific cases, such as in Sinhala, they should compare as equivalent. See Section 2.3 Specific Character Adjustments.

[Note to reviewers: the following text is fairly detailed; should it be an appendix?]

[Note to reviewers: The use of Virama below does not allow for letters with combining marks (eg KA + NUKTA + VIRAMA + ZWJ). Should this be added as well?

The characters and their contexts are given by the following:

A. ZWNJ in the following contexts:

1. **Breaking a cursive connection.** That is, in the context based on the Arabic Shaping property, consisting of:
   - A Left-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 character
   - This corresponds to the following regular expression (in Perl-style syntax): **/$L** $T* **ZWNJ** $T* **$R/** where:
     - $T = [:Joining_Type=Transparent:]
     - $R = [[:Joining_Type=Dual_Joining:] [: Joining_Type=Right_Joining:]]
     - $L = [[:Joining_Type=Dual_Joining:] [:Joining_Type=Left_Joining:]]
   - **Example:** Farsi <Noon, Alef, Meem, Heh, Alef, Farsi Yeh>. Without a ZWNJ, it translates to "names"; with a ZWNJ between Heh and Alef, it means "a letter". Figure 2 illustrates this.

Figure 2.

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

2. **In a conjunct context.** That is, a sequence of the form:
   - A Letter, followed by a Virama, followed by a ZWNJ, followed by an Letter
   - This corresponds to the following regular expression (in Perl-style syntax): **/$L $V ZWNJ $L/** where:
     - $L = [:General_Category=Letter:]
     - $V = [:Canonical_Combining_Class=Virama:]
   - **Example:** In Khmer, U+17A2 U+200D*(ZWNJ)* U+17CA U+17B7 U+17A2 U+17BB U+17CA U+17C7 [□□□□□□□] is a case where the first TRIISAP needs to be escaped, but the second does not (as there is a below base vowel).
   - **Example:** The Malayalam word for *eyewitness*. The form without the ZWNJ is incorrect in this case.

Figure 3.

	Code Points	Names (abbreviated)
ദൃക്‌സാക്ഷി	0D28 + 0D43 + 0D15 + 0D4D + 200C + 0D38 + 0D3E + 0D15 + 0D4D + 0D37	DA + SIGN VOCALIC R + KA + VIRAMA + ZWNJ + SA + SIGN AA + KA + VIRAMA + SSA
ദൃക്സാക്ഷി	0D28 + 0D43 + 0D15 + 0D4D + 0D38 + 0D3E + 0D15 + 0D4D + 0D37	DA + SIGN VOCALIC R + KA + VIRAMA + SA + SIGN AA + KA + VIRAMA + SSA

B. ZWJ in the following context:

1. **In a conjunct context.** That is, a sequence of the form:
   - A Letter, followed by a Virama, followed by a ZWJ

- This corresponds to the following regular expression (in Perl-style syntax): **/$L $V ZWJ/**
  where:
  - $L = [:General_Category=Letter:]
  - $V = [:Canonical_Combining_Class=Virama:]
- **Example:** The Sinhala word for the country 'Sri Lanka' in Figure 4A, which uses both a space character and a ZWJ. Removing the space gives the text in Figure 4B which is still readable, but removing the ZWJ completely modifies the appearance of the 'Sri' cluster and gives the text in Figure 4C.

Figure 4.

Appearance	Codepoints	Names (abbreviated)
A 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
B ශ්‍රීලංකා	0DC1 + 0DCA + 200D + 0DBB + 0DD3 + 0DBD + 0D82 + 0D9A + 0DCF	SHA + VIRAMA + ZWJ + RA + VOWEL SIGN II + LA + ANUSVARA + KA + VOWEL SIGN AA
C ශ්රී ලංකා	0DC1 + 0DCA + 0DBB + 0DD3 + 0020 + 0DBD + 0D82 + 0D9A + 0DCF	SHA + VIRAMA + RA + VOWEL SIGN II + SPACE + LA + ANUSVARA + KA + VOWEL SIGN AA

C. Mongolian Separators (NNBSP or MVSs) in the following context:

1. **Between Mongolian Letters.** That is, a sequence of the form:
   - A Mongolian Letter, followed by NNBSP or a MVS, followed by a Mongolian Letter.
   - This corresponds to the following regular expression (in Perl-style syntax): **/$ML $MS $ML/**
     where:
     - $ML = [[:General_Category=Letter:]&[:Script=Mongolian:]]
     - $MS = [\u202F \u180B \u180C \u180D]
   - **Example:** See pages 454 455 of *The Unicode Standard, Version 5.0.*

2.3 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. In these cases, a profile should consider whether the characters in *Table 3* should be allowed in identifiers, and perhaps others, depending on the languages in question. In some environments even spaces are allowed in identifiers, such as in SQL: *SELECT * FROM Employee Pension.*

Table 3. Characters possibly added for Natural Language Identifiers

0027 (') APOSTROPHE
002D (-) HYPHEN-MINUS
002E (.) FULL STOP
003A (:) COLON
00B7 (·) MIDDLE DOT
058A (֊) ARMENIAN HYPHEN
05F3 (׳) HEBREW PUNCTUATION GERESH
05F4 (״) HEBREW PUNCTUATION GERSHAYIM
200C () ZERO WIDTH NON-JOINER
200D () ZERO WIDTH JOINER
2010 (‐) HYPHEN
2019 (’) RIGHT SINGLE QUOTATION MARK
2027 (‧) HYPHENATION POINT
30A0 (゠) KATAKANA-HIRAGANA DOUBLE HYPHEN

Some characters are not in modern customary use, and thus implementations may want to exclude them from identifiers. The set of characters in Table 4 provides candidates of these, including archaic scripts and archaic or technical blocks.

Table 4. Characters possibly removed from Natural Language Identifiers

[[:script=Bali:] [:script=Bugi:]
[:script=Cari:] [:script=Copt:] [:script=Cprt:]
[:script=Dsrt:]
[:script=Glag:] [:script=Goth:]
[:script=Hano:]
[:script=Ital:]
[:script=Khar:]
[:script=Linb:] [:script=Lyci:] [:script=Lydi:]
[:script=Ogam:] [:script=Osma:]
[:script=Phag:] [:script=Phnx:]
[:script=Runr:]
[:script=Shaw:] [:script=Sylo:] [:script=Syrc:]
[:script=Tagb:] [:script=Tglg:]
[:script=Ugar:]
[:script=Xpeo:] [:script=Xsux:]

[:block=Combining_Diacritical_Marks_for_Symbols:]

[:block=Musical_Symbols:]
[:block=Ancient_Greek_Musical_Notation:]]

For more information on characters that may occur in words, see *Section 4, Word Boundaries*, in [UAX29].

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*.

2.4 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 adds 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. This list consists of four characters:

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, or exceptional characters. This list consists of ~~nine~~ ten characters:

U+00B7 (·) MIDDLE DOT

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.

R1 ***Default Identifiers***

To meet this requirement, an implementation shall use definition D1 and the properties ID_Start and ID_Continue (or XID_Start and XID_Continue) to determine whether a string is an identifier.

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

R1a ***Restricted Format Characters***

To meet this requirement, an implementation shall define a profile for R1 which allows format characters as described in *Section 2.2 Layout and Format Control Characters*. An implementation may further restrict the context for ZWJ or ZWNJ, such as by limiting the scripts, if a clear specification for such a further restriction is supplied.

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. (The typical mechanism used to achieve this is by using grandfathered characters.)

3 Alternative Identifier Syntax

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 1.1 specification by the W3C [XML1.1].

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.

R2 ***Alternative Identifiers***

To meet this requirement, an implementation shall define identifiers to be any string of characters that contains neither Pattern_White_Space nor Pattern_Syntax characters.

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

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 made regarding completely stable identifiers, and the practice that is seen in XML 1.1 [XML1.1]. (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 1.1.)

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.

Example 1:

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 signal an error.

As of [Unicode4.1], two Unicode character properties can be used for for stable syntax: Pattern_White_Space and Pattern_Syntax. Particular pattern languages may, of course, override these recommendations (for example, adding or removing other characters for compatibility in ASCII).

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 add more symbol or whitespace characters, but the syntax and whitespace characters 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.

Example 2:

Consider the following, 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 if in this example \uXXXX is used for hex expression, but resolved before quoting, and single quotes are used for quoting, this might be expressed as

'a\u0020b' => 'x\u200By' + z;

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 list of characters that are added to or removed from the sets of code points defined by these properties.

- All characters other than those defined by these 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.

R4 Equivalent Normalized Identifiers

To meet this requirement, an implementation shall specify the Normalization Form and shall provide a precise list of any 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.

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.

R6 *Filtered Normalized Identifiers*

To meet this requirement, an implementation shall specify the Normalization Form and shall provide a precise list of any 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, no identifiers are allowed that are not in the specified Normalization Form.

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, no identifiers are allowed that are not in the specified Normalization Form.

For R6, this involved removing from identifiers any characters in the set [:NFKC_QuickCheck=No:] (or equivalently, removing [:^isNFKC:]). For R7, this involves removing from identifiers any characters in the set [:^isCaseFolded:].

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. For related information, see Unicode Technical Report #30, "Character Foldings."

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.

1. ~~Middle dot. Because most Catalan legacy data is encoded in Latin-1, U+00B7 MIDDLE DOT is allowed in ID_Continue. If the programming language is using a dot as an operator, then U+2219 BULLET OPERATOR or U+22C5 DOT OPERATOR should be used instead. However, care should be taken when dealing with U+00B7 MIDDLE DOT, as many processes will assume its use as punctuation, rather than as a letter extender.~~

   [Note to reviewers: In Unicode 5.0, middle dot has been added to the identifier definition.]

2. ***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
3. ***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.

With these amendments to the identifier syntax, all identifiers are closed under all four Normalization Forms. Identifiers are also closed under case operations (with one exception). This means that for any string S:

<table>
<tr><td rowspan="2"><code>isIdentifier(S)</code></td><td rowspan="2">implies</td><td><code>isIdentifier(toNFD(S))</code>
<code>isIdentifier(toNFC(S))</code>
<code>isIdentifier(toNFKD(S))</code>
<code>isIdentifier(toNFKC(S))</code></td><td>Normalization Closure</td></tr>
<tr><td><code>isIdentifier(toLowercase(S))</code>
<code>isIdentifier(toUppercase(S))</code>
<code>isIdentifier(toFoldedcase(S))</code></td><td>Case Closure</td></tr>
</table>

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 *not* true in the case of compatibility equivalence: `isIdentifier(toNFC(S))` does not imply `isIdentifier(S)`. There are many characters for which the reverse implication is not true, since there are many character 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:

Code Points	GC	Samples	Names
2070	No	(⁰)	SUPERSCRIPT ZERO
20A8	Sc	(₨)	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 these characters would be need to be tailored so as to be added to identifiers.

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.

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

If an implementation needed to ensure full canonical equivalence of identifiers, then these characters had to be tailored to have the same value, so that either both isIdentifier(S) and isIdentifier(toNFC(S)) are true, or so that both values are false.

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 Character Database (UCD) provides support for handling case folding with normalization: the property FC_NFKC_Closure can be used in case folding, so that a case folding of an NFKC string is itself normalized. These properties, and the files containing them, are described in the UCD documentation [UCD].

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, Julie Allen, 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.”

Modifications

The following summarizes modifications from previous revisions of this annex.

Revision 8

- Added to discussion of canonical equivalence of identifiers.
- Added Filtered identifiers and rules
- Added format character discussion and rule
- Draft 3:
  - Removed restriction on scripts for ZWJ and ZWNJ.
  - Added sentence about further restrictions to R1a
  - Added line pointing to UTR36 for information about further restrictions

Revision 7

- Introduced the term *profile.*
- Added note on profiles of identifiers for natural language in *Section 2.3 Specific Character Adjustments*
- Minor editing for clarity in 2 Default Identifier Syntax
- Added note on spaces in identifiers (eg in SQL)

Revision 6 being a proposed update, only changes between revisions 7 and 5 are noted here.

Revision 5

- Removed section 4.1, because the two properties have been accepted for Unicode 4.1.
- Expanded introduction
- Adding information about stability, and tailoring for identifiers.
- Added the list of characters in Other_ID_Continue .
- Changed <identifier_continue> and <identifier_start> to just use the property names, to avoid confusion.
- Included XID_Start and XID_Continue in R1 and elsewhere.
- Added reference to UTR #36, and the phrase “or a list of additional constraints on identifiers” to R1.
- Changed “Coverage” to “General Description of Coverage,” because the UCD value are definitive.
- Added clarifications in 2.4
- Revamped 2.2 Layout and Format Control Characters
- Minor editing

Revision 3

- Made draft UAX
- Incorporated Annex 7 from UAX #15
- Added Other_ID_Continue for Unicode 4.1
- Added conformance clauses
- Changed <identifier_extend> to <identifier_continue> to better match the property name.
- Some additional edits.

Revision 2

- Modified Pattern_White_Space to remove compatibility characters
- Added example explaining use of Pattern_White_Space

Revision 1

- First version: incorporated section from Unicode 4.0 on Identifiers plus new section on patterns.
