

ISO
International Organization for Standardization
Organisation Internationale de Normalisation

ISO/IEC JTC 1/SC 2/WG 2
Universal Multiple-Octet Coded Character Set (UCS)

ISO/IEC JTC 1/SC 2/WG 2 N2895

L2/05-007

Date: 2005-01-17

Title: Informative Annex on 'Characters in Identifiers'
Source: USA and Unicode Consortium
(prepared by V.S. Umamaheswaran (umavs@ca.ibm.com))
Action: WG 2 members and Liaison organizations
Reference WG2 / N2818 and WG2 N2754 Resolution M45.23
Distribution: ISO/IEC JTC 1/SC 2/WG 2 members and liaison organizations

This document proposes addition of an informative Annex on 'Characters in Identifiers' to ISO/IEC 10646.

Background:

Document N2818 - Letter from SC22 to SC2 on language identifiers; SC22, via SC2 Secretariat; 2004-06-18, had requested WG2:

“SC 22 would like to ask SC 2/WG 2 to specify those characters which they believe are suitable for identifiers, leaving each programming language standard (SC 22 and other SCs and work programs generally) free to specify its own identifier-character list after considering the tradeoffs between its requirements and the advantages of a consistent, single, identifier specification in the same manner as the current TR 10176 recommends that programming language standards can extend or restrict the identifier character list.”

In response, WG2 had taken the following resolution at its meeting M45:

M45.23 (Request from SC22 to SC2/WG2):

With reference to document N2818, WG2 accepts the request from SC22 to classify characters in ISO/IEC 10646 for their suitability for use in identifiers. WG2 understands the importance that such information is available at the same time characters are added to ISO/IEC 10646. SC22 is invited to contribute to this work.

The US national body and the Unicode Consortium believe that WG2 can best meet the above work in the standard by adding suitable text in the standard pointing to the detailed work done by the Unicode Consortium on the topic of identifiers as documented in the Unicode Annex – UAX31 – “Identifier and Pattern Syntax” (see <http://www.unicode.org/reports/tr31/tr31-4.html>). The current version of TR 31 text is attached for information. Even though its status is Draft as of this writing, it is expected to be progressed to 'approved' status in the near future.

The following proposed text should be considered and added to ISO/IEC 10646 preferably in an informative Annex.

Proposed Text:

**Annex XXX (Informative)
Characters in Identifiers**

A common task facing an implementer of UCS is the provision of a parsing and/or lexing engine for identifiers. Each programming language standard has its own identifier syntax; different programming languages have different conventions for the use of certain characters from the ASCII (ISO 646-IRV) range (\$, @, #, _) in identifiers. Questions as to which characters to use for syntactic purposes versus which to be allowed in identifiers, whether case-pairing should be included, normalization should be performed, and other factors enter into the picture when defining the set of permitted characters for a given identification purpose.

To assist in the standard treatment of identifiers in UCS character-based parsers, a set of specifications is provided in UAX31 – “Identifier and Pattern Syntax” (see <http://www.unicode.org/reports/tr31/>). Those specifications are recommended for determining the list of UCS characters suitable for use in identifiers.



Draft Unicode Standard Annex #31

Identifier and Pattern Syntax

| | |
|------------------|---|
| Version | 4 (draft) |
| Authors | Mark Davis (mark.davis@us.ibm.com) |
| Date | 2004-10-08 |
| This Version | http://www.unicode.org/reports/tr31/tr31-4.html |
| Previous Version | http://www.unicode.org/reports/tr31/tr31-3.html |
| Latest Version | http://www.unicode.org/reports/tr31/ |

Summary

This document describes specifications for recommended defaults for the use of Unicode in the definitions of identifiers and in pattern-based syntax. It incorporates the Identifier section of Unicode 4.0 (somewhat reorganized) and a new section on the use of Unicode in patterns. As a part of the latter, it presents recommended new properties for addition to the Unicode Character Database. It also incorporates guidelines for use of normalization with identifiers (from UAX #15).

- Section 2 supersedes Section 5.15 Identifiers from The Unicode Standard 4.0.
- Section 5 supersedes Annex 7 in UAX #15.

Status

This document has been approved by the Unicode Technical Committee for public review as a Draft Unicode Standard Annex. Making this document available for public review does not imply endorsement by the Unicode Consortium. This is a draft document which may be updated, replaced, or superseded by other documents at any time. 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 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 number of the Unicode Standard at the last point that the UAX document was updated.

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

Contents

- [1. Introduction](#)
- [2. Default Identifier Syntax](#)
- [3. Alternative Identifier Syntax](#)
- [4. Pattern Syntax](#)
- [5. Normalization and Case](#)
- [Acknowledgements](#)
- [References](#)
- [Modifications](#)

1. Introduction

A common task facing an implementer of the Unicode Standard is the provision of a parsing and/or lexing engine for identifiers. To assist in the standard treatment of identifiers in Unicode character-based parsers, 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.

In addition, this document provides the definition of the Unicode properties used to define stable pattern syntax: syntax that is stable over future versions of the Unicode Standard. It also provides guidelines for the user of normalization with identifiers, originally in UAX #15.

1.1 Conformance

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

| | |
|-----|---|
| 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. |
| C2. | <p>An implementation claiming conformance to Level 1 of this specification shall describe which of the following it observes:</p> <ul style="list-style-type: none"> • R1 Default Identifiers • R2 Alternative Identifiers • R3 Pattern Whitespace and Syntax Characters • R4 Normalized Identifiers • R5 Case-Insensitive Identifiers |

2. Default Identifier Syntax

The formal syntax provided here is intended to capture the general intent that an identifier consists of a string of characters that begins with a letter or an ideograph, and then includes any number of letters, ideographs, digits, or underscores. Each programming language standard has its own identifier syntax; different programming languages have different conventions for the use of certain characters from the ASCII range (\$, @, #, _) in identifiers. To extend such a syntax to cover the full behavior of a Unicode implementation, implementers need only combine these specific rules with the syntax provided here.

D1. Default Identifier Syntax

`<identifier> := <identifier_start> <identifier_continue>*`

Identifiers are defined by the following sets of character categories from the Unicode Character Database.

Syntactic Classes for Identifiers

| Syntactic Class | Properties | Coverage |
|---------------------------------------|------------|--|
| <code><identifier_start></code> | ID_Start | Uppercase letter, lowercase letter, titlecase letter, modifier letter, other letter, letter number, stability extensions |

| | | |
|-----------------------|-------------|--|
| <identifier_continue> | ID_Continue | Plus nonspacing mark, spacing combining mark, decimal number, connector punctuation, formatting code, stability extensions |
|-----------------------|-------------|--|

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

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 must be valid for an identifier. This requirement results from the requirement for combining marks in the representation of many languages, and the conformance rules in Chapter 3 regarding interpretation of canonical-equivalent character sequences.

Enclosing combining marks (for example, U+20DD. .U+20E0) are excluded from the syntactic definition of <identifier_continue>, because the composite characters that result from their composition with letters (for example, U+24B6 circled latin capital letter a) are themselves not normally considered valid constituents of these identifiers.

2.2 Layout and Format Control Characters

The Unicode characters that are used to control joining behavior, bidirectional ordering control, and alternative formats for display are explicitly defined as not affecting breaking behavior. Unlike space characters or other delimiters, they do not serve to indicate word, line, or other unit boundaries. Accordingly, they should normally be ignored for the purposes of identifier definition. Implementations that cannot ignore characters in identifiers should exclude these characters.

2.3 Specific Character Adjustments

Specific identifier syntaxes can be treated as tailorings of the generic syntax based on character properties. For example, SQL identifiers allow an underscore as an identifier part (but not as an identifier start); C identifiers allow an underscore as either an identifier part 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`.

For programming language identifiers, normalization and case have a number of important implications. For a discussion of these issues, see [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 `Other_ID_Start` property contains a small list of characters that qualified as `<identifier_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. In Unicode 4.0, 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 contains a small list of characters that qualified as `<identifier_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_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 the D1 and the properties `ID_Start` and `ID_Continue` to determine whether a string is an identifier or not;

or shall declare that it uses a modification, and provide a precise list of characters that are added to or removed from `ID_Start` and `ID_Continue`.

3. Alternative Identifier Syntax

The down-side of working with the syntactic classes defined above 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.

One method to address this problem is to turn 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.

Some additional extensions to the list of disallowed code points can be made to further constrain “unnatural” identifiers. For example, one could include unassigned code points in blocks of characters set aside for future encoding as symbols, such as mathematical operators.

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 whitespace. 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;

or shall declare that it uses a modification, and provide 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. These patterns have been very limited in the past, and 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 forwards and backwards compatibility, it is very 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 consortium committed to not allocating characters suitable for identifiers in the range 2190..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. By using this policy, it 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:

In version 1.3 of program X, `'≈'` is a reserved syntax character, e.g. it doesn't perform an operation, but you have to quote it. In version 1.4, `'≈'` gets a real meaning, e.g. uppercase the subsequent characters. In this example, `'\'` quotes the next character; i.e., causes it to be treated as a literal instead of a syntax character.

- The pattern `abc...\≈...xyz` works on both version 1.3 and 1.4, and refers to the literal character since it is quoted in both cases.
- The pattern `abc...≈...xyz` works on 1.1 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 Unicode 4.1, there are two Unicode character properties that 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 property values are absolutely invariant; not changing with successive versions of Unicode. Of course, this doesn't 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 would not change.

When generating rules or patterns, all whitespace and syntax code points that are to be literals would 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: consider the following, where the items in angle brackets indicate literal characters.

- `a<SPACE>b => x<ZERO WIDTH SPACE>y + z;`

Since `<SPACE>` is a `Pattern_White_Space` character, it would require quoting. Since `<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 Syntax Characters

To meet this requirement, an implementation shall use `Pattern_White_Space` characters as all and only those character interpreted as whitespace in parsing, and shall use `Pattern_Syntax` characters as all and only those characters with syntactic use;

or shall declare that it uses a modification, and provide a precise list of characters that are added to or removed from the sets of code points defined by these properties.

- Note: all characters other than those defined by these properties would be available as identifiers or literals.

5. Normalization and Case

[R4](#) 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, and if the normalization form is NFKC, shall apply the modifications in NFKC Modifications. Except for identifiers containing excluded characters, any two identifiers that have the same normalization form shall be treated as equivalent by the implementation.

[R5](#) 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.

This section discusses issues that must be taken into account when considering normalization and case folding of identifiers in programming languages or scripting languages. Normalization can be used to avoid problems where apparently identical identifiers are not treated equivalently. Such problems can appear both during compilation and during linking, in particular also 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 may be used, while if the programming language has case-insensitive identifiers then Normalization Form KC may be more appropriate.

Note: In mathematically oriented programming languages which make distinctive use of the Mathematical Alphanumeric Symbols such as [U+1D400 MATHEMATICAL BOLD CAPITAL A](#), NFKC must not be used without filtering its application to not apply to those characters with the property value `decomposition_type=font`. For related information, see [UTR #30: Character Foldings](#).

If programming languages are using NFKC to level ("fold") differences between characters, then they use the following modification of the identifier syntax from the Unicode Standard to deal with the idiosyncrasies of a small number of characters. These characters fall into three classes:

NFKC Modifications

1. Middle Dot. Because most Catalan legacy data will be encoded in Latin-1, U+00B7 MIDDLE DOT needs to be allowed in `<identifier_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.)
2. Combining-like characters. Certain characters are not formally combining characters, although they behave in most respects as if they were. Ideally, they should not be in `<identifier_start>`, but rather in `<identifier_continue>`, along with combining characters. 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 to be in `<identifier_continue>` and not `<identifier_start>`:
 - 0E33 THAI CHARACTER SARA AM
 - 0EB3 LAO VOWEL SIGN AM
 - FF9E HALFWIDTH KATAKANA VOICED SOUND MARK
 - 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 must be excluded from both `<identifier_start>` and `<identifier_continue>`. It is recommended that all Arabic presentation forms be excluded from identifiers in any event, although only a few of them are required to be excluded for normalization to guarantee identifier closure.

With these amendments to the identifier syntax, all identifiers are closed under all four Normalization forms. This means that for any string *S*,

`isIdentifier(S)` implies

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

Identifiers are also closed under case operations (with one exception), so that for any string *S*,

`isIdentifier(S)` implies

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

The one exception 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 `<identifier_start>`, but its uppercase and case-folder version are. In practice this is not a problem, because of the way normalization is used with identifiers.

Note: Those programming languages with case-insensitive identifiers should use the case foldings described in Section 3.13 Default Case Operations to produce a case-insensitive normalized form.

When source text (such as program source) is parsed for identifiers, the identifiers must be parsed before folding distinctions using case mapping or NFKC.

When source text (such as program source) 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.

Note: The Unicode Character Database [UCD] provides derived properties that can be used by implementations for parsing identifiers, both normalized and unnormalized. These are the properties `ID_Start`, `ID_Continue`, `XID_Start`, and `XID_Continue`. Unicode 3.1 also provides support for handling case folding with normalization: the Unicode Character Database 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 [UCDDoc].

Acknowledgements

Thanks to Eric Muller, Asmus Freytag, and Martin Duerst for feedback on this document.

References

- [Feedback] Reporting Errors and Requesting Information Online
<http://www.unicode.org/reporting.html>
- [Reports] Unicode Technical Reports
<http://www.unicode.org/reports/>
 For information on the status and development process for technical reports, and for a list of technical reports.
- [UCD] Unicode Character Database.
<http://www.unicode.org/ucd>
 For an overview of the Unicode Character Database and a list of its associated files
- [Unicode] The Unicode Consortium. [The Unicode Standard, Version 4.0](#). Reading, MA, Addison-Wesley, 2003. 0-321-18578-1.
- [UAX15] UAX #15, Unicode Normalization Forms
<http://www.unicode.org/reports/tr15/>
- [Versions] Versions of the Unicode Standard
<http://www.unicode.org/versions/>
 For information on version numbering, and citing and referencing the Unicode Standard, the Unicode Character Database, and Unicode Technical Reports.
- [XML1.1] Extensible Markup Language (XML) 1.1
<http://www.w3.org/TR/xml11/>

Modifications

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

- 4
 - Removed section 4.1, since the two properties have been accepted for Unicode 4.1.
 - Minor editing
- 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.

- 2
 - Modified Pattern White Space to remove compatibility characters
 - Added example explaining use of Pattern White Space
- 1
 - First version: incorporated section from Unicode 4.0 on Identifiers plus new section on patterns.

Copyright © 2000-2004 Unicode, Inc. All Rights Reserved. The Unicode Consortium makes no expressed or implied warranty of any kind, and assumes no liability for errors or omissions. No liability is assumed for incidental and consequential damages in connection with or arising out of the use of the information or programs contained or accompanying this technical report. The Unicode [Terms of Use](#) apply.

Unicode and the Unicode logo are trademarks of Unicode, Inc., and are registered in some jurisdictions.