L2/25-127

Working Draft for a Proposed Draft Unicode® Technical Standard #

UNICODE SET NOTATION

Version	1
Editors	Robin Leroy (eggrobin@unicode.org)
Date	2025-04-11
This Version	
Previous Version	n/a
Latest Version	
Latest Proposed Update	
Revision	1

Summary

The description of Unicode properties and algorithms frequently requires referring to sets of code points and strings defined using property assignments. This document defines a notation for such sets. The notation is machine-readable and can be used in APIs.

Status

*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time.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

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

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

Contents

1 Introduction

Sets of code points can be defined by reference to their properties; for instance:

1. “the characters with the property XID_Continue”
2. “the characters whose Line_Break property value is OP and whose East_Asian_Width property value is neither F, W, nor H”
3. “the characters that have the Other_ID_Start property, or the Other_ID_Continue property, or whose General_Category value is one of Nl, Mn, Mc, Nd, Pc, or one of those in the L grouping, but that have neither the Pattern_Syntax property nor the Pattern_White_Space property.”
4. “the characters whose General_Category value is one of Nl, Mn, Mc, Nd, Pc, or one of those in the L grouping, except for the character U+2E2F VERTICAL TILDE.”

These kinds of set definitions are used throughout the Unicode Standard, including its annexes, and in the Unicode Technical Standards. They are necessary to the description of Unicode algorithms, such the line breaking algorithm [UAX14] and text segmentation algorithms [UAX29], of relations between properties, as in the derivations in [UAX29], [UAX31] and [UAX44], or of syntaxes as in [UAX31] or [UTS51]. They are also omnipresent in proposals and reports used in the development of these standards.

The use of plain-language definitions of these sets, as above, can become impractical when the definitions are complicated or when the sets are used in higher-level syntaxes, such as grammar rules or regular expressions. A definition that is not machine readable also prevents its direct use in implementations, or its inspection using tooling.

This document defines a formal syntax, *UnicodeSet notation*, for finite sets of code points and strings. In this syntax, the above examples can be expressed as:

1. `\p{XID_Continue}`
2. `[\p{lb=OP}-[\p{ea=F}\p{ea=W}\p{ea=H}]]`
3. `[\p{Other_ID_Start}\p{Other_ID_Continue}\p{L}\p{Nl}\p{Mn}\p{Mc}\p{Nd}\p{Pc}-\p{Pattern_Syntax}-\p{Pattern_White_Space}]`
4. `[\p{L}\p{Nl}\p{Mn}\p{Mc}\p{Nd}\p{Pc}-[\u2E2F]]`

Besides defining sets that are useful in specifications, this notation, if implemented in a tool that displays the contents of the set, can serve as a query language for the Unicode Character Database, allowing maintainers of the standard to answer questions such as:

1. “Which characters have an Uppercase_Mapping that differs from their Simple_Uppercase_Mapping?” `\p{Uppercase_Mapping≠@Simple_Uppercase_Mapping@}`.
2. “Which characters changed Simple_Case_Folding between Unicode Version 15.0 and Unicode Version 15.1?” `\p{U15.1:Simple_Case_Folding≠@U15.0:Simple_Case_Folding@}`.
3. “Which CJK characters have the word ‘cat’ in their definition, and which Egyptian hieroglyphs have the word ‘cat’ in their description?” `[\p{cjkDefinition=/\bcat\b/} \p{kEH_Desc=/\bcat\b/}]`.
4. “Does Changes_When_Casefolded mean the same as ‘different from its Case_Folding’?” No, the set `[\p{Case_Folding≠@code point@}-\p{Changes_When_Casefolded}]` is nonempty.

The document then discusses what subsets of UnicodeSet notation is appropriate for use in APIs, and how it can be incorporated in higher-level syntaxes.

Review Note: This syntax, which originates in the API of the ICU class UnicodeSet, was previously standardized in [UTS35], see https://unicode.org/reports/tr35/#Unicode_Sets; however, it is only partially defined there, with reference to [UTS18]:

> Unicode property sets are defined as described in UTS #18: Unicode Regular Expressions [UTS18], Level 1 and RL2.5, including the syntax where given. For an example of a concrete implementation of this, see [ICUUnicodeSet].

[UTS18] in turn does not formally define a syntax, but instead presents an example syntax, which differs from UnicodeSet syntax. The UAXes and UTSes that use UnicodeSet syntax currently refer to [UTS35], or sometimes incorrectly refer to [UTS18].

There are five known implementations of UnicodeSet notation maintained by the Unicode Consortium:

1. the ICU4C implementation;
2. the ICU4J implementation;
3. the implementation of the online Unicode tools (referred to as the JSPs), based on ICU4J with extensions and comprehensive property coverage;
4. the implementation used in the invariant tests in the Unicode tools, similar to the preceding one, with slightly different extensions;
5. the ICU4X experimental implementation used in the experimental transliterator module.

In addition, a syntax similar to UnicodeSet is supported by ICU4C regular expressions (but not documented), together with a syntax that uses && and -- for set operations for compatibility with Java. The Unicode Standard itself (Section A.2.1) defines a notation for sets of code points which is similar to, but different from UnicodeSet syntax. That notation uses && and -- for set operations. Many technical reports use UnicodeSet syntax instead.

In practice, any usage in CLDR has needed to lie within the common subset supported by ICU4C and ICU4J, regardless of what was written in the LDML specification. As a result, this document mostly follows the ICU4C implementation. Changes with respect to the current ICU4C implementation that could be in scope for implementation in ICU are highlighted in yellow or cyan in the grammar. Extensions to the ICU4C implementation that are unlikely to be in scope for implementation in ICU are shown with a gray background; these typically originate from the Unicode Tools, and are useful for the development and testing of the Unicode Standard itself, but not for general-purpose internationalization libraries. Divergences in other implementations are described in review notes.

1.1 Terminology and Notation

The context-free UnicodeSet syntax is described using a variant of Backus-Naur Form. Production rules are written using the sign ::=, and alternatives are separated by |. Nonterminal symbols, referred to in this document as *syntactic categories*, are written in a serif font, and are links to their definition. A `monospace font` is used for literal text. The symbol "" is used for the empty string. Some syntactic categories which correspond to character classes, such as white-space, are defined outside of the BNF grammar.

A *construct* is a piece of text that is an instance of a syntactic category. A *constituent* of a construct is the construct itself, or any construct appearing within it. An *immediate* constituent of a construct is one that corresponds to a syntactic category appearing in the right-hand side of the production rule defining the syntactic category of the construct.

Rules shown over a gray background define syntactic categories that are not recommended for support in general-purpose APIs. See *Section 5, Use in APIs*.

Example: The rule

Difference ::= Restriction - UnicodeSet

defines the syntactic category Difference as consisting of a Restriction, followed by the character U+002D HYPHEN-MINUS which is a set-operator, followed by a UnicodeSet.

In the Difference `[A-Z]-[C]`, the Restriction `[A-Z]`, the set-operator `-`, and the UnicodeSet `[C]` are the immediate constituent constructs of the Difference; the substring `[A-Z]-[` is not a construct. Parsing the constituent Restriction `[A-Z]` itself, it consists of set-operators `[` and `]` and of a Range `A-Z`. These are constituent constructs of the Restriction `[A-Z]` as well as of the Difference `[A-Z]-[C]`.

The syntax of UnicodeSet notation is described in two parts: lexical elements, whose grammars are regular and space-sensitive, and the context-free (but not regular) grammar of the ranges and set arithmetic making up the UnicodeSet expression itself, where white space is ignored. Syntactic categories used in the grammars of lexical elements are written in kebab-case; their production rules are space-sensitive. Syntactic categories used in the grammar of UnicodeSet are written in CamelCase; their production rules implicitly allow for optional-white-space between their constituent lexical elements.

Example: `[ A-Z ] - [C]` is a valid Difference, equivalent to `[A-Z]-[C]`.

This allows for a clear separation between lexical analysis (identifying lexical elements independently from context, which can be done using regular expressions) and syntactic analysis (building up syntactic categories up to UnicodeSet itself). In particular, this separation makes it easier to perform the insertion of left-to-right marks described in *Section 5.2, Conversion to Plain Text*, in *Unicode Technical Standard #55, Unicode Source Code Handling* [UTS55]; see also *Section 7.2, Bidirectional Display*.

Review Note: This approach differs from the one taken in [UTS35], where white space is explicit throughout the grammar, and no distinction is made between the syntactic categories for individual characters in string literals, which should not be directionally isolated, and those for individual characters in sets.

2 Lexical Elements

An expression in UnicodeSet notation consists of a sequence of separate *lexical elements*. Each lexical element is either a set-operator, a literal-element, an escaped-element, a named-element, a bracketed-element, or a string-literal, or a property-query.

In this grammar, white-space is defined as any character with the Pattern_White_Space property. One or more white-space character is allowed between any two adjacent lexical elements; this is not indicated explicitly in the grammar for UnicodeSet. Within the grammar for each lexical element, the syntactic category optional-white-space is used where spaces are allowed:

```
optional-white-space ::=
        ""
      | optional-white-space white-space
```

Note: white-space is sometimes necessary to separate consecutive lexical elements. For instance, `\00` consists of a single escaped-element, but `\0 0` consists of an escaped-element followed by a literal-element.

Each lexical element other than a set-operator represents a set of code point sequences.

A set-operator is any of `&`, `-`, `[`, `]`, and the sequence `[^`.

2.1 Literal Elements

A literal-element is a Unicode scalar value that does not have the Pattern_White_Space property, and is neither a set operator nor one of `{`, `}`, `$` or `\`.

2.1.1 Semantics

A literal-element represents a single code point: itself.

2.2 Escaped Elements

An escaped-element is defined by the following regular grammar, where escapable-character is any Unicode scalar value other than the digits `0` through `7` and the letters `u`, `x`, `U`, `N`, `a`, `b`, `t`, `n`, `v`, `f`, and `r`.

```
escaped-element ::=
        \x up-to-two-hexadecimal-digits
      | \u four-hexadecimal-digits
      | \U000 five-hexadecimal-digits
      | \U0010 four-hexadecimal-digits
      | \x{ hexadecimal-digits }
      | \ up-to-three-octal-digits
      | \ escapable-character
      | \a | \b | \t | \n | \v | \f | \r
up-to-three-octal-digits ::=
        octal-digit
      | octal-digit octal-digit
      | octal-digit octal-digit octal-digit
up-to-two-hexadecimal-digits ::=
        hexadecimal-digit
      | hexadecimal-digit hexadecimal-digit
four-hexadecimal-digits ::=
        hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit
five-hexadecimal-digits ::=
        hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit
hexadecimal-digits ::=
```

hexadecimal-digit
| hexadecimal-digits hexadecimal-digit
octal-digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hexadecimal-digit ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| A | B | C | D | E | F
| a | b | c | d | e | f

Note: In this grammar, hexadecimal-digit is not equivalent to the set of characters with the property Hex_Digit: the fullwidth digits and letters are not alowed in an escaped-element.

2.2.1 Semantics

An escaped-element represents a single code point, as follows. An escaped-element consisting of \ followed by an escapable-character represents that escapable-character. Any escaped-element with constituent octal-digits represents the code point whose octal representation is given by its constituent octal-digits. Any escaped-element with constituent hexadecimal-digits represents the code point whose hexadecimal representation is given by its constituent hexadecimal-digits. If the constituent hexadecimal-digits do not represent a code point, the UnicodeSet expression is ill-formed. The remaining escaped-elements are defined by the following table.

escaped-element	Code point (name alias)
\a	U+0007 (ALERT)
\b	U+0008 (BACKSPACE)
\t	U+0009 (HORIZONTAL TABULATION)
\n	U+000A (NEW LINE)
\v	U+000B (VERTICAL TABULATION)
\f	U+000C (FORM FEED)
\r	U+000D (CARRIAGE RETURN)

Example: The escaped-elements \\, \134, \x5C, \u005C,\x{05C}, and \U0000005C all represent the code point U+005C. The escaped-elements \a, \7, and \x7 all represent the code point U+0007. The escaped-element \x{110000} is ill-formed.

Review Note: [UTS35] allows for \u{2F} as well as \x{2F}, and for wholly-escaped strings with the syntax \x{2F 2F} (equivalent to {\x{2F}\x{2F}}). It allows optional-white-space (including line terminators) inside the braces of a \x{} or \u{} escape. This is not supported by ICU4C, ICU4J, the JSPs, nor the invariants, but is supported by the ICU4X experimental implementation. [UTS35] does not allow for octal escapes nor for a single hexadecimal digit after \x, but since this is supported by ICU4C, ICU4J, and the ICU4J-based Unicode tools, as well as consistent with many programming languages, we include these in the specification.

2.3 Named Elements

A named-element is defined by the following regular grammar, where a ucd-identifier-character is any character in the Basic Latin block whose general category is one of Lu, Ll, Nd, Pc, Pd, or Zs, and where a named-literal-element is any Unicode scalar value except : and }.

named-element ::=
\N{ ucd-identifier }
| \xN{ hexadecimal-digits : ucd-identifier }
| \xcN{ hexadecimal-digits : named-literal-element : ucd-identifier }
ucd-identifier ::=
ucd-identifier-character
| ucd-identifier ucd-identifier-character

Note: In UnicodeSet notation, the set of ucd-identifier-characters is [\p{block=Basic_Latin} & [\p{L}\p{Nd}\p{Pc}\p{Pd}\p{Zs}]] = [A-Za-z0-9\N{SPACE}_-].

2.3.1 Semantics

A named-element represents the single character whose Name or Name Alias matches the constituent ucd-identifier according to loose matching rule UAX44-LM2. If there is no such character, the UnicodeSet expression is ill-formed.

If the named-element contains hexadecimal-digits, these shall be a hexadecimal representation of the code point named by the ucd-identifier. If it contains a named-literal-element, that named-literal-element shall be the named character.

Examples: The named-elements \N{SPACE}, \xN{0020:SPACE}, and \xcN{20: :SPACE} all represent U+0020 SPACE. The named-elements \N{THIS IS NOT A CHARACTER}, \xN{0A:LATIN CAPITAL LETTER A}, and \xcN{41:a:LATIN CAPITAL LETTER A} are ill-formed.

The named-elements \N{PRESENTATION FORM FOR VERTICAL RIGHT WHITE LENTICULAR BRAKCET} and \N{PRESENTATION FORM FOR VERTICAL RIGHT WHITE LENTICULAR BRACKET} both represent U+FE18 ﴘ. The named-element \N{Latin small ligature o-e} represents U+0153 œ LATIN SMALL LIGATURE OE. The named-element \N{Hangul jungseong O-E} represents U+1180 ᅀ HANGUL JUNGSEONG O-E. The named-element \N{Hangul jungseong OE} represents U+116C ᅬ HANGUL JUNGSEONG OE.

Review Note: The \xN and \xcN escapes are innovations introduced in this document. The need for them has become apparent in the Unicode invariant tests, especially for property comparisons for character additions. They are approximated in the Unicode invariant tests by the use of \x{code point} \N{name}, combined in some cases with higher-level checks that the sets have the right size (this is done because earlier iterations of those tests failed to catch incorrect code points or names in draft data when they were testing only one of those). This is however quite brittle (for instance, swapped characters would not be detected).

Review Note: [UTS35] allows for arbitrary ignored white-space (including line terminators) after the opening curly bracket and before the closing curly bracket, but not within the character name itself (only U+0020 SPACE is allowed within the name). Spaces other than U+0020 within a \N escape are not supported by any implementation (ICU4C, ICU4J, JSPs, nor invariants; the ICU4X experimental implementation does not support \N at all).

Review Note: Neither the Unicodetools implementation nor the ICU implementation consider name aliases.

Review Note: \N escapes do not allow for the use of named sequences. Should they be allowed?

2.4 Bracketed Elements and Strings

The syntactic categories bracketed-element and string-literal are defined by the following regular grammar, where a bracketed-literal-element is any non-white-space Unicode scalar value except `\` and `}`.

```
bracketed-element ::= { optional-white-space string-element optional-white-space }
string-literal ::=
        { optional-white-space }
      | { optional-white-space string-elements optional-white-space }
string-element ::=
        bracketed-literal-element | escaped-element | named-element
string-elements ::=
        string-element optional-white-space string-element
      | string-elements optional-white-space string-element
```

2.4.1 Semantics

A bracketed-literal-element represents a single code point: itself. A string-element represents the code point represented by its constituent bracketed-literal-element, escaped-element, or named-element.

A bracketed-element represents the code point represented by its constituent string-element. A string-literal represents the sequence of the code points represented by each of its constituent string-elements.

Note: { optional-white-space } represents the empty string. A string-literal represents either the empty string or a string consisting of two or more code points.

Note: The optional-white-space has no effect on the semantics of a string-literal or bracketed-element.

2.5 Property Queries

A property-query is defined by the following regular grammar.

```
property-query ::=
        perl-start query-expression perl-end
      | posix-start query-expression posix-end
perl-start ::= \p{ | \P{
perl-end ::= }
posix-start ::= [: | [:^
posix-end ::= :]
query-expression ::=
        unary-query-expression
      | binary-query-expression
unary-query-expression ::= optional-version-qualifier ucd-identifier
binary-query-expression ::= optional-version-qualifier ucd-identifier query-operator property-predicate
optional-version-qualifier ::=
        ""
      | version-qualifier
version-qualifier ::=
        U version-number :
      | U version-suffix :
      | U-1:
version-number ::=
        digit optional-suffix
      | version-number digit optional-suffix
      | version-number . digit optional-suffix
optional-suffix ::=
        ""
      | version-suffix
version-suffix ::= α | β | dev
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
query-operator ::= = | ≠
property-predicate ::=
        property-value
      | regular-expression-match
      | property-comparison
property-value ::= property-value property-value-element
property-value-element ::=
        literal-value-element
      | escaped-element
      | named-element
```

where literal-value-element is any Unicode scalar value other than `\`, `:`, `{`, `}`, `=`, `≠`, `/`, or `@`.

```
property-comparison ::= @ unary-query-expression @
regular-expression-match ::= / regular-expression /
regular-expression ::=
        ""
      | regular-expression regular-expression-character
regular-expression-character ::= regex-unescaped | \ any
```

where regex-unescaped is any Unicode scalar value other than `/` and `\` and any is any Unicode scalar value.

Review Note: The operator ≠ is not supported by ICU4C and ICU4J, but is specified in [UTS35], and is supported in the JSPs as well as the ICU4X experimental implementation. Experience has shown that the \P syntax can lead to confusion, so \p with ≠ may be preferable. The double negation resulting from \P with ≠ or [:^ with ≠ should be avoided, and implementations should probably reject it.

Review Note: property-comparison and regular-expression-match are supported only in the JSPs and invariants.

Review Note: No implementation supports escapes in property values. This is not a major problem for the ICUs, as they do not support string- or code point-valued properties either, except for Name; but it is a problem in the tools. Since the lack of string- or code point-valued properties seems to be serendipitous, rather than fundamental to the scope of general-purpose internationalization libraries, we propose adding support for escapes generally (so they are in yellow, not in gray).

Review Note: UTS35 allows for unescaped : in Perl-style queries, and for unescaped } in POSIX-style queries. However, non-enumerated properties are not supported in any UnicodeSet implementation other than those of the Unicode tools (JSPs and invariants), so this poses no real compatibility constraints. Since we are using : as a delimiter, it makes sense to require that it be escaped.

2.5.1 Negations

A property-query is *exteriorly negated* if it starts with the posix-start `[:^` or the perl-start `\P{`. It is *interiorly negated* if its query-expression is a binary-query-expression whose query-operator is `≠`.

> **Examples:** The constructs `\P{Cn}`, `[:^Cn:]`, `\P{General_Category=Cn}`, and `[:^General_Category=Cn:]`, and `[:^General_Category≠Cn:]` are exteriorly negated. The constructs `\p{General_Category≠Cn}`, and `[:General_Category≠Cn:]`, and `[:^General_Category≠Cn:]` are interiorly negated.

For a property-query, the *corresponding non-negated* property-query is defined by changing any perl-start to `\p{`, any posix-start to `[:`, and any query-operator to `=`.

> **Examples:**
>
property-query	Corresponding non-negated property-query
`\P{Cn}`	`\p{Cn}`
`\p{General_Category≠Cn}`	`\p{General_Category=Cn}`
`\P{General_Category=Cn}`	`\p{General_Category=Cn}`
`\p{General_Category=Cn}`	`\p{General_Category=Cn}`
`[:^General_Category≠Cn:]`	`[:General_Category=Cn:]`

A property-query is *simply negated* if it is either exteriorly negated or interiorly negated, but not both. A simply negated property-query represents the code point complement of the set represented by the corresponding non-negated property-query.

> **Examples:** `\P{Cn}` and `\p{General_Category≠Cn}` are simply negated. They represent the code point complement of `\p{General_Category=Cn}`.

A property-query is *doubly negated* if it is both exteriorly negated and interiorly negated. A doubly negated property-query represents the same set as the corresponding non-negated property-query.

> **Note:** While they are well-defined, the use of doubly negated property queries is discouraged. Examples of doubly-negated property-queries: `\P{Decomposition_Type≠compat}` (equal to `\p{Decomposition_Type=compat}`), `[:^Noncharacter_Code_Point≠No:]` (equal to `[:Noncharacter_Code_Point=No:]`).

> **Note:** There is no semantic difference between POSIX-style and Perl-style property queries, that is, for any property-query x, `[:`x`:]` is equivalent to `\p{`x`}`, and `[:^`x`:]` is equivalent to `\P{`x`}`.

A property-query which is neither simply negated nor doubly negated is *non-negated*.

> **Note:** For any property-query, the corresponding non-negated property-query is non-negated.

2.5.2 Unary Queries

A non-negated property-query whose query-expression is a unary-query-expression represents a set of code points as follows.

1. If the ucd-identifier matches an alias for a binary property under rule UAX44-LM3, the property-query represents the set of code points for which the given property is True.
2. If the ucd-identifier matches an alias for a Script property value under rule UAX44-LM3, the property-query represents the set of code points whose Script property value has that alias.
3. If the ucd-identifier matches an alias for a General_Category property value under rule UAX44-LM3, then:
    1. if the ucd-identifier matches an alias for a grouping of General_Category values, the property-query represents the set of code points whose General_Category property value is in that grouping;
    2. otherwise, the property-query represents the set of code points whose General_Category property value has the alias matching the ucd-identifier.
4. Otherwise, the UnicodeSet expression is ill-formed.

> **Note:** The invariants of the Unicode character database ensure that only one of these alternatives holds. For example, no Script property value alias matches an alias for a binary property.
>
> No such guarantee is made if unary queries are extended to other properties:
>
> - Properties of other types can match Script or General_Category aliases; for instance, ISO_Comment has the alias isc, which matches the alias C for the General_Category grouping Other.

- Value aliases for properties other than Script and General_Category can match property aliases for binary properties; for instance, White_Space is both a Bidi_Class value and a binary property.
- If P and Q are properties and the pair $\{P, Q\}$ is not {Script, General_Category}, a value alias for P may match a value alias for Q. For instance, with P=Line_Break and Q=Grapheme_Cluster_Break, both properties have a value alias ZWJ. With P=Script and Q=Block, both properties have a value alias Greek.

Review Note: The UnicodeSet implementation of the invariant tests do not implement implicit Script nor implicit General_Category.

If the version-qualifier with a version-number is present, the above set is defined based on the property assignments in the version of the Unicode Character Database given by the version-number. A version-suffix may be used to refer to unpublished versions of the Unicode Character database.

Note: No products or implementations should be released based on the beta, alpha, or earlier draft UCD data files. The use of a version suffix in UnicodeSet expressions should be restricted to documents and tools involved in the development of the Unicode Standard.

Review Note: Only the Unicode tools (JSPs and invariants) support version-qualifiers. This is not expected to change: general-purpose internationalization libraries have no reason to ship the entire history of the UCD.

In the absence of a version qualifier, the version of the UCD used depends on context. The version-qualifier `U-1:` is used to refer to the version of the UCD preceding the one referenced by an absence of version qualifier.

Review Note: The version-qualifier `U-1:` is only supported in the invariant tests, not in the JSPs.

Examples:

By default, within the text of the Unicode Standard, a UnicodeSet expression refers to the property assignments in that version of the standard.

In the sentences “the set `\p{Pattern_Syntax}` is immutable” and “the set `\p{XID_Continue}` can only grow over successive versions of the Unicode Standard”, the expression refers to all versions of the UCD.

The encoding stability policy, applicable to Unicode 2.0+, states that

> Once a character is encoded, it will not be moved or removed.

This policy implies that `\p{GC=unassigned}` ⊆ `\p{U-1:GC=unassigned}`, where the implicit version is any version after 2.0.

2.5.3 Binary Queries

A non-negated property-query whose query-expression is a binary-query-expression represents a set of code points as follows.

The ucd-identifier preceding the query-operator shall match an alias for a property under rule UAX44-LM3. That property is the *queried property*. If the binary-query-expression starts with a version-qualifier, it defines the *queried version*.

Note: The invariants of the Unicode character database ensure that a string matches an alias for at most one property.

If the property-predicate is a property-value, the *queried value* is defined as the sequence of code points represented by each property-value-element, where a literal-value-element represents itself, and an escaped-element and a named-element represent a code point as described by their respective semantics.

A property-value shall consist solely of literal-value-elements unless the queried property is a string-valued or miscellaneous property.

Review Note: The preceding paragraph removes an unnecessary burden on implementers that do not support string properties (they do not need to support `\p{gc=\N{LATIN CAPITAL LETTER L}\N{LATIN SMALL LETTER L}}`), and it establishes some semblance of typing (even though we do not formally have types in this specification).

If the queried version is defined, the property assignments of the queried property used in the definition of the set are those from that version of the Unicode Character Database.

2.5.3.1 Age Queries

If the queried property is the Age property, the property-predicate shall be a property-value, and the queried value shall match a value alias for the Age property under UAX44-LM3. The property-query then represents the set of code points whose Age value is less than or equal to the matching Age value.

Example: The set `\p{Age=6.0}` contains all characters that were assigned in Unicode Version 6.0, as well as noncharacter code points, surrogate code points, and private use area code points. It is equal to the set `[ \P{U6:Cn} \p{U6:Noncharacter_Code_Point} ]`. The expressions `\p{Age=@U6:Age@}` and `\p{Age=/1/}` are ill-formed.

Note: The special handling of the Age property addresses the common use case of matching characters present in some version of Unicode (thus with an age older than or equal to that version of Unicode). This special handling is largely redundant with the more regular version-qualifier mechanism; specifically for an alias x of the Age property which satisfies the version-number grammar, The sets `\p{Ux:gc≠Unassigned}` and `[ \p{Age=x} - \p{Noncharacter_Code_Point} ]` are equal. However, the support of version-qualifier is not recommended for general-purpose APIs, see *Section 5, Use in APIs*.

Review Note: The age property behaves unusually in UnicodeSet, in a way that cannot be unified with the other properties. Contrast the Name property, which we can make regular by treating formal aliases as value aliases. We therefore do not specify property comparisons nor regular expression matching on the Age property.

2.5.3.2 Property Comparisons

If the property-predicate is a property-comparison, the constituent ucd-identifier of the property-comparison shall either match match an alias for a property under rule UAX44-LM3, or it shall match the string `none` or the string `code point` under rule UAX44-LM3. In the first case, that

property is the *comparison property*. In the second case, there is no comparison property. If the constituent unary-query-expression of the property-comparison starts with a version-qualifier, it defines the *comparison version*.

Example: In both `\p{scf=@lc@}` and `\p{U15.1:scf=@U15.1:lc@}`, the queried property is Simple_Case_Folding and the comparison property is Lowercase_Mapping. In `\p{U15.0:Line_Break≠@U15.1:Line_Break@}`, the queried version is 15.0, and the comparison version is 15.1. In `\p{kIRG_GSource=@none@}` and `\p{case folding=@code point@}`, there is no comparison property. The expressions `\p{kIRG_GSource=@U16:none@}` and `\p{case folding=@U16:code point@}` are ill-formed.

If there is no comparison property, the constituent unary-query-expression of the property-comparison shall not start with a version-qualifier.

If the comparison version is defined, the property assignments used of the comparison property used in the definition of the set are those from that version of the Unicode Character Database. For both properties, if the version is absent, it depends on context. If both version qualifiers are absent, the same context-dependent version is used.

Example: The statement “the set `\p{scf=@lc@}` shrank between Unicode 15.0 and Unicode 15.1” is a statement about the sets `\p{U15.1:scf=@U15.1:lc@}` and `\p{U15.0:scf=@U15.0:lc@}`

If there is a comparison property, its type shall be compatible with that of the queried property, that is, one of the following shall hold:

1. Both are binary properties.
2. Both are (possibly multivalued) string-valued properties.
3. Both are (possibly multivalued) numeric properties.
4. Both are (possibly multivalued) enumerated or catalog properties with the same underlying enumeration.
5. They are the same property.

The query-expression then represents the set of code points that have the same value for the queried property and comparison property. For unordered multivalued properties, the sets of values are compared. For ordered multivalued properties, the sequences of values are compared.

Examples: The expression `\p{Decomposition_Mapping=@Ideographic@}` is ill-formed, as the string-valued Decomposition_Mapping property and the binary Ideographic property have incompatible types. The following are well-formed expressions from each of the three categories above:

1. The set `\p{Uppercase≠@Changes_When_Lowercased@}` is the set of characters whose Uppercase value differs from their Changes_When_Lowercased value. It is equal to `[[\p{Uppercase}\p{Changes_When_Lowercased}]-[\p{Uppercase}&\p{Changes_When_Lowercased}]]`, that is, the set of characters that are either Uppercase or Changes_When_Lowercased, but not both.
2. The set `\p{scf≠@cf@}` is the set of characters whose Simple_Case_Folding differs from their (full) Case_Folding.
3. The set `\p{Numeric_Value=@kPrimaryNumeric@}` is the set of characters that either have a single kPrimaryNumeric value, or have neither kPrimaryNumeric nor Numeric_Value (both are NaN).
4. The set `\p{U15.0:Line_Break≠@U15.1:Line_Break@}` is the set of code points whose Line_Break assignment changed betwen Unicode Version 15.0 and Unicode Version 15.1.

The set `\p{U16.0:kPrimaryNumeric≠@U17.0:kPrimaryNumeric@}` contains U+5146, as the values are ordered and the order changed in Unicode Version 17.0. The set `\p{Script_Extensions=@Script@}` is the set of characters whose Script_Extensions value is a single value equal to their Script value. These are the characters not listed in ScriptExtensions.txt, to which the line `@missing: 0000..10FFFF; <script>` applies.

Review Note: We allow only sensible property-comparisons. The UnicodeTools allow \p{Decomposition_Mapping=@Ideographic@}, which is equal to [№] (via the value No), and we don’t want to specify this sort of silliness.

2.5.3.3 Identity and Null Queries

If the property-predicate is a property-comparison and there is no comparison property:

1. If the ucd-identifier matches `code point`, the property shall be a string-valued property. The query-expression represents the set of code points that are mapped to themselves by the queried property.
2. If the ucd-identifier matches `none`, the property shall be a string-valued property or a miscellaneous property. The query-expression represents the set of code points for which no value is defined for the queried property.

Examples: The set `\p{scf=@code point@}` is equal to the set of code points which map to themselves under simple case folding. The set `[:^kIRG_GSource=@none@:]` is the set of CJK ideographs that have a “G” source mapping. The sets `\p{Bidi_Paired_Bracket=@none@}` and `\p{Bidi_Paired_Bracket_Type=None}` are equal.

Review Note: The only known implementation to support identity and null queries is the one used by the invariant tests. UTS #18 suggests @identity@ instead of @code point@ and does not have @none@. The use of @code point@ and @none@ is consistent with the use of <code point> and <none> in UCD @missing lines in a shared namespace with property names, with <script>.

2.5.3.4 Valid Values and Resolved Sets

A string s is a *valid value* for a property p if one of the following holds:

1. p is the Name property and s matches a value of the Name property or a value of the Name_Alias property under matching rule UAX44-LM2.
2. p is the Name_Alias property and s matches one the values of the Name_Alias property under matching rule UAX44-LM2.
3. p is a property for which property value aliases are defined, and s matches a value alias under matching rule UAX44-LM3.
4. p is some other string-valued or miscellaneous property.
5. p is a numeric property, and:
    1. s matches the string `NaN` under matching rule UAX44-LM3, or
    2. s matches the regular expression `[+-]?[0-9]+(/[0-9]*[1-9][0-9]*)?`.

Review Note: We are not accepting decimal marks for now, since then we have to specify rounding rules for consistency with extracted/DerivedNumericValues.txt (that would probably be the nearest binary32). No existing production implementation deals with numeric values.

The *resolved set* of p for s is then respectively:

1. The set whose sole element is the character whose name or name alias matches s.
2. The set whose sole element is the character whose name alias matches s.
3. The set of characters for which the value of p is the string s itself.
4. If p is the General_Category property and s is an alias for a grouping of General_Category values, the set of characters whose General_Category is one of the values in that grouping. Otherwise, the set of characters for which one of the values of p has an alias matching s.
5. The set of characters for which p has the value NaN, or for which the value of p is the rational number expressed by s.

> **Note:** This implements matching rule UAX44-LM1.

2.5.3.5 Property Value Queries

If the property-predicate is a property-value, the queried value shall be a valid value for the queried property.

The query-expression represents the resolved set of the queried property for the property-predicate.

> **Examples:** The set \p{Uppercase=True} is equal to the set \p{Uppercase}. The set \p{Uppercase=NO} is equal to the set \P{Uppercase}. The set \p{Script_Extensions=Latin} is the set of characters that have Latin as one of their Script_Extensions values. The sets \p{nv=2/12} and \p{Numeric_Value=1/6} are equal. For all formal name aliases x, \p{Name_Alias=x} and \p{Name=x} are equal.

2.5.3.6 Regular Expression Queries

If the property-predicate is a regular-expression-match, the queried property shall not be a numeric property. The text of the regular-expression is interpreted as a regular expression. Where ambiguous, the specific regular expression syntax and options used should be described.

Review Note: Defining regular expression matching on numeric values would require us to define a finite set of preferred string representations of the numeric values, filling the same role as the exact spellings of name aliases. This would be a nontrivial exercise, and likely a pointless one, as matching numbers with regular expressions is inconvenient.

If the queried property is the Name property, the query-expression represents the set of code points whose character name matches the regular expression, or that have a formal name alias matching the regular expression. Otherwise the query-expression represents the set of code points for which one of the aliases of one of the values of the queried property matches the regular expression.

> **Examples:** The set \p{Name=/CAPITAL LETTER/} is the set of all characters whose name contains “CAPITAL LETTER”. The set \p{Block=/^Cyrillic/} is the set of all code points in a block whose name starts with “Cyrillic”. The set \p{scx=/Gondi/} contains all code points that have either Gunjala_Gondi or Masaram_Gondi among their Script_Extensions values. The set \p{gc=/^P/} contains punctuation characters (whose short aliases match), as well as private use characters and U+2029 PARAGRAPH SEPARATOR (whose long aliases match).

> **Note:** Neither loose matching rule LM2 nor LM3 is applied in regular expression queries. The set \p{Name=/NO BREAK SPACE/} is empty, whereas the set \p{Name=/NO-BREAK SPACE/} contains NO-BREAK SPACE, NARROW NO-BREAK SPACE, and ZERO WIDTH NO-BREAK SPACE. The set \p{Script=/ Gondi/} is empty, whereas the set \p{Script=/_Gondi/} contains Gunjala Gondi and Masaram Gondi characters. General_Category groupings are not taken into account in regular expression queries: the set \p{gc=/Cased_Letter/} is empty. If x is the exact spelling of a value alias for property p, or if P is Name and x is either the exact spelling of a name or a name alias, the sets \p{p=x} and \p{p=/^$x$$/} are equal.

Review Note: Neither the JSPs nor the invariant tests take Name_Alias into account for regular expression queries on the Name property. We want to take Name_Alias into account for value queries for compatibility with ICU (which follows the recommendations in UTS18), see the review note above. We also want to be consistent between regular expression queries and value queries (specifically, we want the property stated at the end of the note above). We therefore need to consider name aliases as aliases of the Name property here too.

3 Set Operations

UnicodeSet expressions are defined by the syntactic category UnicodeSet in the following context-free space-insensitive grammar, whose terminals are the lexical elements defined in Section 2, Lexical Elements.

```
UnicodeSet ::= Factor | NamedSingleton
Factor ::=
        [ Union ]
      | Complement
      | property-query
      | named-element
NamedSingleton ::= named-element
Complement ::= [^ Union ]
Union ::=
        Terms
      | UnescapedHyphenMinus Terms
      | Terms UnescapedHyphenMinus
      | UnescapedHyphenMinus Terms UnescapedHyphenMinus
UnescapedHyphenMinus ::= -
Terms ::=
        ""
      | Terms Term
Term ::=
        Elements
```

```
        | Restriction
Restriction :=
        Factor~~UnicodeSet~~
        | Intersection
        | Difference
Intersection := Restriction & Factor~~UnicodeSet~~
Difference := Restriction - UnicodeSet
Elements := Element | Range
Range := RangeElement - RangeElement
RangeElement :=
        literal-element
        | escaped-element
        | named-element
        | bracketed-element
Element := RangeElement | string-literal
```

Note: The above grammar is LR(2) rather than LR(1). After `[a`, if the next lexical element is the set-operator `-`, there is an ambiguity between a Range and an Element followed by an UnescapedHyphenMinus (a shift-reduce conflict). This ambiguity is resolved by looking ahead one more lexical element: the `-` is an UnescapedHyphenMinus only if it is followed by the set-operator `]`. The grammar can be rewritten to be LR(1), see [Knuth1965]. However, such a transformation obscures the definition of the syntax, as it requires introducing syntactic categories for constructs such as `a-` that could either be the beginning of a range or an element followed by an unescaped hyphen, and those such as `[a-z]-` that could turn out to be either the beginning of a difference or a restriction followed by an unescaped hyphen; alternatively, this could be achieved by introducing a lexical element `-` optional-white-space `]`.

Review Note: ICU puts named-element as an alternative in UnicodeSet rather than Element, making \N{SPACE} equivalent to [\x{20}] rather than \x{20}; see ICU-22851.

This is misleading, as the expression [\N{LATIN SMALL LETTER A}-\N{LATIN SMALL LETTER Z}] is then valid, but is the singleton [a] rather than the set of 26 letters [a-z]. This has led to bugs in practice.

However, UnicodeSet `-` named-element can make sense in expressions such as [\p{Changes_When_Casefolded}-\N{COMBINING GREEK YPOGEGRAMMENI}], and has been used in practice, so simply moving named-element into Element (yellow) would cause unwanted incompatibilities with reasonable expressions that have been valid for more than twenty years, requiring such expressions to be changed to [\p{Changes_When_Casefolded}-[\N{COMBINING GREEK YPOGEGRAMMENI}]]. Likewise, an unbracketed \N{SPACE} is currently a valid and unproblematic UnicodeSet.

We do not want to make a literal-element a valid UnicodeSet, as many tools built on top of UnicodeSet search for a valid UnicodeSet as part of their parsing logic. Allowing for arbitrary UnicodeSet `-` Element is less problematic, but seems like it loses clarity in expressions involving the already overloaded operator -.

Our solution to these compatibility issues is the set of changes in cyan. The introduction of the distinction between Factor and UnicodeSet is necessary to prohibit named-element `-` named-element from being a Difference (which would be a reduce-reduce conflict with Range). Besides the left-hand side of Difference and Intersection, we use Factor on the right-hand side of Intersection to disallow intersection with a singleton, which is certainly a bug. The following table summarizes the changes to expressions involving named-element.

Expression	ICU 2.4–77	Proposed	Notes
\N{SPACE}	[\u0020]		NamedSingleton
[[\u0000-\u007F]-\N{TILDE}]	[\u0000-\u007D\u007E]		**[** **[** **\u0000** **-** **\u007F** **]** **-** **\N{TILDE}** **]** Factor NamedSingleton Restriction UnicodeSet Difference
[\N{SPACE}-~]	Syntax error: expected `[`, `[^`, property-query, or named-element after `-`.	[\u0020-\u007E]	
[[\u0000-\u007F]&\N{TILDE}]	[\u007E]	Syntax error: expected `[`, `[^`, or property-query after `&`.	
[\N{SPACE}-\N{TILDE}]	[\u0020]	[\u0020-\u007E]	

UTS35 specifies that \N{SPACE} is equivalent to \x{20}, but no implementation follows it.

Review Note: ICU4J allows string ranges such as [{aa}-{zz}] (all 2-letter lowercase ASCII strings). ICU4C disallows string ranges, but also disallows bracketed-element in ranges, thus disallowing [{a}-{z}]. UTS35 used to allow string ranges, but they were retracted, leaving only the single-character [{a}-{z}]. ICU4X follows UTS35 and allows for ranges of bracketed-element, but not string ranges.

Experience in CLDR has shown that the systematic usage of brackets is useful in avoiding surprises with combining marks:
`[\p{Latn} - \p{Changes_When_NFKC_Casefolded} & [a-ä]]` is a set of 31 Latin letters equal to `[a-z áàâäã]`, whereas
`[\p{Latn} - \p{Changes_When_NFKC_Casefolded} & [a-ä]]` is equal to `[a-q]`, because `[a-ä]` is `[a-q \N{COMBINING DIAERESIS}]`. If brackets are used, `[\p{Latn} - \p{Changes_When_NFKC_Casefolded} & [{a}-{ä}]]` remains valid, but
`[\p{Latn} - \p{Changes_When_NFKC_Casefolded} & [{a}-{ä}]]` is a syntax error, exposing the issue.

As a result, we are proposing to allow bracketed-element as a RangeElement, while disallowing string ranges.

3.1 Semantics

A RangeElement represents the single code point represented by its constituent lexical element.

A Range represents the set of code points that are both greater than or equal to the code point represented by the initial RangeElement and less than or equal to the final RangeElement. If the code point represented by the initial RangeElement is greater than the code point represented by the final RangeElement, the UnicodeSet expression is ill-formed.

> **Examples:** The Range a-z represents a set of 26 elements. The Range z-a is not the empty set; it is ill-formed.

An UnescapedHyphenMinus represents the set whose sole element is U+002D - HYPHEN-MINUS.

> Review Note: ICU4C and ICU4J also support a final `$` in a Union, which represents U+FFFF. However, this is better understood as a conformant extension designed for an environment where U+FFFF signals string boundaries, in particular for use in higher-level syntaxes such as transliterator rules. This is therefore discussed in the sections on conformance and higher-level syntaxes. [TODO: Which I have not yet written.]

A Complement represents the code point complement of the set represented by its constituent Union, that is, the set of code points not in the set represented by the Union.

An Intersection represents the intersection of the sets represented by the Restriction and Factor either side of the `&`.

A Difference represents the set of elements of set represented by the Restriction that are not elements of the set represented by the UnicodeSet.

For all other syntactic categories defined in the UnicodeSet grammar, the construct represent the union of the sets represented by their immediate constituent constructs.

> **Examples:** The UnicodeSet [ac-z] contains twenty-five elements; it is the union of the sets represented by the Element `a` and the Range `c-z`.
>
> **Note:** The empty Terms represents the empty set, and the UnicodeSet `[]` is therefore the empty set.
>
> **Note:** The operators `&` (intersection) and `-` (set difference) have equal precedence and left-to-right associativity: `[ [a-z] - [c] & [d] ]` is equal to `[d]`, whereas `[ [a-z] - [[c] & [d]]]` is the empty set. Set union, denoted by juxtaposition, has a lower precedence: `[ [a-z] - [c] [d] ]` is equal to `[a-b d-z]`, whereas `[ [a-z] - [[z] [d]] ]` is equal to `[a-b e-z]`.

4 Conformance

An implementation of UnicodeSet syntax is *consistent* if, for every valid UnicodeSet expression defined by this specification, the implementation either rejects the expression or evaluates it according to this specification.

> **Example:** An implementation that rejects any input string is consistent. An implementation is consistent if it rejects any UnicodeSet expression that makes use of the syntactic categories whose definition has a gray background in the grammar, but accepts and correctly interprets all other UnicodeSet expressions. An implementation which interprets `[a]` and `[b]` as the same set is not consistent. An implementation which interprets `[\d]` as `\p{Nd}` is not consistent.
>
> **Note:** Consistency is not required of conformant implementation, as it prevents the use of notations that are common in regular expressions, such as `\d` for digits, or the use of identifiers without sigils, as in [UAX14]. However, since they lead to interoperability issues when reusing an expression in another implementation, the inconsistencies must be declared.

An implementation that interprets expressions that are not valid UnicodeSet expressions according to this specification implements a *pure extension*.

> **Example:** The following are pure extensions:
>
> - Accepting a final `$` in a Union and interpreting it as representing the character U+FFFF.
> - Interpreting a non-negated property-query whose ucd-identifier is `exemplar` as the set of all characters that are CLDR exemplars for the language whose language code is given by the property-predicate.
> - Accepting the operators `--` as set difference and `&&` as set intersection, in addition to `-` and `&`.
>
> **Note:** The International Components for Unicode interpret a final `$` in a Union as U+FFFF. This is related to the behavior of out-of-range indexing in ICU, which returns U+FFFF as a sentinel value. A character class containing U+FFFF can therefore be used to match the end of a string.

An implementation of UnicodeSet syntax is *syntactically complete* if, for some subset of lexical elements which contains at least all set-operators, it supports all productions of the UnicodeSet grammar and interprets them according to this document.

> **Examples:** As the syntactic categories whose definitions have a gray background in the grammar are part of the grammar of lexical elements, an implementation is syntactically complete if does not support these, but accepts and correctly interprets all other UnicodeSet expressions.
>
> An implementation is not syntactically complete if it supports the entirety of the property-query grammar, but does not support the Complement syntax.
>
> A syntactically complete implementation interprets `[]` as the empty set and `[^]` as the set of all code points.
>
> **Note:** A syntactically complete implementation need not be consistent. For instance, such an implementation can remove `\d` from the set of escaped-elements, give it the meaning of `\p{Nd}`, and add it as an alternative in UnicodeSet. It would therefore give `[\d]` a different meaning than that given by this specification.

A syntactically complete implementation is *minimally consistent* if, for any lexical element in the following list, the implementation either rejects the lexical element, or interprets it according to this specification:

- Any escaped-element with constituent hexadecimal-digits.
- Any named-element.
- Any property-query.

Note: The definition of syntactic completeness requires that a minimally consistent implementation interpret all set-operators according to this specification.

Example: An implementation can be minimally consistent even if it interprets `\d` as the set `\p{Nd}` rather than as an escaped-element. An implementation that interprets `\p{IsGreek}` as the set of code points in the Greek and Coptic block, instead of the set of characters with Script=Greek, is not minimally consistent.

UTS61-C1 *A conformant implementation of UnicodeSet syntax shall be syntactically complete and minimally consistent.*

Example: An implementation that interprets `\p{IsGreek}` as the set of code points in the Greek and Coptic block is not a conformant UnicodeSet implementation.

UTS61-C2 *A conformant implementation of UnicodeSet syntax shall declare any restrictions to the set of lexical elements defined by this syntax.*

Note: A lack of support for the syntactic categories defined with a gray background can be described as “supporting only property queries that are recommended for general-purpose APIs”. Support for a subset of UCD properties in property queries is easiest to describe by enumerating the supported properties.

UTS61-C3 *A conformant implementation of UnicodeSet syntax that is not consistent shall declare itself as a tailoring of UnicodeSet syntax. It shall declare the expressions that are interpreted differently from this specification.*

Example: A syntactically complete and minimally consistent implementation that excludes XID_Continue characters from literal-element, adds default identifiers to the UnicodeSet production, and interprets x as `\p{lb=`x`}` for any default identifier x, is not consistent, since it interprets `[QU]` as a different set from `[{Q} {U}]`. It is a conformant tailoring of UnicodeSet syntax.

5 Use in APIs

The support of version-qualifier require carrying a long-obsolete versions of the Unicode Character Database; this represents a large amount of data, and a burden on implementers to support variations in format over the years. It is therefore not recommended for general-purpose APIs.

Similarly, the support of property-comparison and regular-expression-match in a property-query requires a significant amount of bespoke logic from implementers, and are primarily useful for exploratory queries on the Unicode Character Database, rather than to define character classes used in practical application. It is not recommended for general-purpose APIs.

General-purpose APIs should not expose the properties that are contributory, obsolete, deprecated, or otherwise not recommended for support in public property APIs. See *Section 5.1, Property Index*, in [UAX44].

Note: UnicodeSet expressions using such properties are well-defined, and it is useful for them to be supported in tools used in the development of the Unicode Standard. For instance, the stability policy statement that decomposition mappings are limited to a single value or a pair can be checked by verifying that the sets `[ \p{Decomposition_Type=Canonical} & \p{Decomposition_Mapping=} ]` and `[ \p{Decomposition_Type=Canonical} & \p{Decomposition_Mapping=/.../} ]` are empty, even though Decomposition_Type is not appropriate for general-purpose APIs.

6 Use in Higher-Level Syntaxes

UnicodeSet syntax can be used within higher-level syntaxes. In particular, as it defines a syntax for character classes, it can be used for the character classes in a regular expression syntax.

In many cases, it can be useful to include variables in a higher-level syntax based on UnicodeSet. A syntax allowing variables in UnicodeSet syntax should incorporate the identifiers into the grammar. Textual replacement prior to parsing the UnicodeSet syntax is not advisable, as it results in misleading behaviour: `[ $x $y $z ]` would be the range `[a-z]` for `$x=a`, `$y=-`, `$z=z`, but the three-element set `[az-]` for `$x=a`, `$y=z`, `$z=-`.

The UnicodeSet syntax disallows an unescaped U+0024 $ DOLLAR SIGN, so identifiers starting with $ can be made a lexical element as a pure extension of the syntax. Alternatively, default identifiers as defined in [UAX31] may be used. If default identifiers are used, characters with the XID_Start property must be removed from the syntactic category literal-element.

Example: In [UAX14], short aliases of Line_Break property values stand for the set of code points with that property; for instance, `QU` stands for `\p{lb=QU}`. If the algorithm were to special-case the letter Q in one of its regular expressions, it would need to refer to it using an escaped-element such as `\x51`, a named-element such as `\N{LATIN CAPITAL LETTER Q}`, or a bracketed-element such as `{Q}`.

In addition to defining a lexical element identifier, a syntax using UnicodeSet with identifiers must incorporate this lexical element in the UnicodeSet grammar. If the variables can only represent sets, identifier can be added as an alternative in the UnicodeSet production without further complication: `[$a-$b]` is then always a set difference. If the variables are also allowed to represent single code points for use in ranges, the category variable can be added as an alternative in the RangeElement production. This makes the grammar ambiguous (that is, it has a reduce-reduce conflict), so that the types of the variables must be known to parse it correctly: `[$a-$b]` may be a range, a set difference, or erroneous depending on the types of `$a` and `$b`.

Review Note: The Unicode invariant tests, the implementation of segmentation rules in the Unicode tools, and ICU transliterators all support variables in UnicodeSets, all using variables with `$sigils`.

The invariant tests and segmentation rules use textual replacement, but check that the values of the variables are valid UnicodeSet expressions; except for special handling of \N with the grammar as amended here, this is equivalent to having identifier as an alternative in UnicodeSet.

The ICU4C and ICU4J transliterators use textual replacement, but do not check that the variables are valid UnicodeSet expressions. The variables are used in ranges in practice by some transliterators in CLDR.

The ICU4X implementation of transliterators incorporates variables into its UnicodeSet grammar, using the types to disambiguate, but disallowing a variable from turning into a set operator.

As part of a higher-level syntax that allows comments, it can be useful to allow comments within multiline UnicodeSet expressions. In that case, the comment initiator character must be removed from the literal-element category. The character U+0023 # NUMBER SIGN is a common choice, being compatible with the comment syntax of many space-insensitive regular expression syntaxes.

Review Note: The Unicode invariant tests allow comments in multiline UnicodeSet expressions.

7 Best Practices

7.1 Escaping

The use of an escaped-element with a constituent escapable-character is not recommended when that escapable-character is neither a space (U+0020) nor a Pattern_Syntax character; such unnecessary escaping is especially ill-advised for letters in the Basic Latin block. Indeed, escape sequences consisting of a Basic Latin letter frequently have a different meaning in higher level syntaxes. This is in particular the case in regular expressions, where, for instance, `\d` typically stands for digits (`\p{Nd}` or `[0-9]` depending on the implementation), rather than the letter U+0064 d LATIN SMALL LETTER D.

Conversely, it is recommended to escape the character U+0023 # NUMBER SIGN, as it may be a comment initiator in higher-level syntaxes.

7.2 Bidirectional display

TODO Describe the atoms for the purpose of https://www.unicode.org/reports/tr55/#Conversion-To-Plain-Text.

7.3 Style Guide for Unicode Specifications

Many aspects of UnicodeSet syntax exist for compatibility with existing practice in regular expression and other pattern syntaxes. Prominent examples are the profusion of escape syntaxes, including octal, and the dual POSIX-style `[:...:]` and `\p{...}` options. The specification includes these options to ensure that standard UnicodeSet expressions are interoperable with commonly-used UnicodeSet implementations, and that commonly-used UnicodeSet expressions are well-defined.

However, actually using multiple redundant options is detrimental to the clarity of specifications. As a result, a limited subset of UnicodeSet syntax is used in the text of the Unicode Standard and associated Unicode Technical Reports. The rules in this section define this limited subset.

Besides making a choice between redundant alternatives, the subset of UnicodeSet syntax used in Unicode specifications also excludes some of the advanced features that function as a query language on the UCD. While it is valuable in the preparation of the standard to have a well-defined notation for discussing the relation between properties, or historical values of properties, the actual standard should not rely on these constructs. If a set defined by a relation between properties is useful to an algorithm, it should be turned into a derived binary property, instead of requiring users of the standard to derive it themselves.

UTS61-SG1 Do not use POSIX-style property queries.

UTS61-SG2 Use only the posix-start `\p`, not `\P`. Use a binary-query-expression with `=No` or `≠` instead of negating a unary-query-expression with `\P`.

UTS61-SG3 Prefer changing an intersection to a difference, or vice-versa, to using a negated property query as its right-hand side.

UTS61-SG4 Only use the following escaped-elements:

- `\u` four-hexadecimal-digits
- `\x{` hexadecimal-digits `}`

UTS61-SG5 Do not use regular-expression-match, property-comparison, or version-qualifier.

Table 1. Style Guide Examples

Rule	Do not use	Use instead
UTS61-SG1	`[:Lowercase_Letter:]`	`\p{Lowercase_Letter}`
UTS61-SG2	`\P{Unassigned}`	`\p{General_Category≠Unassigned}`
	`\P{Deprecated}`	`\p{Deprecated=No}`
UTS61-SG3	`[ [\u0000-\uFFFF] & \p{General_Category≠Unassigned} ]`	`[ [\u0000-\uFFFF] - \p{Unassigned} ]`
UTS61-SG4	`\0`	`\u0000`
	`\U00010FFFF`	`\x{10FFFF}`
UTS61-SG5	`\p{Uppercase≠@Changes_When_Lowercased@}`	`[ [\p{Uppercase}\p{Changes_When_Lowercased}] - [\p{Uppercase}&\p{Changes_Wh`
	`\p{Bidi_Paired_Bracket=@none@}`	`\p{Bidi_Paired_Bracket_Type=None}`
	`\p{scf≠@cf@}`	(If this set is useful in an algorithm, a property should be defined for it.)

Review Note: Many more rules will be added in subsequent drafts.

References

Review Note: The list of references will be updated in a future draft of this document.

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

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

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

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

[UTS18] *Unicode Technical Standard #18: Unicode Regular Expressions*
Latest version:
https://www.unicode.org/reports/tr18/

Acknowledgements

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

Thanks also to the following people for their feedback or contributions to this document: Mark Davis, Asmus Freytag,

Modifications

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

Revision 1

-