

## Proposed Update Unicode® Technical Standard #18

# UNICODE REGULAR EXPRESSIONS

Version	24
Editors	Mark Davis
Date	2023-05-11
This Version	<a href="https://www.unicode.org/reports/tr18/tr18-24.html">https://www.unicode.org/reports/tr18/tr18-24.html</a>
Previous Version	<a href="https://www.unicode.org/reports/tr18/tr18-23.html">https://www.unicode.org/reports/tr18/tr18-23.html</a>
Latest Version	<a href="https://www.unicode.org/reports/tr18/">https://www.unicode.org/reports/tr18/</a>
Latest Proposed Update	<a href="https://www.unicode.org/reports/tr18/proposed.html">https://www.unicode.org/reports/tr18/proposed.html</a>
Revision	24

## Summary

*This document describes guidelines for how to adapt regular expression engines to use Unicode.*

## 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*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time.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

- 0 [Introduction](#)
  - 0.1 [Notation](#)
    - 0.1.1 [Character Classes](#)
    - 0.1.2 [Property Examples](#)
  - 0.2 [Conformance](#)
- 1 [Basic Unicode Support: Level 1](#)
  - 1.1 [Hex Notation](#)
    - 1.1.1 [Hex Notation and Normalization](#)
  - 1.2 [Properties](#)
    - 1.2.1 [Domain of Properties](#)
    - 1.2.2 [Codomain of Properties](#)
    - 1.2.3 [Examples of Properties](#)
    - 1.2.4 [Property Syntax](#)
    - 1.2.5 [General Category Property](#)
    - 1.2.6 [Script and Script Extensions Properties](#)
    - 1.2.7 [Age](#)
    - 1.2.8 [Blocks](#)

1.3	<a href="#">Subtraction and Intersection</a>
1.4	<a href="#">Simple Word Boundaries</a>
1.5	<a href="#">Simple Loose Matches</a>
1.6	<a href="#">Line Boundaries</a>
1.7	<a href="#">Code Points</a>
2	<a href="#">Extended Unicode Support: Level 2</a>
2.1	<a href="#">Canonical Equivalents</a>
2.2	<a href="#">Extended Grapheme Clusters and Character Classes with Strings</a>
2.2.1	<a href="#">Character Classes with Strings</a>
2.3	<a href="#">Default Word Boundaries</a>
2.4	<a href="#">Default Case Conversion</a>
2.5	<a href="#">Name Properties</a>
2.5.1	<a href="#">Individually Named Characters</a>
2.6	<a href="#">Wildcards in Property Values</a>
2.7	<a href="#">Full Properties</a>
2.8	<a href="#">Optional Properties</a>
3	<a href="#">Tailored Support: Level 3 (Retracted)</a>
	<a href="#">Annex A: Character Blocks</a>
	<a href="#">Annex B: Sample Collation Grapheme Cluster Code (Retracted)</a>
	<a href="#">Annex C: Compatibility Properties</a>
	<a href="#">Annex D: Resolving Character Classes with Strings and Complement</a>
	<a href="#">Annex E: Notation for Properties of Strings</a>
	<a href="#">Annex F: Parsing Character Classes</a>
	<a href="#">References</a>
	<a href="#">Acknowledgments</a>
	<a href="#">Modifications</a>

## 0 Introduction

Regular expressions are a powerful tool for using patterns to search and modify text. They are a key component of many programming languages, databases, and spreadsheets. Starting in 1999, this document has supplied guidelines and conformance levels for supporting Unicode in regular expressions. The following issues are involved in supporting Unicode.

- Unicode is a large character set—regular expression engines that are only adapted to handle small character sets will not scale well.
- Unicode encompasses a wide variety of languages which can have very different characteristics than English or other western European text.

There are **two** fundamental levels of Unicode support that can be offered by regular expression engines:

- **Level 1: Basic Unicode Support.** At this level, the regular expression engine provides support for Unicode characters as basic logical units. (This is independent of the actual serialization of Unicode as UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, or UTF-32LE.) This is a minimal level for useful Unicode support. It does not account for end-user expectations for character support, but does satisfy most low-level programmer requirements. The results of regular expression matching at this level are independent of country or language. At this level, the user of the regular expression engine would need to write more complicated regular expressions to do full Unicode processing.
- **Level 2: Extended Unicode Support.** At this level, the regular expression engine also accounts for extended grapheme clusters (what the end-user generally thinks of as a character), better detection of word boundaries, and canonical equivalence. This is still a default level—independent of country or language—but provides much better support for end-user expectations than the raw level 1, without the regular-expression writer needing to know about some of the complications of Unicode encoding structure.

In particular:

1. Level 1 is the minimally useful level of support for Unicode. All regex implementations dealing with Unicode should be at least at Level 1.
2. Level 2 is recommended for implementations that need to handle additional Unicode features. This level is achievable without too much effort. However, some of the subitems in Level 2 are more important than others: see [Level 2](#).

One of the most important requirements for a regular expression engine is to document clearly what Unicode features are and are not supported. Even if higher-level support is not currently offered, provision should be made for the syntax to be extended in the future to encompass those features.

**Note:** The Unicode Standard is constantly evolving: new characters will be added in the future. This means that a regular expression that tests for currency symbols, for example, has different results in Unicode 2.0 than in Unicode 2.1, which added the euro sign currency symbol.

At any level, efficiently handling properties or conditions based on a large character set can take a lot of memory. A common mechanism for reducing the memory requirements—while still maintaining performance—is the two-stage table, discussed in Chapter 5 of *The Unicode Standard* [Unicode]. For example, the Unicode character properties required in [RL1.2 Properties](#) can be stored in memory in a two-stage table with only 7 or 8 Kbytes. Accessing those properties only takes a small amount of bit-twiddling and two array accesses.

**Note:** For ease of reference, the section ordering for this document is intended to be as stable as possible over successive versions. That may lead, in some cases, to the ordering of the sections being less than optimal.

## 0.1 Notation

In order to describe regular expression syntax, an extended BNF form is used:

Syntax	Meaning
$x\ y$	the sequence consisting of $x$ then $y$
$x^*$	zero or more occurrences of $x$
$x?$	zero or one occurrence of $x$
$x\  \ y$	either $x$ or $y$
$(\ x\ )$	for grouping
"XYZ"	terminal character(s)

The text also uses the following notation for sets in describing the behavior of Character Classes.

Symbol	Description	Example	Equivalent
$\alpha, \beta, \gamma, \dots$	A code point or multi-code-point string	a, ab, 🐼	n/a
A, B, C, ...	A set of code points and/or strings	A	n/a
$\{\dots\}$	A set of literal items, comma delimited	$\{\alpha, \beta\}$	n/a
$\mathbb{P}$	The set of all code points (= strings with single code points)	$\mathbb{P} \cap \{a, ab, \text{🐼}\}$	$\{a\}$
$\mathbb{S}$	The set of all strings (zero or more codepoints)	$\mathbb{S} \cap \{a, ab, \text{🐼}\}$	$\{a, ab, \text{🐼}\}$
$A \cup B$	Union	$\{\alpha, \beta\} \cup \{\beta, \gamma\}$	$\{\alpha, \beta, \gamma\}$
$A \cap B$	Intersection	$\{\alpha, \beta\} \cap \{\beta, \gamma\}$	$\{\beta\}$
$A \setminus B$	Set Difference	$\{\alpha, \beta\} \setminus \{\beta, \gamma\}$	$\{\alpha\}$
$A \ominus B$	Symmetric Difference	$\{\alpha, \beta\} \ominus \{\beta, \gamma\}$	$\{\alpha, \gamma\}$
$C_S A$	Full <b>Complement</b> (all <b>strings</b> except those in A)	$\mathbb{C}A$ (= $C_S A$ )	$\mathbb{S} \setminus A$
$C_P A$	Code Point <b>Complement</b> (all <b>code points</b> except those in A)	$\mathbb{C}_P A$	$\mathbb{P} \setminus A$

The Full Complement of a finite set results in an infinite set. Because that is not useful for regular expressions, the complement operations such as  $[\^{\dots}]$  are interpreted as Code Point Complement.

Note that the examples of characters having a given property use snapshots from a particular version of Unicode, and may not match those in the latest version of Unicode. In addition, note that the property assignments from the respective data file are normative. The descriptions of any of the character properties in Unicode specifications include examples of representative or interesting characters for each property, but always refer to the respective data file for the complete and up-to-date property values.

### 0.1.1 Character Classes

A Character Class represents a set of characters. When a regex implementation follows [Section 2.2.1 Character Classes with Strings](#) the set can include sequences of characters as well. The following syntax for Character Classes is used and extended in successive sections. This syntax is not normative: regular expression implementations may need to use different syntax to be consistent with their current syntax.

Nonterminal	Production Rule	Comments & Constraints
CHARACTER_CLASS	<code>:= '[' COMPLEMENT? SEQUENCE '']</code>	If complement is present, it is $\mathbb{C}_P A$ , the set of all code points <i>except</i> those in SEQUENCE.
SEQUENCE	<code>:= ITEM+</code>	union of items: $A \cup B \dots$ This is replaced with operators in <a href="#">RL1.3 Subtraction and Intersection</a>
ITEM	<code>:= LITERAL ('-' LITERAL)?</code> <code>:= CHARACTER_CLASS</code>	<i>Constraint:</i> parse error if in range with 1st literal > 2nd literal (some Regex Engines may allow them to be identical without an error)
		[a] $s = a$
		[a-j] $\text{len}(s) == 1 \text{ AND } s \geq a \text{ AND } s \leq j$
LITERAL	<code>:= ESCAPE (SYNTAX_CHAR   SPECIAL_CHAR)</code> <code>:= NON_SYNTAX_CHAR</code>	Different variants of SYNTAX_CHAR, SPECIAL_CHAR, and NON_SYNTAX_CHAR can be used for particular contexts to maintain compatibility
COMPLEMENT	<code>:= '^'</code>	
ESCAPE	<code>:= '\'</code>	
SYNTAX_CHAR	<code>:= [\ - \[ \] \{ \} / \ \ ^ ]</code>	
SPECIAL_CHAR	<code>:= [abcefnrtu]</code>	The exact set of SPECIAL_CHAR may vary across Regex engines
NON_SYNTAX_CHAR	<code>:= [^SYNTAX_CHAR]</code>	$[\^{\text{SYNTAX\_CHAR}}]$ means all valid Unicode code points except for those in SYNTAX_CHAR
SP	<code>:= ' '+</code>	

The EBNF can be enhanced with other features. For example, to allow ignored spaces for readability, it can add  $\backslash u\{20\}$  to SYNTAX\_CHAR, and add SP? around various elements, change ITEM+ to SP? ITEM (SP? ITEM)+, etc. In this document, SP is allowed between any elements in examples, but to simplify the presentation those changes are omitted from the EBNF.

In subsequent sections of this document, additional EBNF lines will be added for additional features. In one case, marked in a comment, one of the above lines will be replaced.

Complementing affects the entire value in square brackets. That is,  $[\^{\text{abcm-z}}] = [\^{\text{abcm-z}}]$ . It is defined to be the *Code Point Complement*  $= \mathbb{P} \setminus A$ , and consists of the set of all code points that are *not* in the enclosed character class. Using syntax introduced below,  $[\^A]$  is equivalent to  $[\text{p}\{\text{any}\}--[\text{A}]]$  or to an expression with the equivalent literal,  $[\backslash u\{0\}-\backslash u\{10FFFF\}--[A]]$ .

See [Annex D: Resolving Character Classes with Strings and Complement](#) for details.

For the purpose of regular expressions, in this document the terms “character” and “code point” are used interchangeably. Similarly, the terms “string” and “sequence of code points” are used interchangeably. Typically the code points of interest will be those representing characters. A Character Class is also referred to as the set of all characters specified by that Character Class.

In addition, for readability the simple parentheses are used where in practice a non-capturing group would be used. That is, (ab|c) is written instead of (?ab|c).

Code points that are syntax characters or whitespace are typically escaped. For more information see [UAX31]. In examples, the syntax “\s” is sometimes used to indicate whitespace. See also [Annex C: Compatibility Properties](#). Also, in many regex implementations, the first position after the opening '[' or '['^ is treated specially, with some syntax chars treated as literals.

**Note:** This is only a **sample** syntax for the purposes of examples in this document. Regular expression syntax varies widely: the issues discussed here would need to be adapted to the syntax of the particular implementation. However, it is important to have a concrete syntax to correctly illustrate the different issues. In general, the syntax here is similar to that of [Perl Regular Expressions](#) [Perl].) In some cases, this gives multiple syntactic constructs that provide for the same functionality.

The following table gives examples of Character Classes:

Character Class	Matches
[a-z    A-Z    0-9]	ASCII alphanumerics
[a-z A-Z 0-9]	
[a-zA-Z0-9]	
[^a-z A-Z 0-9]	all code points except ASCII alphanumerics
[ \ ] \- \ ]	the literal characters ], -, <space>

Where string offsets are used in examples, they are from zero to n (the length of the string), and indicate positions *between* characters. Thus in "abcde", the substring from 2 to 4 includes the two characters "cd".

The following additional notation is defined for use here and in other Unicode specifications:

Syntax	Meaning	Note
\n+	As used within regular expressions, expands to the text matching the <b>n</b> <sup>th</sup> parenthesized group in the regular expression. (à la Perl)	<b>n</b> is an ASCII digit. Implementations may impose limits on the number of digits.
\$n+	As used within replacement strings for regular expressions, expands to the text matching the <b>n</b> <sup>th</sup> parenthesized group in a corresponding regular expression. (à la Perl)	The value of \$0 is the entire expression.

Because any character could occur as a literal in a regular expression, when regular expression syntax is embedded within other syntax it can be difficult to determine where the end of the regex expression is. Common practice is to allow the user to choose a delimiter like '/' in /ab(c)\*/. The user can then simply choose a delimiter that is not in the particular regular expression.

### 0.1.2 Property Examples

All examples of properties being equivalent to certain literal character classes are illustrative. They were generated at a point in time, and are not updated with each release. Thus when an example contains “p{sc=Hira} = [あ-け > -ふ 𐀀𐀁]”, it does not imply that that identity expression would be true for the current version of Unicode.

### 0.2 Conformance

The following section describes the possible ways that an implementation can claim conformance to this Unicode Technical Standard.

All syntax and API presented in this document is *only* for the purpose of illustration; there is absolutely no requirement to follow such syntax or API. Regular expression syntax varies widely: the features discussed here would need to be adapted to the syntax of the particular implementation. In general, the syntax in examples is similar to that of [Perl Regular Expressions](#) [Perl], but it may not be exactly the same. While the API examples generally follow [Java style](#), it is again *only* for illustration.

**C0.** *An implementation claiming conformance to this specification at any Level shall identify the version of this specification and the version of the Unicode Standard.*

**C1.** *An implementation claiming conformance to Level 1 of this specification shall meet the requirements described in the following sections:*

- [RL1.1 Hex Notation](#)
- [RL1.2 Properties](#)
- [RL1.2a Compatibility Properties](#)
- [RL1.3 Subtraction and Intersection](#)
- [RL1.4 Simple Word Boundaries](#)
- [RL1.5 Simple Loose Matches](#)
- [RL1.6 Line Boundaries](#)
- [RL1.7 Supplementary Code Points](#)

**C2.** *An implementation claiming conformance to Level 2 of this specification shall satisfy C1, and meet the requirements described in the following sections:*

- [RL2.1 Canonical Equivalents](#)
- [RL2.2 Extended Grapheme Clusters and Character Classes with Strings](#)
- [RL2.3 Default Word Boundaries](#)
- [RL2.4 Default Case Conversion](#)
- [RL2.5 Name Properties](#)
- [RL2.6 Wildcards in Property Values](#)
- [RL2.7 Full Properties](#)

**C3.** *This conformance clause has been removed.*

**C4.** *An implementation claiming partial conformance to this specification shall clearly indicate which levels are completely supported (C1-C2), plus any additional supported features from higher levels.*

For example, an implementation may claim conformance to Level 1, except for [Subtraction and Intersection](#).

A regular expression engine may be operating in the context of a larger system. In that case some of the requirements may be met by the overall system. For example, the requirements of Section [2.1 Canonical Equivalents](#) might be best met by making normalization available as a part of the larger system, and requiring users of the system to normalize strings where desired before supplying them to the regular-expression engine. Such usage is conformant, as long as the situation is clearly documented.

A conformance claim may also include capabilities added by an optional add-on, such as an optional library module, as long as this is clearly documented.

For backwards compatibility, some of the functionality may only be available if some special setting is turned on. None of the conformance requirements require the functionality to be available by default.

---

## 1 Basic Unicode Support: Level 1


Regular expression syntax usually allows for an expression to denote a set of single characters, such as `[a-z A-Z 0-9]`. Because there are a very large number of characters in the Unicode Standard, simple list expressions do not suffice.

### 1.1 Hex Notation

The character set used by the regular expression writer may not be Unicode, or may not have the ability to input all Unicode code points from a keyboard.

RL1.1 Hex Notation

To meet this requirement, an implementation shall supply a mechanism for specifying any Unicode code point (from U+0000 to U+10FFFF), using the hexadecimal code point representation.

The syntax must use the code point in its hexadecimal representation. For example, syntax such as `\uD834\uDD1E` or `\xF0\x9D\x84\x9E` does not meet this requirement for expressing U+1D11E (  ) because "1D11E" does not appear in the syntax. In contrast, syntax such as `\U0001D11E`, `\x{1D11E}` or `\u{1D11E}` does satisfy the requirement for expressing U+1D11E.

A sample notation for listing hex Unicode characters within strings uses "u" followed by four hex digits or "u{" followed by any number of hex digits and terminated by "}", with multiple characters indicated by separating the hex digits by spaces. This would provide for the following addition:

Nonterminal	Production Rule	Comments & Constraints
LITERAL	<code>:= HEX</code>	Adds to previous LITERAL rules.
HEX	<code>:= '\u' HEX_CHAR{4}</code> <code>:= '\u{' CODEPOINT (SP CODEPOINT)* '}'</code>	<code>\u{3b1 3b3 3b5 3b9}</code> == <code>\u{3b1}\u{3b3}\u{3b5}\u{3b9}</code>
HEX_CHAR	<code>:= [0-9A-Fa-f]</code>	
CODEPOINT	<code>:= '10' HEX_CHAR{4}   HEX_CHAR{1,5}</code>	



**Note:** `\u{3b1 3b3 3b5 3b9}` is semantic sugar — useful for readability and concision but not a requirement. It can be used anywhere the equivalent individual hex escapes could be, thus `[a-\u{3b1 3b3}-\u{3b9}]` behaves like `[a-\u{3b1}\u{3b3}-\u{3b9}]` == `[a-\u{3b1}\u{3b3}-\u{3b9}]`

The following table gives examples of this hex notation:

Syntax	Matches
<code>[\u{3040}-\u{309F}\u{30FC}]</code>	The Hiragana block (which includes some unassigned code points), plus the prolonged sound sign —
<code>[\u{B2} \u{2082}]</code>	superscript ² and subscript ₂
<code>[a \u{10450}]</code>	"a" and U+10450 SHAVIAN LETTER PEEP
<code>ab\u{63 64}</code>	"abcd"

More advanced regular expression engines can also offer the ability to use the Unicode character name for readability. See [2.5 Name Properties](#).

For comparison, the following table shows some additional examples of escape syntax for Unicode code points:

Type	Escaped Characters					Escaped String
Unescaped		€	£	a	<tab>	 €£a<tab>
Code Point†	U+1F47D	U+20AC	U+00A3	U+0061	U+0009	U+1F47D U+20AC U+00A3 U+0061 U+0009
UTS18, Ruby	<code>\u{1F47D}</code>	<code>\u{20AC}</code>	<code>\u{A3}</code>	<code>\u{61}</code>	<code>\u{9}</code>	<code>\u{1F47D 20AC A3 61 9}</code>
Swift, Javascript (ECMAScript)	<code>\u{1F47D}</code>	<code>\u{20AC}</code>	<code>\u{A3}</code>	<code>\u{61}</code>	<code>\u{9}</code>	<code>\u{1F47D}\u{20AC}\u{A3}\u{61}\u{9}</code>



Perl, Java, ICU*	\x{1F47D}	\x{20AC}	\x{A3}	\x{61}	\x{9}	\x{1F47D}\x{20AC}\x{A3}\x{61}\x{9}
C++, Python	\U0001F47D	\u20AC	\u00A3	\u0061	\u0009	\U0001F47D\u20AC\u00A3\u0061\u0009
XML, HTML	&#x1F47D;	&#x20AC;	&#xA3;	&#x61;	&#x9;	&#x1F47D;&#x20AC;&#xA3;&#x61;&#x9;
CSS†	\1F47D	\20AC	\A3	\61	\9	\1F47D \20AC \A3 \61 \9

† Following whitespace is consumed.

\* ICU4C regex + ICU UnicodeSet

### 1.1.1 Hex Notation and Normalization

The Unicode Standard treats certain sequences of characters as equivalent, such as the following:

u + grave	U+0075 ( u ) LATIN SMALL LETTER U + U+0300 ( ◌ ) COMBINING GRAVE ACCENT
u_grave	U+00F9 ( ù ) LATIN SMALL LETTER U WITH GRAVE

Literal text in regular expressions may be normalized (converted to equivalent characters) in transmission, out of the control of the authors of that text. For example, a regular expression may contain a sequence of literal characters 'u' and *grave*, such as the expression [aeiou◌◌◌] (the last three characters being U+0300 ( ◌ ) COMBINING GRAVE ACCENT, U+0301 ( ◌ ) COMBINING ACUTE ACCENT, and U+0308 ( ◌ ) COMBINING DIAERESIS. In transmission, the two adjacent characters in Row 1 might be changed to the different expression containing just one character in Row 2, thus changing the meaning of the regular expression. Hex notation can be used to avoid this problem. In the above example, the regular expression should be written as [aeiou\u{300 301 308}] for safety.

A regular expression engine may also enforce a single, uniform interpretation of regular expressions by always normalizing input text to Normalization Form NFC before interpreting that text. For more information, see UAX #15, *Unicode Normalization Forms* [UAX15].

## 1.2 Properties

Because Unicode is a large character set that is regularly extended, a regular expression engine needs to provide for the recognition of whole categories of characters as well as simply literal sets of characters and strings; otherwise the listing of characters becomes impractical, out of date, and error-prone. This is done by providing syntax for sets of characters based on the Unicode character properties, as well as related properties and functions. Examples of such syntax are \p{Script=Greek} and [:Script=Greek:], which both stand for the set of characters that have the Script value of Greek. In addition to the basic syntax, regex engines also need to allow them to be combined with other sets defined by properties or with literal sets of characters and strings. An example is [\p{Script=Greek}--\p{General\_Category=Letter}], which stands for the set of characters that have the Script value of Greek *and* that do not have the General\_Category value of Letter.

Many character properties are defined in the Unicode Character Database (UCD), which also provides the official data for mapping Unicode characters (and code points) to property values. See UAX #44, *Unicode Character Database* [UAX44] and Chapter 4 in *The Unicode Standard* [Unicode]. For use in regular expressions, properties can also be considered to be defined by Unicode definitions and algorithms, and by data files and definitions associated with other Unicode Technical Standards, such as UTS #51, *Unicode Emoji*. For example, this includes the **Basic\_Emoji** definition from UTS #51. The full list of recommended properties is in Section 2.7, *Full Properties*.

UAX #44, *Unicode Character Database* [UAX44] divides character properties into several types: Catalog, Enumeration, Binary, String, Numeric, and Miscellaneous. Those categories are not all precisely defined or immediately relevant to regular expressions. Some are more pertinent to the maintenance of the Unicode Character Database.

### 1.2.1 Domain of Properties



For regular expressions, it is more helpful to divide up properties by the treatment of their domain (what they are properties of) and their codomain (the values of the properties). Most properties are properties of Unicode code points; thus their domains are simply the full set of Unicode code points. Typically the important information is for the subset of the code points that are characters; therefore, those properties are often also called properties of characters.

In addition to properties of characters, there are also properties of strings (sequences of characters). A property of strings is more general than a property of characters. In other words, any property of characters is also a property of strings; its domain is, however, limited to strings consisting of a single character.

Data, definitions, and properties defined by the Unicode Standard and other Unicode Technical Standards, which map from strings to values, can thus be specified in this document as defining regular-expression properties.

A complement of a property of strings or a Character Class with strings may not be valid in regular expressions. For more information, see [Annex D: Resolving Character Classes with Strings and Complement](#) and [Section 2.2.1 Character Classes with Strings](#).

### 1.2.2 Codomain of Properties

The values (codomain) of properties of characters (or strings) have the following simple types: Binary, Enumerated, Numeric, Code Point, and String. Properties can also have multivalued types: a Set or List of other types.

The Binary type is a special case of an Enumerated type limited to precisely the two values "True" and "False". In general, a property of Enumerated type has a longer list of defined values. Those defined values are abstractions, but they are identified in the Unicode Character Database with labels known as aliases. Thus, the Script value "Devanagari" may also be identified by the abbreviated alias "Deva"—both refer to the same enumerated value, even though the exact label for that value may differ.

The Code Point type is a special case of a String type where the values are always limited to single-code point strings.

The UCD "Catalog" type is the same as Enumerated (the name differs for historical reasons).

### 1.2.3 Examples of Properties

The following tables provide some examples of property values for each domain type.

#### Examples of Properties of Characters

Type	Property Name	Code Point	Character	Value	Regex Literal
Binary	White_Space	U+0020	" "	True	
	Emoji	U+231A	🕒	True	
Enumerated	Script	U+3032	↵	Common	
Code point	Simple_Lowercase_Mapping	U+0041	A	"a"	\u{61}
String	Name	U+0020	" "	"SPACE"	\u{53 50 41 43 45}
Set	Script_Extensions	U+3032	↵	{Hira, Kana}	




**Note:** The Script\_Extensions property maps from code points to a *set* of enumerated Script property values.

Expressions involving Set properties, which have multiple values, are most often tested for containment, not equality. An expression like `\p{Script_Extensions=Hira}` is interpreted as containment: matching each code point *cp* such that `Script_Extensions(cp) ⊇ {Hira}`. Thus, `\p{Script_Extensions=Hira}` will match both U+3032 ↵ VERTICAL KANA REPEAT WITH VOICED SOUND MARK (with value {Hira Kana}) and U+3041 あ HIRAGANA LETTER SMALL A (with value {Hira}). That also allows the natural replacement of

the regular expression `\p{Script=Hira}` by `\p{Script_Extensions=Hira}` — the latter just adds characters that may be *either* Hira or some other script. For a more detailed example, see [Section 1.2.6 Script and Script Extensions Properties](#).

Expressions involving List properties may be tested for containment, but may have different semantics for the elements based on position. For example, each value of the [kMandarin](#) property is a list of up to two String values: the first being preferred for zh-Hans and the second for zh-Hant (where the preference differs).

### Examples of Properties of Strings

Type	Property Name	Code Point(s)	Character(s)	CLDR Name	Value
Binary	Basic_Emoji	U+231A		watch	True
		U+23F2 U+FE0F		timer clock	True
		U+0041	A		False
		U+0041 U+0042	"AB"		False
	RGI_Emoji_Flag_Sequence	U+1F1EB U+1F1F7		flag: France	True

**Note:** Properties of strings can always be “narrowed” to just contain code points. For example, `[\p{Basic_Emoji} && \p{any}]` is the set of characters in Basic\_Emoji.

### 1.2.4 Property Syntax

The recommended names (identifiers) for UCD properties and property values are in [PropertyAliases.txt](#) and [PropertyValueAliases.txt](#). There are both abbreviated names and longer, more descriptive names. It is strongly recommended that both names be recognized, and that loose matching of property names and values be implemented following the guidelines in [Section 5.9 Matching Rules](#) in [UAX44].

**Note:** It may be a useful implementation technique to load the Unicode tables that support properties and other features on demand, to avoid unnecessary memory overhead for simple regular expressions that do not use those properties.

Where a regular expression is expressed as much as possible in terms of higher-level semantic constructs such as *Letter*, it makes it practical to work with the different alphabets and languages in Unicode. The following is an example of a syntax addition that permits properties. Following Perl Syntax, the *p* is lowercase to indicate a positive match, and uppercase to indicate a complemented match.

Nonterminal	Production Rule	Comments & Constraints
CHARACTER_CLASS	<pre> := '\ ' [pP] '{' PROP_SPEC '}' := '[' : ' COMPLEMENT? PROP_SPEC ':' ]' </pre>	<p>Adds to previous CHARACTER_CLASS rules.</p> <p><code>[:X:]</code> is older notation, and is defined to be identical to <code>\p{X}</code></p> <p><code>\P{X}</code> and <code>[:^X:]</code> are defined to be identical to <code>[\p{X}]</code>, that is, the Code Point Complement of <code>\p{X}</code>.</p>
PROP_SPEC	<pre>:= PROP_NAME (RELATION PROP_VALUE)?</pre>	
PROP_NAME	<pre>:= ID_CHAR+</pre>	<p><b>Constraint:</b> PROP_NAME = valid Unicode property name or alias (<a href="#">RL1.2 Properties</a>, <a href="#">2.7 Full Properties</a>, <a href="#">RL2.7 Full Properties</a>), or optional property name or alias (<a href="#">2.8 Optional Properties</a>)</p>
ID_CHAR	<pre>:= [A-Za-z0-9\ \- _]</pre>	
RELATION	<pre>:= '='   '!='   '!='</pre>	
PROP_VALUE	<pre>:= LITERAL*</pre>	<p><b>Constraint:</b> PROP_VALUE = valid Unicode property value for that PROP_NAME</p>

The following table shows examples of this extended syntax to match properties:

Syntax	Matches
<code>[\p{L} \p{Nd}]</code>	all letters and decimal digits
<code>[\p{letter} \p{decimal number}]</code>	
<code>[\p{letter decimal number}]</code>	
<code>[\p{L Nd}]</code>	
<code>\P{script=greek}</code>	all code points except those with the Greek script property
<code>\P{script=greek}</code>	
<code>\p{script≠greek}</code>	
<code>[ :^script=greek: ]</code>	
<code>[ :^script:greek: ]</code>	
<code>[ :script≠greek: ]</code>	anything that has the enumerated property value East_Asian_Width = Narrow
<code>\p{East Asian Width:Narrow}</code>	
<code>\p{Whitespace}</code>	
<code>\p{scx=Kana}</code>	The match is to all characters whose Script_Extensions property value <i>includes</i> the specified value(s). So this expression matches U+30FC, which has the Script_Extensions value {Hira, Kana}

Some properties are binary: they are either true or false for a given code point. In that case, only the property name is required. Others have multiple values, so for uniqueness both the property name and the property value need to be included.

For example, **Alphabetic** is a binary property, but it is also a value of the enumerated Line\_Break property. So `\p{Alphabetic}` would refer to the binary property, whereas `\p{Line Break:Alphabetic}` or `\p{Line_Break=Alphabetic}` would refer to the enumerated Line\_Break property.

There are two exceptions to the general rule that expressions involving properties with multiple value should include both the property name and property value. The **Script** and **General\_Category** properties commonly have their property name omitted. Thus `\p{Unassigned}` is equivalent to `\p{General_Category = Unassigned}`, and `\p{Greek}` is equivalent to `\p{Script=Greek}`.

## RL1.2 Properties

*To meet this requirement, an implementation shall provide at least a minimal list of properties, consisting of the following:*

- *General\_Category*
- *Script and Script\_Extensions*
- *Alphabetic*
- *Uppercase*
- *Lowercase*
- *White\_Space*
- *Noncharacter\_Code\_Point*
- *Default\_Ignorable\_Code\_Point*
- *ANY, ASCII, ASSIGNED*

*The values for these properties must follow the Unicode definitions, and include the property and property value aliases from the UCD. Matching of Binary, Enumerated, Catalog, and Name values must follow the [Matching Rules](#) from [UAX44] with one exception: implementations are not required to ignore an initial prefix string of "is" in property values.*

### RL1.2a Compatibility Properties

To meet this requirement, an implementation shall provide the properties listed in [Annex C: Compatibility Properties](#), with the property values as listed there. Such an implementation shall document whether it is using the Standard Recommendation or POSIX-compatible properties.

In order to meet requirements [RL1.2](#) and [RL1.2a](#), the implementation must satisfy the Unicode definition of the properties for the supported version of The Unicode Standard, rather than other possible definitions. However, the names used by the implementation for these properties may differ from the formal Unicode names for the properties. For example, if a regex engine already has a property called "Alphabetic", for backwards compatibility it may need to use a distinct name, such as "Unicode\_Alphabetic", for the corresponding property listed in [RL1.2](#).

Implementers may add aliases beyond those recognized in the UCD. For example, in the case of the Age property an implementation could match the defined aliases "3.0" and "V3\_0", but also match "3", "3.0.0", "V3.0", and so on. However, implementers must be aware that such additional aliases may cause problems if they collide with future UCD aliases for *different* values.

Ignoring an initial "is" in property values is optional. Loose matching rule [UAX44-LM3](#) in [\[UAX44\]](#) specifies that occurrences of an initial prefix of "is" are ignored, so that, for example, "Greek" and "isGreek" are equivalent as property values. Because existing implementations of regular expressions commonly make distinctions based on the presence or absence of "is", this requirement from [\[UAX44\]](#) is dropped.

For more information on properties, see [UAX #44, Unicode Character Database \[UAX44\]](#).

Of the properties in [RL1.2](#), General\_Category and Script have enumerated property values with more than two values; the other properties are binary. An implementation that does not support non-binary enumerated properties can essentially "flatten" the enumerated type. Thus, for example, instead of `\p{script=latin}` the syntax could be `\p{script_latin}`.

1.2.5 General Category Property

The most basic overall character property is the General\_Category, which is a basic categorization of Unicode characters into: *Letters*, *Punctuation*, *Symbols*, *Marks*, *Numbers*, *Separators*, and *Other*. These property values each have a single letter abbreviation, which is the uppercase first character except for separators, which use Z. The official data mapping Unicode characters to the General\_Category value is in [UnicodeData.txt](#).

Each of these categories has different subcategories. For example, the subcategories for *Letter* are *uppercase*, *lowercase*, *titlecase*, *modifier*, and *other* (in this case, *other* includes uncased letters such as Chinese). By convention, the subcategory is abbreviated by the category letter (in uppercase), followed by the first character of the subcategory in lowercase. For example, *Lu* stands for *Uppercase Letter*.

**Note:** Because it is recommended that the property syntax be lenient **as to spaces, casing, hyphens and underbars**, any of the following should be equivalent: `\p{Lu}`, `\p{lu}`, `\p{uppercase letter}`, `\p{Uppercase Letter}`, `\p{Uppercase_Letter}`, and `\p{uppercaseletter}`. **More precisely, the matching rules from Section 5.9 Matching Rules of [UAX44] should be applied, notably UAX44-LM1, UAX44-LM2, and UAX44-LM3. For example, in \p{numeric-value=-0.5}, hyphen is not significant in numeric-value, but is significant in -0.5.**

The General\_Category property values are listed below. For more information on the meaning of these values, see [UAX #44, Unicode Character Database \[UAX44\]](#).

Abb.	Long form	Abb.	Long form	Abb.	Long form
<b>L</b>	<b>Letter</b>	<b>S</b>	<b>Symbol</b>	<b>Z</b>	<b>Separator</b>
Lu	Uppercase Letter	Sm	Math Symbol	Zs	Space Separator
Ll	Lowercase Letter	Sc	Currency Symbol	Zl	Line Separator
Lt	Titlecase Letter	Sk	Modifier Symbol	Zp	Paragraph Separator
Lm	Modifier Letter	So	Other Symbol	<b>C</b>	<b>Other</b>
Lo	Other Letter	<b>P</b>	<b>Punctuation</b>	Cc	Control

<b>M</b>	<b>Mark</b>	Pc	Connector Punctuation	Cf	Format
Mn	Non-Spacing Mark	Pd	Dash Punctuation	Cs	Surrogate
Mc	Spacing Combining Mark	Ps	Open Punctuation	Co	Private Use
Me	Enclosing Mark	Pe	Close Punctuation	Cn	Unassigned
<b>N</b>	<b>Number</b>	Pi	Initial Punctuation	-	Any*
Nd	Decimal Digit Number	Pf	Final Punctuation	-	Assigned*
NI	Letter Number	Po	Other Punctuation	-	ASCII*
No	Other Number				

Value	Matches	Equivalent to	Notes
Any	all code points, that is: $\mathbb{P}$	$[\backslash u\{0\}-\backslash u\{10FFFF\}]$	In some regular expression languages, $\backslash p\{Any\}$ may be expressed by a period ("."), but that usage may exclude newline characters.
Assigned	all assigned characters (for the target version of Unicode)	$\backslash P\{Cn\}$	This also includes all private use characters. It is useful for avoiding confusing double complements. Note that <i>Cn</i> includes noncharacters, so <i>Assigned</i> excludes them.
ASCII	all ASCII characters	$[\backslash u\{0\}-\backslash u\{7F\}]$	

A regular-expression mechanism may choose to offer the ability to identify characters on the basis of other Unicode properties besides the General Category. In particular, Unicode characters are also divided into scripts as described in UAX #24, *Unicode Script Property* [UAX24] (for the data file, see [Scripts.txt](#)). Using a property such as `\p{sc=Greek}` allows implementations to test whether letters are Greek or not.

Code	Char	Name	sc	scx
U+3042	あ	HIRAGANA LETTER A	Hira	{Hira}
U+30FC	ー	KATAKANA-HIRAGANA PROLONGED SOUND MARK	Zyyy = Common	{Hira, Kana}
U+3099	゛	COMBINING KATAKANA-HIRAGANA VOICED SOUND MARK	Zinh = Inherited	{Hira, Kana}
U+30FB	・	KATAKANA MIDDLE DOT	Zyyy = Common	{Bopo, Hang, Hani, Hira, Kana, Yiii}

Expression	Contents of Set <b>in Unicode 15.0</b>
<code>\p{sc=Hira}</code>	[> ズ あ-うづ え-お かか-ぐ けけげ □ こ-ぽ ぽ ま-よ ろ ら-わ □ あ □ □ へ □ を ん □ □ に □ □]

`\p{scx=Hira}` [ゝ゜〜 = ・・、 、 \ 。。 ミミ <-「」」『-』 [-] " " ° 二XX <-\ > ズ ---ゐ  
へ あ-うづ え-お かか-ぐ けけげ □こ-ぽ 𐄂ま □み-よ ㇿら-わ □ゐ □□ゑ □をん □-□に  
□-□]

See [Section 0.1.2 Property Examples](#) for information about updates to the contents of a literal set across versions.

The expression `\p{scx=Hira}` contains not only the characters in `\p{script=Hira}`, but many other characters such as U+30FC (ー), which are either Hiragana or Katakana.

In most cases, script extensions are a superset of the script values ( $\set{p\{scx=x\}} \supseteq \set{p\{sc=x\}}$ ). However, in some cases that is not true. For example, the Script property value for U+30FC ( — ) is Common, but the Script\_Extensions value for U+30FC ( — ) does not contain the script value Common. In other words,  $\set{p\{scx=Common\}} \not\supseteq \set{p\{sc=Common\}}$ .

The usage model for the `Script` and `Script_Extensions` properties normally requires that people construct somewhat more complex regular expressions, because a great many characters (Common and Inherited) are shared between scripts. Documentation should point users to the description in [UAX24]. The values for `Script_Extensions` are likely to be extended over time as new information is gathered on the use of characters with different scripts. For more information, see [The Script\\_Extensions Property](#) in UAX #24, *Unicode Script Property* [UAX24].

### 1.2.7 Age

As defined in the Unicode Standard, the Age property (in the [DerivedAge](#) data file in the UCD) specifies the first version of the standard in which each character was assigned. It does not refer to how long it has been encoded, nor does it indicate the historic status of the character.

In regex expressions, the `Age` property is used to indicate the characters that were in a particular version of the Unicode Standard. That is, a character has the `Age` property of that version or less. Thus `\p{age=3.0}` includes the letter `a`, which was included in Unicode 1.0. To get characters that are new in a particular version, subtract off the previous version as described in [1.3 Subtraction and Intersection](#). For example: `[\p{age=3.1} -- \p{age=3.0}]`.

### 1.2.8 Blocks

Unicode blocks have an associated enumerated property, the Block property. However, there are some very significant caveats to the use of Unicode blocks for the identification of characters: see [Annex A: Character Blocks](#). If blocks are used, some of the names can collide with Script names, so they should be distinguished, with syntax such as `\p{Greek Block}` or `\p{Block=Greek}`.

### 1.3 Subtraction and Intersection

As discussed earlier, character properties are essential with a large character set. In addition, there needs to be a way to "subtract" characters from what is already in the list. For example, one may want to include all non-ASCII letters without having to list every character in `\p{letter}` that is not one of those 52.

### RL1.3 Subtraction and Intersection

*To meet this requirement, an implementation shall supply mechanisms for union, intersection and set-difference of sets of characters within regular expression character class expressions.*

The following is an example of a syntax extension to handle set operations:

Nonterminal	Production Rule	Comments & Constraints
SEQUENCE	<code>:= ITEM (SEQ_EXTEND)*</code>	<i>Replaces</i> SEQUENCE definition above (which has just ITEM+)
SEQ_EXTEND	<code>:= OPERATOR CHARACTER_CLASS   ITEM</code>	<i>Constraint:</i> the last entity before the OPERATOR can also be required to be a CHARACTER_CLASS. See



		the notes below.
OPERATOR	<div><div>: = '     '</div><div>: = '&amp;&amp;'</div><div>: = '- -'</div><div>: = '~ ~'</div></div>	<div>union: <math>A \cup B</math> (explicit operator where desired for clarity)</div> <div>intersection: <math>A \cap B</math></div> <div>set difference: <math>A \setminus B</math></div> <div>symmetric difference: <math>A \oplus B = (A \cup B) \setminus (A \cap B)</math></div>

The [symmetric difference](#) of two sets is defined as being the union minus the intersection, that is  $(A \cup B) \setminus (A \cap B)$ , or equivalently, the union of the asymmetric differences  $(A \setminus B) \cup (B \setminus A)$ .

For discussions of support by various engines, see:

- <https://www.regular-expressions.info/charclassintersect.html>
- <https://www.regular-expressions.info/charclasssubtract.html>

Either set difference or symmetric difference can be used with union to produce all combinations of sets that can be used in regular expressions. They *cannot* be replaced by  $[\wedge\dots]$ , because it is defined to be Code Point Complement. For example, you cannot express  $[A \setminus B]$  as  $[A \&\&[\wedge B]]$ : the following are *not* equivalent if A contains a string s that is not in B.

Expression	Contains s?	Comment
$[A \setminus B]$	Yes	Remove everything in B from A. Because s is not in B, it remains in A
$[A \&\&[\wedge B]]$	No	Retain only <i>code points</i> that are not in B. So s is removed from A.

Code point complement can also be expressed using the property  $\setminus p\{any\}$  or the equivalent literal  $[\setminus u\{0\} \setminus u\{10FFFF\}]$ . Thus  $[\wedge A]$  is equivalent to  $[\setminus p\{any\} \setminus \setminus A]$  and to  $[[\setminus u\{0\} \setminus u\{10FFFF\}] \setminus \setminus A]$ .

See [Annex D: Resolving Character Classes with Strings and Complement](#) for details.

For clarity, it is common to use doubled symbols, and require a CHARACTER\_CLASS on both sides of the OPERATOR, such as  $[[abc] \setminus [cde]]$ . Thus  $[abc \setminus cde]$  or  $[abc \setminus [cde]]$  or  $[[abc] \setminus cde]$  would be illegal syntax, and cause a parse error. This also decreases the risk that the meaning of an older regular expression accidentally changes.

**Note:** There is no exact analog between arithmetic operations and the set operations. The operator  $||$  *adds* items to the current results, the operators  $\&\&$  and  $\setminus \setminus$  *remove* items, and the operator  $\sim \sim$  both *adds and removes* items.

This specification does not require any particular operator precedence scheme. The illustrative syntax puts all operators on the same precedence level, similar to how in arithmetic expressions work with + and -, where  $a + b - c + d - e$  is the same as  $((((a + b) - c) + d) - e)$ . That is, in the absence of brackets, each operator combines the following CHARACTER\_CLASS with the current accumulated results. Using the same precedence level also works well in parsing (see [Annex F. Parsing Character Classes](#)).

Binding or precedence may vary by regular expression engine, so as a user it is safest to always disambiguate using brackets to be sure. In particular, precedence may put all operators on the same level, or may take union as binding more closely. For example, where A..F stand for expressions, not characters:

Expression	Precedence	Interpreted as	Interpreted as
$[AB \setminus \setminus CD \&\& EF]$	Union, intersection, and difference bind at the same level	$[[[[[AB] \setminus \setminus C]D] \&\& E]F]$	<div>clone(A).add(B)</div> <div>.remove(C).add(D)</div> <div>.retain(E).add(F)</div>
	Union binds more closely than difference or intersection	$[[[AB] \setminus \setminus [CD]] \&\& [EF]]$	<div>clone(A).add(B)</div> <div>.remove(clone(C).add(D))</div> <div>.retain(clone(E).add(F))</div>

Binding at the same level is used in this specification.

The following table shows various examples of set subtraction:

Expression	Matches
<code>[\\p{L}--[QW]]</code>	all letters but Q and W
<code>[\\p{N}--[\\p{Nd}--[0-9]]]</code>	all non-decimal numbers, plus 0-9
<code>[\\u{0}-\\u{7F}--[\\p{letter}]]</code>	all letters in the ASCII range, by subtracting non-letters
<code>[\\p{Greek}--[\\N{GREEK SMALL LETTER ALPHA}]]</code>	Greek letters except alpha
<code>[\\p{Assigned}--[\\p{Decimal Digit Number}--[a-fA-F a - f A - F ]]]</code>	all assigned characters except for hex digits (using a broad definition)
<code>[\\p{letter}~\\p{ascii}]</code>	either <i>letter</i> or <i>ascii</i> , but not both. Equivalent to <code>[\\p{letter}\\p{ascii}--[\\p{letter}&amp;&amp;\\p{ascii}]]</code>

The boolean expressions can also involve properties of strings or [Character Classes with strings](#). Thus the following matches all code points that neither have a Script value of Greek nor are in Basic\_Emoji:

```
[\\P{Script=Greek}&&\\P{Basic_Emoji}]
```

For more information, see [Annex D: Resolving Character Classes with Strings and Complement](#) and [Section 2.2.1 Character Classes with Strings](#).

## 1.4 Simple Word Boundaries

Most regular expression engines allow a test for word boundaries (such as by `"\b"` in Perl). They generally use a very simple mechanism for determining word boundaries: one example of that would be having word boundaries between any pair of characters where one is a `<word_character>` and the other is not, or at the start and end of a string. This is not adequate for Unicode regular expressions.

### RL1.4 Simple Word Boundaries

*To meet this requirement, an implementation shall extend the word boundary mechanism so that:*

1. The class of `<word_character>` includes all the Alphabetic values from the Unicode character database, from [UnicodeData.txt](#), plus the decimals (`General_Category=Decimal_Number`, or equivalently `Numeric_Type=Decimal`), and the U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER (`Join_Control=True`). See also [Annex C: Compatibility Properties](#).
2. Nonspacing marks are never divided from their base characters, and otherwise ignored in locating boundaries.

Level 2 provides more general support for word boundaries between arbitrary Unicode characters which may override this behavior.

## 1.5 Simple Loose Matches

Most regular expression engines offer caseless matching as the only loose matching. If the engine does offer this, then it needs to account for the large range of cased Unicode characters outside of ASCII.

### RL1.5 Simple Loose Matches

*To meet this requirement, if an implementation provides for case-insensitive matching, then it shall provide at least the simple, default Unicode case-insensitive matching, and specify which properties are closed and which are not.*

*To meet this requirement, if an implementation provides for case conversions, then it shall provide at least the simple, default Unicode case folding.*

In addition, because of the vagaries of natural language, there are situations where two different Unicode characters have the same uppercase or lowercase. To meet this requirement, implementations must

implement these in accordance with the Unicode Standard. For example, the Greek U+03C3 "σ" *small sigma*, U+03C2 "ς" *small final sigma*, and U+03A3 "Σ" *capital sigma* all match.

Some caseless matches may match one character against two: for example, U+00DF "ß" matches the two characters "SS". And case matching may vary by locale. However, because many implementations are not set up to handle this, at Level 1 only simple case matches are necessary. To correctly implement a caseless match, see *Chapter 3, Conformance* of [Unicode]. The data file supporting caseless matching is [CaseData].

To meet this requirement, where an implementation also offers case conversions, these must also follow *Chapter 3, Conformance* of [Unicode]. The relevant data files are [SpecialCasing] and [UData].

Matching case-insensitively is one example of matching under an equivalence relation:

A regular expression *R* matches *under an equivalence relation E* whenever for all strings *S* and *T*:

If *S* is equivalent to *T* under *E*, then *R* matches *S* if and only if *R* matches *T*.

In the Unicode Standard, the relevant equivalence relation for case-insensitivity is established according to whether two strings case fold to the same value. The case folding can either be simple (a 1:1 mapping of code points) or full (with some 1:n mappings).

- "ABC" and "Abc" are equivalent under both full and simple case folding.
- "cliff" (with the "ff" ligature) and "CLIFF" are equivalent under full case folding, but not under simple case folding.

In practice, regex APIs are not set up to match parts of characters. For this reason, full case equivalence is difficult to handle with regular expressions. For more information, see *Section 2.1, Canonical Equivalents*.

For case-insensitive matching:

1. Each string literal is matched case-insensitively. That is, it is *logically* expanded into a sequence of OR expressions, where each OR expression lists all of the characters that have a simple case-folding to the same value.
  - For example, /Dåb/ matches as if it were expanded into /(?:d|D)(?:â|Å|u{212B})(?:b|B)/. (The u{212B} is an angstrom sign, identical in appearance to Å.)
  - Back references are subject to this logical expansion, such as /(?:i)(a.c)\1/, where \1 matches what is in the first grouping.
2. **(optional)** Each character class is closed under case. That is, it is logically expanded into a set of code points, and then closed by adding all simple case equivalents of each of those code points.
  - For example, [\p{Block=Phonetic\_Extensions} [A-E]] is a character class that matches 133 code points (under Unicode 6.0). Its case-closure adds 7 more code points: a-e, P, and ð, for a total of 140 code points.

For condition #2, in both property character classes and explicit character classes, closing under simple case-insensitivity means including characters not in the set. For example:

- The case-closure of \p{Block=Phonetic\_Extensions} includes two characters not in that set, namely P and ð.
- The case-closure of [A-E] includes five characters not in that set, namely [a-e].

Conformant implementations can choose whether and how to apply condition #2: the only requirement is that they declare what they do. For example, an implementation may:

- A. uniformly apply condition #2 to all property and explicit character classes
- B. uniformly not apply condition #2 to any property or explicit character classes
- C. apply condition #2 only within the scope of a switch
- D. apply condition #2 to just specific properties and/or explicit character classes

## 1.6 Line Boundaries

Most regular expression engines also allow a test for line boundaries: end-of-line or start-of-line. This presumes that lines of text are separated by line (or paragraph) separators.

### RL1.6 Line Boundaries

*To meet this requirement, if an implementation provides for line-boundary testing, it shall recognize not only CRLF, LF, CR, but also NEL (U+0085), PARAGRAPH SEPARATOR (U+2029) and LINE SEPARATOR (U+2028).*

Formfeed (U+000C) also normally indicates an end-of-line. For more information, see Chapter 3 of [Unicode].

These characters should be uniformly handled in determining logical line numbers, start-of-line, end-of-line, and arbitrary-character implementations. Logical line number is useful for compiler error messages and the like. Regular expressions often allow for SOL and EOL patterns, which match certain boundaries. Often there is also a "non-line-separator" arbitrary character pattern that excludes line separator characters.

The behavior of these characters may also differ depending on whether one is in a "multiline" mode or not. For more information, see *Anchors and Other "Zero-Width Assertions"* in Chapter 3 of [Friedl].

A newline sequence is defined to be any of the following:

`\u{A} | \u{B} | \u{C} | \u{D} | \u{85} | \u{2028} | \u{2029} | \u{D A}`

#### 1. Logical line number

- The line number is increased by one for each occurrence of a newline sequence.
- Note that different implementations may call the first line either line zero or line one.

#### 2. Logical beginning of line (often "^")

- SOL is at the start of a file or string, and depending on matching options, also immediately following any occurrence of a newline sequence.
- There is no empty line within the sequence `\u{D A}`, that is, between the first and second character.
- Note that there may be a separate pattern for "beginning of text" for a multiline mode, one which matches only at the beginning of the first line. For example, in Perl this is `\A`.

#### 3. Logical end of line (often "\$")

- EOL at the end of a file or string, and depending on matching options, also immediately preceding a final occurrence of a newline sequence.
- There is no empty line within the sequence `\u{D A}`, that is, between the first and second character.
- SOL and EOL are not symmetric because of multiline mode: EOL can be interpreted in at least three different ways:
  - a. EOL matches at the end of the string
  - b. EOL matches before final newline
  - c. EOL matches before any newline

#### 4. Arbitrary character pattern (often ".")

- Where the 'arbitrary character pattern' matches a newline sequence, it must match all of the newline sequences, and `\u{D A}` (CRLF) *should* match as if it were a single character. (The recommendation that CRLF match as a single character is, however, not required for conformance to RL1.6.)
- Note that `^$` (an empty line pattern) should not match the empty string within the sequence `\u{D A}`, but should match the empty string within the reversed sequence `\u{A D}`.

It is strongly recommended that there be a regular expression meta-character, such as `"\R"`, for matching all line ending characters and sequences listed above (for example, in #1). This would correspond to something equivalent to the following expression. That expression is slightly complicated by the need to avoid backup.

`(?:\u{D A}|(?:!\u{D A})){\u{A}-\u{D}\u{85}\u{2028}\u{2029}}`

**Note:** For some implementations, there may be a performance impact in recognizing CRLF as a single entity, such as with an arbitrary pattern character ("."). To account for that, an implementation may also satisfy R1.6 if there is a mechanism available for converting the sequence CRLF to a single line boundary character before regex processing.

For more information on line breaking, see [UAX14].

## 1.7 Code Points

A fundamental requirement is that Unicode text be interpreted semantically by code point, not code units.

### RL1.7 Supplementary Code Points

*To meet this requirement, an implementation shall handle the full range of Unicode code points, including values from U+FFFF to U+10FFFF. In particular, where UTF-16 is used, a sequence consisting of a leading surrogate followed by a trailing surrogate shall be handled as a single code point in matching.*

UTF-16 uses pairs of 16-bit code units to express code points above  $FFFF_{16}$ , while UTF-8 uses from two to four 8-bit code units to represent code points above  $7F_{16}$ . Surrogate pairs (or their equivalents in other encoding forms) are to be handled internally as single code point values. In particular, `[\u{0}-\u{10000}]` will match all the following sequence of code units:

Code Point	UTF-8 Code Units	UTF-16 Code Units	UTF-32 Code Units
7F	7F	007F	0000007F
80	C2 80	0080	00000080
7FF	DF BF	07FF	000007FF
800	E0 A0 80	0800	00000800
FFFF	EF BF BF	FFFF	0000FFFF
10000	F0 90 80 80	D800 DC00	00010000

For backwards compatibility, some regex engines allow for switches to reset matching to be by code unit instead of code point. Such usage is discouraged. For example, in order to match 🍌 it is far better to write `\u{1F44E}` rather than `\uD83D\uDC4E` (using UTF-16) or `\xF0\x9F\x91\x8E` (using UTF-8).

**Note:** It is permissible, but not required, to match an isolated surrogate code point (such as `\u{D800}`), which may occur in Unicode 16-bit Strings. See [Unicode String](#) in the Unicode [Glossary].

## 2 Extended Unicode Support: Level 2

Level 1 support works well in many circumstances. However, it does not handle more complex languages or extensions to the Unicode Standard very well. Particularly important cases are canonical equivalence, word boundaries, extended grapheme cluster boundaries, and loose matches. (For more information about boundary conditions, see UAX #29, *Unicode Text Segmentation* [UAX29].)

Level 2 support matches much more what user expectations are for sequences of Unicode characters. It is still locale-independent and easily implementable. However, for compatibility with Level 1, it is useful to have some sort of syntax that will turn Level 2 support on and off.

The features comprising Level 2 are not in order of importance. In particular, the most useful and highest priority features in practice are:

- [RL2.3 Default Word Boundaries](#)
- [RL2.5 Name Properties](#)
- [RL2.6 Wildcards in Property Values](#)
- [RL2.7 Full Properties](#)

## 2.1 Canonical Equivalents

The equivalence relation for canonical equivalence is established by whether two strings are identical when normalized to NFD.

For most full-featured regular expression engines, it is quite difficult to match under canonical equivalence, which may involve reordering, splitting, or merging of characters. For example, all of the following sequences are canonically equivalent:

- A. o + horn + dot\_below
  - 1. U+006F ( o ) LATIN SMALL LETTER O
  - 2. U+031B ( ◌́ ) COMBINING HORN
  - 3. U+0323 ( ◌̣ ) COMBINING DOT BELOW
- B. o + dot\_below + horn
  - 1. U+006F ( o ) LATIN SMALL LETTER O
  - 2. U+0323 ( ◌̣ ) COMBINING DOT BELOW
  - 3. U+031B ( ◌́ ) COMBINING HORN
- C. o-horn + dot\_below
  - 1. U+01A1 ( ø ) LATIN SMALL LETTER O WITH HORN
  - 2. U+0323 ( ◌̣ ) COMBINING DOT BELOW
- D. o-dot\_below + horn
  - 1. U+1ECD ( ȯ ) LATIN SMALL LETTER O WITH DOT BELOW
  - 2. U+031B ( ◌́ ) COMBINING HORN
- E. o-horn-dot\_below
  - 1. U+1EE3 ( Ȱ ) LATIN SMALL LETTER O WITH HORN AND DOT BELOW

The regular expression pattern `/o\u{31B}/` matches the first two characters of A, the first and third characters of B, the first character of C, part of the first character together with the third character of D, and part of the character in E.

In practice, regex APIs are not set up to match parts of characters or handle discontinuous selections. There are many other edge cases: a combining mark may come from some part of the pattern far removed from where the base character was, or may not explicitly be in the pattern at all. It is also unclear what `./.` should match and how back references should work.

It is feasible, however, to construct patterns that will match against NFD (or NFKD) text. That can be done by:

1. Putting the text to be matched into a defined normalization form (NFD or NFKD).
2. Having the user design the regular expression pattern to match against that defined normalization form. For example, the pattern should contain no characters that would not occur in that normalization form, nor sequences that would not occur.
3. Applying the matching algorithm on a code point by code point basis, as usual.

## 2.2 Extended Grapheme Clusters and Character Classes with Strings

One or more Unicode characters may make up what the user thinks of as a character. To avoid ambiguity with the computer use of the term *character*, this is called a *grapheme cluster*. For example, "G" + *acute-accent* is a grapheme cluster: it is thought of as a single character by users, yet is actually represented by two Unicode characters. The Unicode Standard defines *extended grapheme clusters* that treat certain sequences as units, including Hangul syllables and base characters with combining marks. The precise definition is in UAX #29, *Unicode Text Segmentation* [UAX29]. However, the boundary definitions in CLDR are strongly recommended: they are more comprehensive than those defined in [UAX29] and include Indic extended grapheme clusters such as *ksha*.

### RL2.2 Extended Grapheme Clusters and Character Classes with Strings

*To meet this requirement, an implementation shall provide a mechanism for matching against an arbitrary extended grapheme cluster, Character Classes with Strings, and extended grapheme cluster boundaries.*



For example, an implementation could interpret `\x` as matching any extended grapheme cluster, while interpreting `".` as matching any single code point. It could interpret `\b{g}` as a zero-width match against any extended grapheme cluster boundary, and `\B{g}` as the complement of that.

More generally, it is useful to have zero width boundary detections for each of the different kinds of segment boundaries defined by Unicode ([UAX29] and [UAX14]). For example:

Syntax	Zero-width Match at
<code>\b{g}</code>	a Unicode extended grapheme cluster boundary
<code>\b{w}</code>	a Unicode word boundary. Note that this is different than <code>\b</code> alone, which corresponds to <code>\w</code> and <code>\W</code> . See <a href="#">Annex C: Compatibility Properties</a> .
<code>\b{l}</code>	a Unicode line break boundary
<code>\b{s}</code>	a Unicode sentence boundary

Thus `\x` is equivalent to `.+?\b{g}`; proceed the minimal number of characters (but at least one) to get to the next extended grapheme cluster boundary.

2.2.1 Character Classes with Strings

Regular expression engines should also provide some mechanism for easily matching against *Character Classes with Strings*, because they are more likely to match user expectations for many languages. One mechanism for doing that is to have explicit syntax for strings in Character Classes, as in the following addition to the syntax of Section 0.1.1 Character Classes:

Nonterminal	Production Rule	Comments & Constraints
ITEM	<code>:= '\q{ ' LITERAL* ( '   ' LITERAL* ) * ' }</code>	Adds to previous ITEM rules. Represents one or more literal strings of characters.

The `|` separator is used to make an expression more readable. Some implementations may choose to drop the `\q`, although many will choose to retain it for backwards compatibility.

Compact Notation	<code>[a-z🧐\q{ch sch BE BF BG }]</code>
Equivalent Expanded Notation	<code>[a-z🧐\q{ch}\q{sch}\q{BE }\q{BF }\q{BG }]</code>

The following table shows examples of use of the `\q` syntax:

Expression	Matches
<code>[a-z\q{x\u{323}}]</code>	The characters a-z, and the string <i>x with an under-dot</i> (used in American Indian languages)
<code>[a-z\q{aa}]</code>	The characters a-z, and the string <i>aa</i> (treated as a single character in Danish)
<code>[a-z ñ \q{ch ll rr}]</code>	Some lowercase characters in traditional Spanish
<code>[a-z \q{🧐 FR }]</code>	Characters a-z and two emoji. Note that this is equivalent to <code>[a-z 🧐\q{FR }]</code> because the first emoji is a single code point, while the second is two codepoints and thus requires the <code>\q</code> syntax. However, users of regex can not be expected to always know which sequences are single code points,

In implementing Character Classes with strings, the expression `/[a-m \q{ch|chh|rr}] β-ξ/` should behave as the alternation `/(chh | ch | rr | [a-mβ-ξ] | )/`. Note that such an alternation must have the multi-code point strings ordered as longest-first to work correctly in arbitrary regex engines, because some regex engines try the leftmost matching alternative first. Therefore it does not work to have shorter strings first. The exception is where those shorter strings are not initial substrings of longer strings.

String literals in character classes are especially useful in combination with a property of strings. String literals can be used to modify the property by removing exceptions. Such exceptions cannot be expressed

by other means. The only workaround would be to hard-code the result in an alternation, creating a large expression that loses the automatic updates of properties. For example, the following could not be expressed with alternation, except by replacing the property by hard-coded current contents (that would get out of date):

```
[p\{RGI_Emoji}--[a-z🤪\q{ch|sch|BE|BF|BG }]]
```

If the implementation supports empty alternations, such as `(ab[[ac-m]])`, then it can also handle empty strings: `[q{ab}[ac-m]\q{ }]`.

Of course, such alternations can be optimized internally for speed and/or memory, such as `(ab[[ac-m]]) → ((ab?)[[c-m]])`.

Like properties of strings, complemented Character Classes with strings need to be handled specially: see [Annex D: Resolving Character Classes with Strings and Complement](#).

## 2.3 Default Word Boundaries

### RL2.3 Default Word Boundaries

*To meet this requirement, an implementation shall provide a mechanism for matching Unicode default word boundaries.*

The simple Level 1 support using simple `<word_character>` classes is only a very rough approximation of user word boundaries. A much better method takes into account more context than just a single pair of letters. A general algorithm can take care of character and word boundaries for most of the world's languages. For more information, see UAX #29, *Unicode Text Segmentation* [UAX29].

**Note:** Word boundaries and "soft" line-break boundaries (where one could break in line wrapping) are not generally the same; line breaking has a much more complex set of requirements to meet the typographic requirements of different languages. See UAX #14, *Line Breaking Properties* [UAX14] for more information. However, soft line breaks are not generally relevant to general regular expression engines.

A fine-grained approach to languages such as Chinese or Thai—languages that do not use spaces—requires information that is beyond the bounds of what a Level 2 algorithm can provide.

## 2.4 Default Case Conversion

### RL2.4 Default Case Conversion

*To meet this requirement, if an implementation provides for case conversions, then it shall provide at least the full, default Unicode case folding.*

Previous versions of RL2.4 included full default Unicode case-insensitive matching. For most full-featured regular expression engines, it is quite difficult to match under code point equivalences that are not 1:1. For more discussion of this, see 1.5 [Simple Loose Matches](#) and 2.1 [Canonical Equivalents](#). Thus that part of RL2.4 has been retracted.

Instead, it is recommended that implementations provide for full, default Unicode case conversion, allowing users to provide both patterns and target text that has been fully case folded. That allows for matches such as between U+00DF "ß" and the two characters "SS". Some implementations may choose to have a mixed solution, where they do full case matching on literals such as "Strauß", but simple case folding on character classes such as `[ß]`.

To correctly implement case conversions, see [\[Case\]](#). For ease of implementation, a complete case folding file is supplied at [\[CaseData\]](#). Full case mappings use the data files [\[SpecialCasing\]](#) and [\[UData\]](#).

## 2.5 Name Properties

### RL2.5 Name Properties

*To meet this requirement, an implementation shall support individually named characters.*

When using names in regular expressions, the data is supplied in both the **Name (na)** and **Name\_Alias** properties in the UCD, as described in UAX #44, *Unicode Character Database* [UAX44], or computed as in the case of CJK Ideographs or Hangul Syllables. Name matching rules follow [Matching Rules](#) from [UAX44#UAX44-LM2].

The following provides examples of usage:

Syntax	Set	Note
<code>\p{name=ZERO WIDTH NO-BREAK SPACE}</code>	<code>[u{FEFF}]</code>	using the Name property
<code>\p{name=zerowidthno breakspace}</code>	<code>[u{FEFF}]</code>	using the Name property, and <a href="#">Matching Rules</a> [UAX44]
<code>\p{name=BYTE ORDER MARK}</code>	<code>[u{FEFF}]</code>	using the Name_Alias property
<code>\p{name=BOM}</code>	<code>[u{FEFF}]</code>	using the Name_Alias property (a second value)
<code>\p{name=HANGUL SYLLABLE GAG}</code>	<code>[u{AC01}]</code>	with a computed name
<code>\p{name=BEL}</code>	<code>[u{7}]</code>	the control character
<code>\p{name=BELL}</code>	<code>[u{1F514}]</code>	the graphic symbol 🛎

Certain code points are not assigned names or name aliases in the standard. With the exception of "reserved", these should be given names based on [Code Point Label Tags](#) table in [UAX44], as shown in the following examples:

Syntax	Set	Note
<code>\p{name=private-use-E000}</code>	<code>[u{E000}]</code>	
<code>\p{name=surrogate-D800}</code>	<code>[u{D800}]</code>	would only apply to isolated surrogate code points
<code>\p{name=noncharacter-FDD0}</code>	<code>[u{FDD0}]</code>	
<code>\p{name=control-0007}</code>	<code>[u{7}]</code>	

Characters with the <reserved> tag in the [Code Point Label Tags](#) table of [UAX44] are *excluded*: the syntax `\p{reserved-058F}` would mean that the code point U+058F is unassigned. While this code point was unassigned in Unicode 6.0, it *is* assigned in Unicode 6.1 and thus no longer "reserved".

Implementers may add aliases beyond those recognized in the UCD. They must be aware that such additional aliases may cause problems if they collide with future character names or aliases. For example, implementations that used the name "BELL" for U+0007 broke when the new character U+1F514 ( 🛎 ) BELL was introduced.

Previous versions of this specification recommended supporting ISO control names from the Unicode 1.0 name field. These names are now covered by the name aliases (see [NameAliases.txt](#)). In four cases, the name field included both the ISO control name as well as an abbreviation in parentheses.

U+000A LINE FEED (LF)  
 U+000C FORM FEED (FF)  
 U+000D CARRIAGE RETURN (CR)  
 U+0085 NEXT LINE (NEL)

These abbreviations were intended as alternate aliases, not as part of the name, but the documentation did not make this sufficiently clear. As a result, some implementations supported the entire field as a name. Those implementations might benefit from continuing to support them for compatibility. Beyond that, their use is not recommended.

The `\p{name=...}` syntax can be used meaningfully with wildcards (see [Section 2.6 Wildcards in Property Values](#)). For example, in Unicode 6.1, `\p{name=/ALIEN/}` would include a set of two characters:

- U+1F47D ( 🛶 ) EXTRATERRESTRIAL ALIEN,
- U+1F47E ( 🛷 ) ALIEN MONSTER

The namespace for the `\p{name=...}` syntax is the [Unicode namespace for character names \[UAX34-D3\]](#).

### 2.5.1 Individually Named Characters

The following provides syntax for specifying a code point by supplying the precise name. This syntax specifies a single code point, which can thus be used wherever `\u{...}` can be used. Note that `\N` and `\p{name}` may be extended to match *sequences* if `NamedSequences.txt` is supported as in Section 2.7 [Full Properties](#).

Nonterminal	Production Rule	Comments & Constraints
LITERAL	<code>:= '\N{' ID_CHAR+ '}'</code>	Adds to previous LITERAL rules. <i>Constraint:</i> ID_CHAR+ = valid Unicode name or alias

The `\N` syntax is related to the syntax `\p{name=...}`, but there are important distinctions:

1. `\N` matches a single character, while `\p` matches a set of characters (when using wildcards).
2. The `\p{name=<character_name>}` may silently fail, if no character exists with that name. The `\N` syntax should instead cause a syntax error for an undefined name.

The namespace for the `\N{name=...}` syntax is the [Unicode namespace for character names \[UAX34-D3\]](#). Name matching rules follow [Matching Rules](#) from [\[UAX44#UAX44-LM2\]](#).

The following table gives examples of the `\N` syntax:

Expression	Equivalent to
<code>\N{WHITE SMILING FACE}</code>	<code>\u{263A}</code>
<code>\N{whitesmilingface}</code>	
<code>\N{GREEK SMALL LETTER ALPHA}</code>	<code>\u{3B1}</code>
<code>\N{FORM FEED}</code>	<code>\u{C}</code>
<code>\N{SHAVIAN LETTER PEEP}</code>	<code>\u{10450}</code>
<code>[\N{GREEK SMALL LETTER ALPHA}-\N{GREEK SMALL LETTER BETA}]</code>	<code>[\u{3B1}-\u{3B2}]</code>

## 2.6 Wildcards in Property Values

### RL2.6 Wildcards in Property Values

*To meet this requirement, an implementation shall support wildcards in Unicode property values.*

Instead of a single property value, this feature allows the use of a regular expression to pick out a set of characters (or strings) based on whether the property values match the regular expression. The regular expression must support at least wildcards; other regular expressions features are recommended but optional.

Nonterminal	Production Rule	Comments & Constraints
PROP_VALUE	<code>:= '/' &lt;regex expression&gt; '/'</code>	<code>\p{PROP_NAME=&lt;regex expression&gt;}</code> is set of all characters (or strings) whose property value matches the regular expression. See below for examples.
	<code>:= '@' PROP_NAME '@'</code>	<code>\p{PROP_NAME1=@PROP_NAME2@}</code> is set of all characters (or strings) whose property value for PROP_NAME1 is identical to the property value for PROP_NAME2. See below for examples.

#### Notes:

- Where regular expressions are used in matching, the case, spaces, hyphen, and underbar are significant; it is presumed that users will make use of regular-expression features to ignore these if desired.

- In this syntax, the syntax characters are doubled at the start and end to avoid colliding with actual property values. For example, this prevents problems with properties with string values. In the unusual case that a desired property value happens to start and end with, say, @, the expression can use quoted characters such as `\u{40}`
- As usual, the syntax in this document is illustrative: characters other than `'` and `@` can be chosen if these are not appropriate for the environment used by the regular expression engine.

The `@...@` syntax is used to compare property values, and is primarily intended for string properties. It allows for expressions such as `[.^toNFKC_Casefold=@toNFKC@:]`, which expresses the set of all and only those code points **CP** such that **toNFKC\_Casefold(CP) = toNFKC(CP)**. The value *identity* can be used in this context. For example, `\p{toLowercase#@identity@}` expresses the set of all characters that are changed by the toLowercase mapping.

The following table shows examples of the use of wildcards.

Expression	Matched Set <span style="background-color: yellow;">in Unicode 5.0*</span>
Characters whose NFD form contains a "b" (U+0062) in the value:	
<code>\p{toNfd=/b/}</code>	U+0062 ( <b>b</b> ) LATIN SMALL LETTER B U+1E03 ( <b>ḃ</b> ) LATIN SMALL LETTER B WITH DOT ABOVE U+1E05 ( <b>ḅ</b> ) LATIN SMALL LETTER B WITH DOT BELOW U+1E07 ( <b>ḇ</b> ) LATIN SMALL LETTER B WITH LINE BELOW
Characters with names containing "SMILING FACE" or "GRINNING FACE":	
<code>\p{name=/(SMILING GRINNING) FACE/}</code>	U+263A ( 😊 ) WHITE SMILING FACE U+263B ( 🙄 ) BLACK SMILING FACE U+1F601 ( 😄 ) GRINNING FACE WITH SMILING EYES U+1F603 ( 😆 ) SMILING FACE WITH OPEN MOUTH U+1F604 ( 😊 ) SMILING FACE WITH OPEN MOUTH AND SMILING EYES U+1F605 ( 😄 ) SMILING FACE WITH OPEN MOUTH AND COLD SWEAT U+1F606 ( 😏 ) SMILING FACE WITH OPEN MOUTH AND TIGHTLY-CLOSED EYES U+1F607 ( 😇 ) SMILING FACE WITH HALO U+1F608 ( 😈 ) SMILING FACE WITH HORNS U+1F60A ( 😊 ) SMILING FACE WITH SMILING EYES U+1F60D ( 😊 ) SMILING FACE WITH HEART-SHAPED EYES U+1F60E ( 😎 ) SMILING FACE WITH SUNGLASSES U+1F642 ( 😊 ) SLIGHTLY SMILING FACE U+1F929 ( 😄 ) GRINNING FACE WITH STAR EYES U+1F92A ( 😄 ) GRINNING FACE WITH ONE LARGE AND ONE SMALL EYE U+1F92D ( 😊 ) SMILING FACE WITH SMILING EYES AND HAND COVERING MOUTH U+1F970 ( 😊 ) SMILING FACE WITH SMILING EYES AND THREE HEARTS
Characters with names containing "VARIATION" or "VARIANT":	
<code>\p{name=/VARIA(TION NT)/}</code>	U+180B ( ) MONGOLIAN FREE VARIATION SELECTOR ONE ... U+180D ( ) MONGOLIAN FREE VARIATION SELECTOR THREE U+299C ( ∟ ) RIGHT ANGLE VARIANT WITH SQUARE U+303E ( ☰ ) IDEOGRAPHIC VARIATION INDICATOR U+FE00 ( ) VARIATION SELECTOR-1 ... U+FE0F ( ) VARIATION SELECTOR-16 U+121AE ( 𐀞 ) CUNEIFORM SIGN KU4 VARIANT FORM U+12425 ( 𐎥 ) CUNEIFORM NUMERIC SIGN THREE SHAR2

	VARIANT FORM U+1242F (𐎢) CUNEIFORM NUMERIC SIGN THREE SHARU VARIANT FORM U+12437 (𐎣) CUNEIFORM NUMERIC SIGN THREE BURU VARIANT FORM U+1243A (𐎤) CUNEIFORM NUMERIC SIGN THREE VARIANT FORM ESH16 ... U+12449 (𐎥) CUNEIFORM NUMERIC SIGN NINE VARIANT FORM ILIMMU A U+12453 (𐎦) CUNEIFORM NUMERIC SIGN FOUR BAN2 VARIANT FORM U+12455 (𐎧) CUNEIFORM NUMERIC SIGN FIVE BAN2 VARIANT FORM U+1245D (𐎩) CUNEIFORM NUMERIC SIGN ONE THIRD VARIANT FORM A U+1245E (𐎪) CUNEIFORM NUMERIC SIGN TWO THIRDS VARIANT FORM A U+E0100 ( ) VARIATION SELECTOR-17 ... U+E01EF ( ) VARIATION SELECTOR-256
Characters in the Letterlike symbol block with different toLowercase values:	
\p{Block=Letterlike Symbols} --\p{toLowercase=@identity@}	U+2126 ( Ω ) OHM SIGN U+212A ( K ) KELVIN SIGN U+212B ( Å ) ANGSTROM SIGN U+2132 ( Ⓐ ) TURNED CAPITAL F
Greek characters whose toLowercase and toUppercase values are different, excluding decomposable characters	
\p{script=Grek} --\p{toLowercase=@toUppercase@} --\p{nfdqcn}	[αΑ βΒ γΓ δΔ εΕ ϜϞ Ϡϡ ζΖ ηΗ θΘ ιΙ jJ κΚ χΧ λΛ μΜ νΝ ξΞ οΟ πΠ Ϙϙ Ϛϛ ϟϠ ϡϢ τΤ υΥ φΦ χΧ ψΨ ωΩ ϣϛ ϭϮ ϯϰ]

⚠ The lists in the examples above were extracted on the basis of Unicode 5.0; different Unicode versions may produce different results.

See [Section 0.1.2 Property Examples](#) for information about updates to the contents of a literal set across versions.

The following table some additional samples, illustrating various sets. A click on the link will use the online Unicode utilities on the Unicode website to show the contents of the sets. Note that these online utilities currently use single-letter operations.

Expression	Description
<code>[[:name=/CJK/:]-[:ideographic:]]</code>	The set of all characters with names that contain CJK that are not Ideographic
<code>[[:name=/\bDOT\$/:]]</code>	The set of all characters with names that end with the word DOT
<code>[[:block=/(?i)arab/:]]</code>	The set of all characters in blocks that contain the sequence of letters "arab" (case-insensitive)
<code>[[:toNFKC=/\./:]]</code>	the set of all characters with toNFKC values that contain a literal period

## 2.7 Full Properties

## RL2.7 Full Properties

*To meet this requirement, an implementation shall support all of the properties listed below that are in the supported version of the Unicode Standard (or Unicode Technical Standard, respectively), with values that match the Unicode definitions for that version.*



To meet requirement RL2.7, the implementation must satisfy the Unicode definition of the properties for the supported version of Unicode (*or Unicode Technical Standard, respectively*), rather than other possible definitions. However, the names used by the implementation for these properties may differ from the formal Unicode names for the properties. For example, if a regex engine already has a property called "Alphabetic", for backwards compatibility it may need to use a distinct name, such as "Unicode\_Alphabetic", for the corresponding property listed in [RL1.2](#).

The list excludes provisional, contributory, obsolete, and deprecated properties. It also excludes specific properties: Unicode\_1\_Name, Unicode\_Radical\_Stroke, and the Unihan properties. The properties shown in the table with a gray background are covered by [RL1.2](#) Properties. For more information on properties, see UAX #44, *Unicode Character Database* [[UAX44](#)].

Property Domains: All listed properties marked with \* are properties of strings. All other listed properties are properties of code points. The domain of these properties (strings vs code points) will not change in subsequent versions.

General	Case	Shaping and Rendering
Name (Name_Alias)	Uppercase	Join_Control
Block	Lowercase	Joining_Group
Age	Simple_Lowercase_Mapping	Joining_Type
General_Category	Simple_Titlecase_Mapping	Vertical_Orientation
Script (Script_Extensions)	Simple_Uppercase_Mapping	Line_Break
White_Space	Simple_Case_Folding	Grapheme_Cluster_Break
Alphabetic	Soft_Dotted	Sentence_Break
Hangul_Syllable_Type	Cased	Word_Break
Noncharacter_Code_Point	Case_Ignorable	East_Asian_Width
Default_Ignorable_Code_Point	Changes_When_Lowercased	Prepended_Concatenation_Mark
Deprecated	Changes_When_Uppercased	
Logical_Order_Exception	Changes_When_Titlecased	<b>Bidirectional</b>
Variation_Selector	Changes_When_Casefolded	Bidi_Class
	Changes_When_Casemapped	Bidi_Control
<b>Numeric</b>		Bidi_Mirrored
Numeric_Value	<b>Normalization</b>	Bidi_Mirroring_Glyph
Numeric_Type	Canonical_Combining_Class	Bidi_Paired_Bracket
Hex_Digit	Decomposition_Type	Bidi_Paired_Bracket_Type
ASCII_Hex_Digit	NFC_Quick_Check	
	NFKC_Quick_Check	<b>Miscellaneous</b>
<b>Identifiers</b>	NFD_Quick_Check	Math
ID_Continue	NFKD_Quick_Check	Quotation_Mark
ID_Start	NFKC_Casefold	Dash
XID_Continue	Changes_When_NFKC_Casefolded	Sentence_Terminal
XID_Start	NFKC_Simple_Casefold	Terminal_Punctuation
Pattern_Syntax		Diacritic
Pattern_White_Space	<b>Emoji</b>	Extender
Identifier_Status	Emoji	Grapheme_Base

Identifier_Type	Emoji_Presentation	Grapheme_Extend
ID_Compat_Math_Start	Emoji_Modifier	Regional_Indicator
ID_Compat_Math_Continue	Emoji_Modifier_Base	
	Emoji_Component	
CJK	Extended_Pictographic	
Ideographic	Basic_Emoji*	
Unified_Ideograph	Emoji_Keycap_Sequence*	
Radical	RGI_Emoji_Modifier_Sequence*	
IDS_Binary_Operator	RGI_Emoji_Flag_Sequence*	
IDS_Tertiary_Operator	RGI_Emoji_Tag_Sequence*	
Equivalent_Unified_Ideograph	RGI_Emoji_ZWJ_Sequence*	
IDS_Unary_Operator	RGI_Emoji*	

The properties that are not in the UCD provide property metadata in their data file headers that can be used to support property syntax. That information is used to match and validate properties and property values for syntax such as `\p{pname=pvalue}`, so that they can be used in the same way as UCD properties. These include the [Identifier Status](#) and [Identifier Type](#), and the Emoji sequence properties.

The `Name` and `Name_Alias` properties are used in `\p{name=...}` and `\N{...}`. The data in `NamedSequences.txt` is also used in `\N{...}`. For more information see [Section 2.5, \*Name Properties\*](#). The `Script` and `Script_Extensions` properties are used in `\p{scx=...}`. For more information, see [Section 1.2.6, \*Script and Script Extensions Properties\*](#).

To test whether a *string* is in a normalization format such as NFC requires special code. However, there are "quick-check" properties that can detect whether characters are allowed in a normalization format at all. Those can be used for cases like the following, which removes characters that cannot occur in NFC:

```
\p{Alphabetic}--\p{NFC Quick Check=No}
```

The Emoji properties can be used to precisely parse text for valid emoji of different kinds, while the [Equivalent Unified Ideograph](#) can be used to find radicals for unified ideographs (or vice versa):

$\backslash p\{\text{Equivalent Unified Ideograph}=\sqrt{\quad}\}$  matches  $[\sqrt{\quad}\sqrt{\quad}\sqrt{\quad}]$ .

See also [2.5 Name Properties](#) and [2.6 Wildcards in Property Values](#).

## 2.8 Optional Properties

Implementations may also add other regular expression properties based on Unicode data that are not listed above. Some possible candidates include the following. These are optional, and are not required by any conformance clauses in this document, nor is the example syntax required.

[illegible]

		to the Name property, NamedAliases.txt, and NamedSequences.txt, so that <code>\p{Named_Sequence=X}</code> is a drop-in for <code>\p{Name=X}</code> .
Standardized Variants.txt	<code>\p{Standardized_Variant}</code>	The set of all standardized variant sequences.
UCD	<code>\p{Indic_Positional_Category=Left_And_Right}</code>	See UCD description
UCD	<code>\p{Indic_Syllabic_Category=Avagraha}</code>	See UCD description
	<code>\p{identity=a}</code>	The identity property maps each code point to itself. For example, this expression is a character class containing the one character 'a'. It is primarily useful in wildcard property values.

See [Section 0.1.2 Property Examples](#) for information about updates to the contents of a literal set across versions.

### 3 Tailored Support: Level 3

This section has been retracted. It last appeared in [version 19](#).

## Annex A: Character Blocks

The Block property from the Unicode Character Database can be a useful property for quickly describing a set of Unicode characters. It assigns a name to segments of the Unicode codepoint space; for example, `[\u{370}-\u{3FF}]` is the Greek block.

However, block names need to be used with discretion; they are very easy to misuse because they only supply a very coarse view of the Unicode character allocation. For example:

- **Blocks are not at all exclusive.** There are many mathematical operators that are not in the Mathematical Operators block; there are many currency symbols not in Currency Symbols, and so on.
- **Blocks may include characters not assigned in the current version of Unicode.** This can be both an advantage and disadvantage. Like the General Property, this allows an implementation to handle characters correctly that are not defined at the time the implementation is released. However, it also means that depending on the current properties of assigned characters in a block may fail. For example, all characters in a block may currently be letters, but this may not be true in the future.
- **Writing systems may use characters from multiple blocks:** English uses characters from Basic Latin and General Punctuation, Syriac uses characters from both the Syriac and Arabic blocks, various languages use Cyrillic plus a few letters from Latin, and so on.
- **Characters from a single writing system may be split across multiple blocks.** See the following table on Writing Systems versus Blocks. Moreover, presentation forms for a number of different scripts may be collected in blocks like Alphabetic Presentation Forms or Halfwidth and Fullwidth Forms.

The following table illustrates the mismatch between writing systems and blocks. These are only examples; this table is not a complete analysis. It also does not include common punctuation used with all of these writing systems.

### Writing Systems Versus Blocks

Writing System	Associated Blocks
Latin	Basic Latin, Latin-1 Supplement, Latin Extended-A, Latin Extended-B, Latin Extended-C, Latin Extended-D, Latin Extended-E, Latin Extended Additional, Combining Diacritical Marks
Greek	Greek, Greek Extended, Combining Diacritical Marks

Arabic	Arabic, Arabic Supplement, Arabic Extended-A, Arabic Presentation Forms-A, Arabic Presentation Forms-B
Korean	Hangul Jamo, Hangul Jamo Extended-A, Hangul Jamo Extended-B, Hangul Compatibility Jamo, Hangul Syllables, CJK Unified Ideographs, CJK Unified Ideographs Extension A, CJK Compatibility Ideographs, CJK Compatibility Forms, Enclosed CJK Letters and Months, Small Form Variants
Yi	Yi Syllables, Yi Radicals
Chinese	CJK Unified Ideographs, CJK Unified Ideographs Extension A, CJK Unified Ideographs Extension B, CJK Unified Ideographs Extension C, CJK Unified Ideographs Extension D, CJK Unified Ideographs Extension E, CJK Compatibility Ideographs, CJK Compatibility Ideographs Supplement, CJK Compatibility Forms, Kangxi Radicals, CJK Radicals Supplement, Enclosed CJK Letters and Months, Small Form Variants, Bopomofo, Bopomofo Extended, CJK Unified Ideographs Extension F, CJK Unified Ideographs Extension G, ...

For the above reasons, Script values are generally preferred to Block values. Even there, they should be used in accordance with the guidelines in UAX #24, *Unicode Script Property* [UAX24].

## Annex B: Sample Collation Grapheme Cluster Code

*This annex was retracted at the same time that Level 3 was retracted.*

## Annex C: Compatibility Properties

The following table shows recommended assignments for compatibility property names, for use in Regular Expressions. The standard recommendation is shown in the column labeled "Standard"; applications should use this definition wherever possible. If populated with a different value, the column labeled "POSIX Compatible" shows modifications to the standard recommendation required to meet the formal requirements of [POSIX], and also to maintain (as much as possible) compatibility with the POSIX usage in practice. That modification involves some compromises, because POSIX does not have as fine-grained a set of character properties as in the Unicode Standard, and also has some additional constraints. So, for example, POSIX does not allow more than 20 characters to be categorized as digits, whereas there are many more than 20 digit characters in Unicode.

### Compatibility Property Names

Property	Standard	POSIX Compatible	Comments
<b>alpha</b>	<code>\p{Alphabetic}</code>		Alphabetic includes more than <code>gc = Letter</code> . Note that combining marks (Me, Mn, Mc) are required for words of many languages. While they could be applied to non-alphabetics, their principal use is on alphabetics. See <a href="#">DerivedCoreProperties</a> for Alphabetic. See also <a href="#">DerivedGeneralCategory</a> . Alphabetic should <i>not</i> be used as an approximation for word boundaries: see <a href="#">word</a> below.
<b>lower</b>	<code>\p{Lowercase}</code>		Lowercase includes more than <code>gc = Lowercase_Letter (Li)</code> . See <a href="#">DerivedCoreProperties</a> .
<b>upper</b>	<code>\p{Uppercase}</code>		Uppercase includes more than <code>gc = Uppercase_Letter (Lu)</code> .
<b>punct</b>	<code>\p{gc=Punctuation}</code>	<code>\p{gc=Punctuation}</code> <code>\p{gc=Symbol}</code> <code>-- \p{alpha}</code>	POSIX adds symbols. Not recommended generally, due to the confusion of having <i>punct</i> include non-punctuation marks.

<b>digit (\d)</b>	<code>\p{gc=Decimal_Number}</code>	<code>[0..9]</code>	Non-decimal numbers (like Roman numerals) are normally excluded. In U4.0+, the recommended column is the same as <code>gc = Decimal_Number (Nd)</code> . See <a href="#">DerivedNumericType</a> .
<b>xdigit</b>	<code>\p{gc=Decimal_Number}</code> <code>\p{Hex_Digit}</code>	<code>[0-9 A-F a-f]</code>	Hex_Digit contains 0-9 A-F, fullwidth and halfwidth, upper and lowercase.
<b>alnum</b>	<code>\p{alpha}</code> <code>\p{digit}</code>		Simple combination of other properties
<b>space (\s)</b>	<code>\p{Whitespace}</code>		See <a href="#">PropList</a> for the definition of Whitespace.
<b>blank</b>	<code>\p{gc=Space_Separator}</code> <code>\N{CHARACTER TABULATION}</code>		"horizontal" whitespace: space separators plus U+0009 <i>tab</i> . Engines implementing older versions of the Unicode Standard may need to use the longer formulation: <code>\p{Whitespace} --</code> <code>[\N{LF} \N{VT} \N{FF} \N{CR} \N{NEL}</code> <code>\p{gc=Line_Separator}</code> <code>\p{gc=Paragraph_Separator}]</code>
<b>cntrl</b>	<code>\p{gc=Control}</code>		The characters in <code>\p{gc=Format}</code> share some, but not all aspects of control characters. Many format characters are required in the representation of plain text.
<b>graph</b>	<code>[^</code> <code>\p{space}</code> <code>\p{gc=Control}</code> <code>\p{gc=Surrogate}</code> <code>\p{gc=Unassigned}]</code>		<i>Warning:</i> the set shown here is defined by <i>excluding</i> space, controls, and so on with <code>^</code> .
<b>print</b>	<code>\p{graph}</code> <code>\p{blank}</code> <code>-- \p{cntrl}</code>		Includes graph and space-like characters.
<b>word (\w)</b>	<code>\p{alpha}</code> <code>\p{gc=Mark}</code> <code>\p{digit}</code> <code>\p{gc=Connector_Punctuation}</code> <code>\p{Join_Control}</code>	n/a	This is only an approximation to Word Boundaries (see <a href="#">b</a> below). The Connector Punctuation is added in for programming language identifiers, thus adding "_" and similar characters.
<b>\X</b>	Extended Grapheme Clusters	n/a	See <a href="#">[UAX29]</a> . Other functions are used for programming language identifier boundaries.
<b>\b</b>	Default Word Boundaries	n/a	If there is a requirement that <code>\b</code> align with <code>\w</code> , then it would use the approximation above instead. See <a href="#">[UAX29]</a> . Note that different functions are used for programming language identifier boundaries. See also <a href="#">[UAX31]</a> .

## Annex D: Resolving Character Classes with Strings and Complement

The operators and contents of a character class correspond to a set of strings. With full complement, the normal set-theoretic equivalences are maintained:

- $A \cup B = B \cup A$
- $A \cap B = B \cap A$
- $A \cup (B \cap C) = (A \cup B) \cap C$
- $A \cap (B \cup C) = (A \cap B) \cup C$
- $C(C(A)) = A$

- $A \setminus B = A \cap \complement B$
- $A \setminus (B \setminus C) = (A \setminus B) \cup (A \cap C)$
- ...

See [https://en.wikipedia.org/wiki/Set\\_\(mathematics\)#Basic\\_operations](https://en.wikipedia.org/wiki/Set_(mathematics)#Basic_operations) for more examples. (Note that that page uses one of the alternate notations for complement:  $A'$ .)

However, the full complement turns a finite set into an infinite set. This is a problem for regular expressions. If  $[\^a]$  were defined to be the full complement of  $[a]$ , then it would include every string except for 'a'. Matching a finite set of strings can be represented in regular expression implementations using alternation, in a straightforward way. Matching an infinite set of strings fails badly:  $[\^a]$  would match "ab", since the string "ab" is not in  $[a]$ . So  $[\^a]$  cannot be interpreted as full complement, since that would break well-established behavior.

This is not a problem for the other set operations:  $A \cup B$ ,  $A \cap B$ ,  $A \setminus B$ ,  $A \ominus B$ . None of them can produce an infinite set from finite sets. Moreover, the operator for full complement of strings is not necessary for regular expressions: that is, with the operations  $A \cup B$ ,  $A \cap B$ ,  $A \setminus B$ ,  $A \ominus B$ , all combinations of character classes resulting in a finite set of strings can be formed.

For this reason,  $[\^...]$  remains as code point complement even when other regular expression syntax is extended to allow for strings. The normal set-theoretic equivalences still hold for all operations, except that those involving code point complement are qualified, so:

- $C_P(C_P(A)) = A$ , if  $P \supseteq A$
- $A \setminus B = A \cap C_P B$ , if  $P \supseteq A$
- ...

These can be derived by converting  $C_P A$  to the equivalent  $(P \setminus A)$ . For example,  $C_P(C_P(A)) = P \setminus (P \setminus A) = P \cap A$ .

**Note:** Some implementations may choose to throw exceptions when complement is applied to an expression that contains (or could contain) strings. For those implementations,  $[\^A]$  would not always be equivalent to  $[\p{any}--[A]]$ , since the former could throw an exception, while the latter would always resolve to the code point complement.

However, the full complement of a Character Class with strings or of a property of strings could be allowed **internal** to a character class expression as long as the fully resolved version of the outermost expression does not contain an infinite number of strings. If an implementation is to support Full Complement, then the following section describes how this can be done. First is to provide an additional operator for Full Complement:

Nonterminal	Production Rule	Comments & Constraints
ITEM	<code>:= '[' FULL_COMPLEMENT CHARACTER_CLASS ']'</code>	<i>Adds to ITEM definition above. Forms the <b>full complement</b>: <math>S \setminus \text{CHARACTER\_CLASS}</math></i>
FULL_COMPLEMENT	<code>:= '!'</code>	

For example, suppose that  $C$  is a Character Class without strings or property of characters, and  $S$  is a Character Class with strings or property of strings.

- $[[[![[S]]]]$  is allowable
- $[C--[S]]$  is allowable
- $[C\&\&[[S]]]$  resolves to  $[C--[S]]$  and is thus allowable — it does not contain any strings.
- $[[[C--[S]]]$  is allowable
- $[[[S--[C]]]$  is not allowable (on the top level)

A narrowed set of single characters can always be represented by intersecting with the set of single characters, such as  $[\p{Basic_Emoji}\&\&\p{any}]$ .



The following describes how a boolean expression can be resolved to a Character Class with *only* characters, a Character Class with strings, or a full-complemented Character Class with *only* characters. As usual, this is a logical expression of the process; implementations can optimize as long as they get the same results.

When incrementally parsing and building a resolved boolean expression, the process can be analyzed in terms of a series of core operations. In parsing Character Classes, the intermediate objects are logically *enhanced sets* of strings, such as A and B. The enhancement is the addition of a flag to indicate whether the internal set is *full-complemented* or not. The symbol  $\oplus$  stands for the flag value = *normal*. The symbol  $\ominus$  stands for the flag value = *full-complemented*. Thus:

$\oplus$  means that the internal set is treated normally; the enhanced set is the same as the internal set.

$\ominus$  means that the internal set is full-complemented; the logical contents of the enhanced set are every possible string *except those in the internal set*. Where  $\mathcal{S}$  stands for the set of all strings, and  $\{\alpha, \beta\}$  is the internal set, then the semantics is:  $(\mathcal{S} \setminus \{\alpha, \beta\})$ , that is, the set of all strings *except for*  $\{\alpha, \beta\}$ .

When the flag is full-complemented, adding or removing from the enhanced set has the reverse effect on the internal set.

- *adding*  $\beta$  to  $(\mathcal{S} \setminus \{\alpha, \beta\})$  is the same as *removing* from the internal set:  $\Rightarrow (\mathcal{S} \setminus \{\alpha\})$
- *removing*  $\gamma$  from  $(\mathcal{S} \setminus \{\alpha, \beta\})$  is the same as *adding* to the internal set:  $\Rightarrow (\mathcal{S} \setminus \{\alpha, \beta, \gamma\})$

For brevity in the table below,  $C_{\mathcal{S}}\{\alpha, \beta\}$  is used to express  $(\mathcal{S} \setminus \{\alpha, \beta\})$ .

While logically the enhanced set can contain an infinite set of strings, internally there is only ever a finite set.

### Creation and Unary Operations

- [expression] and  $\backslash p\{\text{expression}\}$  (without full-complementing) create enhanced sets with the internal sets corresponding to the expression, and the flags set to  $\oplus$ .
- [!expression] and  $\backslash P\{\text{expression}\}$  (with full-complementing) create enhanced sets with the internal sets corresponding to the expression, and the flags set to  $\ominus$ .

[!A] where A is an enhanced set with (set, flag) results in the flag being toggled:  $\oplus \Leftrightarrow \ominus$

### Binary Operations

The table shows how to process binary operations on enhanced sets, with each result being the internal set plus flag. Examples are provided with two overlapping sets:  $A = \{\alpha, \beta\}$  and  $B = \{\beta, \gamma\}$ .

Syntax	Flag of A	Flag of B	Result Set	Flag of Result	Example Input	Example Result
A    B (union)	$\oplus$	$\oplus$	$\text{setA} \cup \text{setB}$	$\oplus$	$\{\alpha, \beta\} \cup \{\beta, \gamma\}$	$\{\alpha, \beta, \gamma\}$
	$\oplus$	$\ominus$	$\text{setB} \setminus \text{setA}$	$\ominus$	$\{\alpha, \beta\} \cup C\{\beta, \gamma\}$	$C\{\gamma\}$
	$\ominus$	$\oplus$	$\text{setA} \setminus \text{setB}$	$\ominus$	$C\{\alpha, \beta\} \cup \{\beta, \gamma\}$	$C\{\alpha\}$
	$\ominus$	$\ominus$	$\text{setA} \cap \text{setB}$	$\ominus$	$C\{\alpha, \beta\} \cup C\{\beta, \gamma\}$	$C\{\beta\}$
A && B (intersection)	$\oplus$	$\oplus$	$\text{setA} \cap \text{setB}$	$\oplus$	$\{\alpha, \beta\} \cap \{\beta, \gamma\}$	$\{\beta\}$

	$\oplus$	$\ominus$	$\text{setA} \setminus \text{setB}$	$\oplus$	$\{\alpha, \beta\} \cap \mathcal{C}\{\beta, \gamma\}$	$\{\alpha\}$
	$\ominus$	$\oplus$	$\text{setB} \setminus \text{setA}$	$\oplus$	$\mathcal{C}\{\alpha, \beta\} \cap \{\beta, \gamma\}$	$\{\gamma\}$
	$\ominus$	$\ominus$	$\text{setA} \cup \text{setB}$	$\ominus$	$\mathcal{C}\{\alpha, \beta\} \cap \mathcal{C}\{\beta, \gamma\}$	$\{\alpha, \beta, \gamma\}$
A -- B (set difference)	$\oplus$	$\oplus$	$\text{setA} \setminus \text{setB}$	$\oplus$	$\{\alpha, \beta\} \setminus \{\beta, \gamma\}$	$\{\alpha\}$
	$\oplus$	$\ominus$	$\text{setA} \cap \text{setB}$	$\oplus$	$\{\alpha, \beta\} \setminus \mathcal{C}\{\beta, \gamma\}$	$\{\beta\}$
	$\ominus$	$\oplus$	$\text{setA} \cup \text{setB}$	$\ominus$	$\mathcal{C}\{\alpha, \beta\} \setminus \{\beta, \gamma\}$	$\mathcal{C}\{\alpha, \beta, \gamma\}$
	$\ominus$	$\ominus$	$\text{setB} \setminus \text{setA}$	$\oplus$	$\mathcal{C}\{\alpha, \beta\} \setminus \mathcal{C}\{\beta, \gamma\}$	$\{\gamma\}$
A ~ B (symmetric difference)	$\oplus$	$\oplus$	$\text{setA} \setminus \text{setB}$	$\oplus$	$\{\alpha, \beta\} \ominus \{\beta, \gamma\}$	$\{\alpha, \gamma\}$
	$\oplus$	$\ominus$	$\text{setA} \cap \text{setB}$	$\oplus$	$\{\alpha, \beta\} \ominus \mathcal{C}\{\beta, \gamma\}$	$\mathcal{C}\{\alpha, \gamma\}$
	$\ominus$	$\oplus$	$\text{setA} \cup \text{setB}$	$\ominus$	$\mathcal{C}\{\alpha, \beta\} \ominus \{\beta, \gamma\}$	$\mathcal{C}\{\alpha, \gamma\}$
	$\ominus$	$\ominus$	$\text{setB} \setminus \text{setA}$	$\oplus$	$\mathcal{C}\{\alpha, \beta\} \ominus \mathcal{C}\{\beta, \gamma\}$	$\{\alpha, \gamma\}$

The normal set equivalences hold, such as  $\mathcal{C}_{\mathbb{S}}(\text{A} \cup \text{B}) = \mathcal{C}_{\mathbb{S}}\text{A} \cap \mathcal{C}_{\mathbb{S}}\text{B}$

## Annex E: Notation for Properties of Strings

Properties of strings are properties that can apply to, or match, sequences of two or more characters (in addition to single characters). This is in contrast to the more common case of properties of characters, which are functions of individual code points only. Those properties marked with an asterisk in the [Full Properties](#) table are properties of strings. See, for example, [Basic\\_Emoji](#).

The preferred notation for properties of strings is `\p{Property_Name}`, the same as for the traditional properties of characters. For regular expressions, properties of strings may appear both within and outside of character class expressions. As described in [Annex D](#), some character class expressions are invalid when they contain properties of strings. Detection of such invalid expressions should happen early, when the regular expression is first compiled or processed.

Implementations that are constrained in that they do not support strings in character classes may use `\m{Property_Name}` as an alternate notation for properties of strings appearing outside of character class expressions. However:

- `\m` should also accept ordinary properties of characters. If a property that applies to strings later changes to only apply to characters, a regex with such a `\m{property}` should not become invalid. Also, being able to use the same `\m` syntax outside of a character class for any property would be simpler for a regex writer.
- Implementations with full support for `\p` and properties of strings in character class expressions may also optionally support the `\m` syntax.
- Implementations that initially adopt `\m` only for properties of strings, then later add support for strings in character classes, should also add support for `\p` as alternate syntax for properties of strings.

Annex F. Parsing Character Classes

It is reasonably straightforward to build a parser for Character Classes. While there are many ways to do this, the following describes one example of a logical process for building such a parser. Implementations can use optimized code, such as a DFA ([Deterministic Finite Automaton](#)) for processing.

Storage

The description uses Java syntax to illustrate the code, but of course would be expressed in other programming languages. At the core is a class (here called CharacterClass) that stores the information that is being built, typically a set of strings optimized for compact storage of ranges of characters, such as ICU's [UnicodeSet](#) (C++).

The methods needed are the following:

Method	Meaning
CharacterClass create();	$A = \{\}$
void addAll(CharacterClass other);	$A = A \cup \text{other}$
void retainAll(CharacterClass other);	$A = A \cap \text{other}$
void removeAll(CharacterClass other);	$A = A \setminus \text{other}$
void symmetricDiffAll(CharacterClass other);	$A = A \oplus \text{other}$
void add(int cp);	$A = A \cup \{cp\}$
void addRange(int cpStart, int cpEnd);	$A = A \cup \{cpStart .. cpEnd\}$
void addString(String stringToAdd);	$A = A \cup \{\text{stringToAdd}\}$
void codePointComplement();	$A = \mathbb{C}_p A$
void setToProperty(String propertyString);	$A = \text{propertySet}$


Building

At the top level a method parseCharacterClass can recognize and branch on '\p{', '\P{', '[', and '['^'. For '\p{' and '\P{' , it calls a parseProperty method that parses up to an unescaped '}', and returns a set based on Unicode properties. See [RL1.2 Properties](#), [2.7 Full Properties](#), [RL2.7 Full Properties](#), and [2.8 Optional Properties](#).

For '[', and '['^', it calls a parseSequence method that parses out items, stopping when it hits ']'. The type of each item can be determined by the initial characters. There is a special check for '-' so that it can be interpreted according to context. The targetSet is set to the first item. All successive items at that level are combined with the targetSet, according to the specified operation (union, intersection, etc.). Note that other binding/precedence options would require somewhat more complicated parsing.

For the Character Class item, a recursive call is made on the parseCharacterClass method. The other initial characters that are branched on are '\u{' , '\u' , '\q{' , '\N{' , '\', the operators, and literal and escaped characters.

Examples

In the following examples,  is a cursor marking how the parsing progresses. For brevity, intermediate steps that only change state are omitted. The two examples are the same, except that in the right-hand example the second and third character classes are grouped.

<table><tr><th>Input</th><th>Action</th><th>Result</th></tr><tr><td></td><td></td><td></td></tr></table>	Input	Action	Result				<table><tr><th>Input</th><th>Action</th><th>Result</th></tr><tr><td></td><td></td><td></td></tr></table>	Input	Action	Result			
Input	Action	Result											
Input	Action	Result											

$\frac{\circ}{\circ}[[abc] \text{ -- } [bcd] \&\& [c-e]]$	A = create()	A = []
$[[a \frac{\circ}{\circ} bc] \text{ -- } [bcd] \&\& [c-e]]$	A.add('a')	A = [a]
$[[ab \frac{\circ}{\circ} c] \text{ -- } [bcd] \&\& [c-e]]$	A.add('b')	A = [ab]
$[[abc \frac{\circ}{\circ}] \text{ -- } [bcd] \&\& [c-e]]$	A.add('c')	A = [a-c]
$[[abc] \text{ -- } \frac{\circ}{\circ}[bcd] \&\& [c-e]]$	B = create()	A = [a-c]
$[[abc] \text{ -- } [b \frac{\circ}{\circ} cd] \&\& [c-e]]$	B.add('b')	
$[[abc] \text{ -- } [bc \frac{\circ}{\circ} d] \&\& [c-e]]$	B.add('c')	B = [b-c]
$[[abc] \text{ -- } [bcd \frac{\circ}{\circ}] \&\& [c-e]]$	B.add('d')	B = [b-d]
$[[abc] \text{ -- } [bcd] \frac{\circ}{\circ} \&\& [c-e]]$	A.removeAll(B)	A = [a]
$[[abc] \text{ -- } [bcd] \&\& \frac{\circ}{\circ}[c-e]]$	B.clear()	B = []
$[[abc] \text{ -- } [[bcd] \&\& [c \frac{\circ}{\circ} -e]]]$	B.add('c')	B = [c]
$[[abc] \text{ -- } [[bcd] \&\& [c-e \frac{\circ}{\circ}]]]$	B.addRange('d', 'e')	B = [c-e]
$[[abc] \text{ -- } [[bcd] \&\& [c-e] \frac{\circ}{\circ}]]]$	A.retainAll(C)	A = []

$\frac{\circ}{\circ}[[abc] \text{ -- } [[bcd] \&\& [c-e]]]$	A = create()	A = []
$[[a \frac{\circ}{\circ} bc] \text{ -- } [[bcd] \&\& [c-e]]]$	A.add('a')	A = [a]
$[[ab \frac{\circ}{\circ} c] \text{ -- } [[bcd] \&\& [c-e]]]$	A.add('b')	A = [ab]
$[[abc \frac{\circ}{\circ}] \text{ -- } [[bcd] \&\& [c-e]]]$	A.add('c')	A = [a-c]
$[[abc] \text{ -- } \frac{\circ}{\circ}[[bcd] \&\& [c-e]]]$	B = create()	A = [a-c]
$[[abc] \text{ -- } [[b \frac{\circ}{\circ} cd] \&\& [c-e]]]$	B.add('b')	
$[[abc] \text{ -- } [[bc \frac{\circ}{\circ} d] \&\& [c-e]]]$	B.add('c')	B = [b-d]
$[[abc] \text{ -- } [[bcd \frac{\circ}{\circ}] \&\& [c-e]]]$	B.add('d')	B = [b-d]
$[[abc] \text{ -- } [[bcd] \frac{\circ}{\circ} \&\& [c-e]]]$		
$[[abc] \text{ -- } [[bcd] \&\& \frac{\circ}{\circ}[c-e]]]$	C = create()	C = []
$[[abc] \text{ -- } [[bcd] \&\& [c \frac{\circ}{\circ} -e]]]$	C.add('c')	C = [c]
$[[abc] \text{ -- } [[bcd] \&\& [c-e \frac{\circ}{\circ}]]]$	C.addRange('d', 'e')	C = [c-e]
$[[abc] \text{ -- } [[bcd] \&\& [c-e] \frac{\circ}{\circ}]]]$	B.retainAll(C)	B = [cd]
$[[abc] \text{ -- } [[bcd] \&\& [c-e]]] \frac{\circ}{\circ}$	A.removeAll(B)	A = [ab]

## References

- [Case] Section 3.13, *Default Case Algorithms* in [Unicode]
- [CaseData] <https://www.unicode.org/Public/UCD/latest/ucd/CaseFolding.txt>
- [Friedl] Jeffrey Friedl, "Mastering Regular Expressions", 2nd Edition 2002, O'Reilly and Associates, ISBN 0-596-00289-0
- [Glossary] Unicode Glossary  
<https://www.unicode.org/glossary/>  
*For explanations of terminology used in this and other documents.*
- [Perl] <https://perldoc.perl.org/>  
 See especially:  
<https://perldoc.perl.org/charnames.html>  
<https://perldoc.perl.org/perlre.html>

<https://perldoc.perl.org/perluniintro.html>  
<https://perldoc.perl.org/perlunicode.html>

- [POSIX] The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition, "Locale" chapter  
[https://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd\\_chap07.html](https://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap07.html)
- [Prop] <https://www.unicode.org/Public/UCD/latest/ucd/PropertyAliases.txt>
- [PropValue] <https://www.unicode.org/Public/UCD/latest/ucd/PropertyValueAliases.txt>
- [ScriptData] <https://www.unicode.org/Public/UCD/latest/ucd/Scripts.txt>
- [SpecialCasing] <https://www.unicode.org/Public/UCD/latest/ucd/SpecialCasing.txt>
- [UAX14] UAX #14, *Unicode Line Breaking Algorithm*  
<https://www.unicode.org/reports/tr14/>
- [UAX15] UAX #15, *Unicode Normalization Forms*  
<https://www.unicode.org/reports/tr15/>
- [UAX24] UAX #24, *Unicode Script Property*  
<https://www.unicode.org/reports/tr24/>
- [UAX29] UAX #29, *Unicode Text Segmentation*  
<https://www.unicode.org/reports/tr29/>
- [UAX31] UAX #31, *Unicode Identifier and Pattern Syntax*  
<https://www.unicode.org/reports/tr31/>
- [UAX38] UAX #38, *Unicode Han Database (UniHan)*  
<https://www.unicode.org/reports/tr38/>
- [UAX44] UAX #44, *Unicode Character Database*  
<https://www.unicode.org/reports/tr44/>
- [UData] <https://www.unicode.org/Public/UCD/latest/ucd/UnicodeData.txt>
- [Unicode] The Unicode Standard  
*For the latest version, see:*  
<https://www.unicode.org/versions/latest/>
- [UTR50] UTR #50, *Unicode Vertical Text Layout*  
<https://www.unicode.org/reports/tr50/>
- [UTR51] UTR #51, *Unicode Emoji*  
<https://www.unicode.org/reports/tr51/>
- [UTS10] UTS #10, *Unicode Collation Algorithm (UCA)*  
<https://www.unicode.org/reports/tr10/>
- [UTS35] UTS #35, *Unicode Locale Data Markup Language (LDML)*  
<https://www.unicode.org/reports/tr35/>
- [UTS39] UTS #39, *Unicode Security Mechanisms*  
<https://www.unicode.org/reports/tr39/>
- [UTS46] UTS #46, *Unicode IDNA Compatibility Processing*  
<https://www.unicode.org/reports/tr46/>

## Acknowledgments

Mark Davis created the initial version of this annex and maintains the text, with significant contributions from Andy Heninger. Andy also served as co-editor for many years.

Thanks to Julie Allen, Mathias Bynens, Tom Christiansen, David Corbett, Michael D'Errico, Asmus Freytag, Jeffrey Friedl, Norbert Lindenberg, Peter Linsley, Alan Liu, Kent Karlsson, Jarkko Hietaniemi, Ivan Panchenko, Michael Saboff, Gurusamy Sarathy, Markus Scherer, Xueming Shen, Henry Spencer, Kento Tamura, Philippe Verdy, Tom Watson, Ken Whistler, Karl Williamson, and Richard Wordingham for their feedback on the document.

## Modifications

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

## Revision 24

### Summary:

Added 5 additional properties to the Full Properties list, and referenced UAX#34 and UAX44 where needed.

### Details:

- Added IDS\_Unary\_Operator, NFKC\_Simple\_Casefold, ID\_Compat\_Math\_Start, and ID\_Compat\_Math\_Continue to the Full Properties list in *Section 2.7 Full Properties*
- Fixed the references to the namespace for character names to reference the *Unicode namespace for character names* [UAX34-D3].
- Clarified that the the matching rules from *Section 5.9 Matching Rules* of [UAX44] should be used for property names and values.
- Fixed the last example in *Section 1.3 Subtraction and Intersection* to be `[\P{Script=Greek}&&\P{Basic_Emoji}]`
- Clarified that the results in the table of wildcard examples are for Unicode 5.0, in *Section 2.6 Wildcards in Property Values*

Modifications for previous versions are listed in those respective versions.

---

Copyright © 2023 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.