L2/16-150

Proposed Update Unicode® Technical Standard #18

UNICODE REGULAR EXPRESSIONS

Version	18 (Draft 1)
Editors	Mark Davis, Andy Heninger
Date	2016-04-26
This Version	http://www.unicode.org/reports/tr18/tr18-18.html
Previous Version	http://www.unicode.org/reports/tr18/tr18-17.html
Latest Version	http://www.unicode.org/reports/tr18/
Latest Proposed Update	http://www.unicode.org/reports/tr18/proposed.html
Revision	18

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*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time.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

The following describes general guidelines for extending regular expression engines (Regex) to handle Unicode. The following issues are involved in such extensions.

- 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 three 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.
- **Level 3: Tailored Support.** At this level, the regular expression engine also provides for tailored treatment of characters, including country- or language-specific behavior. For example, the characters *ch* can behave as a single character in Slovak or traditional Spanish. The results of a particular regular expression reflect the end-users' expectations of what constitutes a character in their language, and the order of the characters. However, there is a performance impact to support at this level.

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.
3. Level 3 contains information about extensions only useful for specific applications. Features at this level may require further investigation for effective implementation.

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:*** *Unicode is a constantly evolving standard: 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, where the Euro currency symbol was added.*

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:

`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 following syntax for character ranges will be used in successive examples.

> ***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.*

```
LIST := "[" NEGATION? ITEM (SEP? ITEM)* "]"
ITEM := CODE_POINT2
     := CODE_POINT2 "-" CODE_POINT2 // range

CODE_POINT2 := ESCAPE CODE_POINT
            := CODE_POINT

NEGATION := "^"
SEP := ""   // no separator = union
    := "||" // union
ESCAPE := "\"
```

CODE_POINT refers to any Unicode code point from U+0000 to U+10FFFF, although typically the only ones of interest will be those representing characters. Whitespace is allowed between any elements, but to simplify the presentation the many occurrences of " "* are omitted.

Code points that are syntax characters or whitespace are typically escaped. For more information see [UAX31]. In examples, the syntax \s to mean white space is sometimes used. See also *Annex C: Compatibility Properties*.

Examples:

`[a-z || A-Z || 0-9]`	Match ASCII alphanumerics
`[a-z A-Z 0-9]`	
`[a-zA-Z0-9]`	
`[^a-z A-Z 0-9]`	Match anything but ASCII alphanumerics
`[\] \- \ ]`	Match 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 notation is defined for use here and in other Unicode documents:

`\n`	As used within regular expressions, expands to the text matching the nth parenthesized group in regular expression. (à la Perl). Note that most engines limit n to be [1-9]; thus \456 would be the reference to the 4th group followed by the literal '56'.
`$n`	As used within replacement strings for regular expressions, expands to the text matching the nth parenthesized group in a corresponding regular expression. The

	value of $0 is the entire expression.(à la Perl)
`$xyz`	As used within regular expressions or replacement strings, expands to an assigned variable value. The 'xyz' is of the form of an identifier. For example, given `$greek_lower = [[:greek:]&&[:lowercase:]]`, the regular expression pattern "`ab$greek_lower`" is equivalent to "`ab[[:greek:]&&[:lowercase:]]`".

Note: *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.*

0.2 Conformance

The following describes the possible ways that an implementation can claim conformance to this 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
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. An implementation claiming conformance to Level 3 of this specification shall satisfy C1 and C2, and meet the requirements described in the following sections:

RL3.1 Tailored Punctuation
RL3.2 Tailored Grapheme Clusters
RL3.3 Tailored Word Boundaries
RL3.6 Context Matching
RL3.7 Incremental Matches
RL3.9 Possible Match Sets
RL3.11 Submatchers

C4. An implementation claiming partial conformance to this specification shall clearly indicate which levels are completely supported (C1-C3), plus any additional supported features from higher levels.

For example, an implementation may claim conformance to Level 1, plus Context Matching, and Incremental Matches. Another 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 \U0001**1D11E,** \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:

```
<codepoint>  := <character>
<codepoint>  := "\u" HEX_CHAR HEX_CHAR HEX_CHAR HEX_CHAR
<codepoint>  := "\u{" HEX_CHAR+ "}"
<codepoints> := "\u{" HEX_CHAR+ (SEP HEX_CHAR+)* "}"
<sep>        := \s+

U_SHORT_MARK := "u"
```

Examples:

`[\u{3040}-\u{309F} \u{30FC}]`	Match Hiragana characters, plus prolonged sound sign
`[\u{B2} \u{2082}]`	Match superscript and subscript 2
`[a \u{10450}]`	Match "a" or U+10450 SHAVIAN LETTER PEEP
`ab\u{63 64}`	Match "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, here are some examples of (current) escape syntax for Unicode code points:

Characters	👽€£a<tab>
Code Point†	U+1F47D U+20AC U+00A3 U+0061 U+0009
CSS†	\1F47D \20AC \A3 \61 \9
UTS18, Ruby	\u{1F47D 20AC A3 61 9}
Perl	\x{1F47D}\x{20AC}\x{A3}\x{61}
XML/HTML	&#x1F47D;&#x20AC;&#xA3;&#x61;&#x9;
C++/Python/ICU	\U0001F47D\u20AC\u00A3\u0061\u0009
Java/JS/ICU*	\uD83D\uDC7D\u20AC\u00A3\u0061\u0009
URL*	%F0%9F%91%BD%E2%82%AC%C2%A3%61%09
XML/HTML*	&#128125;&#8364;&#163;&#97;&#9;

† Following whitespace is consumed.
* Does not satisfy RL1.1

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 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 character 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, a regular expression engine needs to provide for the recognition of whole categories of characters as well as simply ranges of characters; otherwise the listing of characters becomes impractical and error-prone. This is done by providing syntax for sets of characters based on the Unicode character properties, and allowing them to be mixed with lists and ranges of individual code points.

There are a large number of Unicode Properties defined in the Unicode Character Database (UCD), which also provides the official data for mapping Unicode characters (and code points) to property values. See Section 2.7, *Full Properties*; UAX #44: *Unicode Character Database* [UAX44]; and Chapter 4 in *The Unicode Standard* [Unicode]. The defined Unicode string functions, such as isNFC() and isLowercase(), also apply to single code points and are useful to support in regular expressions.

The recommended names for UCD properties and property values are in PropertyAliases.txt [Prop] and PropertyValueAliases.txt [PropValue]. 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 be used, whereby the case distinctions, whitespace, hyphens, and underbar are ignored.

> ***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 negative match.

```
ITEM := POSITIVE_SPEC | NEGATIVE_SPEC
POSITIVE_SPEC := ("\p{" PROP_SPEC "}") | ("[:" PROP_SPEC ":]")
NEGATIVE_SPEC := ("\P{" PROP_SPEC "}") | ("[:^" PROP_SPEC ":]")
PROP_SPEC  := <binary_unicode_property>
PROP_SPEC  := <unicode_property> (":" | "=" | "≠" | "!=" ) VALUE
PROP_SPEC  := <script_or_category_property_value>  ("|"
<script_or_category_property_value>)*
PROP_VALUE := <unicode_property_value> ("|" <unicode_property_value>)*
```

Examples:

[\p{L} \p{Nd}]	Match all letters and decimal digits
[\p{letter} \p{decimal number}]	
[\p{letter|decimal number}]	
[\p{L|Nd}]	
\P{script=greek}	Match anything that does not have the Greek script
\P{script:greek}	
\p{script≠greek}	
[:^script=greek:]	
[:^script:greek:]	
[:script≠greek:]	
\p{East Asian Width:Narrow}	Match anything that has the `East Asian Width` property value of Narrow
\p{Whitespace}	Match anything that has the binary property Whitespace

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 both a binary property and a value of the Line_Break enumeration, so \p{Alphabetic} would mean the binary property, and \p{Line Break:Alphabetic} or \p{Line_Break=Alphabetic} would mean the enumerated property. There are two exceptions to this: the properties *Script* and *General Category* commonly have the property name omitted. Thus \p{Not_Assigned} is equivalent to \p{General_Category = Not_Assigned}, 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].

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 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.

For more information on properties, see UAX #44: *Unicode Character Database* [UAX44].

Of the properties in RL1.2, General Category and Script have enumeration 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}`.

When property$_x$ is defined to have values that are sets of other values, the notation \p{property$_x$=value$_y$} represents the set of all code points whose property values *contain* value$_y$. For example, the Script_Extensions property value for U+30FC (ー) is the set {Hiragana, Katakana}. So U+30FC (ー) is contained in \p{Script_Extensions=Hiragana}, and is also contained in \p{Script_Extensions=~~Hiragana~~Katakana}.

1.2.1 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 [UData].

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}`

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
L	Letter
Lu	Uppercase Letter
Ll	Lowercase Letter

Abb.	Long form
S	Symbol
Sm	Math Symbol
Sc	Currency Symbol

Abb.	Long form
Z	Separator
Zs	Space Separator
Zl	Line Separator

Lt	Titlecase Letter
Lm	Modifier Letter
Lo	Other Letter
M	Mark
Mn	Non-Spacing Mark
Mc	Spacing Combining Mark
Me	Enclosing Mark
N	Number
Nd	Decimal Digit Number
Nl	Letter Number
No	Other Number

Sk	Modifier Symbol
So	Other Symbol
P	Punctuation
Pc	Connector Punctuation
Pd	Dash Punctuation
Ps	Open Punctuation
Pe	Close Punctuation
Pi	Initial Punctuation
Pf	Final Punctuation
Po	Other Punctuation

Zp	Paragraph Separator
C	Other
Cc	Control
Cf	Format
Cs	Surrogate
Co	Private Use
Cn	Not Assigned
-	Any*
-	Assigned*
-	ASCII*

* The last few properties are not part of the General Category.

- *Any* matches all code points. This could also be captured with `[\u{0}-\u{10FFFF}]`. In some regular expression languages, `\p{Any}` may be expressed by a period, but that may exclude newline characters.
- *Assigned* is equivalent to `\P{Cn}`, and matches all assigned characters (for the target version of Unicode). It also includes all private use characters. It is useful for avoiding confusing double negatives. Note that *Cn* includes noncharacters, so *Assigned* excludes them.
- ASCII is equivalent to `[\u{0}-\u{7F}]`.

1.2.2 Script and Script Extensions Properties

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 [ScriptData]). Using a property such as `\p{sc=Greek}` allows implementations to test whether letters are Greek or not.

~~There are situations where characters are regularly used with multiple scripts, including common characters such as U+30FC (ー) KATAKANA-HIRAGANA PROLONGED SOUND MARK. To account for such cases, support of the Script_Extensions property (abbreviated as **scx**) is recommended. The Script_Extensions property maps a code point to a set of one or more scripts.~~ Some characters, such as U+30FC (ー) KATAKANA-HIRAGANA PROLONGED SOUND MARK, are regularly used with multiple scripts. For such characters the Script_Extensions property (abbreviated as **scx**) identifies the set of associated scripts. The following shows some sample characters with their Script and Script_Extensions property values:

Code	Char	Name	sc	scx
U+3042	あ	HIRAGANA LETTER A	Hira	{Hira}
U+30FC	ー	KATAKANA-HIRAGANA PROLONGED	Zyyy =	{Hira, Kana}

		SOUND MARK	Common	
U+3099	゙	COMBINING KATAKANA-HIRAGANA VOICED SOUND MARK	Zinh = Inherited	{Hira, Kana}
U+30FB	・	KATAKANA MIDDLE DOT	Zyyy = Common	{Bopo, Hang, Hani, Hira, Kana, Yiii}

The expression \p{sc=Hira} includes those characters whose *Script* value *is* Hira, while the expression \p{scx=Hira} includes all the characters whose *Script_Extensions* value *contains* Hira. The following shows the difference:

Expression	Contents
\p{sc=Hira}	[ぁ-ゖゝ-ゟ𛀁🈀]
\p{scx=Hira}	[、-〃〆〈-】〓-〟〰-〵〷〼-〿ぁ-ゖ゙-゠・-ー㆐-㆟㇀-㇣㈠-㉇㊀-㊰㋀-㋋㍘-㍰㍻-㍿㏠-㏾﹅﹆｡-･ﾞﾟ𛀁🈀]

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 (\p{scx=X} ⊇ \p{sc=X}). However, in some cases that is not true. For example, the Script property value for U+30FC (ー) is Common, but the Script_Extensions value for U+30FC (ー) does not contain the script value Common. In other words, \p{scx=Common} ⊉ \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 UAX #24. The values for Script_Extensions are likely 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].

1.2.3 Other Properties

Other recommended properties are described in 2.7 Full Properties. See also 2.5 Name Properties and 2.6 Wildcards in Property Values.

Implementations may also add other regular expression properties based on Unicode data that are not listed under RL1.2. Some useful candidates include:

- isCased, isLowercase, toLowercase, and so on from Chapter 3 in [Unicode]
- cjkTraditionalVariant, cjkSimplifiedVariant, CJK_Radical number from the Unihan data in the UCD [Unicode]
- isNF*, toNF* (* = D, C, K, KC from [UAX15])
- toNFKC_Casefold (see [Case])
- exemplar characters from [UTS35]
- IDNA status and mapping from [UTS46]
- identifier restriction status and type from [UTS39]
- DUCET primary values from [UTS10]
- Emoji, Emoji_Presentation, Emoji_Modifier and Emoji_Modifier_Base from [UTR51]

- vertical orientation from [UTR50]

The following tables gives examples of such properties in use:

String properties	Description
[:toNFC=Å:]	The set of all characters X such that toNFC(X) = "a"
[:toNFD=A\u{300}:]	The set of all characters X such that toNFD(X) = "A\u{300}"
[:toNFKC=A:]	The set of all characters X such that toNFKC(X) = "A"
[:toNFKD=A\u{300}:]	The set of all characters X such that toNFKD(X) = "a"
[:toLowercase=a:]	The set of all characters X such that toLowercase(X) = "a"
[:toUppercase=A:]	The set of all characters X such that toUppercase(X) = "A"
[:toTitlecase=A:]	The set of all characters X such that toTitlecase(X) = "A"
[:toCaseFold=a:]	The set of all characters X such that toCasefold(X) = "A"
\p{exemplars=zh-Hant}	The exemplar characters from LDML.
Binary properties	**Description**
[:isNFC:]	The set of all characters X such that toNFC(X) = X
[:isNFD:]	The set of all characters X such that toNFD(X) = X
[:isNFKC:]	The set of all characters X such that toNFKC(X) = X
[:isNFKD:]	The set of all characters X such that toNFKD(X) = X
[:isLowercase:]	The set of all characters X such that toLowercase(X) = X
[:isUppercase:]	The set of all characters X such that toUppercase(X) = X
[:isTitlecase:]	The set of all characters X such that toTitlecase(X) = X
[:isCaseFolded:]	The set of all characters X such that toCasefo(X) = X
[:isCased:]	The set of all cased characters.

1.2.4 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}].

1.2.5 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 ~~Unicode sets~~ sets of characters within regular expression character class expressions.

`ITEM      := "[" ITEM "]"`	// for grouping
`OPERATOR := ""`	// no separator = union
`         := "||"`	// union: A∪B
`         := "&&"`	// intersection: A∩B
`         := "--"`	// set difference: A-B
`         := "~~"`	// symmetric difference: A⊖B = (A∪B)-(A∩B)

Implementations may also choose to offer other set operations. The symmetric difference of two sets is particularly useful. It is defined as being the union minus the intersection. Thus `[\p{letter}~~\p{ascii}]` is equivalent to `[[\p{letter}\p{ascii}]--[\p{letter}&&\p{ascii}]]`.

For compatibility with industry practice, symbols are doubled in the above notation. This practice provides for better backwards compatibility with expressions using older syntax, because they are unlikely to contain doubled characters. It also allows the operators to appear adjacent to ranges without ambiguity, such as `[\p{letter}--a-z]`.

Binding or precedence may vary by regular expression engine, so 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..E` stand for expressions, not characters:

Expression	Equals	When
[ABC--DE]	[[AB]C]--[DE]]	Union binds more closely. That is, it means: Form the union of A, B, and C, and then subtract the union of D and E.
	[[[[AB]C]--D]E]]	Operators are on the same level. That is, it means: Form the union of A, B, and C, and then subtract D, and then add E.

Even where an expression is not ambiguous, extra grouping brackets may be useful for clarity.

Examples:

`[\p{L}--QW]`	Match all letters but Q and W
`[\p{N}--[\p{Nd}--0-9]]`	Match all non-decimal numbers, plus 0-9
`[\u{0}-\u{7F}--\P{letter}]`	Match all letters in the ASCII range, by subtracting non-letters
`[\p{Greek}--\N{GREEK SMALL LETTER ALPHA}]`	Match Greek letters except alpha
`[\p{Assigned}--\p{Decimal Digit Number}--a-fA-Fa-fA-F]`	Match all assigned characters except for hex digits (using a broad definition)

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 [UData], 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 offers 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)(?:å|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, Ᵽ, and Ᵹ, 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 Ᵽ and Ᵹ.

- 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. uniformally 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 Unicode code units to express code points above FFFF$_{16}$. Surrogate pairs (or their equivalents in other encoding forms) are 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

Note: It is permissible, but not required, to match an isolated surrogate code point (such as \u{D800}), which may occur in Unicode 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:

1. o + horn + dot_below
    1. U+006F (o) LATIN SMALL LETTER O
    2. U+031B (◌̛) COMBINING HORN
    3. U+0323 (◌̣) COMBINING DOT BELOW
2. o + dot_below + horn
    1. U+006F (o) LATIN SMALL LETTER O
    2. U+0323 (◌̣) COMBINING DOT BELOW
    3. U+031B (◌̛) COMBINING HORN
3. o-horn + dot_below
    1. U+01A1 (ơ) LATIN SMALL LETTER O WITH HORN
    2. U+0323 (◌̣) COMBINING DOT BELOW
4. o-dot_below + horn
    1. U+1ECD (ọ) LATIN SMALL LETTER O WITH DOT BELOW
    2. U+031B (◌̛) COMBINING HORN
5. o-horn-dot_below
    1. U+1EE3 (ợ) LATIN SMALL LETTER O WITH HORN AND DOT BELOW

The regular expression pattern /o\u{31B}/ matches the first two characters of #1, the first and third characters of #2, the first character of #3, part of the first character together with the third character of #4, and part of the character in #5.

In practice, regex APIs are not set up to match parts of characters or handle discontiguous 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

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 keep Hangul syllables together and do not break between base characters and combining marks. The precise definition is in UAX #29: *Unicode Text Segmentation* [UAX29]. These *extended* grapheme clusters are not the same as *tailored* grapheme clusters, which are covered in *Section 3.2, Tailored Grapheme Clusters*.

RL2.2 Extended Grapheme Clusters

To meet this requirement, an implementation shall provide a mechanism for matching against an arbitrary extended grapheme cluster, a literal cluster, and matching 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 negation 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	Description
`\b{g}`	Zero-width match at a Unicode extended grapheme cluster boundary
`\b{w}`	Zero-width match at a Unicode word boundary. Note that this is different than `\b` alone, which corresponds to `\w` and `\W`. See Annex C: Compatibility Properties.
`\b{l}`	Zero-width match at a Unicode line break boundary
`\b{s}`	Zero-width match at 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.

Regular expression engines should also provide some mechanism for easily matching against literal clusters, because they are more likely to match user expectations for many languages. One mechanism for doing that is to have explicit syntax for literal clusters, as in the following syntax:

```
ITEM := "\q{" CODE_POINT + "}"
```

This syntax can also be used for tailored grapheme clusters (see Tailored Grapheme Clusters).

Examples:

`[a-z\q{x\u{323}}]`	Match a–z, and x with an under-dot (used in American Indian languages).
`[a-z\q{aa}]`	Match a–z, and aa (treated as a single character in Danish).
`[a-z ñ \q{ch} \q{ll} \q{rr}]`	Match some lowercase characters in traditional Spanish.

In implementing extended grapheme clusters, the expression `/[a-m \q{ch} \q{rr}]/` should behave roughly like `/(?: ch | rr | [a-m])/`. That is, the expression would:

- match ch or rr and advance by two code points, or
- match a-m and advance one code point, or
- fail to match

Note that the strings need to be ordered as longest first to work correctly in arbitrary regex engines, because some regex engines try the leftmost matching alternative first. For example, the expression `/[a-m {ch} {chh}]/` would need to behave like `/(?: chh | ch | [a-m])/`, with "chh" before "ch".

Matching a *complemented* set containing strings like \q{ch} may behave differently in the two different modes: the normal mode where code points are the unit of matching, or the mode where extended grapheme clusters are the unit of matching. That is, the expression `[^ a-m \q{ch} \q{rr}]` should behave in the following way:

Mode	Behavior	Description
normal	`(?! ch | rr | [a-m] ) [\u{0}-\u{10FFFF}]`	failing with strings starting with a–m, ch, or rr, and otherwise advancing by one code point
grapheme cluster	`(?! ch | rr | [a-m] ) \X`	failing with strings starting with a–m, ch, or rr, and otherwise advancing by one extended grapheme cluster

A complex character set containing strings like `\q{ch}` plus embedded complement operations is interpreted as if the complement were pushed up to the top of the expression, using the following rewrites recursively:

Original	Rewrite
^^x	x
^x || ^y	^(x && y)
^x || y	^(x -- y)
x || ^y	^(y -- x)
^x && ^y	^(x || y)
^x -- y	
^x && y	y -- x
^x -- ^y	
x && ^y	x -- y

x -- ^y	x && y
^x ~~ ^y	x ~~ y
^x ~~ y	^(x ~~ y)
x ~~ ^y	

Applying these rewrites results in a simplification of the regex expression. Either the complement operations will be completely eliminated, or a single remaining complement operation will remain at the top level of the expression. Logically, then, the rest of the expression consists of a flat list of characters and/or multi-character strings; matching strings can then can be handled as described above.

2.2.1 Grapheme Cluster Mode

A grapheme cluster mode behaves more like users' expectations for character boundaries, and is especially useful for handling canonically equivalent matching. In a grapheme cluster mode, matches are guaranteed to be on extended grapheme cluster boundaries. Each atomic literal of the pattern matches complete extended grapheme clusters, and thus behaves as if followed by `\b{g}`. Atomic literals include: a dot, a character class (like `[a-m]`), a sequence of characters (perhaps with some being escaped) that matches as a unit, or syntax that is equivalent to these. Note that in `/abc?/`, the `"abc"` is not matching as a unit; the `?` modifier is only affecting the last character, and thus the `ab` and the `c` are separate atomic literals. To summarize:

Syntax	Description
`.`	Behaves like `\X`; that is, matches a full extended grapheme cluster going forward.
`[abc{gh}]`	Behaves like `[abc{gh}]\b{g}`; that is, matches only if the end point of the match is at a grapheme cluster boundary
`abcd`	Behaves like `abcd\b{g}`; that is, matches only if the end point of the match is at a grapheme cluster boundary

Note that subdivisions can modify the behavior in this mode. Normally `/(xy)/` is equivalent to `/(x)(y)/` in terms of matching (where x and y are arbitrary literal character strings); that is, only the grouping is different. That is not true in grapheme cluster mode, where each atomic literal acts like it is followed by `\b{g}`. For example, `/(x\u{308})/` is not the same as `/(x)(\u{308})/` in matching. The former behaves like `/(x\u{308}\b{g})/` while the latter behaves like `/(x\b{g})(\u{308}\b{g})/`. The latter will never match in grapheme cluster mode, since it would only match if there were a grapheme cluster boundary after the x and if x is followed by \u{308}, but that can never happen simultaneously.

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].

The following provides examples of usage:

Syntax	Description
\p{name=ZERO WIDTH NO-BREAK SPACE}	[\u{FEFF}], using the Name property.
\p{name=zerowidthno breakspace}	[\u{FEFF}], using the Name property, and Matching Rules [UAX44].

\p{name=BYTE ORDER MARK}	[\u{FEFF}], using the Name_Alias property.
\p{name=BOM}	[\u{FEFF}], using the Name_Alias property (a second value).
\p{name=HANGUL SYLLABLE GAG}	[\u{AC01}], with a computed name.
\p{name=BEL}	[\u{7}], the control character.
\p{name=BELL}	[\u{1F514}, 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]:

`\p{name=private-use-E000}`	[\u{E000}]
`\p{name=surrogate-D800}`	[\u{D800}] *Note: this would only apply to isolated surrogate code points.*
`\p{name=noncharacter-FDD0}`	[\u{FDD0}]
`\p{name=control-0007}`	[\u{7}]

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. 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.

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 in ranges.

```
<codepoint> := "\N{" <character_name> "}"
```

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

1. \N matches a single character or a sequence, while \p matches a set of characters.
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.
3. 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 designate a set of two characters:

- U+1F47D (👽) EXTRATERRESTRIAL ALIEN,
- U+1F47E (👾) ALIEN MONSTER

4. The namespace for the \p{name=...} syntax is the namespace for character names plus name aliases. The namespace for the \N syntax includes named sequences defined in NamedSequences.txt, such as \N{KHMER CONSONANT SIGN COENG KA}. Sequences behave as a single element, so \N{KHMER CONSONANT SIGN COENG KA}* should be treated as if it were the expression (\u{17D2 1780})*.

As with other property values, names should use a loose match, disregarding case, spaces and hyphen (the underbar character "_" cannot occur in Unicode character names). An implementation may also choose to allow namespaces, where some prefix like "LATIN LETTER" is set globally and used if there is no match otherwise.

There are, however, three instances that require special-casing with loose matching, where an extra test shall be made for the presence or absence of a hyphen.

- U+0F68 TIBETAN LETTER A and
  U+0F60 TIBETAN LETTER -A
- U+0FB8 TIBETAN SUBJOINED LETTER A and
  U+0FB0 TIBETAN SUBJOINED LETTER -A
- U+116C HANGUL JUNGSEONG OE and
  U+1180 HANGUL JUNGSEONG O-E

Examples:

- \N{WHITE SMILING FACE} or \N{whitesmilingface} is equivalent to \u{263A}
- \N{GREEK SMALL LETTER ALPHA} is equivalent to \u{3B1}
- \N{FORM FEED} is equivalent to \u{C}
- \N{SHAVIAN LETTER PEEP} is equivalent to \u{10450}
- [\N{GREEK SMALL LETTER ALPHA}-\N{GREEK SMALL LETTER BETA}] is equivalent to [\u{3B1}-\u{3B2}]

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 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.

```
PROP_VALUE := <value>
            | "/" <regex expression> "/"
            | "@" <unicode_property> "@"
```

Note: 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.

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.

Examples:

Expression	Description/Contents
\p{toNfd=/b/}	Characters whose NFD form contains a "b" (U+0062) in the value.
	U+0062 (b) LATIN SMALL LETTER B U+1E03 (ḃ) LATIN SMALL LETTER B WITH DOT ABOVE U+1E05 (ḅ) LATIN SMALL LETTER B WITH DOT BELOW U+1E07 (ḇ) LATIN SMALL LETTER B WITH LINE BELOW
\p{name=/^LATIN LETTER.*P$/}	Characters with names starting with "LATIN LETTER" and ending with "P"
	U+01AA (ƪ) LATIN LETTER REVERSED ESH LOOP U+0294 (ʔ) LATIN LETTER GLOTTAL STOP U+0296 (ʖ) LATIN LETTER INVERTED GLOTTAL STOP U+1D18 (ᴘ) LATIN LETTER SMALL CAPITAL P
\p{name=/VARIA(TION|NT)/}	Characters with names containing "VARIATION" or "VARIANT"
	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 (𒆮) CUNEIFORM SIGN KU4 VARIANT FORM U+12425 (𒐥) CUNEIFORM NUMERIC SIGN THREE SHAR2 VARIANT FORM U+1242F (𒐯) CUNEIFORM NUMERIC SIGN THREE SHARU VARIANT FORM U+12437 (𒐷) CUNEIFORM NUMERIC SIGN THREE BURU VARIANT FORM U+1243A (𒐺) CUNEIFORM NUMERIC SIGN THREE VARIANT FORM ESH16 … U+12449 (𒑉) CUNEIFORM NUMERIC SIGN NINE VARIANT FORM ILIMMU A U+12453 (𒑓) CUNEIFORM NUMERIC SIGN FOUR BAN2 VARIANT FORM

	U+12455 (𒑕) CUNEIFORM NUMERIC SIGN FIVE BAN2 VARIANT FORM U+1245D (𒑝) CUNEIFORM NUMERIC SIGN ONE THIRD VARIANT FORM A U+1245E (𒑞) CUNEIFORM NUMERIC SIGN TWO THIRDS VARIANT FORM A U+E0100 ( ) VARIATION SELECTOR-17 ... U+E01EF ( ) VARIATION SELECTOR-256
[\p{toLowercase≠@cp@} & \p{Block=Letterlike Symbols}]	**Characters in the Letterlike symbol block with different toLowercase values**
	U+2126 (Ω) OHM SIGN U+212A (K) KELVIN SIGN U+212B (Å) ANGSTROM SIGN U+2132 (Ⅎ) TURNED CAPITAL F

The above are all on the basis of Unicode 5.0; different versions of Unicode may produce different results.

Here are 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 curently use single-letter operations:

Expression	Description
`[[:name=/CJK/:]-[:ideographic:]]`	The set of all characters with names that contain CJK that are not Ideographic
`[:name=/\bDOT$/:]`	The set of all characters with names that end with the word DOT
`[:block=/(?i)arab/:]`	The set of all characters in blocks that contain the sequence of letters "arab" (case-insensitive)
`[:toNFKC=/\./:]`	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 Unicode, 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, 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 in gray are covered by RL1.2 Properties. For more information on properties, see UAX #44: *Unicode Character Database* [UAX44].

General	Case	Shaping and Rendering
Name (Name_Alias)	Uppercase	Join_Control
Block	Lowercase	Joining_Group
Age	Lowercase_Mapping	Joining_Type
General_Category	Titlecase_Mapping	Line_Break
Script (Script_Extensions)	Uppercase_Mapping	Grapheme_Cluster_Break
White_Space	Case_Folding	Sentence_Break
Alphabetic	Simple_Lowercase_Mapping	Word_Break
Hangul_Syllable_Type	Simple_Titlecase_Mapping	East_Asian_Width
Noncharacter_Code_Point	Simple_Uppercase_Mapping	Prepended_Concatenation_Mark
Default_Ignorable_Code_Point	Simple_Case_Folding	
Deprecated	Soft_Dotted	Bidirectional
Logical_Order_Exception	Cased	Bidi_Class
Variation_Selector	Case_Ignorable	Bidi_Control
	Changes_When_Lowercased	Bidi_Mirrored
Numeric	Changes_When_Uppercased	Bidi_Mirroring_Glyph
Numeric_Value	Changes_When_Titlecased	Bidi_Paired_Bracket
Numeric_Type	Changes_When_Casefolded	Bidi_Paired_Bracket_Type
Hex_Digit	Changes_When_Casemapped	
ASCII_Hex_Digit		CJK
	Normalization	Ideographic
Identifiers	Canonical_Combining_Class	Unified_Ideograph
ID_Continue	~~Decomposition_Mapping~~	Radical
ID_Start	~~Composition_Exclusion~~	IDS_Binary_Operator
XID_Continue	~~Full_Composition_Exclusion~~	IDS_Trinary_Operator
XID_Start	Decomposition_Type	
Pattern_Syntax	NFC_Quick_Check	Miscellaneous
Pattern_White_Space	NFKC_Quick_Check	Math
	NFD_Quick_Check	Quotation_Mark
	NFKD_Quick_Check	Dash
	NFKC_Casefold	Sentence_Term
	Changes_When_NFKC_Casefolded	Terminal_Punctuation
		Diacritic
		Extender

		Grapheme_Base
		Grapheme_Extend

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 1.2.2, Script_Property.*

3 Tailored Support: Level 3

All of the above deals with a default specification for a regular expression. However, a regular expression engine also may want to support tailored specifications, typically tailored for a particular language or locale. This may be important when the regular expression engine is being used by end-users instead of programmers, such as in a word-processor allowing some level of regular expressions in searching.

For example, the order of Unicode characters may differ substantially from the order expected by users of a particular language. The regular expression engine has to decide, for example, whether the list `[a-ä]` means:

- the Unicode characters in binary order between `0061`$_{16}$ and `00E5`$_{16}$ (including '`z`', '`Z`', '[', and '¼'), *or*
- the letters in that order in the users' locale (which *does not* include '`z`' in English, but *does* include it in Swedish).

If both tailored and default regular expressions are supported, then a number of different mechanism are affected. There are two main alternatives for control of tailored support:

- *coarse-grained support:* the whole regular expression (or the whole script in which the regular expression occurs) can be marked as being tailored.
- *fine-grained support:* any part of the regular expression can be marked in some way as being tailored.

For example, fine-grained support could use some syntax such as the following to indicate tailoring to a locale within a certain range. Locale (or language) IDs should use the syntax from locale identifier definition in [UTS35], *Section 3. Identifiers* . Note that the locale id of "root" or "und" indicates the root locale, such as in the CLDR root collation.

```
\T{<locale_id>}..\E
```

There must be some sort of syntax that will allow Level 3 support to be turned on and off, for two reasons. Level 3 support may be considerably slower than Level 2, and most regular expressions may require Level 1 or Level 2 matches to work properly. The syntax should also specify the particular locale or other tailoring customization that the pattern was designed for, because tailored regular expression patterns are usually quite specific to the locale, and will generally not work across different locales.

Sections 3.6 and following describe some additional capabilities of regular expression engines that are very useful in a Unicode environment, especially in dealing with the complexities of the large number of writing systems and languages expressible in Unicode.

3.1 Tailored Punctuation

The Unicode character properties for punctuation may vary from language to language or from country to country. In most cases, the effects of such changes will be apparent in other operations, such as a determination of word breaks. But there are other circumstances where the effects should be apparent in the general APIs, such as when testing whether a curly quotation mark is *opening* or *closing* punctuation.

***RL3.1* Tailored Punctuation**

> *To meet this requirement, an implementation shall allow for punctuation properties to be tailored according to locale, using the locale identifier definition in [UTS35], Section 3. Identifiers.*

As just described, there must be the capability of turning this support on or off.

3.2 Tailored Grapheme Clusters

***RL3.2* Tailored Grapheme Clusters**

> *To meet this requirement, an implementation shall provide for collation grapheme clusters matches based on a locale's collation order.*

Tailored grapheme clusters may be somewhat different than the extended grapheme clusters discussed in Level 2. They are coordinated with the collation ordering for a given language in the following way. A collation ordering determines a *collation grapheme cluster*, which is a sequence of characters that is treated as a unit by the ordering. For example, *ch* is a collation grapheme cluster for a traditional Spanish ordering.

The tailored grapheme clusters for a particular locale are the collation grapheme clusters for the collation ordering for that locale. The determination of tailored grapheme clusters requires the regular expression engine to either draw upon the platform's collation data, or incorporate its own tailored data for each supported locale.

For example, an implementation could interpret `\X{es-u-co-trad}` as matching a collation grapheme cluster for a traditional Spanish ordering, or use a switch to change the meaning of **\X** during some span of the regular expression.

See *Section 6.9, Handling Collation Graphemes* in UTS #10: Unicode Collation Algorithm [UTS10] for the definition of collation grapheme clusters, and *Annex B: Sample Collation Grapheme Cluster Code* for sample code.

3.3 Tailored Word Boundaries

***RL3.3* Tailored Word Boundaries**

> *To meet this requirement, an implementation shall allow for the ability to have word boundaries to be tailored according to locale.*

For example, an implementation could interpret `\b{x:…}` as matching the word break positions according to the locale information in CLDR [UTS35] (which are tailorings of word break positions in [UAX29]). Thus it could interpret

- **\b{w:und}** or **\b{w}** as matching a *root* word break
- **\b{w:ja}** as matching a Japanese word break
- **\b{l:ja}** as matching a Japanese line break

Alternatively, it could use a switch to change the meaning of **\b** and **\B** during some span of the regular expression.

Semantic analysis may be required for correct word boundary detection in languages that do not require spaces, such as Thai. This can require fairly sophisticated support if Level 3 word boundary detection is required, and usually requires drawing on platform OS services.

3.4 Tailored Loose Matches (Retracted)

RL3.4 Tailored Loose Matches (Retracted)

Previous versions of RL3.4 described loose matches based on collation order. However, 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 RL3.4 has been retracted.

3.5 Tailored Ranges (Retracted)

RL3.5 Tailored Ranges (Retracted)

Previous versions of RL3.5 described ranges based on collation order. However, tailored ranges can be quite difficult to implement properly, and can have very unexpected results in practice. For example, languages may also vary whether they consider lowercase below uppercase or the reverse. This can have some surprising results: `[a-Z]` may not match anything if *Z* < *a* in that locale. Thus RL3.5 has been retracted.

3.6 Context Matching

RL3.6 Context Matching

> *To meet this requirement, an implementation shall provide for a restrictive match against input text, allowing for context before and after the match.*

For parallel, filtered transformations, such as those involved in script transliteration, it is important to restrict the matching of a regular expression to a substring of a given string, and yet allow for context before and after the affected area. Here is a sample API that implements such functionality, where m is an extension of a Regex Matcher.

```
if (m.matches(text, contextStart, targetStart, targetLimit, contextLimit)) {
  int end = p.getMatchEnd();
}
```

The range of characters between `contextStart` and `targetStart` define a *precontext*; the characters between `targetStart` and `targetLimit` define a *target*, and the offsets between `targetLimit` and `contextLimit` define a *postcontext*. Thus `contextStart` ≤ `targetStart` ≤ `targetLimit` ≤ `contextLimit`. The meaning of this function is that:

- a match is attempted beginning at `targetStart`.
- the match will only succeed with an endpoint at or less than `targetLimit`.
- any zero-width look-arounds (look-aheads or look-behinds) can match characters inside or outside of the target, but cannot match characters outside of the context.

Examples:

In these examples, the text in the pre- and postcontext is italicized and the target is underlined. In the output column, the text in **bold** is the matched portion. The pattern syntax "(←x)" means a backwards match for *x* (without moving the cursor) This would be `(?<=x)` in Perl. The pattern "(→x)" means a forwards match for *x* (without moving the cursor). This would be `(?=x)` in Perl.

Pattern	Input	Output	Comment
/(←a) (bc)* (→d)/	1 *a*bcbc*d*2	1 *a***bcbc***d*2	matching with context
/(←a) (bc)* (→bcd)/	1 *a*bcbc*d*2	1 *a***bc**bc*d*2	stops early, because otherwise 'd' would not match.
/(bc)*d/	1 *a*bcbc*d*2	*no match*	'd' cannot be matched in the target, only in the postcontext
/(←a) (bc)* (→d)/	1abcbc*d*2	*no match*	'a' cannot be matched, because it is before the precontext (which is zero-length, in this case)

While it would be possible to simulate this API call with other regular expression calls, it would require subdividing the string and making multiple regular expression engine calls, significantly affecting performance.

There should also be pattern syntax for matches (like ^ and $) for the `contextStart` and `contextLimit` positions.

> Internally, this can be implemented by modifying the regular expression engine so that all matches are limited to characters between `contextStart` and `contextLimit`, and so that all matches that are not zero-width look-arounds are limited to the characters between `targetStart` and `targetLimit`.

3.7 Incremental Matches

RL3.7 Incremental Matches

> *To meet this requirement, an implementation shall provide for incremental matching.*

For buffered matching, one needs to be able to return whether there is a partial match; that is, whether there *would be* a match if additional characters were added after the `targetLimit`. This can be done with a separate method having an enumerated return value: *match*, *no_match*, or *partial_match*.

```
if (m.incrementalmatches(text, cs, ts, tl, cl) == Matcher.MATCH) {
  ...
}
```

Thus performing an incremental match of `/bcbce(→d)/` against "1abcbc*d*2" would return a *partial_match* because the addition of an *e* to the end of the target would allow it to match. Note that `/(bc)*(→d)/` would *also* return a partial match, because if *bc* were added at the end of the target, it would match.

Here is the above table, when an incremental match method is called:

Pattern	Input	Output	Comment

/(←a) (bc)* (→d)/	1*a*bcbc*d*2	*partial match*	'bc' could be inserted
/(←a) (bc)* (→bcd)/	1*a*bcbc*d*2	*partial match*	'bc' could be inserted
/(bc)*d/	1*a*bcbc*d*2	*partial match*	'd' could be inserted
/(←a) (bc)* (→d)/	1abcbc*d*2	*no match*	as with the matches function; the backwards search for 'a' fails

The typical usage of incremental matching is to make a series of incremental match calls, marching through a buffer with each successful match. At the end, if there is a partial match, one loads another buffer (or waits for other input). When the process terminates (no more buffers or input are available), then a regular match call is made.

Internally, incremental matching can be implemented in the regular expression engine by detecting whether the matching process ever fails when the current position is at or after `targetLimit`, and setting a flag if so. If the overall match fails, and this flag is set, then the return value is set to *partial_match*. Otherwise, either *match* or *no_match* is returned, as appropriate.

The return value *partial_match* indicates that there was a partial match: if further characters were added there could be a match to the resulting string. It may be useful to divide this return value into two, instead:

- *extendable_match*: in addition to there being a partial match, there was also a match somewhere in the string. For example, when matching /(ab)*/ against "aba", there is a match, *and* if other characters were added ("a", "aba",...) there could also be another match.
- *only_partial_match*: there was no other match in the string. For example, when matching /abcd/ against "abc", there is only a partial match; there would be no match unless additional characters were added.

3.8 Unicode Set Sharing (Retracted)

Previous versions described a technique to reduce memory consumption by sharing the underlying implementation data structures for character classes. Retracted because it assumed a very specific implementation environment and did not specify any Unicode related pattern or matching features.

~~For script transliteration and similar applications, there may be a hundreds of regular expressions, sharing a number of Unicode sets in common. These Unicode sets, such as `[\p{Alphabetic} -- \p{Latin}]`, could take a fair amount of memory, because they would typically be expanded into an internal memory representation that allows for fast lookup. If these sets are separately stored, this means an excessive memory burden.~~

~~To reduce the storage requirements, an API may allow regular expressions to share storage of these and other constructs, by having a 'pool' of data associated with a set of compiled regular expressions.~~

```
~~rules.registerSet("$lglow", "[\p{lowercase}&&[\p{latin}\p{greek}]] ");~~
~~rules.registerSet("$mark", "[\p{Mark}]");~~
~~...~~
~~rules.add("θ", "th");~~
~~rules.add("Θ(→$mark*$lglow)", "Th");~~
~~rules.add("Θ", "TH");~~
~~...~~
~~rules.add("φ", "ph");~~
```

```
~~rules.add("Φ(→$mark*$lglow)", "Ph");~~
~~rules.add("Φ", "Ph");~~
~~...~~
```

3.9 Possible Match Sets

RL3.9 Possible Match Sets

> *To meet this requirement, an implementation shall provide for the generation of possible match sets from any regular expression pattern.*

There are a number of circumstances where additional functions on regular expression patterns can be useful for performance or analysis of those patterns. These are functions that return information about the sets of characters that a regular expression can match.

When applying a list of regular expressions (with replacements) against a given piece of text, one can do that either serially or in parallel. With a serial application, each regular expression is applied the text, repeatedly from start to end. With parallel application, each position in the text is checked against the entire list, with the first match winning. After the replacement, the next position in the text is checked, and so on.

For such a parallel process to be efficient, one needs to be able to winnow out the regular expressions that simply could not match text starting with a given code point. For that, it is very useful to have a function on a regular expression pattern that returns a set of all the code points that the pattern would partially or fully match.

```
myFirstMatchingSet = pattern.getFirstMatchSet(Regex.POSSIBLE_FIRST_CODEPOINT);
```

For example, the pattern `/[[\u{0}-\u{FF}] && [:Latin:]] * [0-9]/` would return the set {0..9, A..Z, a..z}. Logically, this is the set of all code points that would be at least partial matches (if considered in isolation).

> **Note:** An additional useful function would be one that returned the set of all code points that could be matched at any point. Thus a code point outside of this set cannot be in any part of a matching range.

The second useful case is the set of all code points that could be matched in any particular group, that is, that could be set in the standard $0, $1, $2, ... variables.

```
myAllMatchingSet = pattern.getAllMatchSet(Regex.POSSIBLE_IN$0);
```

Internally, this can be implemented by analysing the regular expression (or parts of it) recursively to determine which characters match. For example, the first match set of an alternation *(a | b)* is the union of the first match sets of the terms *a* and *b*.

The set that is returned is only guaranteed to *include* all possible first characters; if an expression gets too complicated it could be a proper superset of all the possible characters.

3.10 Folded Matching (Retracted)

RL3.10 Folded Matching

Previous versions of RL3.10 described tailored folding. However, for most full-featured regular expression engines, it is quite difficult to match under folding equivalences that are not 1:1. For more discussion of this, see 1.5 Simple Loose Matches and 2.1 Canonical Equivalents. Thus RL3.10 has been retracted.

3.11 Submatchers

RL3.11 Submatchers

> *To meet this requirement, an implementation shall provide for general registration of matching functions for providing matching for general linguistic features.*

There are over 70 properties in the Unicode character database, yet there are many other sequences of characters that users may want to match, many of them specific to given languages. For example, characters that are used as vowels may vary by language. This goes beyond single-character properties, because certain sequences of characters may need to be matched; such sequences may not be easy themselves to express using regular expressions. Extending the regular expression syntax to provide for registration of arbitrary properties of characters allows these requirements to be handled.

The following provides an example of this. The actual function is just for illustration.

```
class MultipleMatcher implements RegExSubmatcher {
// from RegExFolder, must be overridden in subclasses
  /**
   * Returns -1 if there is no match; otherwise returns the endpoint;
   * an offset indicating how far the match got.
   * The endpoint is always between targetStart and targetLimit, inclusive.
   * Note that there may be zero-width matches.
   */
int match(String text, int contextStart, int targetStart, int targetLimit, int contextLimit) {
// code for matching numbers according to numeric value.
}

// from RegExFolder, may be overridden for efficiency
  /**
   * The parameter is a number. The match will match any numeric value that is a multiple.
   * Example: for "2.3", it will match "0002.3000", "4.6", "11.5", and any non-Western
   * script variants, like Indic numbers.
   */
RegExSubmatcher clone(String parameter, Locale locale) {...}
}
  ...

  RegExSubmatcher.registerMatcher("multiple", new MultipleMatcher());

  ...

  p = Pattern.compile("xxx\M{multiple=2.3}xxx");
```

In this example, the match function can be written to parse numbers according to the conventions of different locales, based on OS functions available for such parsing. If there are mechanisms for setting a locale for a portion of a regular expression, then that locale would be used; otherwise the default locale would be used.

> **Note:** It might be advantageous to make the Submatcher API identical to the Matcher API; that is, only have one base class "Matcher", and have user extensions derive from the base class. The base class itself can allow for nested matchers.

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 Systems	Blocks
Latin	Basic Latin, Latin-1 Supplement, Latin Extended-A, Latin Extended-B, Latin Extended C, Latin Extended D, Latin Extended Additional, Diacritics
Greek	Greek, Greek Extended, Diacritics
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 Compatibility Ideographs, CJK Compatibility Forms, Enclosed CJK Letters and Months, Small Form Variants, Bopomofo, Bopomofo Extended

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

The following provides sample code for doing Level 3 collation grapheme cluster detection. This code is meant to be illustrative, and has not been optimized. Although written in Java, it could be easily expressed in any programming language that allows access to the Unicode Collation Algorithm mappings.

```
                        /**
 * Return the end of a collation grapheme cluster.
 * @param s          the source string
 * @param start      the position in the string to search
 *                   forward from
 * @param collator   the collator used to produce collation elements.
 * This can either be a custom-built one, or produced from
 * the factory method Collator.getInstance(someLocale).
 * @return           the end position of the collation grapheme cluster
 */

static int getLocaleCharacterEnd(String s,
  int start, RuleBasedCollator collator) {
    int lastPosition = start;
    CollationElementIterator it
      = collator.getCollationElementIterator(
          s.substring(start, s.length()));
    it.next(); // discard first collation element
int primary;

// accumulate characters until we get to a non-zero primary

do {
        lastPosition = it.getOffset();
        int ce = it.next();
        if (ce == CollationElementIterator.NULLORDER) break;
        primary = CollationElementIterator.primaryOrder(ce);
    } while (primary == 0);
    return lastPosition;
}
```

Annex C: Compatibility Properties

The following are recommended assignments for compatibility property names, for use in Regular Expressions. There are two alternatives: the Standard Recommendation and the POSIX Compatible versions. Applications should use the former wherever possible. The latter is modified to meet the formal requirements of [POSIX], and also to maintain (as much as possible) compatibility with the POSIX usage in practice. That 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.

Property	Standard Recommendation	POSIX Compatible (where different)	Comments
alpha	\p{Alphabetic}		Alphabetic includes more than gc = Letter. Note that 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 DerivedCoreProperties in [UAX44]

			for Alphabetic, also DerivedGeneralCategory in [UAX44]. Alphabetic should *not* be used as an approximation for word boundaries: see word below.
lower	`\p{Lowercase}`		Lowercase includes more than gc = Lowercase_Letter (Ll). See DerivedCoreProperties in [UAX44].
upper	`\p{Uppercase}`		Uppercase includes more than gc = Uppercase_Letter (Lu).

punct	`\p{gc=Punctuation}`	`\p{gc=Punctuation}`
		`\p{gc=Symbol}`
		`-- \p{alpha}`
			POSIX adds symbols. Not recommended generally, due to the confusion of having *punct* include non-punctuation marks.

digit (\d)	`\p{gc=Decimal_Number}`	`[0..9]`	Non-decimal numbers (like Roman numerals) are normally excluded. In U4.0+, the recommended column is the same as gc = Decimal_Number (Nd). See DerivedNumericType in [UAX44].

xdigit	`\p{gc=Decimal_Number}`
	`\p{Hex_Digit}`
		`[0-9 A-F a-f]`	Hex_Digit contains 0-9 A-F, fullwidth and halfwidth, upper and lowercase.

alnum	`\p{alpha}`
	`\p{digit}`
			Simple combination of other properties

space \s	`\p{Whitespace}`		See PropList in [UAX44] for the definition of Whitespace.

blank	\p{gc=Space_Separator}
	\N{CHARACTER TABULATION}
			"horizontal" whitespace: space separators plus U+0009 *tab. Engines implementing older versions of the Unicode Standard may need to use the longer formulation:*
			\p{Whitespace} --
			[\N{LF} \N{VT} \N{FF} \N{CR} \N{NEL} \p{gc=Line_Separator} \p{gc=Paragraph_Separator}]

cntrl	\p{gc=Control}		The characters in \p{gc=Format} share some, but not all aspects of control characters. Many format characters are required in the representation of plain text.

graph	[^
	\p{space}
	\p{gc=Control}
	\p{gc=Surrogate}
	\p{gc=Unassigned}]
			Warning: the set to the left is defined by *excluding* space, controls, and so on with ^.

print	\p{graph}
	\p{blank}
	-- \p{cntrl}
			Includes graph and space-like characters.

word
(\w)
	\p{alpha}
	\p{gc=Mark}
	\p{digit}
	\p{gc=Connector_Punctuation}
	\p{Join_Control}
		n/a	This is only an approximation to Word Boundaries (see b below). The Connector Punctuation is added in for programming language identifiers, thus adding "_" and similar characters.

\X	Extended Grapheme Clusters	n/a	See [UAX29]. Other functions are used for programming language identifier boundaries.
\b	Default Word Boundaries	n/a	If there is a requirement that \b align with \w, then it would use the approximation above instead. See [UAX29]. Note that different functions are used for programming language identifier boundaries. See also [UAX31].

References

[Case] Section 3.13, *Default Case Algorithms* [Unicode]

[CaseData] http://www.unicode.org/Public/UNIDATA/CaseFolding.txt

[FAQ] Unicode Frequently Asked Questions
http://www.unicode.org/faq/
For answers to common questions on technical issues.

[Feedback] Reporting Form
http://www.unicode.org/reporting.html
For reporting errors and requesting information online.

[Friedl] Jeffrey Friedl, "Mastering Regular Expressions", 2nd Edition 2002, O'Reilly and Associates, ISBN 0-596-00289-0

[Glossary] Unicode Glossary
http://www.unicode.org/glossary/
For explanations of terminology used in this and other documents.

[Online] http://www.unicode.org/onlinedat/online.html

[Perl] http://perldoc.perl.org/
See especially:
http://perldoc.perl.org/charnames.html
http://perldoc.perl.org/perlre.html
http://perldoc.perl.org/perluniintro.html
http://perldoc.perl.org/perlunicode.html

[POSIX] The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition, "Locale" chapter
http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap07.html

[Prop] http://www.unicode.org/Public/UNIDATA/PropertyAliases.txt

[PropValue] http://www.unicode.org/Public/UNIDATA/PropertyValueAliases.txt

[Reports] Unicode Technical Reports
http://www.unicode.org/reports/
For information on the status and development process for technical reports, and for a list of technical reports.

[ScriptData] http://www.unicode.org/Public/UNIDATA/Scripts.txt

[SpecialCasing] http://www.unicode.org/Public/UNIDATA/SpecialCasing.txt

[UAX14] UAX #14: *Unicode Line Breaking Algorithm*
http://www.unicode.org/reports/tr14/

[UAX15] UAX #15: *Unicode Normalization Forms*
http://www.unicode.org/reports/tr15/

[UAX24] UAX #24: *Unicode Script Property*
http://www.unicode.org/reports/tr24/

[UAX29] UAX #29: *Unicode Text Segmentation*
http://www.unicode.org/reports/tr29/

[UAX31] UAX #31: *Unicode Identifier and Pattern Syntax*
http://www.unicode.org/reports/tr31/

[UAX44] UAX #44: *Unicode Character Database*
http://www.unicode.org/reports/tr44/

[UTS39] UTS #39: Unicode Security Mechanisms
http://www.unicode.org/reports/tr39/

[UTS46] Unicode IDNA Compatibility Processing
http://www.unicode.org/reports/tr46/

[UData] http://www.unicode.org/Public/UNIDATA/UnicodeData.txt

[Unicode] The Unicode Standard
For the latest version, see:
http://www.unicode.org/versions/latest/

[UTS10] UTS #10: *Unicode Collation Algorithm (UCA)*
http://www.unicode.org/reports/tr10/

[UTS35] UTS #35: *Unicode Locale Data Markup Language (LDML)*
http://www.unicode.org/reports/tr35/

[Versions] Versions of the Unicode Standard
http://www.unicode.org/versions/
For information on version numbering, and citing and referencing the Unicode Standard, the Unicode Character Database, and Unicode Technical Reports.

Acknowledgments

Mark Davis created the initial version of this annex and maintains the text, with significant contributions from Andy Heninger.

Thanks to Julie Allen, Tom Christiansen, Michael D'Errico, Asmus Freytag, Jeffrey Friedl, Norbert Lindenberg, Peter Linsley, Alan Liu, Kent Karlsson, Jarkko Hietaniemi, Gurusamy Sarathy, Xueming Shen, Henry Spencer, Kento Tamura, Philippe Verdy, Tom Watson, and Karl Williamson for their feedback on the document.

Modifications

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

Revision 18

- Section 1.2, fixed typographical errors.
- Section 1.2.2, revised description of Script Extension property.
- Section 1.2.3, Other Properties, added emoji properties and vertical orientation property.
- RL1.3, revised description of set operations to remove use of the term "Unicode Set".
- Section 2.7 Full Properties, added new property Prepended_Concatenation_Mark, and renamed STerm to Sentence_Term. Removed Decomposition_Mapping, Composition_Exclusion and Full_Composition_Exclusion.
- Retracted Section 3.8, Unicode Set Sharing.

Revision 17

Revision 16 being a Proposed Update, only changes between Revision 15 and 17 are listed here.

- Added new Unicode 6.3 properties in *Section 2.7 Full Properties* : Bidi_Paired_Bracket and Bidi_Paired_Bracket_Type
- Expanded the discussion of Script_Extensions in Section *1.2.2 Script Property*, and changed "Script" in RL1.2 Properties to "Script and Script_Extensions"
- Dropped two links to specific versions of Unicode in the references, and to two versioned files in Annex C.
- Minor edits.

Revision 15

Revision 14 being a Proposed Update, only changes between Revision 13 and 15 are listed here.

- Allowed case-folding to optionally close under character classes.
- Retracted clauses involving equivalences that were not 1:1.
- Added Name_Alias matching, and clarified the text, distinguishing \N{somename} from \p{name=somename}
- Fixed the table of "general category" names.
- Cited the Matching Rules from [UAX44].
- Added new properties in UAX #44 Table 7. Property Index by Scope of Use
- Added compact form of \u{...} for multiple characters. For example, using \u{1234 1235 4567} instead of \u{1234\u{1235}\u{4567}, and used in examples.
- Aligned RL1.4 with Annex C \w.
- Added the @ syntax for wildcards.
- Made changes from PRI **#179** Changes to Unicode Regular Expression Guidelines to the following.
    - RL2.4 Default Case Conversion
    - RL2.1 Canonical Equivalents
    - RL1.5 Simple Loose Matches
- Added new conformance clause at Level 2: RL2.7 Full Properties.
- Clarified syntax requirements in RL1.1 Hex Notation.
- Added note clarifying matching of isolated surrogates in RL1.7 Supplementary Code Points.
- Made it clear that the Unicode property definitions must be used to satisfy RL1.2 Properties and RL1.2a Compatibility Properties.
- Replaced use of [UCD] and [UCDDoc] by [UAX44].
- Added updates for the new Script Extensions [scx] property under RL1.2 Properties and RL2.7 Full Properties.
- Simplified the definition of \p{blank} in Annex C Compatibility Properties.
- Added clarification on use of the Age property.
- Changed "collation character" to "collation grapheme cluster" to match [UTS10] usage. Instances are not highlighted.
- Misc editing and clarifications.

Revision 13

Revision 12 being a Proposed Update, only changes between Revision 11 and 13 are listed here.

- Revised Section 2.2 handling of Extended Grapheme Clusters
- Added Section 2.2.1, Grapheme Cluster Mode
- Tailored Loose Matches, add example of matching Traditional and Simplified Chinese

characters

- Clearer discussion of the importance of levels, and features within level 2.
- Updated syntax
- Fixed precedence to be neutral, just noting the two main alternatives.
- Discussion of the use of hex notation to prevent unwanted normalization in literals
- Examples of normalization and casing properties
- Improved end-of-line treatment
- Revised treatment of (extended) grapheme clusters (U5.1), and the connection to normalization support. (Instances of changes from "default" to "extended" are not flagged.)
- Clearer description of the use of wildcards in property values
- Clarified conformance requirements for "." and CRLF.
- Pointed to LDML for the locale ID syntax
- Made the importance of the levels (and sublevels) clearer.
- Added ≠ in property expressions, ~~ for symmetric difference
- Changed operators to use doubled characters: --, &&, ||, ~~
- Added multiple property values. \p{gc=L|M|Nd} is equivalent to [\p{gc=L}\p{gc=M}\p{gc=Nd}]
- Fixed case where 'arbitrary character pattern' matches a newline sequence
- Added order of priority for level 2 items
- Described implementation of canonical equivalence through extended grapheme clusters
- Moved extended grapheme clusters (2.2) to level 3.
- Added named sequences, such as \N{KHMER CONSONANT SIGN COENG KA}
- Added some example links to Unicode utilities.

Revision 11

- Annex C:
  - Clarified first paragraph and removed review notes.
  - Changed *upper* definition in Annex C, because the UTC has changed the properties so that it will always be the case (from 4.1.0 onward) that Alphabetic ⊇ Uppercase and Alphabetic ⊇ Lowercase
  - Added \p{gc=Format} to graph, for better compatibility with POSIX usage.
- Added a caution about use of Tailored Ranges, and a note about the option of pre-normalization with newlines.
- Removed conformance clause for Unicode Set Sharing
- Misc Edits, including:
  - Added note on limit of 1-9 for \n
  - Fixed ^.*$ to ^$
  - Added parentheses to ([a-z ä] | (a \u{308}))

Revision 10

- R1.4, item 2 changed for ZW(N)J
- Added conformance clause to allow a claim of conformance to the Compatibility properties.
- Split the Compatibility properties into two, to allow for regular vs. strict POSIX properties.
- Added other notation for use here and in other Unicode Standards
- Added vertical tab to newline sequences. Reorganized text slightly to only list codepoints once.
- Minor Editing

Revision 9

- Split 2.5 into two sections, expanding latter.
- Misc. editing and clarifications.

Revision 8

- Renumbered sections to match levels
- Introduced "RL" numbering on clauses
- Misc. editing and clarifications.

Revision 7

- Now proposed as a UTS, adding Conformance and specific wording in each relevant section.
- Move hex notation for surrogates from 1.7 Surrogates into 1.1 Hex notation.
- Added 3.6 Context Matching and following.
- Updated to Unicode 4.0
- Minor editing
- **Note:** paragraphs with major changes are highlighted in this document; less substantive wording changes may not be.

Revision 6

- Fixed 16-bit reference, moved Supplementary characters support (surrogates) to level 1.
- Generally changed "locale-dependent" to "default", "locale-independent" to "tailored" and "grapheme" to "grapheme cluster"
- Changed syntax slightly to be more like Perl
- Added explicit table of General Category values
- Added clarifications about scripts and blocks
- Added descriptions of other properties, and a pointer to the default names
- Referred to TR 29 for grapheme cluster and word boundaries
- Removed old annex B (word boundary code)
- Removed spaces from anchors
- Added references, modification sections
- Rearranged property section
- Minor editing