Technical Reports

L2/08-211

Proposed Update Unicode Technical Standard #18

UNICODE REGULAR EXPRESSIONS

Version	12 (draft 3)
Authors	Mark Davis, Andy Heninger
Date	2008-05-01
This Version	http://www.unicode.org/reports/tr18/tr18-12.html
Previous Version	http://www.unicode.org/reports/tr18/tr18-11.html
Latest Version	http://www.unicode.org/reports/tr18/
Revision	12

Summary

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

Status

~~*This document has been reviewed by Unicode members and other interested parties, and has been approved by the Unicode Technical Committee as a **Unicode Technical Standard**. This is a stable document and may be used as reference material or cited as a normative reference by other specifications.*~~
*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time.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 [Syntax]. 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 **n**th 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 **n**th 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 Loose Matches
RL2.5 Name Properties
RL2.6 Wildcards in Property Values

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
RL3.4 Tailored Loose Matches
RL3.5 Tailored Ranges
RL3.6 Context Matching
RL3.7 Incremental Matches
RL3.9 Possible Match Sets
RL3.10 Folded Matching
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.

Notes:

- 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).

A sample notation for listing hex Unicode characters within strings is by prefixing four hex digits with "\u" and prefixing eight hex digits with "\U". This would provide for the following addition:

```
<codepoint> := <character>
<codepoint> := ESCAPE U_SHORT_MARK
               HEX_CHAR HEX_CHAR HEX_CHAR HEX_CHAR

<codepoint> := ESCAPE U_LONG_MARK
               HEX_CHAR HEX_CHAR HEX_CHAR HEX_CHAR
               HEX_CHAR HEX_CHAR HEX_CHAR HEX_CHAR

U_SHORT_MARK := "u"
U_LONG_MARK := "U"
```

Examples:

`[\u3040-\u309F \u30FC]`	Match Hiragana characters, plus prolonged sound sign
`[\u00B2 \u2082]`	Match superscript and subscript 2
`[a \U00010450]`	Match "a" or U+10450 SHAVIAN LETTER PEEP

- **Note**: instead of `[...\u3040...]`, an alternate syntax is `[...\x{3040}...]`, as in Perl 5.6 and later.
- **Note**: more advanced regular expression engines can also offer the ability to use the Unicode character name for readability. See 2.5 Name Properties.

1.1.1 Hex Notation and Normalization

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

1	u + grave	U+0075 (u) LATIN SMALL LETTER U + U+0300 (̀) COMBINING GRAVE ACCENT
2	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 [aeioù́̈] (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\u0300\u0301\u0308]` 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* [Norm].

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): for the list, see Unicode Character Database properties. The official data mapping Unicode characters (and code points) to properties is the Unicode Character Database [UCD]. See also 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. Here is an example of a syntax addition that permits properties.

> Notice that 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*
- *Alphabetic*
- *Uppercase*
- *Lowercase*
- *White_Space*
- *Noncharacter_Code_Point*
- *Default_Ignorable_Code_Point*
- *ANY, ASCII, ASSIGNED*

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.

Of the properties in RL1.2, only General Category and Script have multiple values; the rest 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}`.

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 UCD.html [UDataDoc].

Abb.	Long form
L	Letter
Lu	Uppercase Letter
Ll	Lowercase Letter
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

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

Abb.	Long form
Z	Separator
Zs	Space Separator
Zl	Line Separator
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 `[\u0000-\u10FFFF]`, but with Tailored Ranges off. 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 `[\u0000-\u007F]`, but with Tailored Ranges off.

Script Property

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 UTR #24: Script Names [ScriptDoc] (for the data file, see Scripts.txt [ScriptData]). Using a property such as `\p{Greek}` allows implementations to test whether letters are Greek or not.

Note, however, that the usage model for the script property normally requires that people construct somewhat more complex regular expressions, because a great many characters are shared between scripts. Documentation should point users to the description in UTR #24.

Other Properties

Other recommended properties are described in *Section 2* of *UAX #15: Unicode Normalization Forms* [Norm], in *Section 3.13* of *The Unicode Standard, Version 5.0* [Case], and in the documentation for the Unicode Character Database [UCD]. See also "Clarification of Lowercase and Uppercase" in [Unicode5.1].

The binary properties include:

- *Bidi_Control, Join_Control*
- *ASCII_Hex_Digit, Hex_Digit*
- *ID_Start, ID_Continue, XID_Start, XID_Continue*
- *isLowercase, isUppercase, isTitlecase, isCasefolded, isCased*
- *isNFC, isNFD, isNFKC, isNFKD*

The enumerated non-binary properties include:

- *Decomposition_Type*
- *Numeric_Type*
- *East_Asian_Width*
- *Line_Break*

The numeric properties include:

- *Numeric_Value*

The string properties include:

- *Name*
  See also 2.5 Name Properties and 2.6 Wildcards in Property Values.
- *toLowercase, toUppercase, toTitlecase, toCasefolded*
- *toNFC, toNFD, toNFKC, toNFKD*
- *Age*

Caution: The DerivedAge data file in the UCD provides the deltas between versions, for compactness. However, when using the property all characters included in that version are included. 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}]`

For example:

String properties	Description
[:toNFC=Å:]	The set of all characters X such that toNFC(X) = "a"
[:toNFD=A\u0300:]	The set of all characters X such that toNFD(X) = "A\u0300"
[:toNFKC=A:]	The set of all characters X such that toNFKC(X) = "A"
[:toNFKD=A\u0300:]	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"

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

A full list of the available UCD properties is on UCD Properties. Of those, the following are only useful in very restricted cases, such as in the internal implementation of normalization or case conversions:

Rarely Needed Properties
Composition_Exclusion
Decomposition_Mapping
Special_Case_Condition
FC_NFKC_Closure
ISO_Comment
NFC_Quick_Check NFKC_Quick_Check NFD_Quick_Check NFKD_Quick_Check
Expands_On_NFC Expands_On_NFD Expands_On_NFKC Expands_On_NFKD
all *Contributory Properties*.

Blocks

Unicode blocks can sometimes also be a useful enumerated 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.

`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 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.
`[\u0000-\u007F--\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 U+200C ZERO WIDTH NON-JOINER and U+200D ZERO WIDTH JOINER. 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.

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

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 of the Unicode Standard [Unicode]. The data file supporting caseless matching is CaseFolding.txt [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.txt [SpecialCasing] and UnicodeData.txt [UData].

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), PS (U+2029) and LS (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:

```
\u000A | \u000B | \u000C | \u000D | \u0085 | \u2028 | \u2029 | \u000D\u000A
```

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 `\u000D\u000A`, 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, e.g., in Perl \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 `\u000D\u000A`, 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 `\u000D\u000A` (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 `\u000D\u000A`, but should match the empty string within the reversed sequence `\u000A\u000D`.

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 (e.g. in #1). It would thus be shorthand for:

```
( \u000D\u000A | [\u000A\u000B\u000C\u000D\u0085\u2028\u2029] )
```

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 [LineBreak].

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 FFFF16. Surrogate pairs (or their equivalents in other encoding forms) are be handled internally as single code point values. In particular, `[\u0000-\U0010000]` 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

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 UTR #29: Text Boundaries [Boundaries].)

Level 2 support matches much more what user expectations are for sequences of Unicode characters. It is still locale-independent and easily implementable. However, ~~the implementation may be slower when supporting Level 2, and some expressions may require Level 1 matches. Thus it is often~~ 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

2.1 Canonical Equivalents

There are many instances where a character can be equivalently expressed by two different sequences of Unicode characters. For example, `[ä]` should match both "ä" and "a\u0308". (See UAX #15: Unicode Normalization [Norm] and *Sections 2.5 and 3.9* of *The Unicode Standard* for more information.)

RL2.1 Canonical Equivalents

To meet this requirement, an implementation shall provide a mechanism for ensuring that all canonically equivalent literal characters match.

One of the most effective ways to implement canonical equivalence is by having a special mode that only matches on extended grapheme cluster boundaries, since it avoids the reordering problems that can happen in normalization. See RL2.2 Extended Grapheme Clusters.

There are two other options for implementing this:

1. Before (or during) processing, translate text (and pattern) into a normalized form. This is the simplest to implement, because there are available code libraries for doing normalization.

2. Expand the regular expression internally into a more generalized regular expression that takes canonical equivalence into account. For example, the expression `[a-z ä]` can be internally turned into `([a-z ä] | (a \u0308))`. While this can be faster, it may also be substantially more difficult to generate expressions capturing all of the possible equivalent sequences.

It may be useful to distinguish a regular-expression engine from the larger software package which uses it. For example, the requirements of this section can be met by requiring the package to normalize text before supplying it to the regular expression engine. However, where the regular expression engine returns offsets into the text, the package may need to map those back to what the offsets would be in the original, unnormalized text.

Note: Combining characters are required for many languages. Even when text is in Normalization Form C, there may be combining characters in the text.

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 UTR #29: Text Boundaries [Boundaries]. These *extended* grapheme clusters are not the same as *tailored* grapheme clusters, which are covered in Level 3, Tailored Grapheme Clusters.

RL3.12 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 "\h" as a zero-width match against any extended grapheme cluster boundary, and "\H" as the negation of that.

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\u0323}]`	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 like `(?> ch | rr | [am]){1}+` as interpreted in Java-like regex engines (the {1}+ syntax is not in Perl). 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 `/(?> ch | rr | [a-m])heese/` will match "chheese" but not "cheese"; that is, the *c* in `[a-m]` will not match if the "ch" has already been matched.

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-z \q{ch} \q{rr}]` should behave in the following way:

Mode	Behavior	Description

normal	`(?! ch | rr | [a-m] ) [\x{0}-\x{10FFFF}]`	failing with strings starting with a-z, ch, or rr, and otherwise advancing by one code point
grapheme cluster	`(?! ch | rr | [a-m] ) \x`	failing with strings starting with a-z, 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.

2.3 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 UTR #29: Text Boundaries [Boundaries].

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 [LineBreak] 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 Loose Matches

RL2.4 Default Loose Matches

To meet this requirement:

- *if an implementation provides for case-insensitive matching, then it shall provide at least the full, default Unicode case-insensitive matching.*
- *if an implementation provides for case conversions, then it shall provide at least the full, default Unicode case conversion.*

At Level 1, caseless matches do not need to handle cases where one character matches against two. Level 2 includes caseless matches where one character may match against two (or more) characters. For example, 00DF "ß" will match against the two characters "SS".

To correctly implement a caseless match and case conversions, see UAX #21: Case Mappings [Case]. For ease of implementation, a complete case folding file is supplied at CaseFolding.txt [CaseData].

If the implementation containing the regular expression engine also offers case conversions, then these should also be done in accordance with UAX #21, with the full mappings. The relevant data files are SpecialCasing.txt [SpecialCasing] and UnicodeData.txt [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 main data is supplied in the Name property in the UCD, as described in [UDataDoc], or computed as in the case of CJK Ideographs or Hangul Syllables. Certain code points are not assigned names in the standard. These should be given names based on the General_Category:

Control:	The Unicode 1.0 name field (ISO control names).
Private Use:	<no name>
Unassigned:	

The ISO names for the control characters may be unfamiliar, especially because many people are not familiar with changes in the formal ISO names to make them more language neutral, so it is recommended that they be supplemented with other aliases. For example, for U+0009 the implementation could accept the official name CHARACTER TABULATION, and also the aliases HORIZONTAL TABULATION, HT, and TAB.

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> "}"
```

This is equivalent to using the property *name*, as in `\p{name=WHITE SMILING FACE}`. The only distinction between them is that \N should cause a syntax error if it fails to match a character. This may be extended to match named character sequences, such as \N{KHMER CONSONANT SIGN COENG KA}. Note that because this is a sequence it behaves as a single element, so \N{KHMER CONSONANT SIGN COENG KA}* should be treated as if it were the expression (\x{17D2|\x{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 `\u263A`
- `\N{GREEK SMALL LETTER ALPHA}` is equivalent to `\u03B1`
- `\N{FORM FEED}` is equivalent to `\u000C`
- `\N{SHAVIAN LETTER PEEP}` is equivalent to `\U00010450`
- `[\N{GREEK SMALL LETTER ALPHA}-\N{GREEK SMALL LETTER BETA}]` is equivalent to `[\u03B1-\u03B2]`

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> "/"
```

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

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

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. If you click on them, they 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

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 like 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 [LDML], *Section 3. Identifiers*.

```
\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 [LDML], *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 character for a traditional Spanish ordering. More specifically, a collation character is the longest sequence of characters that maps to a sequence of one or more collation elements where the first collation element has a primary weight and subsequent elements do not, and no completely ignorable characters are included.

The tailored grapheme clusters for a particular locale are the collation characters 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.

See UTS #10: Unicode Collation Algorithm [Collation] for more information about collation, and Annex B. Sample Collation Character 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.

Semantic analysis may be required for correct word boundary detection in languages that do not require spaces, such as Thai, Japanese, Chinese or Korean. 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

RL3.4 Tailored Loose Matches

To meet this requirement, an implementation shall provide for loose matches based on a locale's collation order, with at least 3 levels.

In Level 1 and 2, caseless matches are described, but there are other interesting linguistic features that users may want to match. For example, *V* and *W* are considered equivalent in Swedish collations, and so `[v]` should match *W* in Swedish. In line with the UTS #10: Unicode Collation Algorithm [Collation], the following four levels of equivalences are recommended:

- exact match: bit-for-bit identity
- tertiary match: disregard 4th level differences (language tailorings)
- secondary match: disregard 3rd level differences such as upper/lowercase and compatibility variation (that is, matching both half-width and full-width katakana).
- primary match: disregard accents, case and compatibility variation; also disregard differences between katakana and hiragana.

If users are to have control over these equivalence classes, here is an example of how the sample syntax could be modified to account for this. The syntax for switching the strength or type of matching varies widely. Note that these tags switch behavior on and off in the middle of a regular expression; they do not match a character.

```
ITEM := \v{PRIMARY}   // match primary only
ITEM := \v{SECONDARY} // match primary and secondary only
ITEM := \v{TERTIARY}  // match primary, secondary, and tertiary
ITEM := \v{EXACT}     // match all levels, normal state
```

Examples:

`[\v{SECONDARY}a-m]`	Match a-m, plus case variants A-M, plus compatibility variants

Basic information for these equivalence classes can be derived from the data tables referenced by UTS #10: Unicode Collation Algorithm [Collation].

3.5 Tailored Ranges

RL3.5 Tailored Ranges

To meet this requirement, an implementation shall provide for ranges based on a locale's collation order.

Tailored character ranges will include tailored grapheme clusters, as discussed above. This broadens the set of grapheme clusters — in traditional Spanish, for example, `[b-d]` would match against "`c`" and against "`ch`". That is because in that collation, "`ch`" sorts as a single letter between "`c`" and "`d`".

> **Note**: This is another reason why a property for all characters `\p{Any}` is needed—it is possible for a locale's collation to not have `[\u0000-\U0010FFFF]` encompass all characters.

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 implementers should be cautious about implementing this feature.

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' ca not be matched in the target, only in the postcontext
/(←a) (bc)* (→d)/	1abcbc*d*2	*no match*	'a' ca not 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)/	1 abcbc*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

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", "th");
rules.add("Ψ(→$mark*$lglow)", "Ps");
rules.add("Ψ", "PS");
...
```

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 `/[[\u0000-\u00FF] && [: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

RL3.10 Folded Matching

To meet this requirement, an implementation shall provide for registration of folding functions for providing insensitive matching for linguistic features other than case.

Regular expressions typically provide for case-sensitive or case-insensitive matching. This accounts for the fact that in English and many other languages, users quite often want to disregard the differences between characters that are solely due to case. It would be quite awkward to do this manually: for example, to do a caseless match against the last name in `/Mark\sDavis/`, one would have to use the pattern `/Mark\s[Dd][Aa][Vv][Ii][Ss]/`, instead of some syntax that can indicate the target text is to be matched after folding case, such as `/Mark\s\CDavis\E/`.

For many languages and writing systems, there are other differences besides case where users want to allow a loose match. Once such way to do this was discussed earlier, in the discussion of matching according to collation strength. There are others: for example, for Ethiopic one may need to match characters independent of their inherent vowel, or match certain types of vowels. It is difficult to tell exactly which ways users might want to match text for different languages, so the most flexible way to provide such support is by giving a general mechanism for overriding the way that regular expressions match literals.

One way to do this is to use *folding* functions. These are functions that map strings to strings, and are idempotent (applying a function more than once produces the same result: *f(f(x)) = f(x)*. There are two parts to this: (a) allow folding functions to be registered, and (b) extend patterns so that registered folding functions can be activated. During the span of text in which a folding function is activated, both the pattern literals and the input text will be processed according to the folding before comparing. For example:

```
// Folds katakana and hiragana together
class KanaFolder implements RegExFolder {
// from RegExFolder, must be overridden in subclasses
String fold(String source) {...}

// from RegExFolder, may be overridden for efficiency
RegExFolder clone(String parameter, Locale locale) {...}
  int fold(int source) {...}
  UnicodeSet fold(UnicodeSet source) {...}
}
  ...

  RegExFolder.registerFolding("k_h", new KanaFolder());

  ...

  p = Pattern.compile("(\F{k_h=argument}マルク (\s)* ダ (ギ | ビ) ス \E : \s+)*");
```

In the above example, the Kana folding is in force until terminated with \E. Within the scope of the folding, all text in the target would be folded before matching (the literal text in the pattern would also be folded). This only affects literals; regular expression syntax such as '(' or '*' are unaffected.

While it is sufficient to provide a folding function for strings, for efficiency one can also provide functions for folding single code points and Unicode sets (such as `[a-z...]`). For more information, see [Folding].

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, `[\u0370-\u03FF]` 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 Additional, Diacritics
Greek	Greek, Greek Extended, Diacritics
Arabic	Arabic, Arabic Presentation Forms-A, Arabic Presentation Forms-B
Korean	Hangul Jamo, 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 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 UTR #24: Script Names [ScriptDoc].

Annex B: Sample Collation Character Code

The following provides sample code for doing Level 3 collation character 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 character.
 * @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 character
 */

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

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

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

blank	`\p{Whitespace} --`
	`[\N{LF} \N{VT} \N{FF} \N{CR}`
	`\N{NEL}`
	`\p{gc=Line_Separator}`
	`\p{gc=Paragraph_Separator}]`
		`\p{Whitespace} --`
		`[\N{LF} \N{VT} \N{FF}`
		`\N{CR} \N{NEL}`
		`\p{gc=Line_Separator}`
		`\p{gc=Paragraph_Separator}]`
			"horizontal" whitespace.

cntrl	`\p{gc=Control}`	`\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}]`
		`[^`
		`\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 ^.
			
			~~Perl 5.8.0 is similar except that it excludes: Z, Cc, Cf, Cs, Cn. The intent is for Perl 5.8.1 to align with the specification here.~~

print	`\p{graph}`
	`\p{blank}`
	`-- \p{cntrl}`
		`\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}`
		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 [Boundaries], also GraphemeClusterBreakTest. 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 [Boundaries], also WordBreakTest. Note that different functions are used for programming language identifier boundaries. See also [Syntax].

References

[Boundaries]	Unicode Standard Annex #29: Text Boundaries http://www.unicode.org/reports/tr29/
[Case]	Section 3.13 [Unicode] http://www.unicode.org/versions/Unicode4.0.0/ch03.pdf#G33992
[CaseData]	http://www.unicode.org/Public/UNIDATA/CaseFolding.txt
[Collation]	Unicode Technical Standard #10: Unicode Collation Algorithm http://www.unicode.org/reports/tr10/
[FAQ]	Unicode Frequently Asked Questions http://www.unicode.org/faq/ *For answers to common questions on technical issues.*
[Feedback]	Reporting Errors and Requesting Information Online http://www.unicode.org/reporting.html
[Folding]	UTR #30 Character Foldings *(Current Status: Draft)* *http://www.unicode.org/reports/tr30/*
[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.*
[LineBreak]	Unicode Standard Annex #14: Line Breaking Properties http://www.unicode.org/reports/tr14/
[Norm]	Unicode Standard Annex #15: Unicode Normalization http://www.unicode.org/reports/tr15/
[Online]	http://www.unicode.org/onlinedat/online.html
[Perl]	http://www.perl.com/pub/q/documentation See especially: http://www.perldoc.com/perl5.8.0/lib/charnames.html http://www.perldoc.com/perl5.8.0/pod/perlre.html http://www.perldoc.com/perl5.8.0/pod/perluniintro.html http://www.perldoc.com/perl5.8.0/pod/perlunicode.html
[POSIX]	The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2003 Edition, "Locale" chapter http://www.opengroup.org/onlinepubs/007904975/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
[ScriptDoc]	Unicode Technical Report #24: Script Names http://www.unicode.org/reports/tr24/
[SpecialCasing]	http://www.unicode.org/Public/UNIDATA/SpecialCasing.txt
[Syntax]	Unicode Standard Annex #31: Identifier and Pattern Syntax *http://www.unicode.org/reports/tr31/*

Acknowledgments

Thanks to Jeffrey Friedl, Andy Heninger, Peter Linsley, Alan Liu, Kent Karlsson, Jarkko Hietaniemi, Gurusamy Sarathy, Henry Spencer, Tom Watson, and Kento Tamura for their feedback on the document.

Modifications

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

12 Draft 3

- 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

Draft 2

- 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.

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 \u0308))

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

9
- Split 2.5 into two sections, expanding latter.
- Misc. editing and clarifications.

8
- Renumbered sections to match levels
- Introduced "RL" numbering on clauses
- Misc. editing and clarifications.

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.

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