Technical Reports

Proposed Update Unicode Technical Standard #18

UNICODE REGULAR EXPRESSIONS

Version	8 (DRAFT)
Authors	Mark Davis (mark.davis@us.ibm.com)
Date	2003-05-06
This Version	http://www.unicode.org/reports/tr18/tr18-8.html
Previous Version	http://www.unicode.org/reports/tr18/tr18-7.html
Latest Version	http://www.unicode.org/reports/tr18
Base Unicode Version	Unicode 3.2
Tracking Number	7

Summary

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

Status

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 This is a draft document which may be updated, replaced, or superseded by other documents at any time.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

A Unicode Technical Standard (UTS) is an independent specification. Conformance to the Unicode Standard does not imply conformance to any UTS. Each UTS specifies a base version of the Unicode Standard. Conformance to the UTS requires conformance to that version or higher.

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

Contents

1 Introduction

The following describes general guidelines for extending regular expression engines 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, or UTF-32.) 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 is 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 default grapheme clusters (what the end-user generally thinks of as a character), better word-break, 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, whereby 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 what order the characters are in. However, there is a performance impact to support at this level.

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, say, currency symbols will have 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 can be stored in memory in a two-stage table with only 7 or 8Kbytes. Accessing those properties only takes a small amount of bit-twiddling and two array accesses.

1.1 Notation

In order to describe regular expression syntax, we will use an extended BNF form:

`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. In general, the syntax here is similar to that of Perl Regular Expressions [Perl].)

```
LIST := "[" NEGATION? ITEM (SEP? ITEM)* "]"
ITEM := CODE_POINT
      := <character> "-" CODE_POINT // range
      := ESCAPE 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 occurances of " "* are omitted.

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, 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".

1.2 Conformance

The following describes the possible ways that an implementation can claim conformance to this technical standard. In the sections indicated below, only statements using the term "must" are required for conformance, and they are only required for those implementations claiming conformance to that particular section. Other forms of statements are informative or may indicate recommendations, but are not required for conformance.

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 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:

- Hex notation
- Properties
- Subtraction and Intersection
- Simple Word Boundaries
- Simple Loose Matches
- End Of Line
- Surrogates

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

- Canonical Equivalents
- Default Grapheme Clusters
- Default Words
- Default Loose Matches
- Extended 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:

- Tailored Grapheme Clusters
- Tailored Loose Matches
- Tailored Ranges
- Context Matching
- Incremental Matches
- Unicode Set Sharing
- Possible Match Sets
- Folded Matching
- 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 3.1 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.

2 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]`. Since there are a very large number of characters in the Unicode standard, simple list expressions do not suffice.

2.1 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.

To meet this requirement, an implementation must supply some 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
`[\U00010450]`	Match 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 3.5 Extended Properties.

2.2 Properties

Since 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 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 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 don't 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 the *p* is 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> (":" | "=")
<unicode_property_value>
PROP_SPEC := <script_or_category_property_value>
```

Examples:

`[\p{L} \p{Nd}]`	Match all letters and decimal digits
`[\p{letter} \p{decimal number}]`	
`\P{L}`	Match anything that is not a letter
`\P{letter}`	
`\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}.

To meet this requirement, an implementation must provide a 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

Of these, only General Category and Script are not 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:** Since 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 people test letters for whether they are Greek or not.

Other Properties

Other useful properties are described in the documentation for the Unicode Character Database, cited above. The binary properties include:

- *Bidi_Control, Join_Control*
- *ASCII_Hex_Digit, Hex_Digit*
- *ID_Start, ID_Continue, XID_Start, XID_Continue*
- *NF*_NO, NF*_MAYBE*

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*
- *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 include. 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 2.3 Subtraction and Intersection. E.g. [\p{age=3.1} - \p{age=3.0]

A full list of the available properties is on UCD Properties. See also 3.5 Extended 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, such as in \p{Greek Block} or \p{Block=Greek}.

2.3 Subtraction and Intersection

As discussed above, with a large character set character properties are essential. 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 letters but *Q* and *W* without having to list every character in `\p{letter}` that is neither *Q* nor *W*.

To meet this requirement, an implementation must supply mechanisms for both intersection and set-difference of Unicode sets.

```
ITEM := "[" ITEM "]" // for grouping
SEP  := " "          // no separator = union
     := "|"          // union
     := "&"          // intersection
     := "-"          // removal = set difference
```

Implementations may also choose to offer other set operations, such as symmetric difference: [\p{letter} ⊖ \p{ascii}]

Note: In the sample syntax used here,

1. The symbol "-" between two characters still means a range, not a set-difference. That is:
   - [\p{ascii} - aeiouy] is equivalent to [\p{ascii} - [aeiouy]]
   - [aeiouy & \p{ascii}] is equivalent to [[aeiouy] & \p{ascii}]
2. Union binds more closely than intersection, which binds more closely than removal, so [A|B|C-D|E] is the same as [[A|B|C] - [D|E]]. Otherwise items bind from the left, so [A - B - C - D & E] is the same as [[[A - B] - C] - [D & E]]. However, such binding or precedence may vary by regular expression engine.

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-f A-F - - ]`	Match all assigned characters except for hex digits (using a broad definition).

2.4 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: a word boundary is between any pair of characters where one is a `<word_character>` and the other is not. This is not adequate for Unicode regular expressions.

To meet this requirement, an implementation must 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]. See also Annex C: Compatibility Properties.
2. Non-spacing 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.

2.5 Simple Loose Matches

To meet this requirement, if an implementation provides for case-insensitive matching, then it must provide at least the simple, default Unicode case-insensitive matching.

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

The only loose matches that most regular expression engines offer is caseless matching. If the engine does offers this, then it needs to account for the large range of cased Unicode characters outside of ASCII. In addition, because of the vagaries of natural language, there are situations where two different Unicode characters have the same uppercase or lowercase. Level 1 implementations need to handle these cases. 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].

If the implementation containing the regular expression engine also offers case conversions, then these are also to follow Chapter 3 of the Unicode Standard [Unicode].. The relevant data files are SpecialCasing.txt [SpecialCasing] and UnicodeData.txt [UData].

2.6 End Of Line

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.

To meet this requirement, if an implementation provides for line-boundary testing, then it must 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 The Unicode Standard [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.

1. **Logical line number**
   - The line number is increased by one for each occurrence of:
     `\u2028 | \u2029 | \u000D\u000A | \u000A | \u000C | \u000D | \u0085`
2. **Logical beginning of line (often "^")**
   - SOL is at the start of a file or string, and also immediately following any occurrence of:
     `\u2028 | \u2029 | \u000D\u000A | \u000A | \u000C | \u000D | \u0085`
   - Note that there is no empty line within the sequence `\u000D\u000A`.
3. **Logical end of line (often "$")**
   - EOL at the end of a file or string, and also immediately preceding any occurrence of:
     `\u2028 | \u2029 | \u000D\u000A | \u000A | \u000C | \u000D | \u0085`
   - Note that there is no empty line within the sequence `\u000D\u000A`.
4. **Arbitrary character pattern (often ".")**
   - should *not* match any of
     `\u2028 | \u2029 | \u000A | \u000C | \u000D | \u0085`
   - In "multiline mode", these *would* match, and `\u000D\u000A` matches as if it were a

single character.

- 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 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 | [\u2028\u2029\u000A\u000C\u000D\u0085]).

2.7 Surrogates

To meet this requirement, an implementation must treat surrogate pairs as single code points.

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, [\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

3 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, default grapheme cluster boundaries, and loose matches. (For more information about boundary conditions, see *The Unicode Standard, Section 5-15.*)

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 useful to have some sort of syntax that will turn Level 2 support on and off.

3.1 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.)

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

There are two main options for implementing this:

1. Before (or during) processing, translate text (and pattern) into a normalized form. This is the simplest to implement, since 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.

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

3.2 Default 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.

Note: default grapheme clusters were previously referred to as "locale-independent graphemes". The term *cluster* has been added to emphasize that the term *grapheme* as used differently in linguistics. For simplicity and to align with UTS #10: Unicode Collation Algorithm [Collation], the terms "locale-independent" and "locale-dependent" been also changed to "default" and "tailored" respectively.

Essentially, the default grapheme clusters do only two things: they keep Hangul syllables together, and they don't break before non-spacing marks.

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

To meet this requirement, an implementation must provide a mechanism for matching against an arbitrary default grapheme cluster, a literal cluster, and matching default grapheme cluster boundaries.

For example, an implementation could interpret "\X" as matching any default grapheme cluster, while interpreting "." as matching any single code point. It could interpret "\h" as a zero-width match against any grapheme cluster boundary, and "\H" as the negation of that.

Regular expression engines should also provide some mechanism for easily matching against literal clusters, since 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. This syntax can also be used for tailored grapheme clusters (Tailored Grapheme Clusters).

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

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 lowercase characters in traditional Spanish.

These can be expressed syntactically by breaking them into combinations of code point sets and other constructs: `[a-z ñ \q{ch} \q{ll} \q{rr}]` is equivalent to `([a-z ñ] | ch | ll | rr)`, but that is not as convenient when other options are added, as in Level 3.

A typical implementation of the inverse of a set containing literal clusters simply removes those strings, thus `[^a-z ñ \q{ch} \q{ll} \q{rr}]` is equivalent to `[^a-z ñ]`. Without literal clusters, intersection and set-difference can be expressed simply as a combination of the other with inverse. However, this is not the case if the implementation of inverse simply removes the strings. Thus:

- `[\p{letter} - \p{ascii}]` is equivalent to `[\p{letter} & [^\p{ascii}]]`
- `[\p{letter} & \p{ascii}]` is equivalent to `[\p{letter} - [^\p{ascii}]]`
- `[[a\q{rr}] - [a\q{rr}]]` is *not* equivalent to `[[a\q{rr}] & [^a\q{rr}]]`

3.3 Default Words

To meet this requirement, an implementation must provide a mechanism for matching 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-break boundaries and line-break boundaries 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, 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 have spaces, requires information that is beyond the bounds of what a Level 2 algorithm can provide.

3.4 Default Loose Matches

To meet this requirement, if an implementation provides for case-insensitive matching, then it must provide at least the full, default Unicode case-insensitive matching.

To meet this requirement, if an implementation provides for case conversions, then it must 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].

3.5 Extended Properties

To meet this requirement, an implementation must support individually named characters and

the Unicode name property with wildcards.

Individually Named Characters

This facility provides syntax for specifying a code point by supplying the precise name, such as the following. For control characters (marked with "<control>" in the Unicode Character Database), the Unicode 1.0 name can be used. This syntax specifies a single code point, which can thus be used in ranges.

```
<codepoint> := "\N{" <character name "}"
```

Examples:

- `\N{WHITE SMILING FACE}` 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]`

As with other property values, names should use a loose match, disregarding spaces and "-" (the character "_" cannot occur in Unicode character names). The match can even be slightly looser than with property aliases: one can also remove all instances of the letter sequences "LETTER", "CHARACTER", "DIGIT", and still not have collisions. 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 three instances that require special-casing with loose matching:

- 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

Unicode Name Property

The Unicode Name Property with wildcards is slightly different. Instead of a single code point, it represents a set, and thus cannot be used in character ranges. It must support wild-cards in the name field; general regular expressions are recommended but optional.

Examples:

`{name=LATIN LETTER.*P}` is equivalent to:

```
[\u01AA\u0294\u0296\u1D18]
```

```
01AA          # LATIN LETTER REVERSED ESH LOOP
0294          # LATIN LETTER GLOTTAL STOP
0296          # LATIN LETTER INVERTED GLOTTAL STOP
1D18          # LATIN LETTER SMALL CAPITAL P
```

`{name=.*VARIA(TION|NT).*}` is equivalent to:

```
[\u180B-\u180D\u299C\u303E\uFE00-\uFE0F\U000E0100-\U000E01EF]
180B..180D    # MONGOLIAN FREE VARIATION SELECTOR ONE..----THREE
299C          # RIGHT ANGLE VARIANT WITH SQUARE
303E          # IDEOGRAPHIC VARIATION INDICATOR
FE00..FE0F    # VARIATION SELECTOR-1..VARIATION SELECTOR-16
E0100..E01EF  # ?) VARIATION SELECTOR-17..VARIATION SELECTOR-256
```

String Functions

The there are certain functions defined in the Unicode Standard as operating on strings, with the result either being another string or a boolean value.

- The functions isNFC, toNFC, ... are defined in the Notation section of [Norm].
- The functions toUppercase, toLowercase, toCaseFold,..., isUppercase,... are defined in Section 3.13 of Unicode 4.0 [Unicode].

Corresponding to each of these is a Unicode set property that is a set of strings (not just characters). Thus for example, \p{isNFC=10Å} is the set of all strings whose NFC form is equal to "10Å", and \p{isLowercase=fi} is the set of all strings whose lowercase is "fi".

It is not a requirement to support these properties at this level, but they can be a useful addition.

---

4 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 a 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 indicate tailoring to a locale within a certain range. (Marking locales is generally specified by means of the common ISO 639 and 3166 tags, such as "en_US". For more information on these tags, see the online data in [Online].)

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

Level 3 support may be considerably slower than Level 2, and some scripts may require either Level 1 or Level 2 matches instead. Thus it is usually required to have some sort of syntax that will turn Level 3 support on and off. Because tailored regular expression patterns are usually quite specific to the locale, and will generally not work across different locales, the syntax should also specify the particular locale or other tailoring customization that the pattern was designed for.

Sections 4.6 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.

4.1. Tailored Properties

Some of Unicode character properties, such as punctuation, may in a few cases vary from language to language or from country to country. For example, whether a curly quotation mark is *opening* or *closing* punctuation may vary. For those cases, the mapping of the properties to sets of characters will need to be dependent on the locale or other tailoring.

4.2 Tailored Grapheme Clusters

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

Tailored grapheme clusters may be somewhat different than the default 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.

4.3 Tailored Words

Semantic analysis may be required for correct word-break in languages that don't 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.

4.4 Tailored Loose Matches

To meet this requirement, an implementation must 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], at 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 (e.g. 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 & secondary only
ITEM := \v{TERTIARY}  // match primary, secondary, 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].

4.5. Tailored Ranges

To meet this requirement, an implementation must 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 "`ch`".

> **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.

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!

4.6 Context Matching

[Note to Reviewers: this and the following sections are provisional, and need careful review. They may be significantly revised or even omitted depending the results of discussions in the UTC.]

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

For parallel, filtered transformations, such as 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***bcbc***d*2	stops early, since otherwise 'd' wouldn't match.
/(bc)*d/	1*a*bcbc*d*2	*no match*	'd' can't be matched in the target, only in the postcontext
/(←a) (bc)* (→d)/	1abcbc*d*2	*no match*	'a' can't be matched, since it is before the precontext (which is zero-length, here)

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 that all matches that are not zero-width look-arounds are limited to the characters between `targetStart` and `targetLimit`.

4.7 Incremental Matches

To meet this requirement, an implementation must 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 match 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/input 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.

4.8 Unicode Set Sharing

To meet this requirement, an implementation must provide for shared storage of Unicode sets.

For script transliteration, 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, since they would typically be expanded into an internal memory representation that allows for fast lookup. If these sets 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");
...
```

4.9 Possible Match Sets

To meet this requirement, an implementation must 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).

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, etc. 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.

4.10 Folded Matching

To meet this requirement, an implementation must 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 without 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 that 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 is given above 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 people might want to match text for different languages, so the most flexible way to provide such support is to provide 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}     (\s)*    (   |   )    \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 (e.g. [a-z...]).

4.11 Submatchers

To meet this requirement, an implementation must 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 people 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, since 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 doing 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 since 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, etc.
- **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, etc.
- **Characters from a single writing system may be split across multiple blocks.** See the table below. Moreover, presentation forms for a number of different scripts may be collected in blocks like Alphabetic Presentation Forms or Halfwidth and Fullwidth Forms.

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 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
Diacritics	Combining Diacritical Marks, Combining Marks for Symbols, Combining Half Marks
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
others	IPA Extensions, Spacing Modifier Letters, Cyrillic, Armenian, Hebrew, Syriac, Thaana, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Sinhala, Thai, Lao, Tibetan, Myanmar, Georgian, Ethiopic, Cherokee, Unified Canadian Aboriginal Syllabics, Ogham, Runic, Khmer, Mongolian, CJK Radicals Supplement, Kangxi Radicals, Ideographic Description Characters, CJK Symbols and Punctuation, Hiragana, Katakana, Kanbun, Alphabetic Presentation Forms, Halfwidth and Fullwidth Forms, General Punctuation, Superscripts and Subscripts, Currency Symbols, Letterlike Symbols, Number Forms, Arrows, Mathematical Operators, Miscellaneous Technical, Control Pictures, Optical Character Recognition, Enclosed Alphanumerics, Box Drawing, Block Elements, Geometric Shapes, Miscellaneous Symbols, Dingbats, Braille Patterns, High Surrogates, High Private Use Surrogates, Low Surrogates, Private Use, Specials

For that reason, Script values are generally preferred to Block values.

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. These are informative recommendations only.

Property	Recommended	Comments
punct	\p{gc=P}	For a better match to the POSIX locale, add \p{gc=S}. Not recommended generally, due to the confusion of having punct include non-punctuation marks.
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 principle use is on alphabetics. See DerivedCoreProperties for Alphabetic, also DerivedGeneralCategory
lower	\p{Lowercase}	Lowercase includes more than gc = Lowercase_Letter (Ll). See DerivedCoreProperties. *For strict POSIX, intersect recommendation with {alpha}. One may also add Lt, although it logically doesn't belong.*
upper	\p{Uppercase}	Uppercase includes more than gc = Uppercase_Letter (Lu). *For strict POSIX, intersect recommendation with {alpha}. One may also add Lt, although it logically doesn't belong.*
digit (\d)	\p{gc=Nd}	Non-decimal numbers (like Roman numerals) are normally excluded. In U4.0+, this is the same as gc = Decimal_Number (Nd). See DerivedNumericType

		For strict POSIX, intersect recommendation with {ASCII}
xdigit	\p{gc=Nd} a-f, A-F, ａ-ｆ, Ａ-Ｆ	The A-F are upper & lower, normal and fullwidth. The POSIX spec requires that xdigit contains digit. This also matches Java.
		For strict POSIX, intersect recommendation with {ASCII}
alnum	\p{alpha} \p{digit}	Simple combination of other properties
cntrl	\p{gc=Control}	
graph	[^\p{space} \p{gc=Cc} \p{gc=Cs} \p{gc=Cn}]	Perl 5.8.0 is similar except that it excludes: Z, Cc, Cf, Cs, Cn. The intend is for Perl 5.8.1 to align with the specification here.
		POSIX: includes alpha, digit, punct, excludes cntrl
print	\p{graph} \p{space}	POSIX: includes graph, <space>
space	\p{Whitespace}	See PropList for the definition of Whitespace (also in U3.1, U3.2)
		Note: ZWSP, while a Z character, is for line break control and should not be included.
blank	\p{Whitespace} - [\N{LF} \N{VT} \N{FF} \N{CR} \N{NEL} \p{gc=Zl} \p{gc=Zp}]	"horizontal" whitespace.
word (\w)	\p{alpha} \p{digit} \p{gc=Pc}	This is only an approximation to Word Boundaries (see below). The gc=Pc is added in for programming language identifiers, thus adding "_".
\X	Default Grapheme Clusters	See UAX #29: Text Boundaries, also GraphemeClusterBreakTest.html. Other functions are used for programming language identifier boundaries.
\b	Default Word Boundaries	If there is a requirement that \b align with \w, then it would use the approximation above instead. See UAX #29: Text Boundaries, also WordBreakTest.html. Other functions are used for programming language identifier boundaries.

References

[Boundaries] UAX #29: Text Boundaries
http://www.unicode.org/reports/tr29/

[Case] UAX #21: Case Mappings
http://www.unicode.org/reports/tr21/

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

[Collation] UTS #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
[Glossary]	Unicode Glossary http://www.unicode.org/glossary/ *For explanations of terminology used in this and other documents.*
[LineBreak]	UAX #14: Line Breaking Properties http://www.unicode.org/reports/tr14/
[NewLine]	UAX #13, Unicode Newline Guidelines http://www.unicode.org/reports/tr13/
[Norm]	UAX #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
[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]	UTR #24: Script Names http://www.unicode.org/reports/tr24/
[SpecialCasing]	http://www.unicode.org/Public/UNIDATA/SpecialCasing.txt
[UCD]	http://www.unicode.org/ucd/
[UData]	http://www.unicode.org/Public/UNIDATA/UnicodeData.txt
[UDataDoc]	http://www.unicode.org/Public/UNIDATA/UnicodeData.html
[Unicode]	The Unicode Consortium. The Unicode Standard, Version 4.0. Reading, MA, Addison-Wesley, 2003. 0-321-18578-1.
[Versions]	Versions of the Unicode Standard http://www.unicode.org/versions/ *For details on the precise contents of each version of the Unicode Standard, and how to cite them.*

Acknowledgments

Thanks to Andy Heninger, Alan Liu, Karlsson Kent, Jarkko Hietaniemi, Gurusamy Sarathy, Tom

Watson and Kento Tamura for their feedback on the document.

Modifications

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

8	•
7	• Now proposed as a UTS, adding Conformance and specific wording in each relevant section. • Move hex notation for surrogates from 2.7 Surrogates into 2.1 Hex notation. • Added 4.6 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