# Analyzing Unicode Text with Regular Expressions

Andy Heninger

IBM Corporation

heninger@us.ibm.com

## Abstract

For decades now, Regular Expressions have been used in the analysis of text data, for searching for key words, for extracting out desired fields or substrings from larger bodies of text and in editing or transforming text.

This paper will discuss the application of regular expressions to Unicode text data, including the approaches and extensions that are required to work effectively with the very large Unicode character repertoire. The emphasis is on Unicode specifically, not on the features of regular expressions in general, which is a subject about which entire books can, and have, been written.

## A Very Quick Look At Regular Expressions

Although this paper will primarily be dealing with Unicode related questions, a regular expression language is still needed for discussion and for use in examples Here is a minimalist one, smaller than most real implementations, but sufficient for the purpose.

| ***Item*** | ***Definition*** |
|---|---|
| **.** | Match any single character |
| [*range or set of characters*] | Match any character of a class or set of characters. Set expressions will be described later. |
| * | Match 0 or more occurrences of the preceding item. |
| + | Match 1 or more occurrences of the preceding item. |
| *Literal Characters* | Match themselves. |
| *\udddd*<br>*\Udddddddd* | Unicode Code Point Values, 16 or 32 bits. |
| *( sub-expression )* | Grouping. (abc)*, for example. |
| *a*\|*b*\|*c* | Alternation. Match any one of 'a' or 'b' or 'c'. |

And, to make things more concrete, here are a few samples of simple expressions

| ***Expression*** | ***Description*** |
|---|---|
| Hello | Match or select appearances of the word “Hello” in the target text. |
| aa[a-z]* | Match any word beginning with “aa” and consisting of only the lower case letters a-z. (Just what is in the range [a-z] is another question.) |
| .* | Match everything. |

A large number of implementations and specifications for regular expressions exist, in editors, in programming languages and libraries, in stand-alone search tools, in query languages, and elsewhere. While the basics are pretty much the same among them, differences do exist, especially in the areas of newer or more advanced features, including the level of Unicode support.

Implementations with a reasonable level of Unicode support include Perl, Java (JDK 1.4 or newer), Microsoft .NET and IBM's ICU4C library.

## The Evolution of Character Ranges, from [a-z] to [\p{Letter}]

Access to character properties from regular expression character classes is the the single most important key to accessing the power of Unicode from regular expressions. Let's begin with a very brief history of the use of character sets, or ranges.

In the beginning, regular expressions were assumed to work with text in ASCII, or in some similar legacy character set. A simple expression to match a word might look something like this

[a-zA-Z]+

which is an RE that matches one or more contiguous letters. This only works, more or less, for English. It doesn't work for other languages using letters with accents or other marks (niño, bébé), or for those using non-latin alphabets, such as Greek, Russian or Arabic.

### *POSIX Character Classes*

The first steps towards improving the situation, predating Unicode, came from the POSIX definition of regular expressions. It introduced several named classes of characters, as shown in this table:

| | | | | | |
|---|---|---|---|---|---|
| `[:alnum:]` | `[:cntrl:]` | `[:lower:]` | `[:space:]` | `[:alpha:]` | `[:digit:]` |
| `[:xdigit:]` | `[:print:]` | `[:upper:]` | `[:blank:]` | `[:graph:]` | `[:punct:]` |

Updating our first expression to use the alpha character class rather than the a-to-z range,

[:alpha:]+

will take care of recognizing niño or bébé. And, if the implementation supports the text's character encoding, non-Latin languages will be supported as well.

The POSIX specification for regular expressions is character encoding agnostic. It does not define a precise set of characters to be included in the character classes, and indeed, without knowing the character encoding, it could not. Implementations must make appropriate character class assignments for each encoding or code page that is supported, including Unicode.

The Unicode consortium, in http://www.unicode.org/reports/tr18/#Compatibility_Properties, provides recommended mappings for each of the POSIX character classes in terms of Unicode character properties. These provide for some compatibility when moving between POSIX and Unicode environments, however working directly with Unicode character properties provides for a far more expressive and powerful search.

### *Collation Ranges*

POSIX also introduced the idea of collation, or natural language ordering for character ranges in regular expressions. The idea is that range of characters [a-z] would not be

interpreted based on the numerical character code values for 'a' and 'z', but instead be based on a dictionary sort order. The range [a-z] then includes all representable characters that sort between 'a' and 'z', and will match our Spanish niña and French bébé There are, however, some surprising aspects to this approach.

In collation sort ordering, uppercase letters sort before lower case, that is, the ordering of the letters of the English alphabet is AaBbCcDd...YyZz Thus, [a-z] matches A[aBbCcDd...YyZz], including uppercase B-Z, but not uppercase A. Surprising. The collation based range [a-z] will include some accented characters, but the exact details depend on the locale and the implementation, and can be difficult to determine.

Although the intent was certainly noble, the non-obvious implications of collation based range expressions have limited the spread of this approach. It does not appear in Perl, or in implementations tracing their ancestry in some way to Perl, a group that includes most packages supporting Unicode character properties.

### *Unicode Property based Character Classes*

With Unicode text data, Unicode Character Properties make possible a much more complete and capable system for categorizing and selecting characters. For our original example, we can replace the expression

```
[a-zA-Z]+
```

with this

```
[\p{Alphabetic}]+
```

Note that the set of letters [\p{Alphabetic}] contains many more characters than just the Latin letters [a-zA-Z], including accented Latin letters, letters from other alphabets, and some marks and signs that are normally treated as if they were letters.

The set can be restricted to include Latin letters only in this way:

```
[\p{Alphabetic}&\p{Script=Latin}]+
```

Unicode defines more than 70 character properties; here are some that are likely to especially useful in Regular Expressions.

| ***Property Name*** | ***Description*** |
|---|---|
| General_Category | Categorizes all characters as Letters, Numbers, Separators, Punctuation, Marks, Symbols or Other. Also defines sub-groups within each of these main groups, e.g. Letter, uppercase. |

| ***Property Name*** | ***Description*** |
|---|---|
| Script | The script with which a character is associated. Latin, Greek, Cyrillic, Han, Hebrew, etc. |
| Alphabetic | Letters plus additional letter-like marks. |
| Uppercase | Uppercase characters. Consists mostly of letters, but there are a few other characters with separate upper case forms that are not letters. Note that many characters are not cased – are neither upper nor lower. |
| Lowercase | Lowercase characters. |
| White_Space | White space characters |
| NonCharacter_Code_Point | Code points that are explicitly defined as illegal for the encoding of characters. |

The syntax for using a Unicode property in a character set expression typically has this form (originally from Perl):

[\p{*property_name = property value*}]

For example,

[\p{General_Category=Currency_Symbol}]

Because the Unicode General Category is probably the most commonly referenced property in set expressions, the syntax allows the property name to be omitted, like this:

[\p{Currency_Symbol}]

Each General Category value has both a short and a long name, allowing us to shorten the set expression still further, to

[\p{Sc}]

The properties Alphabetic, Uppercase, Lowercase, White_Space and NonCharacter_Code_Point are boolean – either true or false. For these, only the property name is needed; the value of "true" is implied.

[\p{Alphabetic}]

and

[\p{Alphabetic=TRUE}]

are equivalent.

For reference, here is a list of the Unicode General Category property values, with a brief description and/or representative characters for each.

| ***Long Name*** | ***Abbr*** | ***Examples or Description*** |
|---|---|---|
| Letter | L | All letters. ABC áçî ΓΔΠ |
| Uppercase Letter | Lu | ABC ΩΣΔ |
| Lowercase Letter | Ll | abc |
| Titlecase Letter | Lt | A small number of composite characters containing both an initial capital and following small letter. \u01f2, Dz |
| Modifier Letter | Lm | Small signs that are generally used to indicate modifications of a previous letter. |
| Other Letter | Lo | Letters that do not distinguish case. Includes Chinese, Japanese, Korean ideographs. |
| Mark | M | All Marks |
| Non-Spacing Mark | Mn | Marks that combine with a preceding base character to form a single combined character.<br>\u0300, combing grave accent<br>\u0307, combining diaeresis |
| Spacing Combining Mark | Mc | Combining Marks that display after their base character, rather than over/under it.<br>\u094c, Devanagari vowel sign<br>\u0d46, Malayalam vowel sign E |
| Enclosing Mark | Me | Non-spacing marks that display as completely enclosing their base character.<br>\u20dd, combining enclosing circle<br>\u0489, combining Cyrillic millions sign |
| Number | N | All Numbers |
| Decimal Digit Number | Nd | 123 ١٢٣٤ |
| Letter Number | Nl | Roman Numerals, and a few other relatively rare forms. |

| ***Long Name*** | ***Abbr*** | ***Examples or Description*** |
|---|---|---|
| Other Number | No | Superscripts, Subscripts, fractions, other non-decimal-digit values encoded as single characters.<br>\u0bf0, Tamil number ten<br>\u0f2a, Tibetan digit half one<br>\u2154, vulgar fraction two thirds |
| Punctuation | P | All Punctuation |
| Connector Punctuation | Pc | \u005f, _ (ASCII underscore)<br>\u203f, undertie<br>and a few more. |
| Dash Punctuation | Pd | \u002d, hyphen-minus<br>\u2014, em dash<br>\u301c, wave dash<br>and a variety of other dashes and hyphens. |
| Open Punctuation | Ps | ( [ {<br>and a long list of additional left brackets and parentheses. |
| Close Punctuation | Pe | ) ] }<br>and a long list of additional right brackets and parentheses. |
| Initial Punctuation | Pi | Opening quotation marks, many forms. |
| Final Punctuation | Pf | Closing quotation marks, many forms. |
| Other Punctuation | Po | ¿ ¡ ' $ % & * , . ; : / · ! ?<br>and many more, from many scripts. |
| Symbol | S | All Symbols |
| Math Symbol | Sm | + < = > \| ÷ ∑ ⁄ ∩ ∫<br>and many more |
| Currency Symbol | Sc | $ ¢ £ ¤ ¥<br>and many more |
| Modifier Symbol | Sk | Spacing (non-combining) forms of accents, other diacritics. |

| ***Long Name*** | ***Abbr*** | ***Examples or Description*** |
|---|---|---|
| Other Symbol | So | A grab-bag<br>00a7, section sign<br>00ae, registered sign<br>00b0, degree sign<br>0482, Cyrillic thousands sign<br>060e, Arabic poetic verse sign<br>09fa, Bengali isshar<br>and many more. |
| Separator | Z | All Separators |
| Space Separator | Zs | Spaces. 18 in all, including the classic ASCII \u0020. Not the same as "white space", does not include tabs, new lines, etc. |
| Line Separator | Zl | Includes exactly one character,<br>\u2028, Line Separator<br>Carriage Return and New Line are categorized as Control characters, Cc. |
| Paragraph Separator | Zp | Includes exactly one character,<br>\u2029, paragraph separator |
| Other | C | All other characters |
| Control | Cc | ASCII control codes, characters in the ranges<br>\u0000-\u001f and<br>\u0080-\u009f |
| Format | Cf | Soft hyphens, bidi controls, zero-width spaces, joiners, and more. |
| Surrogate | Cs | The range \ud800-\udbff<br>An occurrence of characters within the surrogate range indicates a problem – either unpaired surrogates in the input text, or an implementation that does not handle supplementary characters. |
| Private Use | Co | The ranges<br>\ue000-\ue8ff and<br>\U000f0000 - \U000ffffd<br>\U00100000 - \U0010fffd<br>Reserved for application specific use. |
| Not Assigned | Cn | Unassigned characters. |

## Scripts and Blocks

The script of a character is the alphabet that the character belongs to. Being a Unicode character property, script is accessible to regular expressions in the same way as any other property. For example, text containing Hebrew characters can be selected with this expression:

```
[\p{script=Hebrew}]+
```

With some implementations, the expression can be shortened to the script name alone, omitting the "script =".

```
[\p{Hebrew}]+
```

This does not conflict with the similar short form for General Category because there is no overlap in the names – there is no script with the same name as a General Category value.

Here is a slightly extended expression that matches entire lines that contain any Hebrew characters:

```
.*[\p{Hebrew}].*
```

The ".*"s match all characters following or preceding the Hebrew on the line.

### *Script is not the same as Language*

Regular expressions can identify text written with a particular script, but in most cases this is not the same as identifying the language of the text. The Latin script, to take the obvious case, is used for English, French, German, Spanish, etc. Cyrillic is used for Russian, Serbian, Bulgarian and more. Arabic, Hausa, Kashmiri, Kazak, Kurdish, Kyrghyz, Pashto, Farsi, Sindhi, Tatar, Turkish, Uyghur and Urdu are all written with the Arabic script.

Determining the language of a sample of text is well beyond what can easily be accomplished with regular expressions alone.

### *Characters shared between Scripts*

Many characters are shared between scripts, including numbers, punctuation, symbols, formatting and control characters. All of these shared characters have been assigned the script value of "Common."

A related class of shared characters are non-spacing marks that combine with a base character to form what is logically a single character. Combining accent marks are an example. The script property of these characters is "Inherited", and they are considered to have the same script as the base character that precedes them.

Taking Common and Inherited script characters into account, here is a regular expression to match a range of text written in the Cyrillic script:

```
[\p{Common}\p{Inherited}]*([\p{Cyrillic}][\p{Common}\p{Inherited}]*)+
```

This will match one or more Cyrillic characters (the '+' at the end of the expression), together with any number of intervening, preceding or following Common or Inherited characters.

### *Blocks*

*Block* is a Unicode property that is related to script, but not quite as useful. Because some regular expression implementations make the Block property available, but not the Script, it is important to understand the differences.

A block is a contiguous range of Unicode characters (code points) that has been designated for a particular use, often for encoding a script.

Block property and script names do overlap. Some examples of the property names are shown in the chart below; the full list is available at http://www.unicode.org/Public/UNIDATA/Blocks.txt and

http://www.unicode.org/Public/UNIDATA/Scripts.txt

| ***Block*** | ***Script*** |
|---|---|
| Basic Latin | Latin |
| Latin-1 Supplement | |
| Latin Extended-A | |
| Latin Extended-B | |
| Greek and Coptic | Greek |
| Hebrew | Hebrew |
| Arabic | Arabic |

The block property has some limitations that usually make the script property a better choice if it is available. The problems with relying on Blocks included

1. Blocks are simply ranges, and may contain reserved code points.

2. The characters from an alphabet or writing system may be spread between several blocks.
3. A single block may contain characters that are used in several scripts. There is nothing corresponding to the "Common" and "Inherited" script properties to enable the identification of these shared characters.

Despite these limitations, the block property can still be useful in identifying text that contains characters from desired scripts.

## More Set Operations

### *Lose the Brackets*

When specifying a class of characters from a single Unicode property, the [brackets] around the set expression may be omitted. Thus, these two expressions

```
[\p{Letter}]+
\p{Letter}+
```

are equivalent.

### *Negated Sets*

The set of characters not having a property can be specified by \P{property} Again, the following two expressions are equivalent:

```
[^\p{Letter}]
\P{Letter}
```

The "^" in a [^set] expression complements or inverts the set. This "^" notation dates back to the prehistory of regular expressions; thus [^a-z] matches everything except the lower case letters a through z.

### *Unions and Intersections*

A set specifying two or more properties contains the union of the characters matched by the properties. Thus the expression

```
[\p{Lowercase_Letter}\p{UpperCase_Letter}]
```

will match all upper or lower case letters. It differs from \p{Letter} in that it does not match un-cased letters such has Han ideographs.

In this case, the [brackets] on the set expression are definitely required; without them, the expression

```
\p{Lowercase_Letter}\p{UpperCase_Letter}
```

matches a sequence of two characters, a lower case letter followed by an upper case letter.

Intersection, specified with an "&" makes it possible to define sets of characters for which two or more properties must be true. For example,

```
[\p{Cyrillic}&\p{UpperCase_Letter}]
```

would match only uppercase Cyrillic letters.

*Caution:* The set intersection operator is "&&" in Java regular expressions, "&" in ICU regular expressions, and is not available in Perl.

## Code Points, Code Units, UTF 8/16/32

When we say that a regular expression matches a "character" from some input text, what does that really mean?

The answer is that the basic unit of matching is a Unicode "Code Point", which is an abstract character represented by an integer in the range from 0 to 0x10ffff. Code points can have different physical representations in memory, and it is the responsibility of the implementation to ensure that any underlying byte or word values are transformed to code point character values before any matching operations are carried out.

Viewed from the opposite side, it is not possible, or at least shouldn't be possible, to write a regular expression that would match half of a UTF-16 surrogate pair, or a single byte from a multiple-byte UTF-8 sequence.

Under normal circumstances (no unpaired surrogates or malformed UTF-8 sequences), it should not be possible for an application to determine, based on any regular expression behavior, what underlying storage format was used for input text.

## Normalization

Some characters can have alternative Unicode representations.

For example, the character Ñ can be represented either as the single code point
\u00d1, Latin capital letter N with tilde

Or by the sequence of two code points,
\u004e, Latin capital letter N
\u0303, combining tilde

These are two different normalization forms for Ñ. Both are perfectly valid. Which is chosen can definitely affect the match results obtained with regular expressions.

- Searching for a plain 'N' will match the second (decomposed) form, but not the first.
- Searching for \p{Non Spacing Mark} will match the second form (the tilde, not the N), but not the first
- A single '.' (match any character) in a pattern will consume the entire composed form, but only half of the decomposed form.

There are a few different approaches to dealing with these problems

- Design the regular expression patterns in such a way that the results obtained are independent of the normalization form. This can work well if the expressions are being used to parse apart text based on fixed keywords or punctuation characters, with the intervening text being skipped over, or perhaps captured, but otherwise not analyzed.
- Have independent knowledge, based on the source of the data or the design of the system, that the text to be analyzed will arrive with a known normalization form, and design the regular expression patterns accordingly.
- Normalize the data in a separate step.
- For regular expression implementations that support it, specify the normalization, or Canonical Equivalence option. Note that this option is available in relatively few implementations. For Java, this option will force matching to occur as if the text were in decomposed form.

Here is an example of the first approach of creating a pattern that is independent of the Unicode normalization form of the data. This pattern will scan for the content of a particular element from a string with XML-like syntax.

"<LastName>(.*?)</LastName >"

The markup tags being matched have only one normalization form and can be easily matched with literal strings. The text between the tags, which with Perl compatible expressions will be captured and available to the application, can be in any normalization form.

When matching words that may themselves appear in more than one normalization form, such as

"Möller|Mönch|Mörk"

it's probably better to choose one of the other approaches and put the incoming data into a known form before doing the match.

## Line Endings, New Lines

Most regular expression implementations, by default, restrict the range of a pattern match to a single line of text. Because of this, it is important to know how lines are terminated; that is, which characters or sequences separate one line from another.

(Perversely enough, in most Regular Expression packages, the option to alter the default behavior to allow a match to proceed across a line boundaries into multiple lines is "single line mode".)

The set of Unicode line endings is larger than that of older legacy character sets. The line ending characters recognized by regular expressions are

| | |
|---|---|
| \u000A | Line Feed |
| \u000C | Form Feed |
| \u000D | Carriage Return |
| \u0085 | Next Line (NEL) |
| \u2028 | Line Separator |
| \u2029 | Paragraph Separator |

and the sequence

\u000D \u000A

When a match can cross line boundaries, a "." will match the Carriage Return / Line Feed sequence as if it were a single character.

With the implementations available today, there is no single pattern meta-character for matching any of the allowed line endings. There should be, but there isn't.

## Case Insensitive Matches

All regular expression implementations provide the ability to do a case insensitive, or loose, pattern matches.

Full Unicode caseless matching, as described in Section 3.13 of the Unicode Standard, is

a relatively complicated operation. With full caseless matching it is possible for a single lower case letter to match two upper case letters, with the most famous example being the German character ß, which is equivalent to “ss” and which upper-cases to “SS”.

The Unicode-enabled regular expression packages that are available today provide a more limited form of caseless matching, not handling situations where changing the case of a string could change its length.

## Grapheme Clusters

A *grapheme cluster* is the name given to what an end user would consider to be a character when looking at displayed or printed text. A grapheme cluster may be composed of more than one Unicode code point.

The simplest examples of multi-codepoint grapheme clusters are letters with accents or other diacritical marks, such as our Spanish ñ that appeared in earlier examples. For ñ, and other letters that are commonly used in European languages, single code point forms of the characters do exist, but for many other characters a multi-codepoint representation is required.

Unicode-aware regular expression pattern languages have an additional meta-character, often \X, that will match a complete grapheme cluster, or test whether the current match position is at the boundary of a grapheme cluster.

There are some differences between implementations - not all have the definition of a grapheme cluster quite right, or even implement this at all - so it is important to check the documentation of the specific package being used.

## Word Boundaries (\b and \B)

Regular Expressions patterns have traditionally provided operators to test whether the current position in the text is a word boundary (\b) or not (\B). The exact definition of what constitutes a word for this purpose is usually pretty simple, typically a contiguous run of “word” characters.

Here is an example of traditional RE word boundary positions

```
Hello there.  G'day 123.456
^    ^^    ^  ^^^  ^^  ^^  ^
```

The Unicode Consortium has published a more sophisticated set of rules for determining word boundaries (UAX 29, Text Boundaries). The Unicode definitions do a better job of

not breaking apart contractions or numbers containing decimal points, and they take a fundamentally different approach to non-word characters, such as punctuation or white space. The concept is to have word boundaries around any piece of text that one would expect to be matched by a "match whole words only" option in a word processor.

With the Unicode definitions, the word boundary positions are

```
Hello there.  G'day 123.456
^    ^^    ^^^^    ^^      ^
```

This is different enough from the conventional RE behavior and expectations for \b that the default has not been changed. Unicode behavior may be available as an option, however. Check the docs for your implementation.

## Summary

Regular expressions are a powerful tool for analyzing and extracting information from Unicode text data. Many implementations now have good support for Unicode, including access to character properties, case insensitive matching, grapheme and word boundaries and normalization, all of which greatly increase the power and flexibility of regular expression pattern matching.

## References

References to additional sources of information may be the most important part of this paper. The whole topic of using regular expressions with Unicode text is far too big to be covered in any detail in a single paper.

**Mastering Regular Expressions, Second Edition. Jeffrey Friedl, O'Reilly, 2002**

Mastering Regular Expressions is an essential reference for anyone doing substantial work with regular expressions. It provides clear and detailed descriptions of how regular expression pattern matching really works and of all the pattern operators and meta characters available. Unicode is touched on briefly, but the main emphasis is on general regular expression behavior.

**Unicode Regular Expressions, Unicode Technical Standard #18.**
http://www.unicode.org/reports/tr18/

This document from the Unicode Consortium describes the set of features that should be provided by regular expression implementations that wish to claim to support Unicode. Three levels of conformance are defined. Basic, or level 1, support provides the minimum set of features needed for regular expressions to work effectively with Unicode text. Level 2 provides for additional features, either more advanced and/or harder to implement. Level 3 adds the ability to tailor matching behavior based on Locale or Language.

The implementations available today are mostly at Level 1, sometimes with an occasional level 2 feature.

While intended primarily for those designing or implementing regular expression packages, the document will still be of interest to anyone curious about the future directions of regular expressions, or in evaluating the feature set of an existing implementation.

**The Unicode Character Database, or UCD**
http://www.unicode.org/Public/UNIDATA/UCD.html
A guide to the content and format of the Unicode Character Database, a set of files that define the character properties and their mappings onto the set of Unicode characters.

Although the sheer mass of information included here can be overwhelming at first, anyone doing much work with Unicode character properties will eventually find themselves referring to this set of data.

**Unicode Script and Block Names.**
These are the UCD files that list all available Script and Block names, and of are particular interest because these two character properties are so useful in regular expressions.
http://www.unicode.org/Public/UNIDATA/Blocks.txt
http://www.unicode.org/Public/UNIDATA/Scripts.txt

**The Unicode Standard 4.0, The Unicode Consortium, Addison-Wesley, 2003**
Also available on line, http://www.unicode.org/versions/Unicode4.0.1/
Section 3.5 and all of Chapter 4 contain essential definitions and descriptions of character properties.

### *Implementations of Unicode Regular Expressions*

Regular Expression packages with good Unicode support are readily available. While none of the packages listed here provides every feature mentioned in this paper, they all implement most of them.

**Perl Regular Expressions**
http://www.perldoc.com/perl5.8.4/pod/perlre.html
Perl is a scripting language that includes integrated regular expressions. The innovations in RE pattern features and matching behavior pioneered by Perl have gone on to become defacto standards, and are widely copied.

**IBM ICU4C**
http://oss.software.ibm.com/icu/userguide/regexp.html
A C / C++ Unicode support library, including Perl compatible regular expressions. The API is loosely based on the Java JDK classes for regular expressions.

**JAVA**
http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/package-summary.html
With version 1.4, Sun added a regular expression package to the JDK. The pattern features are based on Perl.

**Microsoft .NET**
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconCOMRegularExpressions.asp
The .NET Framework regular expression classes are part of the base class library and can be used with any language or tool that targets the common language runtime. Perl compatible.

**XML Schema**
http://www.w3.org/TR/xmlschema-2/#regexs
A specification, not an implementation. XML Schemas can use regular expression matches as one of the mechanisms for validating the contents of XML documents. Once again, the regular expression features are based on Perl.

Multiple implementations are available, including Xerces XML from the Apache Software Foundation
http://xml.apache.org/xerces-c/index.html