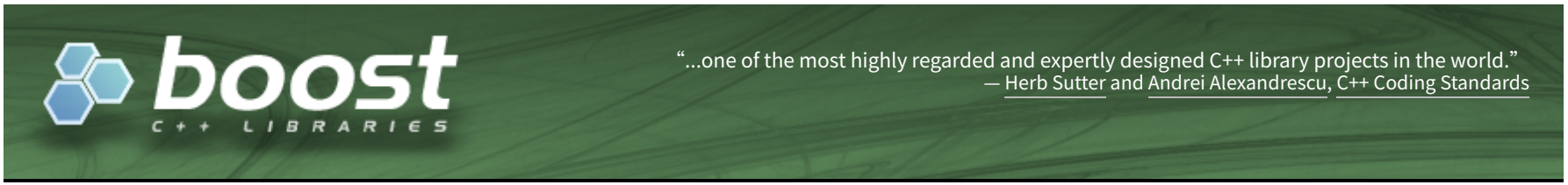

Perl Regular Expression Syntax

Synopsis

The Perl regular expression syntax is based on that used by the programming language Perl . Perl regular expressions are the default behavior in Boost.Regex or you can pass the flag `perl` to the `basic_regex` constructor, for example:

```
// e1 is a case sensitive Perl regular expression:
// since Perl is the default option there's no need to explicitly specify the syntax used here:
boost::regex e1(my_expression);
// e2 a case insensitive Perl regular expression:
boost::regex e2(my_expression, boost::regex::perl|boost::regex::icase);
```

Perl Regular Expression Syntax

In Perl regular expressions, all characters match themselves except for the following special characters:

```
.[{}()\*+?|^$
```

Other characters are special only in certain situations - for example `]` is special only after an opening `[`.

Wildcard

The single character '.' when used outside of a character set will match any single character except:

- The NULL character when the flag `match_not_dot_null` is passed to the matching algorithms.
- The newline character when the flag `match_not_dot_newline` is passed to the matching algorithms.

Anchors

A '^' character shall match the start of a line.

A '$' character shall match the end of a line.

Marked sub-expressions

A section beginning `(` and ending `)` acts as a marked sub-expression. Whatever matched the sub-expression is split out in a separate field by the matching algorithms. Marked sub-expressions can also repeated, or referred to by a back-reference.

Non-marking grouping

A marked sub-expression is useful to lexically group part of a regular expression, but has the side-effect of spitting out an extra field in the result. As an alternative you can lexically group part of a regular expression, without generating a marked sub-expression by using `(?:` and `)`, for example `(?:ab)+` will repeat `ab` without splitting out any separate sub-expressions.

Repeats

Any atom (a single character, a marked sub-expression, or a character class) can be repeated with the *, +, ?, and {} operators.

The * operator will match the preceding atom zero or more times, for example the expression `a*b` will match any of the following:

```
b
ab
aaaaaaaab
```

The + operator will match the preceding atom one or more times, for example the expression `a+b` will match any of the following:

```
ab
aaaaaaaab
```

But will not match:

```
b
```

The ? operator will match the preceding atom zero or one times, for example the expression ca?b will match any of the following:

```
cb
cab
```

But will not match:

```
caab
```

An atom can also be repeated with a bounded repeat:

`a{n}` Matches 'a' repeated exactly n times.

`a{n,}` Matches 'a' repeated n or more times.

`a{n, m}` Matches 'a' repeated between n and m times inclusive.

For example:

```
^a{2,3}$
```

Will match either of:

```
aa
aaa
```

But neither of:

```
a
aaaa
```

Note that the "{" and "}" characters will treated as ordinary literals when used in a context that is not a repeat: this matches Perl 5.x behavior. For example in the expressions "ab{1", "ab1}" and "a{b}c" the curly brackets are all treated as literals and *no error will be raised*.

It is an error to use a repeat operator, if the preceding construct can not be repeated, for example:

```
a(*)
```

Will raise an error, as there is nothing for the * operator to be applied to.

Non greedy repeats

The normal repeat operators are "greedy", that is to say they will consume as much input as possible. There are non-greedy versions available that will consume as little input as possible while still producing a match.

*? Matches the previous atom zero or more times, while consuming as little input as possible.

+? Matches the previous atom one or more times, while consuming as little input as possible.

?? Matches the previous atom zero or one times, while consuming as little input as possible.

{n,}? Matches the previous atom n or more times, while consuming as little input as possible.

{n,m}? Matches the previous atom between n and m times, while consuming as little input as possible.

Possessive repeats

By default when a repeated pattern does not match then the engine will backtrack until a match is found. However, this behaviour can sometime be undesireble so there are also "possessive" repeats: these match as much as possible and do not then allow backtracking if the rest of the expression fails to match.

*+ Matches the previous atom zero or more times, while giving nothing back.

++ Matches the previous atom one or more times, while giving nothing back.

?+ Matches the previous atom zero or one times, while giving nothing back.

{n,}+ Matches the previous atom n or more times, while giving nothing back.

{n,m}+ Matches the previous atom between n and m times, while giving nothing back.

Back references

An escape character followed by a digit *n*, where *n* is in the range 1-9, matches the same string that was matched by sub-expression *n*. For example the expression:

```
^(a*)[^a]*\1$
```

Will match the string:

```
aaabbaaa
```

But not the string:

```
aaabba
```

You can also use the \g escape for the same function, for example:

Escape	**Meaning**
\g1	Match whatever matched sub-expression 1
\g{1}	Match whatever matched sub-expression 1: this form allows for safer parsing of the expression in cases like \g{1}2 or for indexes higher than 9 as in \g{1234}

Escape	Meaning
\g-1	Match whatever matched the last opened sub-expression
\g{-2}	Match whatever matched the last but one opened sub-expression
\g{one}	Match whatever matched the sub-expression named "one"

Finally the \k escape can be used to refer to named subexpressions, for example \k<two> will match whatever matched the subexpression named "two".

Alternation

The | operator will match either of its arguments, so for example: abc|def will match either "abc" or "def".

Parenthesis can be used to group alternations, for example: ab(d|ef) will match either of "abd" or "abef".

Empty alternatives are not allowed (these are almost always a mistake), but if you really want an empty alternative use (?:) as a placeholder, for example:

|abc is not a valid expression, but

(?:)|abc is and is equivalent, also the expression:

(?:abc)?? has exactly the same effect.

Character sets

A character set is a bracket-expression starting with [] and ending with , it defines a set of characters, and matches any single character that is a member of that set.

A bracket expression may contain any combination of the following:

Single characters

For example [abc], will match any of the characters 'a', 'b', or 'c'.

Character ranges

For example `[a-c]` will match any single character in the range 'a' to 'c'. By default, for Perl regular expressions, a character x is within the range y to z, if the code point of the character lies within the codepoints of the endpoints of the range. Alternatively, if you set the `collate` flag when constructing the regular expression, then ranges are locale sensitive.

Negation

If the bracket-expression begins with the `^` character, then it matches the complement of the characters it contains, for example `[^a-c]` matches any character that is not in the range `a-c`.

Character classes

An expression of the form `[[:name:]]` matches the named character class "name", for example `[[:lower:]]` matches any lower case character. See character class names.

Collating Elements

An expression of the form `[[.col.]]` matches the collating element *col*. A collating element is any single character, or any sequence of characters that collates as a single unit. Collating elements may also be used as the end point of a range, for example: `[[.ae.]-c]` matches the character sequence "ae", plus any single character in the range "ae"-c, assuming that "ae" is treated as a single collating element in the current locale.

As an extension, a collating element may also be specified via it's symbolic name, for example:

```
[[.NUL.]]
```

matches a `\0` character.

Equivalence classes

An expression of the form `[[=col=]]`, matches any character or collating element whose primary sort key is the same as that for collating element *col*, as with collating elements the name *col* may be a symbolic name. A primary sort key is one that ignores case, accentation, or locale-specific tailorings; so for example `[[=a=]]` matches any of the characters: a, À, Á, Â, Ã, Ä, Å, A, à, á, â, ã, ä and å. Unfortunately implementation of this is reliant on the platform's collation and localisation support; this feature can not be relied upon to work portably

across all platforms, or even all locales on one platform.

Escaped Characters

All the escape sequences that match a single character, or a single character class are permitted within a character class definition. For example `[\[\]]` would match either of `[` or `]` while `[\W\d]` would match any character that is either a "digit", *or* is *not* a "word" character.

Combinations

All of the above can be combined in one character set declaration, for example: `[[:digit:]a-c[.NUL.]]`.

Escapes

Any special character preceded by an escape shall match itself.

The following escape sequences are all synonyms for single characters:

Escape	**Character**
`\a`	`\a`
`\e`	`0x1B`
`\f`	`\f`
`\n`	`\n`
`\r`	`\r`
`\t`	`\t`
`\v`	`\v`
`\b`	`\b` (but only inside a character class declaration).
`\cX`	An ASCII escape sequence - the character whose code point is X % 32

Escape	Character
`\xdd`	A hexadecimal escape sequence - matches the single character whose code point is 0xdd.
`\x{dddd}`	A hexadecimal escape sequence - matches the single character whose code point is 0xdddd.
`\0ddd`	An octal escape sequence - matches the single character whose code point is 0ddd.
`\N{name}`	Matches the single character which has the symbolic name *name*. For example `\N{newline}` matches the single character \n.

"Single character" character classes:

Any escaped character *x*, if *x* is the name of a character class shall match any character that is a member of that class, and any escaped character *X*, if *x* is the name of a character class, shall match any character not in that class.

The following are supported by default:

Escape sequence	Equivalent to
`\d`	`[[:digit:]]`
`\l`	`[[:lower:]]`
`\s`	`[[:space:]]`
`\u`	`[[:upper:]]`
`\w`	`[[:word:]]`
`\h`	Horizontal whitespace
`\v`	Vertical whitespace
`\D`	`[^[:digit:]]`
`\L`	`[^[:lower:]]`
`\S`	`[^[:space:]]`

Escape sequence	Equivalent to
\U	[^[:upper:]]
\W	[^[:word:]]
\H	Not Horizontal whitespace
\V	Not Vertical whitespace

Character Properties

The character property names in the following table are all equivalent to the names used in character classes.

Form	Description	Equivalent character set form
\pX	Matches any character that has the property X.	[[:X:]]
\p{Name}	Matches any character that has the property Name.	[[:Name:]]
\PX	Matches any character that does not have the property X.	[^[:X:]]
\P{Name}	Matches any character that does not have the property Name.	[^[:Name:]]

For example \pd matches any "digit" character, as does \p{digit}.

Word Boundaries

The following escape sequences match the boundaries of words:

< Matches the start of a word.

> Matches the end of a word.

\b Matches a word boundary (the start or end of a word).

\B Matches only when not at a word boundary.

Buffer boundaries

The following match only at buffer boundaries: a "buffer" in this context is the whole of the input text that is being matched against (note that ^ and $ may match embedded newlines within the text).

\` Matches at the start of a buffer only.

\' Matches at the end of a buffer only.

\A Matches at the start of a buffer only (the same as \`).

\z Matches at the end of a buffer only (the same as \').

\Z Matches a zero-width assertion consisting of an optional sequence of newlines at the end of a buffer: equivalent to the regular expression `(?=\v*\z)`. Note that this is subtly different from Perl which behaves as if matching `(?=\n?\z)`.

Continuation Escape

The sequence `\G` matches only at the end of the last match found, or at the start of the text being matched if no previous match was found. This escape useful if you're iterating over the matches contained within a text, and you want each subsequence match to start where the last one ended.

Quoting escape

The escape sequence `\Q` begins a "quoted sequence": all the subsequent characters are treated as literals, until either the end of the regular expression or \E is found. For example the expression: `\Q*+\Ea+` would match either of:

```
\*+a
\*+aaa
```

Unicode escapes

\C Matches a single code point: in Boost regex this has exactly the same effect as a "." operator. \X Matches a combining character sequence: that is any non-combining character followed by a sequence of zero or more combining characters.

Matching Line Endings

The escape sequence \R matches any line ending character sequence, specifically it is identical to the expression `(?>\x0D\x0A?|[\x0A-\x0C\x85\x{2028}\x{2029}])`.

Keeping back some text

\K Resets the start location of $0 to the current text position: in other words everything to the left of \K is "kept back" and does not form part of the regular expression match. $` is updated accordingly.

For example `foo\Kbar` matched against the text "foobar" would return the match "bar" for $0 and "foo" for $`. This can be used to simulate variable width lookbehind assertions.

Any other escape

Any other escape sequence matches the character that is escaped, for example \@ matches a literal '@'.

Perl Extended Patterns

Perl-specific extensions to the regular expression syntax all start with `(?`.

Named Subexpressions

You can create a named subexpression using:

```
(?<NAME>expression)
```

Which can be then be referred to by the name *NAME*. Alternatively you can delimit the name using 'NAME' as in:

```
(?'NAME'expression)
```

These named subexpressions can be referred to in a backreference using either `\g{NAME}` or `\k<NAME>` and can also be referred to by name in a Perl format string for search and replace operations, or in the `match_results` member functions.

Comments

`(?# ... )` is treated as a comment, it's contents are ignored.

Modifiers

`(?imsx-imsx ... )` alters which of the perl modifiers are in effect within the pattern, changes take effect from the point that the block is first seen and extend to any enclosing `)`. Letters before a '-' turn that perl modifier on, letters afterward, turn it off.

`(?imsx-imsx:pattern)` applies the specified modifiers to pattern only.

Non-marking groups

`(?:pattern)` lexically groups pattern, without generating an additional sub-expression.

Branch reset

`(?|pattern)` resets the subexpression count at the start of each "|" alternative within *pattern*.

The sub-expression count following this construct is that of whichever branch had the largest number of sub-expressions. This construct is useful when you want to capture one of a number of alternative matches in a single sub-expression index.

In the following example the index of each sub-expression is shown below the expression:

```
# before  ---------------branch-reset----------- after
/ ( a )  (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1            2         2  3        2     3     4
```

Lookahead

`(?=pattern)` consumes zero characters, only if pattern matches.

`(?!pattern)` consumes zero characters, only if pattern does not match.

Lookahead is typically used to create the logical AND of two regular expressions, for example if a password must contain a lower case letter,

an upper case letter, a punctuation symbol, and be at least 6 characters long, then the expression:

```
(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}
```

could be used to validate the password.

Lookbehind

`(?<=pattern)` consumes zero characters, only if pattern could be matched against the characters preceding the current position (pattern must be of fixed length).

`(?<!pattern)` consumes zero characters, only if pattern could not be matched against the characters preceding the current position (pattern must be of fixed length).

Independent sub-expressions

`(?>pattern)` *pattern* is matched independently of the surrounding patterns, the expression will never backtrack into *pattern*. Independent sub-expressions are typically used to improve performance; only the best possible match for pattern will be considered, if this doesn't allow the expression as a whole to match then no match is found at all.

Recursive Expressions

`(?`*N*`)` `(?-`*N*`)` `(?+`*N*`)` `(?R)` `(?0)` `(?&NAME)`

`(?R)` and `(?0)` recurse to the start of the entire pattern.

`(?`*N*`)` executes sub-expression *N* recursively, for example `(?2)` will recurse to sub-expression 2.

`(?-`*N*`)` and `(?+`*N*`)` are relative recursions, so for example `(?-1)` recurses to the last sub-expression to be declared, and `(?+1)` recurses to the next sub-expression to be declared.

`(?&NAME)` recurses to named sub-expression *NAME*.

Conditional Expressions

`(?(condition)yes-pattern|no-pattern)` attempts to match *yes-pattern* if the *condition* is true, otherwise attempts to match *no-*

pattern.

`(?(condition)yes-pattern)` attempts to match *yes-pattern* if the *condition* is true, otherwise matches the NULL string.

condition may be either: a forward lookahead assert, the index of a marked sub-expression (the condition becomes true if the sub-expression has been matched), or an index of a recursion (the condition become true if we are executing directly inside the specified recursion).

Here is a summary of the possible predicates:

- `(?(?=assert)yes-pattern|no-pattern)` Executes *yes-pattern* if the forward look-ahead assert matches, otherwise executes *no-pattern*.
- `(?(?!assert)yes-pattern|no-pattern)` Executes *yes-pattern* if the forward look-ahead assert does not match, otherwise executes *no-pattern*.
- `(?(`*N*`)yes-pattern|no-pattern)` Executes *yes-pattern* if subexpression *N* has been matched, otherwise executes *no-pattern*.
- `(?(<`*name*`>)yes-pattern|no-pattern)` Executes *yes-pattern* if named subexpression *name* has been matched, otherwise executes *no-pattern*.
- `(?('`*name*`')yes-pattern|no-pattern)` Executes *yes-pattern* if named subexpression *name* has been matched, otherwise executes *no-pattern*.
- `(?(R)yes-pattern|no-pattern)` Executes *yes-pattern* if we are executing inside a recursion, otherwise executes *no-pattern*.
- `(?(R`*N*`)yes-pattern|no-pattern)` Executes *yes-pattern* if we are executing inside a recursion to sub-expression *N*, otherwise executes *no-pattern*.
- `(?(R&`*name*`)yes-pattern|no-pattern)` Executes *yes-pattern* if we are executing inside a recursion to named sub-expression *name*, otherwise executes *no-pattern*.
- `(?(DEFINE)never-exectuted-pattern)` Defines a block of code that is never executed and matches no characters: this is usually used to define one or more named sub-expressions which are referred to from elsewhere in the pattern.

Backtracking Control Verbs

This library has partial support for Perl's backtracking control verbs, in particular (*MARK) is not supported. There may also be detail differences in behaviour between this library and Perl, not least because Perl's behaviour is rather under-documented and often somewhat random in how it behaves in practice. The verbs supported are:

- `(*PRUNE)` Has no effect unless backtracked onto, in which case all the backtracking information prior to this point is discarded.
- `(*SKIP)` Behaves the same as `(*PRUNE)` except that it is assumed that no match can possibly occur prior to the current point in the string being searched. This can be used to optimize searches by skipping over chunks of text that have already been determined can

not form a match.
- `(*THEN)` Has no effect unless backtracked onto, in which case all subsequent alternatives in a group of alternations are discarded.
- `(*COMMIT)` Has no effect unless backtracked onto, in which case all subsequent matching/searching attempts are abandoned.
- `(*FAIL)` Causes the match to fail unconditionally at this point, can be used to force the engine to backtrack.
- `(*ACCEPT)` Causes the pattern to be considered matched at the current point. Any half-open sub-expressions are closed at the current point.

Operator precedence

The order of precedence for of operators is as follows:

1. Collation-related bracket symbols `[==] [::] [..]`
2. Escaped characters `\`
3. Character set (bracket expression) `[]`
4. Grouping `()`
5. Single-character-ERE duplication `* + ? {m,n}`
6. Concatenation
7. Anchoring `^$`
8. Alternation `|`

What gets matched

If you view the regular expression as a directed (possibly cyclic) graph, then the best match found is the first match found by a depth-first-search performed on that graph, while matching the input text.

Alternatively:

The best match found is the leftmost match, with individual elements matched as follows;

Construct	**What gets matched**
`AtomA AtomB`	Locates the best match for *AtomA* that has a following match for *AtomB*.

Construct	What gets matched
`Expression1 | Expression2`	If *Expresion1* can be matched then returns that match, otherwise attempts to match *Expression2*.
`S{N}`	Matches *S* repeated exactly N times.
`S{N,M}`	Matches S repeated between N and M times, and as many times as possible.
`S{N,M}?`	Matches S repeated between N and M times, and as few times as possible.
`S?, S*, S+`	The same as `S{0,1}`, `S{0,UINT_MAX}`, `S{1,UINT_MAX}` respectively.
`S??, S*?, S+?`	The same as `S{0,1}?`, `S{0,UINT_MAX}?`, `S{1,UINT_MAX}?` respectively.
`(?>S)`	Matches the best match for *S*, and only that.
`(?=S), (?<=S)`	Matches only the best match for *S* (this is only visible if there are capturing parenthesis within *S*).
`(?!S), (?<!S)`	Considers only whether a match for S exists or not.
`(?(condition)yes-pattern | no-pattern)`	If condition is true, then only yes-pattern is considered, otherwise only no-pattern is considered.

Variations

The options `normal`, `ECMAScript`, `JavaScript` and `JScript` are all synonyms for `perl`.

Options

There are a variety of flags that may be combined with the `perl` option when constructing the regular expression, in particular note that the `newline_alt` option alters the syntax, while the `collate`, `nosubs` and `icase` options modify how the case and locale sensitivity are to be applied.

Pattern Modifiers

The perl `smix` modifiers can either be applied using a `(?smix-smix)` prefix to the regular expression, or with one of the regex-compile time flags `no_mod_m`, `mod_x`, `mod_s`, and `no_mod_s`.

References

Perl 5.8.