Modified ECMAScript regular expression grammar

This page describes the regular expression grammar that is used when `std::basic_regex` is constructed with `syntax_option_type` set to `ECMAScript` (the default). See `syntax_option_type` for the other supported regular expression grammars.

The `ECMAScript` 3 regular expression grammar in C++ is ECMA-262 grammar (https://ecma-international.org/ecma-262/5.1/#sec-15.10) with modifications marked with (C++ only) below.

Overview

The modified regular expression grammar (https://eel.is/c++draft/re.grammar) is mostly ECMAScript RegExp grammar with a POSIX-type expansion on locales under *ClassAtom*. Some clarifications on equality checks and number parsing is made. For many of the examples here, you can try this equivalent in your browser console:

```
function match(s, re) { return s.match(new RegExp(re)); }
```

The "normative references" in the standard specifies ECMAScript 3. We link to the ECMAScript 5.1 spec here because it is a version with only minor changes from ECMAScript 3, and it also has an HTML version. See the MDN Guide on JavaScript RegExp (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions) for an overview on the dialect features.

Alternatives

A regular expression pattern is a sequence of one or more *Alternative*s, separated by the disjunction operator **|** (in other words, the disjunction operator has the lowest precedence).

Pattern ::

Disjunction

Disjunction ::

Alternative
Alternative **|** *Disjunction*

The pattern first tries to skip the *Disjunction* and match the left *Alternative* followed by the rest of the regular expression (after the Disjunction).

If it fails, it tries to skip the left *Alternative* and match the right *Disjunction* (followed by the rest of the regular expression).

If the left *Alternative*, the right *Disjunction*, and the remainder of the regular expression all have choice points, all choices in the remainder of the expression are tried before moving on to the next choice in the left *Alternative*. If choices in the left *Alternative* are exhausted, the right *Disjunction* is tried instead of the left *Alternative*.

Any capturing parentheses inside a skipped *Alternative* produce empty submatches.

Run this code

```
#include <cstddef>
#include <iostream>
#include <regex>
#include <string>

void show_matches(const std::string& in, const std::string& re)
{
    std::smatch m;
    std::regex_search(in, m, std::regex(re));
    if (!m.empty())
    {
        std::cout << "input=[" << in << "], regex=[" << re << "]\n  "
                     "prefix=[" << m.prefix() << "]\n  smatch: ";
        for (std::size_t n = 0; n < m.size(); ++n)
            std::cout << "m[" << n << "]=[" << m[n] << "] ";
        std::cout << "\n  suffix=[" << m.suffix() << "]\n";
    }
    else
```

```
        std::cout << "input=[" << in << "], regex=[" << re << "]: NO MATCH\n";
}

int main()
{
    show_matches("abcdef", "abc|def");
    show_matches("abc", "ab|abc"); // left Alternative matched first

    // Match of the input against the left Alternative (a) followed
    // by the remained of the regex (c|bc) succeeds, which results
    // in m[1]="a" and m[4]="bc".
    // The skipped Alternatives (ab) and (c) leave their submatches
    // m[3] and m[5] empty.
    show_matches("abc", "((a)|(ab))((c)|(bc))");
}
```

Output:

```
input=[abcdef], regex=[abc|def]
  prefix=[]
  smatch: m[0]=[abc]
  suffix=[def]
input=[abc], regex=[ab|abc]
  prefix=[]
  smatch: m[0]=[ab]
  suffix=[c]
input=[abc], regex=[((a)|(ab))((c)|(bc))]
  prefix=[]
  smatch: m[0]=[abc] m[1]=[a] m[2]=[a] m[3]=[] m[4]=[bc] m[5]=[] m[6]=[bc]
  suffix=[]
```

Terms

Each *Alternative* is either empty or is a sequence of *Term*s (with no separators between the *Term*s)

Alternative ::

> *[empty]*
> *Alternative Term*

Empty *Alternative* always matches and does not consume any input.

Consecutive *Term*s try to simultaneously match consecutive portions of the input.

If the left *Alternative*, the right *Term*, and the remainder of the regular expression all have choice points, all choices in the remained of the expression are tried before moving on to the next choice in the right *Term*, and all choices in the right *Term* are tried before moving on to the next choice in the left *Alternative*.

Run Share Exit GCC 13.1 (C++23) Powered by Coliru online compiler

```
#include <iostream>
#include <regex>
#include <string>

void show_matches(const std::string& in, const std::string& re)
{
    std::smatch m;
    std::regex_search(in, m, std::regex(re));
    if (!m.empty())
    {
        std::cout << "input=[" << in << "], regex=[" << re << "]\n  "
                     "prefix=[" << m.prefix() << "]\n  smatch: ";
        for (std::size_t n = 0; n < m.size(); ++n)
            std::cout << "m[" << n << "]=[" << m[n] << "] ";
        std::cout << "\n  suffix=[" << m.suffix() << "]\n";
    }
    else
        std::cout << "input=[" << in << "], regex=[" << re << "]: NO MATCH\n
}
```

```

int main()
{
    show_matches("abcdef", ""); // empty regex is a single empty Alternative
    show_matches("abc", "abc|"); // left Alternative matched first
    show_matches("abc", "|abc"); // left Alternative matched first, leaving a
```

Output:

```
input=[abcdef], regex=[]
  prefix=[]
  smatch: m[0]=[]
  suffix=[abcdef]
input=[abc], regex=[abc|]
  prefix=[]
  smatch: m[0]=[abc]
  suffix=[]
input=[abc], regex=[|abc]
  prefix=[]
  smatch: m[0]=[]
  suffix=[abc]
```

Quantifiers

- Each *Term* is either an *Assertion* (see below), or an *Atom* (see below), or an *Atom* immediately followed by a *Quantifier*

Term ::

Assertion
Atom
Atom Quantifier

Each *Quantifier* is either a *greedy* quantifier (which consists of just one *QuantifierPrefix*) or a *non-greedy* quantifier (which consists of one *QuantifierPrefix* followed by the question mark **?**).

Quantifier ::

QuantifierPrefix
QuantifierPrefix **?**

Each *QuantifierPrefix* determines two numbers: the minimum number of repetitions and the maximum number of repetitions, as follows:

QuantifierPrefix	Minimum	Maximum
*****	zero	infinity
+	one	infinity
?	zero	one
{ *DecimalDigits* **}**	value of DecimalDigits	value of DecimalDigits
{ *DecimalDigits* **, }**	value of DecimalDigits	infinity
{ *DecimalDigits* **,** *DecimalDigits* **}**	value of DecimalDigits before the comma	value of DecimalDigits after the comma

The values of the individual *DecimalDigits* are obtained by calling `std::regex_traits::value`(C++ only) on each of the digits.

An *Atom* followed by a *Quantifier* is repeated the number of times specified by the *Quantifier*. A *Quantifier* can be *non-greedy*, in which case the *Atom* pattern is repeated as few times as possible while still matching the remainder of the regular expression, or it can be *greedy*, in which case the *Atom* pattern is repeated as many times as possible while still matching the remainder of the regular expression.

The *Atom* pattern is what is repeated, not the input that it matches, so different repetitions of the *Atom* can match different input substrings.

If the *Atom* and the remainder of the regular expression all have choice points, the *Atom* is first matched as many (or as few, if *non-greedy*) times as possible. All choices in the remainder of the regular expression are tried before moving on to the next choice in the last repetition of *Atom*. All choices in the last (nth) repetition of *Atom* are tried before moving on to the next choice in the next-to-last (n–1)st repetition of *Atom*; at which point it may turn out that more or fewer repetitions of *Atom* are now possible; these are exhausted (again, starting with either as few or as many as possible) before moving on to the next choice in the (n-1)st repetition of *Atom* and so on.

The *Atom*'s captures are cleared each time it is repeated (see the `"(z)((a+)?(b+)?(c))*"` example below)

Run this code

```
#include <cstddef>
#include <iostream>
#include <regex>
#include <string>

void show_matches(const std::string& in, const std::string& re)
{
    std::smatch m;
    std::regex_search(in, m, std::regex(re));
    if (!m.empty())
    {
        std::cout << "input=[" << in << "], regex=[" << re << "]\n  "
                     "prefix=[" << m.prefix() << "]\n  smatch: ";
        for (std::size_t n = 0; n < m.size(); ++n)
            std::cout << "m[" << n << "]=[" << m[n] << "] ";
        std::cout << "\n  suffix=[" << m.suffix() << "]\n";
    }
    else
        std::cout << "input=[" << in << "], regex=[" << re << "]: NO MATCH\n";
}

int main()
{
    // greedy match, repeats [a-z] 4 times
    show_matches("abcdefghi", "a[a-z]{2,4}");
    // non-greedy match, repeats [a-z] 2 times
    show_matches("abcdefghi", "a[a-z]{2,4}?");

    // Choice point ordering for quantifiers results in a match
    // with two repetitions, first matching the substring "aa",
    // second matching the substring "ba", leaving "ac" not matched
    // ("ba" appears in the capture clause m[1])
    show_matches("aabaac", "(aa|aabaac|ba|b|c)*");

    // Choice point ordering for quantifiers makes this regex
    // calculate the greatest common divisor between 10 and 15
    // (the answer is 5, and it populates m[1] with "aaaaa")
    show_matches("aaaaaaaaaa,aaaaaaaaaaaaaaa", "^(a+)\\1*,\\1+$");

    // the substring "bbb" does not appear in the capture clause m[4]
    // because it is cleared when the second repetition of the atom
    // (a+)?(b+)?(c) is matching the substring "ac"
    // NOTE: gcc gets this wrong - it does not correctly clear the
    // matches[4] capture group as required by ECMA-262 21.2.2.5.1,
    // and thus incorrectly captures "bbb" for that group.
    show_matches("zaacbbbcac", "(z)((a+)?(b+)?(c))*");
}
```

Output:

```
input=[abcdefghi], regex=[a[a-z]{2,4}]
  prefix=[]
  smatch: m[0]=[abcde]
  suffix=[fghi]
input=[abcdefghi], regex=[a[a-z]{2,4}?]
  prefix=[]
  smatch: m[0]=[abc]
  suffix=[defghi]
input=[aabaac], regex=[(aa|aabaac|ba|b|c)*]
  prefix=[]
  smatch: m[0]=[aaba] m[1]=[ba]
  suffix=[ac]
input=[aaaaaaaaaa,aaaaaaaaaaaaaaa], regex=[^(a+)\1*,\1+$]
```

```
  prefix=[]
  smatch: m[0]=[aaaaaaaaaa,aaaaaaaaaaaaaaa] m[1]=[aaaaa]
  suffix=[]
input=[zaacbbbcac], regex=[(z)((a+)?(b+)?(c))*]
  prefix=[]
  smatch: m[0]=[zaacbbbcac] m[1]=[z] m[2]=[ac] m[3]=[a] m[4]=[] m[5]=[c]
  suffix=[]
```

Assertions

*Assertion*s match conditions, rather than substrings of the input string. They never consume any characters from the input. Each *Assertion* is one of the following

Assertion ::

^
$
\ b
\ B
(? = *Disjunction* **)**
(? ! *Disjunction* **)**

The assertion **^** (beginning of line) matches

1) The position that immediately follows a *LineTerminator* character (this may not be supported)(until C++17) (this is only guaranteed if `std::regex_constants::multiline`(C++ only) is enabled)(since C++17)
2) The beginning of the input (unless `std::regex_constants::match_not_bol`(C++ only) is enabled)

The assertion **$** (end of line) matches

1) The position of a *LineTerminator* character (this may not be supported)(until C++17)(this is only guaranteed if `std::regex_constants::multiline`(C++ only) is enabled)(since C++17)
2) The end of the input (unless `std::regex_constants::match_not_eol`(C++ only) is enabled)

In the two assertions above and in the Atom **.** below, *LineTerminator* is one of the following four characters: `U+000A` (`\n` or line feed), `U+000D` (`\r` or carriage return), `U+2028` (line separator), or `U+2029` (paragraph separator)

The assertion **\b** (word boundary) matches

1) The beginning of a word (current character is a letter, digit, or underscore, and the previous character is not)
2) The end of a word (current character is not a letter, digit, or underscore, and the previous character is one of those)
3) The beginning of input if the first character is a letter, digit, or underscore (unless `std::regex_constants::match_not_bow`(C++ only) is enabled)
4) The end of input if the last character is a letter, digit, or underscore (unless `std::regex_constants::match_not_eow`(C++ only) is enabled)

The assertion **\B** (negative word boundary) matches everything EXCEPT the following

1) The beginning of a word (current character is a letter, digit, or underscore, and the previous character is not one of those or does not exist)
2) The end of a word (current character is not a letter, digit, or underscore (or the matcher is at the end of input), and the previous character is one of those)

The assertion **(? =** *Disjunction* **)** (zero-width positive lookahead) matches if *Disjunction* would match the input at the current position

The assertion **(? !** *Disjunction* **)** (zero-width negative lookahead) matches if *Disjunction* would NOT match the input at the current position.

For both Lookahead assertions, when matching the *Disjunction*, the position is not advanced before matching the remainder of the regular expression. Also, if *Disjunction* can match at the current position in several ways, only the first one is tried.

ECMAScript forbids backtracking into the lookahead Disjunctions, which affects the behavior of backreferences into a positive lookahead from the remainder of the regular expression (see example below). Backreferences into the negative lookahead from the rest of the regular expression are always undefined (since the lookahead Disjunction must fail to proceed).

Note: Lookahead assertions may be used to create logical AND between multiple regular expressions (see example below).

Run this code

```
#include <cstddef>
#include <iostream>
#include <regex>
#include <string>

void show_matches(const std::string& in, const std::string& re)
{
    std::smatch m;
    std::regex_search(in, m, std::regex(re));
    if (!m.empty())
    {
        std::cout << "input=[" << in << "], regex=[" << re << "]\n  "
                     "prefix=[" << m.prefix() << "]\n  smatch: ";
        for (std::size_t n = 0; n < m.size(); ++n)
            std::cout << "m[" << n << "]=[" << m[n] << "] ";
        std::cout << "\n  suffix=[" << m.suffix() << "]\n";
    }
    else
        std::cout << "input=[" << in << "], regex=[" << re << "]: NO MATCH\n";
}

int main()
{
    // matches the a at the end of input
    show_matches("aaa", "a$");

    // matches the o at the end of the first word
    show_matches("moo goo gai pan", "o\\b");

    // the lookahead matches the empty string immediately after the first b
    // this populates m[1] with "aaa" although m[0] is empty
    show_matches("baaabac", "(?=(a+))");

    // because backtracking into lookaheads is prohibited,
    // this matches aba rather than aaaba
    show_matches("baaabac", "(?=(a+))a*b\\1");

    // logical AND via lookahead: this password matches IF it contains
    // at least one lowercase letter
    // AND at least one uppercase letter
    // AND at least one punctuation character
    // AND be at least 6 characters long
    show_matches("abcdef", "(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}");
    show_matches("aB,def", "(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}");
}
```

Output:

```
input=[aaa], regex=[a$]
  prefix=[aa]
  smatch: m[0]=[a]
  suffix=[]
input=[moo goo gai pan], regex=[o\b]
  prefix=[mo]
  smatch: m[0]=[o]
  suffix=[ goo gai pan]
input=[baaabac], regex=[(?=(a+))]
  prefix=[b]
  smatch: m[0]=[] m[1]=[aaa]
  suffix=[aaabac]
input=[baaabac], regex=[(?=(a+))a*b\1]
  prefix=[baa]
  smatch: m[0]=[aba] m[1]=[a]
  suffix=[c]
input=[abcdef], regex=[(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}]: NO MATCH
input=[aB,def], regex=[(?=.*[[:lower:]])(?=.*[[:upper:]])(?=.*[[:punct:]]).{6,}]
```

```
prefix=[]
smatch: m[0]=[aB,def]
suffix=[]
```

Atoms

An *Atom* can be one of the following:

Atom ::

PatternCharacter
.
**** *AtomEscape*
CharacterClass
(*Disjunction* **)**
(? : *Disjunction* **)**

where *AtomEscape* ::

DecimalEscape
CharacterEscape
CharacterClassEscape

Different kinds of atoms evaluate differently.

Sub-expressions

The *Atom* **(** *Disjunction* **)** is a marked subexpression: it executes the *Disjunction* and stores the copy of the input substring that was consumed by *Disjunction* in the submatch array at the index that corresponds to the number of times the left open parenthesis **(** of marked subexpressions has been encountered in the entire regular expression at this point.

Besides being returned in the `std::match_results`, the captured submatches are accessible as backreferences (`\1`, `\2`, ...) and can be referenced in regular expressions. Note that `std::regex_replace` uses `$` instead of `\` for backreferences (`$1`, `$2`, ...) in the same manner as `String.prototype.replace` (ECMA-262, part 15.5.4.11).

The *Atom* **(? :** *Disjunction* **)** (non-marking subexpression) simply evaluates the *Disjunction* and does not store its results in the submatch. This is a purely lexical grouping.

> This section is incomplete
> Reason: no example

Backreferences

DecimalEscape ::

DecimalIntegerLiteral [*lookahead* ∉ *DecimalDigit*]

If **** is followed by a decimal number `N` whose first digit is not `0`, then the escape sequence is considered to be a *backreference*. The value `N` is obtained by calling `std::regex_traits::value`(C++ only) on each of the digits and combining their results using base-10 arithmetic. It is an error if `N` is greater than the total number of left capturing parentheses in the entire regular expression.

When a backreference `\N` appears as an *Atom*, it matches the same substring as what is currently stored in the N'th element of the submatch array.

The decimal escape `\0` is NOT a backreference: it is a character escape that represents the `NUL` character. It cannot be followed by a decimal digit.

As above, note that `std::regex_replace` uses `$` instead of `\` for backreferences (`$1`, `$2`, ...).

> This section is incomplete
> Reason: no example

Single character matches

The *Atom* **.** matches and consumes any one character from the input string except for *LineTerminator* (`U+000D`, `U+000A`, `U+2029`, or `U+2028`)

The *Atom PatternCharacter*, where *PatternCharacter* is any *SourceCharacter* EXCEPT the characters **`^ $ \ . * + ? ( ) [ ] { } |`**, matches and consumes one character from the input if it is equal to this *PatternCharacter*.

The equality for this and all other single character matches is defined as follows:

1) If `std::regex_constants::icase` is set, the characters are equal if the return values of `std::regex_traits::translate_nocase` are equal (C++ only).
2) Otherwise, if `std::regex_constants::collate` is set, the characters are equal if the return values of `std::regex_traits::translate` are equal (C++ only).
3) Otherwise, the characters are equal if `operator==` returns `true`.

Each *Atom* that consists of the escape character \ followed by *CharacterEscape* as well as the special DecimalEscape **`\0`**, matches and consumes one character from the input if it is equal to the character represented by the *CharacterEscape*. The following character escape sequences are recognized:

CharacterEscape ::

ControlEscape
`c` *ControlLetter*
HexEscapeSequence
UnicodeEscapeSequence
IdentityEscape

Here, *ControlEscape* is one of the following five characters: **`f n r t v`**

ControlEscape	Code Unit	Name
f	U+000C	form feed
n	U+000A	new line
r	U+000D	carriage return
t	U+0009	horizontal tab
v	U+000B	vertical tab

ControlLetter is any lowercase or uppercase ASCII letters and this character escape matches the character whose code unit equals the remainder of dividing the value of the code unit of *ControlLetter* by `32`. For example, **`\cD`** and **`\cd`** both match code unit U+0004 (EOT) because 'D' is U+0044 and `0x44 % 32 == 4`, and 'd' is U+0064 and `0x64 % 32 == 4`.

HexEscapeSequence is the letter **`x`** followed by exactly two *HexDigit*s (where *HexDigit* is one of **`0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`**). This character escape matches the character whose code unit equals the numeric value of the two-digit hexadecimal number.

UnicodeEscapeSequence is the letter **`u`** followed by exactly four *HexDigit*s. This character escape matches the character whose code unit equals the numeric value of this four-digit hexadecimal number. If the value does not fit in this `std::basic_regex`'s `CharT`, `std::regex_error` is thrown (C++ only).

IdentityEscape can be any non-alphanumeric character: for example, another backslash. It matches the character as-is.

Run this code

```
#include <cstddef>
#include <iostream>
#include <regex>
#include <string>

void show_matches(const std::wstring& in, const std::wstring& re)
{
    std::wsmatch m;
    std::regex_search(in, m, std::wregex(re));
    if (!m.empty())
    {
        std::wcout << L"input=[" << in << L"], regex=[" << re << L"]\n  "
                      L"prefix=[" << m.prefix() << L"]\n  wsmatch: ";
        for (std::size_t n = 0; n < m.size(); ++n)
            std::wcout << L"m[" << n << L"]=[" << m[n] << L"] ";
        std::wcout << L"\n  suffix=[" << m.suffix() << L"]\n";
    }
    else
        std::wcout << L"input=[" << in << "], regex=[" << re << L"]: NO MATCH\n";
}

int main()
```

```
{
    // Most escapes are similar to C++, save for metacharacters. You will have to
    // double-escape or use raw strings on the slashes though.
    show_matches(L"C++\\", LR"(C\+\+\\)");

    // Escape sequences and NUL.
    std::wstring s(L"ab\xff\0cd", 5);
    show_matches(s, L"(\\0|\\u00ff)");

    // No matching for non-BMP Unicode is defined, because ECMAScript uses UTF-16
    // atoms. Whether this emoji banana matches can be platform dependent:
    // These need to be wide-strings!
    show_matches(L"\U0001f34c", L"[\\u0000-\\ufffe]+");
}
```

Possible output:

```
input=[C++\], regex=[C\+\+\\]
  prefix=[]
  wsmatch: m[0]=[C++\]
  suffix=[]
input=[ab?c], regex=[(\0{{!}}\u00ff)]
  prefix=[ab]
  wsmatch: m[0]=[?] m[1]=[?]
  suffix=[c]
input=[?], regex=[[\u0000-\ufffe]+]: NO MATCH
```

Character classes

An Atom can represent a character class, that is, it will match and consume one character if it belongs to one of the predefined groups of characters.

A character class can be introduced through a character class escape:

Atom ::

\ *CharacterClassEscape*

or directly

Atom ::

CharacterClass

The character class escapes are shorthands for some of the common characters classes, as follows:

CharacterClassEscape	ClassName expression(C++ only)	Meaning
d	**[[:digit:]]**	digits
D	**[^[:digit:]]**	non-digits
s	**[[:space:]]**	whitespace characters
S	**[^[:space:]]**	non-whitespace characters
w	**[_[:alnum:]]**	alphanumeric characters and the character **_**
W	**[^_[:alnum:]]**	characters other than alphanumeric or **_**

The exact meaning of each of these character class escapes in C++ is defined in terms of the locale-dependent named character classes, and not by explicitly listing the acceptable characters as in ECMAScript.

A *CharacterClass* is a bracket-enclosed sequence of *ClassRanges*, optionally beginning with the negation operator **^**. If it begins with **^**, this *Atom* matches any character that is NOT in the set of characters represented by the union of all *ClassRanges*. Otherwise, this *Atom* matches any character that IS in the set of the characters represented by the union of all *ClassRanges*.

CharacterClass ::

[*[lookahead ∉ {*^*}] ClassRanges* **]**
[^ *ClassRanges* **]**

ClassRanges ::

[empty]
NonemptyClassRanges

NonemptyClassRanges ::

ClassAtom
ClassAtom NonemptyClassRangesNoDash
ClassAtom - ClassAtom ClassRanges

If non-empty class range has the form ***`ClassAtom - ClassAtom`***, it matches any character from a range defined as follows: (C++ only)

The first *ClassAtom* must match a single collating element `c1` and the second *ClassAtom* must match a single collating element `c2`. To test if the input character `c` is matched by this range, the following steps are taken:

1) If `std::regex_constants::collate` is not on, the character is matched by direct comparison of code points: `c` is matched if `c1 <= c && c <= c2`
1) Otherwise (if `std::regex_constants::collate` is enabled):
   1) If `std::regex_constants::icase` is enabled, all three characters (`c`, `c1`, and `c2`) are passed `std::regex_traits::translate_nocase`
   2) Otherwise (if `std::regex_constants::icase` is not set), all three characters (`c`, `c1`, and `c2`) are passed `std::regex_traits::translate`
2) The resulting strings are compared using `std::regex_traits::transform` and the character `c` is matched if `transformed c1 <= transformed c && transformed c <= transformed c2`

The character **`-`** is treated literally if it is

- the first or last character of *ClassRanges*
- the beginning or end ClassAtom of a dash-separated range specification
- immediately follows a dash-separated range specification.
- escaped with a backslash as a *CharacterEscape*

NonemptyClassRangesNoDash ::

ClassAtom
ClassAtomNoDash NonemptyClassRangesNoDash
ClassAtomNoDash - ClassAtom ClassRanges

ClassAtom ::

`-`
ClassAtomNoDash
ClassAtomExClass(C++ only)
ClassAtomCollatingElement(C++ only)
ClassAtomEquivalence(C++ only)

ClassAtomNoDash ::

SourceCharacter but not one of **`\ or ] or -`**
`\` *ClassEscape*

Each *ClassAtomNoDash* represents a single character -- either *SourceCharacter* as-is or escaped as follows:

ClassEscape ::

DecimalEscape
`b`
CharacterEscape
CharacterClassEscape

The special *ClassEscape* **`\b`** produces a character set that matches the code unit U+0008 (backspace). Outside of *CharacterClass*, it is the word-boundary *Assertion*.

The use of **`\B`** and the use of any backreference (*DecimalEscape* other than zero) inside a *CharacterClass* is an error.

The characters **`-`** and **`]`** may need to be escaped in some situations in order to be treated as atoms. Other characters that have special meaning outside of *CharacterClass*, such as **`*`** or **`?`**, do not need to be escaped.

This section is incomplete
Reason: no example

POSIX-based character classes

These character classes are an extension to the ECMAScript grammar, and are equivalent to character classes found in the POSIX regular expressions.

ClassAtomExClass(C++ only) ::

[: *ClassName* **:]**

Represents all characters that are members of the named character class *ClassName*. The name is valid only if `std::regex_traits::lookup_classname` returns non-zero for this name. As described in `std::regex_traits::lookup_classname`, the following names are guaranteed to be recognized: **`alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit, d, s, w`**. Additional names may be provided by system-supplied locales (such as **`jdigit`** or **`jkanji`** in Japanese) or implemented as a user-defined extension.

ClassAtomCollatingElement(C++ only) ::

[. *ClassName* **.]**

Represents the named collating element, which may represent a single character or a sequence of characters that collates as a single unit under the imbued locale, such as `[.tilde.]` or `[.ch.]` in Czech. The name is valid only if `std::regex_traits::lookup_collatename` is not an empty string.

When using `std::regex_constants::collate`, collating elements can always be used as ends points of a range (e.g. `[[.dz.]-g]` in Hungarian).

ClassAtomEquivalence(C++ only) ::

[= *ClassName* **=]**

Represents all characters that are members of the same equivalence class as the named collating element, that is, all characters whose primary collation key is the same as that for collating element *ClassName*. The name is valid only if `std::regex_traits::lookup_collatename` for that name is not an empty string and if the value returned by `std::regex_traits::transform_primary` for the result of the call to `std::regex_traits::lookup_collatename` is not an empty string.

A primary sort key is one that ignores case, accentuation, 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.`

ClassName(C++ only) ::

ClassNameCharacter
ClassNameCharacter ClassName

ClassNameCharacter(C++ only) ::

SourceCharacter but not one of **`. = :`**

This section is incomplete
Reason: no example