

Lookahead and Lookbehind Zero-Length Assertions

Lookahead and lookbehind, collectively called “lookaround”, are zero-length assertions just like the [start and end of line](#), and [start and end of word](#) anchors explained earlier in this tutorial. The difference is that lookaround actually matches characters, but then gives up the match, returning only the result: match or no match. That is why they are called “assertions”. They do not consume characters in the string, but only assert whether a match is possible or not. Lookaround allows you to create regular expressions that are impossible to create without them, or that would get very longwinded without them.

Positive and Negative Lookahead

Negative lookahead is indispensable if you want to match something not followed by something else. When explaining [character classes](#), this tutorial explained why you cannot use a negated character class to match a `q` not followed by a `u`. Negative lookahead provides the solution: `q(?:!u)`. The negative lookahead construct is the pair of parentheses, with the opening parenthesis followed by a question mark and an exclamation point. Inside the lookahead, we have the trivial regex `u`.

Positive lookahead works just the same. `q(?:=u)` matches a `q` that is followed by a `u`, without making the `u` part of the match. The positive lookahead construct is a pair of parentheses, with the opening parenthesis followed by a question mark and an equals sign.

You can use any regular expression inside the lookahead (but not lookbehind, as explained below). Any valid regular expression can be used inside the lookahead. If it contains [capturing groups](#) then those groups will capture as normal and backreferences to them will work normally, even outside the lookahead. (The only exception is [Tcl](#), which treats all groups inside lookahead as non-capturing.) The lookahead itself is not a capturing group. It is not included in the count towards numbering the backreferences. If you want to store the match of the regex inside a lookahead, you have to put capturing parentheses around the regex inside the lookahead, like this: `(?=(regex))`. The other way around will not work, because the lookahead will already have discarded the regex match by the time the capturing group is to store its match.

Regex Engine Internals

First, let's see how the engine applies `q(?:!u)` to the string `Iraq`. The first token in the regex is the [literal](#) `q`. As we already know, this causes the engine to traverse the string until the `q` in the string is matched. The position in the string is now the void after the string. The next token is the lookahead. The engine takes note that it is inside a lookahead construct now, and begins matching the regex inside the lookahead. So the next token is `u`. This does not match the void after the string. The engine notes that the regex inside the lookahead failed. Because the lookahead is negative, this means that the lookahead has successfully matched at the current position. At this point, the entire regex has matched, and `q` is returned as the match.

Let's try applying the same regex to `quit`. `q` matches `q`. The next token is the `u` inside the lookahead. The next character is the `u`. These match. The engine advances to the next character: `i`. However, it is done with the regex inside the lookahead. The engine notes success, and discards the regex match. This causes the engine to step back in the string to `u`.

Because the lookahead is negative, the successful match inside it causes the lookahead to fail. Since there are no other permutations of this regex, the engine has to start again at the beginning. Since `q` cannot match anywhere else, the engine reports failure.

Let's take one more look inside, to make sure you understand the implications of the lookahead. Let's apply `q(?=u)i` to `quit`. The lookahead is now positive and is followed by another token. Again, `q` matches `q` and `u` matches `u`. Again, the match from the lookahead must be discarded, so the engine steps back from `i` in the string to `u`. The lookahead was successful, so the engine continues with `i`. But `i` cannot match `u`. So this match attempt fails. All remaining attempts fail as well, because there are no more `q`'s in the string.

The regex `q(?=u)i` can never match anything. It tries to match `u` and `i` at the same position. If there is a `u` immediately after the `q` then the lookahead succeeds but then `i` fails to match `u`. If there is anything other than a `u` immediately after the `q` then the lookahead fails.

Positive and Negative Lookbehind

Lookbehind has the same effect, but works backwards. It tells the regex engine to temporarily step backwards in the string, to check if the text inside the lookbehind can be matched there. `(?<!a)b` matches a "b" that is not preceded by an "a", using negative lookbehind. It doesn't match `cab`, but matches the `b` (and only the `b`) in `bed` or `debt`. `(?<=a)b` (positive lookbehind) matches the `b` (and only the `b`) in `cab`, but does not match `bed` or `debt`.

The construct for positive lookbehind is `(?<=text)`: a pair of parentheses, with the opening parenthesis followed by a question mark, "less than" symbol, and an equals sign. Negative lookbehind is written as `(?<!\text)`, using an exclamation point instead of an equals sign.

More Regex Engine Internals

Let's apply `(?<=a)b` to `thingamabob`. The engine starts with the lookbehind and the first character in the string. In this case, the lookbehind tells the engine to step back one character, and see if `a` can be matched there. The engine cannot step back one character because there are no characters before the `t`. So the lookbehind fails, and the engine starts again at the next character, the `h`. (Note that a negative lookbehind would have succeeded here.) Again, the engine temporarily steps back one character to check if an "a" can be found there. It finds a `t`, so the positive lookbehind fails again.

The lookbehind continues to fail until the regex reaches the `m` in the string. The engine again steps back one character, and notices that the `a` can be matched there. The positive lookbehind matches. Because it is zero-length, the current position in the string remains at the `m`. The next token is `b`, which cannot match here. The next character is the second `a` in the string. The engine steps back, and finds out that the `m` does not match `a`.

The next character is the first `b` in the string. The engine steps back and finds out that `a` satisfies the lookbehind. `b` matches `b`, and the entire regex has been matched successfully. It matches one character: the first `b` in the string.

Important Notes About Lookbehind

The good news is that you can use lookbehind anywhere in the regex, not only at the start. If you want to find a word not ending with an "s", you could use `\b\w+(?<!s)\b`. This is definitely not the same as `\b\w+[!s]\b`. When applied to `John's`, the former matches `John` and the latter matches `John'` (including the apostrophe). I will leave it up to you to figure out why. (Hint: `\b` matches between the apostrophe and the `s`). The latter also doesn't match single-letter words like "a" or "I". The correct regex without using lookbehind is `\b\w*[!s\W]\b` (star instead of plus, and `\W` in the character class). Personally, I find the lookbehind easier to understand. The last regex, which works correctly, has a double negation (the `\W` in the negated character class). Double negations tend to be

confusing to humans. Not to regex engines, though. (Except perhaps for Tcl, which treats negated shorthands in negated character classes as an error.)

The bad news is that most regex flavors do not allow you to use just any regex inside a lookbehind, because they cannot apply a regular expression backwards. The regular expression engine needs to be able to figure out how many characters to step back before checking the lookbehind. When evaluating the lookbehind, the regex engine determines the length of the regex inside the lookbehind, steps back that many characters in the subject string, and then applies the regex inside the lookbehind from left to right just as it would with a normal regex.

Many regex flavors, including those used by [Perl](#), [Python](#), and [Boost](#) only allow fixed-length strings. You can use [literal text](#), [character escapes](#), [Unicode escapes](#) other than `\x`, and [character classes](#). You cannot use [quantifiers](#) or [backreferences](#). You can use [alternation](#), but only if all alternatives have the same length. These flavors evaluate lookbehind by first stepping back through the subject string for as many characters as the lookbehind needs, and then attempting the regex inside the lookbehind from left to right.

Perl 5.30 supports variable-length lookbehind as an experimental feature. But there are many cases in which it does not work correctly. So in practice, the above is still true for Perl 5.30.

[PCRE](#) is not fully Perl-compatible when it comes to lookbehind. While Perl requires alternatives inside lookbehind to have the same length, PCRE allows alternatives of variable length. [PHP](#), [Delphi](#), [R](#), and [Ruby](#) also allow this. Each alternative still has to be fixed-length. Each alternative is treated as a separate fixed-length lookbehind.

[Java](#) takes things a step further by allowing finite repetition. You can use the [question mark](#) and the [curly braces](#) with the *max* parameter specified. Java determines the minimum and maximum possible lengths of the lookbehind. The lookbehind in the regex `(?<!ab{2,4}c{3,5}d)test` has 5 possible lengths. It can be from 7 through 11 characters long. When Java (version 6 or later) tries to match the lookbehind, it first steps back the minimum number of characters (7 in this example) in the string and then evaluates the regex inside the lookbehind as usual, from left to right. If it fails, Java steps back one more character and tries again. If the lookbehind continues to fail, Java continues to step back until the lookbehind either matches or it has stepped back the maximum number of characters (11 in this example). This repeated stepping back through the subject string kills performance when the number of possible lengths of the lookbehind grows. Keep this in mind. Don't choose an arbitrarily large maximum number of repetitions to work around the lack of infinite quantifiers inside lookbehind. Java 4 and 5 have bugs that cause lookbehind with alternation or variable quantifiers to fail when it should succeed in some situations. These bugs were fixed in Java 6.

Java 13 allows you to use the [star](#) and [plus](#) inside lookbehind, as well as [curly braces](#) without an upper limit. But Java 13 still uses the laborious method of matching lookbehind introduced with Java 6. Java 13 also does not correctly handle lookbehind with multiple quantifiers if one of them is unbounded. In some situations you may get an error. In other situations you may get incorrect matches. So for both correctness and performance, we recommend you only use quantifiers with a low upper bound in lookbehind with Java 6 through 13.

The only regex engines that allow you to use a full regular expression inside lookbehind, including infinite repetition and backreferences, are the [JGsoft engine](#) and the [.NET framework RegEx classes](#). These regex engines really apply the regex inside the lookbehind backwards, going through the regex inside the lookbehind and through the subject string from right to left. They only need to evaluate the lookbehind once, regardless of how many different possible lengths it has.

Finally, flavors like [std::regex](#) and [Tcl](#) do not support lookbehind at all, even though they do support lookahead. [JavaScript](#) was like that for the longest time since its inception. But now lookbehind is part of the ECMAScript 2018 specification. As of this writing (late 2019), Google's Chrome browser is the only popular JavaScript implementation that supports lookbehind. So if cross-browser compatibility matters, you can't use lookbehind in JavaScript.

Lookaround Is Atomic

The fact that lookahead is zero-length automatically makes it [atomic](#). As soon as the lookahead condition is satisfied, the regex engine forgets about everything inside the lookahead. It will not backtrack inside the lookahead to try different permutations.

The only situation in which this makes any difference is when you use [capturing groups](#) inside the lookahead. Since the regex engine does not backtrack into the lookahead, it will not try different permutations of the capturing groups.

For this reason, the regex `(?=(\d+)\w+)\1` never matches `123x12`. First the lookahead captures `123` into `\1`. `\w+` then matches the whole string and backtracks until it matches only `1`. Finally, `\w+` fails since `\1` cannot be matched at any position. Now, the regex engine has nothing to backtrack to, and the overall regex fails. The backtracking steps created by `\d+` have been discarded. It never gets to the point where the lookahead captures only `12`.

Obviously, the regex engine does try further positions in the string. If we change the subject string, the regex `(?=(\d+)\w+)\1` does match `56x56` in `456x56`.

If you don't use capturing groups inside lookahead, then all this doesn't matter. Either the lookahead condition can be satisfied or it cannot be. In how many ways it can be satisfied is irrelevant.

Make a Donation

Did this website just save you a trip to the bookstore? Please [make a donation](#) to support this site, and you'll get a **lifetime of advertisement-free access** to this site!

Page URL: <https://www.regular-expressions.info/lookaround.html>

Page last updated: 09 March 2020

Site last updated: 08 April 2020

Copyright © 2003-2020 Jan Goyvaerts. All rights reserved.