If-Then-Else Conditionals in Regular Expressions

A special construct `(?ifthen|else)` allows you to create conditional regular expressions. If the *if* part evaluates to true, then the regex engine will attempt to match the *then* part. Otherwise, the *else* part is attempted instead. The syntax consists of a pair of parentheses. The opening bracket must be followed by a question mark, immediately followed by the *if* part, immediately followed by the *then* part. This part can be followed by a vertical bar and the *else* part. You may omit the *else* part, and the vertical bar with it.

For the *if* part, you can use the lookahead and lookbehind constructs. Using positive lookahead, the syntax becomes `(?(?=regex)then|else)`. Because the lookahead has its own parentheses, the *if* and *then* parts are clearly separated.

Remember that the lookaround constructs do not consume any characters. If you use a lookahead as the *if* part, then the regex engine will attempt to match the *then* or *else part* (depending on the outcome of the lookahead) at the same position where the *if* was attempted.

Alternatively, you can check in the *if* part whether a capturing group has taken part in the match thus far. Place the number of the capturing group inside parentheses, and use that as the if part. Note that although the syntax for a conditional check on a backreference is the same as a number inside a capturing group, no capturing group is created. The number and the parentheses are part of the if-then-else syntax started with `(?`.

For the *then* and *else*, you can use any regular expression. If you want to use alternation, you will have to group the *then* or *else* together using parentheses, like in `(?(?=condition)(then1|then2|then3)|(else1|else2|else3))`. Otherwise, there is no need to use parentheses around the *then* and *else* parts.

Looking Inside The Regex Engine

The regex `(a)?b(?(1)c|d)` consists of the optional capturing group `(a)?`, the literal `b`, and the conditional `(?(1)c|d)` that tests the capturing group. This regex matches `bd` and `abc`. It does not match `bc`, but does match `bd` in `abd`. Let's see how this regular expression works on each of these four subject strings.

When applied to `bd`, `a` fails to match. Since the capturing group containing `a` is optional, the engine continues with `b` at the start of the subject string. Since the whole group was optional, the group did not take part in the match. Any subsequent backreference to it like `\1` will fail. Note that `(a)?` is very different from `(a?)`. In the former regex, the capturing group does not take part in the match if `a` fails, and backreferences to the group will fail. In the latter group, the capturing group always takes part in the match, capturing either `a` or nothing. Backreferences to a capturing group that took part in the match and captured nothing always succeed. Conditionals evaluating such groups execute the "then" part. In short: if you want to use a reference to a group in a conditional, use `(a)?` instead of `(a?)`.

Continuing with our regex, `b` matches `b`. The regex engine now evaluates the conditional. The first capturing group did not take part in the match at all, so the "else" part or `d` is attempted. `d` matches `d` and an overall match is found.

Moving on to our second subject string `abc`, `a` matches `a`, which is captured by the capturing group. Subsequently, `b` matches `b`. The regex engine again evaluates the conditional. The capturing group took part in the match, so the "then" part or `c` is attempted. `c` matches `c` and an overall match is found.

Our third subject `bc` does not start with `a`, so the capturing group does not take part in the match attempt, like we saw with the first subject string. `b` still matches `b`, and the engine moves on to the conditional. The first capturing group did not take part in the match at all, so the "else" part or `d` is attempted. `d` does not match `c` and the match

attempt at the start of the string fails. The engine does try again starting at the second character in the string, but fails since `b` does not match `c`.

The fourth subject `abd` is the most interesting one. Like in the second string, the capturing group grabs the `a` and the `b` matches. The capturing group took part in the match, so the "then" part or `c` is attempted. `c` fails to match `d`, and the match attempt fails. Note that the "else" part is not attempted at this point. The capturing group took part in the match, so only the "then" part is used. However, the regex engine isn't done yet. It restarts the regular expression from the beginning, moving ahead one character in the subject string.

Starting at the second character in the string, `a` fails to match `b`. The capturing group does not take part in the second match attempt which started at the second character in the string. The regex engine moves beyond the optional group, and attempts `b`, which matches. The regex engine now arrives at the conditional in the regex, and at the third character in the subject string. The first capturing group did not take part in the current match attempt, so the "else" part or `d` is attempted. `d` matches `d` and an overall match `bd` is found.

If you want to avoid this last match result, you need to use anchors. `^(a)?b(?(1)c|d)$` does not find any matches in the last subject string. The caret fails to match before the second and third characters in the string.

Named and Relative Conditionals

Conditionals are supported by the JGsoft engine, Perl, PCRE, Python, and the .NET framework. Ruby supports them starting with version 2.0. Languages such as Delphi, PHP, and R that have regex features based on PCRE also support conditionals.

All these flavors also support named capturing groups. You can use the name of a capturing group instead of its number as the *if* test. The syntax is slightly inconsistent between regex flavors. In Python, .NET, and the JGsoft applications, you simply specify the name of the group between parentheses. `(?<test>a)?b(?(test)c|d)` is the regex from the previous section using named capture. In Perl or Ruby, you have to put angle brackets or quotes around the name of the group, and put that between the conditional's parentheses: `(?<test>a)?b(?(<test>)c|d)` or `(?'test'a)?b(?('test')c|d)`. PCRE supports all three variants.

PCRE 7.2 and later and JGsoft V2 also support relative conditionals. The syntax is the same as that of a conditional that references a numbered capturing group with an added plus or minus sign before the group number. The conditional then counts the opening parentheses to the left (minus) or to the right (plus) starting at the `(?(` that opens the conditional. `(a)?b(?(-1)c|d)` is another way of writing the above regex. The benefit is that this regex won't break if you add capturing groups at the start or the end of the regex.

Python supports conditionals using a numbered or named capturing group. Python does not support conditionals using lookaround, even though Python does support lookaround outside conditionals. Instead of a conditional like `(?(?=regex)then|else)`, you can alternate two opposite lookarounds: `(?=regex)then|(?!regex)else`.

Conditionals Referencing Non-Existent Capturing Groups

Boost and Ruby treat a conditional that references a non-existent capturing group as an error. The latest versions of all other flavors discussed in this tutorial don't. They simply let such conditionals always attempt the "else" part. A few flavors changed their minds, though. Python 3.4 and prior and PCRE 7.6 and prior (and thus PHP 5.2.5 and prior) used to treat them as errors.

Example: Extract Email Headers

The regex `^((From|To)|Subject): ((?(2)\w+@\w+\.[a-z]+|.+))` extracts the From, To, and Subject headers from an email message. The name of the header is captured into the first backreference. If the header is the From or To header, it is captured into the second backreference as well.

The second part of the pattern is the if-then-else conditional `(?(2)\w+@\w+\.[a-z]+|.+)`. The *if* part checks whether the second capturing group took part in the match thus far. It will have taken part if the header is the From or To header. In that case, the *then* part of the conditional `\w+@\w+\.[a-z]+` tries to match an email address. To keep the example simple, we use an overly simple regex to match the email address, and we don't try to match the display name that is usually also part of the From or To header.

If the second capturing group did not participate in the match this far, the *else* part `.+` is attempted instead. This simply matches the remainder of the line, allowing for any test subject.

Finally, we place an extra pair of parentheses around the conditional. This captures the contents of the email header matched by the conditional into the third backreference. The conditional itself does not capture anything. When implementing this regular expression, the first capturing group will store the name of the header ("From", "To", or "Subject"), and the third capturing group will store the value of the header.

You could try to match even more headers by putting another conditional into the "else" part. E.g. `^((From|To)|(Date)|Subject): ((?(2)\w+@\w+\.[a-z]+|(?(3)mm/dd/yyyy|.+)))` would match a "From", "To", "Date" or "Subject", and use the regex `mm/dd/yyyy` to check whether the date is valid. Obviously, the date validation regex is just a dummy to keep the example simple. The header is captured in the first group, and its validated contents in the fourth group.

As you can see, regular expressions using conditionals quickly become unwieldy. I recommend that you only use them if one regular expression is all your tool allows you to use. When programming, you're far better off using the regex `^(From|To|Date|Subject): (.+)` to capture one header with its unvalidated contents. In your source code, check the name of the header returned in the first capturing group, and then use a second regular expression to validate the contents of the header returned in the second capturing group of the first regex. Though you'll have to write a few lines of extra code, this code will be much easier to understand and maintain. If you precompile all the regular expressions, using multiple regular expressions will be just as fast, if not faster, than the one big regex stuffed with conditionals.

Make a Donation

Page URL: https://www.regular-expressions.info/conditional.html
Page last updated: 22 November 2019
Site last updated: 08 April 2020