Mastering Conditional Regex

Conditionals are one of the least used components of regex syntax. Granted, not all engines support them. But in my view, the main reason for the low use of conditionals is that the situations in which they do a better job than alternate constructs is poorly known. This page aims to explain the details of regex conditional syntax and to present the typical situations where using conditionals makes sense.

Jumping Points
For easy navigation, here are some jumping points to various sections of the page:

(direct link)

Support for Regex Conditionals

You can use conditionals in the following engines:

✽ .NET (C#, VB.NET etc.)
✽ PCRE (C, PHP, R…)
✽ Perl
✽ Python
✽ Ruby 2+

Some of these engines are able to test a richer set of conditions than others. We'll see these differences as we go.

(direct link)

Basic Conditional Syntax

The regex conditional is an *IF…THEN…ELSE* construct. Its basic form is this:

`(?(A)X|Y)`
This means "if proposition *A* is true, then match pattern X; otherwise, match pattern Y."

Often, you don't need the *ELSE* case or the *THEN* case:

✽ `(?(A)X)` says "if proposition *A* is true, then match pattern *X*." `(?(A)X|)` means the same — but the alternation bar can be dropped.

✽ `(?(A)|X)` amounts to saying "if proposition *A* is **not** true, then match pattern *X*." If you translate the *IF…THEN…ELSE* construction literally, it says "if proposition *A* is true, then match the empty string (which always matches at every position), otherwise match pattern *X*."

Proposition *A*
Proposition *A* can be one of several kinds of assertions that the regex engine can test and determine to be true or false. These various kinds of assertions are expressed by small variations in the conditional syntax. Proposition *A* can assert that:

✽ a numbered capture group has been set
✽ a named capture group has been set
✽ a capture group at a relative position to the current position in the pattern has been set
✽ a lookaround has been successful
✽ a subroutine call has been made
✽ a recursive call has been made
✽ embedded code evaluates to TRUE

(direct link)

Checking if a Numbered Capture Group has been Set

To check if a numbered capture group has been set, we use something like:

`(?(1)foo|bar)`
In this exact pattern, if Group 1 has been set, the engine must match the literal characters *foo*. If not, it must match the literal characters *bar*. But the alternation can contain any regex pattern, for instance (?(1)\d{2}\b|\d{3}\b)

A realistic use of a conditional that checks whether a group has been set would be something like this:

```
^(START)?\d+(?(1)END|\b)
```

Here is how this works:

✽ The ^ anchor asserts that the current position is the beginning of the string
✽ The parentheses around (START) capture the string *START* to Group 1, but the ? "zero-or-one" quantifier makes the capture optional
✽ \d+ matches one or more digits
✽ The conditional (?(1)END|\b) checks whether Group 1 has been set (i.e., whether *START* has been matched). If so, the engine must match *END*. If not, the engine must match a word boundary.

The net result is that the pattern matches digits that are either embedded within *START…END* at the beginning of the string, or standing by themselves at the beginning of the string.

To achieve the same effect without a conditional, we could use `^(?:START\d+END|\d+\b)`, which forces us to repeat the \d+ token.

(direct link)
Checking if a Named Capture Group has been Set

To check whether the Group named *foo* has been set, use this syntax:

✽ (?(foo)…|…) works in .NET, PCRE (C, PHP, R…) and Python
✽ (?(<foo>)…|…) works in Perl, PCRE (C, PHP, R…) and Ruby
✽ (?('foo')…|…) works in Perl, PCRE (C, PHP, R…) and Ruby

This example would work in .NET and PCRE:

```
^(?<UC>[A-Z])?\d+(?(UC)_END)$
```
✽ With (?<UC>[A-Z]) the optional capture group named *UC* captures one upper-case letter
✽ \d+ matches digits
✽ The conditional (?(UC)_END) checks whether the group named *UC* has been set. If so, it matches the characters *_END*

This pattern would match the string *A55_END* as well as the string *123*.

(direct link)
Checking if a Capture Group at a Relative Position has been Set

In PCRE (but not .NET, Perl, Python and Ruby), you can check whether a capture group at a relative position has been set. The relative position can be to the left of to the right of the conditional.

Checking a relative group to the left
To specify that the relative group to be checked is back from our current position in the pattern, we place a minus sign - in front of an integer. For instance, (?(-2)…|…) checks whether the second capture group to the left of our current position in the pattern has been set. Therefore,

```
(?(-1)X|Y)
```
says: if the nearest capture group to the left of this conditional has been set, match pattern X; otherwise, match pattern Y.

Using a relative group in a conditional comes in handy when you are working on a large pattern, some of whose parts you may later decide to move. It can be easier to count a relative position such as -2 than an absolute position.

Checking a relative group to the right
Although this is far less common, you can also use a forward relative group. This time, we use a + sign in front of an integer:

```
(?(+1)X|Y)
```
This says: if the nearest capture group to the right of this conditional has been set, match pattern X; otherwise, match pattern Y.

But how, you may ask, can a capture group to the right of the current position in the pattern already have been set? This can happen in various ways:

✽ The conditional and the group live inside a quantified group. For instance,
`(?:A(?(+1)B)(C))+` matches *ACABC*. On the first pass through the repeated group, the conditional fails as *C* has not yet been captured. On the second pass, the conditional succeeds.

✽ The conditional has been reached through a subroutine call. For instance,
`(A(?(+1)B)(C))(?1)` matches *ACABC*. Inside the parentheses that define Group 1, the conditional fails as *C* has not been captured. On the subroutine call (?1), the conditional succeeds.

✽ The conditional has been reached through a recursive call. For instance,
`(A(?(+1)B)(C)(?R)?D)` matches *ACABCDD*. At the outer level, the conditional fails as *C* as not been captured. At the first depth of recursion, it succeeds.

(direct link)
Checking if a Lookaround has been Successful

In .NET and PCRE (C, PHP, R…), a conditional can check whether a lookaround can succeed at the current position. For instance, suppose you wish to match the first word of a string, which by default is a vegetable. However, if the string ends with *_FRUIT*, the first word must be a fruit rather than the default vegetable. You can use this:

```
^(?(?=.*_FRUIT$)(?:apple|banana)|(?:carrot|pumpkin))\b
```
After the ^ anchor asserts that the current position is the beginning of the string, the conditional (?(?=.*_FRUIT$)…|…) checks whether the lookahead (?=.*_FRUIT$) can succeed. That lookahead asserts that at the current position, the engine can match any characters, then *_FRUIT* and the end of the string.

If the lookahead succeeds, we match a fruit: (?:apple|banana). Otherwise, we match a vegetable: (?:carrot|pumpkin)

Without a conditional, this would be a bit heavier to express:

```
^(?:(?:apple|banana)(?=.*_FRUIT$)|(?:carrot|pumpkin)(?!.*_FRUIT$))\b
```

(direct link)

Checking if a Subroutine Call has been Made

In Perl and PCRE (C, PHP, R…) you can check whether we are currently in the middle of a call to a specific subroutine. In the case one subroutine call is nested within another, the conditional test succeeds only if the specific subroutine being tested was the last one called.

For these tests, we can use both named and numbered subroutines. For instance, (?(R1)…|…)) checks whether we are in the middle of a call to subroutine 1, and (?(R&foo)…|…)) checks whether we are in the middle of a call to a subroutine named *foo*. Consider this pattern:

```
(A(?(R1)B|C))(?1)
```
It matches the string *ACAB*.

✽ The parentheses around (A…) define Group 1 and Subroutine 1. First, we match the character *A*.
✽ The conditional (?(R1)B|C) checks whether we are in the middle of a call to subroutine 1. After matching the string's initial *A*, it is **not true** that we have reached this point in the pattern via a subroutine call, so we must match the pattern in the *ELSE* branch of the conditional, which is the letter *C*.
✽ (?1) is a call to subroutine 1. First, we match another *A*. The conditional check succeeds as we have reached this point via a call to subroutine 1, so we must match the pattern in the *THEN* branch, which is the letter *B*.

Here is the same, but using a named subroutine:

```
(?<foo>A(?(R&foo)B|C))(?&foo)
```
✽ The parentheses around (?<foo>A…) define a capture group and subroutine named *foo*. First, we match the character *A*.
✽ The conditional (?(R&foo)B|C) checks whether we are in the middle of a call to the subroutine named *foo*. After matching the string's initial *A*, it is **not true** that we have reached this point in the pattern via a subroutine call, so we must match the pattern in the *ELSE* branch of the conditional, which is the letter *C*.
✽ (?&foo) is a call to the subroutine named *foo*. First, we match another *A*. The conditional check succeeds as we have reached this point via a call to the subroutine named *foo*, so we must match the pattern in the *THEN* branch, which is the letter *B*.

Nested Subroutine Calls
Suppose a part of the pattern calls subroutine 2, which then calls subroutine 1. Once inside subroutine 1, the engine encounters a conditional check on whether subroutine 2 has been called. Even though we are currently within a call to subroutine 2, the conditional test fails because what matters is the last subroutine call that was made—which is the call to subroutine 1.

We can see this with these two patterns:

✽ `(A(?(R1)C))(B(?1))(?2)` matches *ABACBAC*. Within it, (A(?(R1)C)) matches *A*, (B(?1)) matches *BAC* and (?2) matches *BAC* again.

✽ `(A(?(R2)C))(B(?1))(?2)` matches *ABABA* but not all of *ABABAC*. Within it, (A(?(R2)C)) matches *A*, (B(?1)) matches *BA*, and (?2) matches *BA* again. The conditional (?(R2)C) fails even when reached via (?2), as the most recent subroutine call when it is reached is the one made by (?1).

(direct link)

Checking if a Recursive Call has been Made

In Perl and PCRE (C, PHP, R…), the conditional (?(R)…|…)) checks whether we have reached this point in the pattern via a recursive call. Consider this pattern:

```
A(?(R)B)(?R)?C
```
It matches the string *AABCC*.

✽ The first time we encounter the conditional, we have not made a recursive call, so we do not have to match a *B*. The outer level of the recursive match will be *A…C*
✽ The second time we encounter the conditional, we are in the middle of a recursive call, so we must match a *B*. If we don't recurse again, the depth 1 match is *ABC*, and the pattern can match *ACABC*.

(direct link)

Checking that Embedded Code Evaluates to TRUE

In Perl, a conditional can check that an embedded fragment of Perl code evaluates to TRUE. The basic syntax for this is (?(?{*Perl code*})…|…)

For instance, suppose you are using the variable $currency as a Boolean flag. The pattern

```
\d+(?(?{$currency}) dollars)
```
matches two kinds of strings.

✽ When $currency is set to FALSE, the conditional test fails and the pattern only matches a series of digits, such as *122*.

✽ When $currency is set to TRUE, the conditional test succeeds and the pattern matches strings such as *55 dollars*.

(direct link)

Conditionals At Work: Balancing Delimiters

Suppose that in a body of text we want to match strings enclosed in two kinds of delimiters:
✽ If the string starts with *{{* it must end with *}}*
✽ If the string starts with *BEGIN:* it must end with *:END*

We can use this conditional regex:

```
(?:(BEGIN:)|({{)).*?(?(1):END)(?(2)}})
```
This will match *{{foo}}* and *BEGIN:bar:END*

✽ The non-capturing group (?:(BEGIN:)|({{)) matches the opening delimiter, either capturing *BEGIN:* to Group 1 or capturing *{{* to Group 2.

✽ .*? matches any characters, lazily expanding up to a point where the rest of the pattern can match.
✽ The conditional (?(1):END) checks if Group 1 has been set. If so, the engine must match *:END*
✽ The conditional (?(2)}}) checks if Group 2 has been set. If so, the engine must match *}}*

Alternative Solution
This can also be solved with a plain alternation:

```
BEGIN:.*?:END|{{.*?}}
```
However, this expression becomes increasingly more complex when
✽ we add potential delimiter pairs, such as <== … ==>, or
✽ the content to be matched between the delimiters turns into a longer pattern—as this pattern must be repeated on each branch of the alternation.

(direct link)
Conditionals At Work: Controlling Failure

This section relies on the classic trick (?!) to force failure. As a reminder, Perl and PCRE (C, PHP, R…) also allow you to use (*F) and (*FAIL)

Just as we can use a conditional to match a certain pattern if (or unless) condition X is met, we can use a conditional to force a match attempt to fail if (or unless) condition Y is met.

Fail If X
Suppose we're interested in matching digits \d+ in certain contexts. The digits must be followed by either *END* or *_end*. However, if they are preceded by *BEG*, then *END* is the only allowable suffix. Therefore, *BEG12_end* cannot match, whereas *BEG00END*, *00END* and *00_end* all match.

We can use this pattern:

```
^(BEG)?\d+(?:END|_end(?(1)(?!)))$
```
✽ (BEG)? optionally matches *BEG*, capturing the characters to Group 1.
✽ \d+ matches the digits.
✽ (?:END|_end(?(1)(?!))) matches either *END* or *_end*. On the *_end* branch, the conditional (?(1)(?!)) checks if Group 1 has been set (i.e., we matched *BEG* earlier), and if so, the *THEN* branch (?!) forces the match attempt to fail.

Fail Unless Y
Let's give a slight tweak to the context in which we'd like to match digits. The digits must still be followed by either *END* or *_end*. However, if they end with *END*, then *BEG* is the only allowable prefix. Therefore, *00END* cannot match, whereas *BEG00END*, *BEG12_end* and *00_end* all match.

We can use this pattern:

```
^(BEG)?\d+(?:_end|END(?(1)|(?!)))$
```
✽ (BEG)? optionally matches *BEG*, capturing the characters to Group 1.
✽ \d+ matches the digits.
✽ (?:_end|END(?(1)|(?!))) matches either *_end* or *END*. On the *END* branch, the conditional (?(1)|(?!)) checks if Group 1 has been set (i.e., we matched *BEG* earlier); if **not so**, the *ELSE* branch (?!) forces the match attempt to fail.

In the example on self-referencing groups, one of the alternate solutions will show a powerful way to use conditionals to control failure in the context of .NET balancing groups.

(direct link)
Conditionals At Work: Self-Referencing Group

This is an advanced technique that you should feel free to skip if you just want to get the gist of conditionals. However, it is required for the black belt program. :)

Suppose we want to match strings such as *AAA foo BBB*, which is framed by the same number of *A*s and *B*s. In Perl and PCRE (C, PHP, R…) we could use a recursive solution, such as
```
\A(A(?:(?1)|[^AB]*)B)\z
```
(This also works in Ruby if we replace the (?1) with a \g<1>)

But if we want to balance a greater number of tokens, as in *AAA foo BBB bar CCC baz DDD*, it can becomes interesting to use self-referencing groups, as seen on the page about Quantifier Capture and on the trick to match line numbers. For our task of balancing *A*s with *B*s in strings such as *AAA foo BBB*, we could use something like:

```
^(?:A(?=A*+[^AB]*+((?(1)\1)B)))++[^B]*+\1$
```
I know… Please don't scream, we'll ease in gently.

One feature of this pattern is that capture Group 1 ((?(1)\1)B) refers to itself with the conditional (?(1)\1). This conditional says:
✽ If Group 1 has already been set, match the current content of the Group 1 capture buffer.
✽ Match *B* — regardless of whether Group 1 has been set.

This construction has the effect that with each pass through Group 1, the Group 1 capture buffer gets longer by one character *B*.
✽ On the first pass, Group 1 has not been set, so the *THEN* branch of the conditional does not apply, and Group 1 captures one single *B*.
✽ On the second pass, the conditional applies, so the parentheses must match \1 (a back-reference to Group 1, which at this stage is *B*) and one additional *B*. At this stage, Group 1 contains *BB*.
✽ On the third pass, \1 is *BB*, so the parentheses must capture *BBB*… and so on.

Thanks to this construction, the quantified group (?:A…)+ matches all the characters one by one, and for each *A* that is matched, the Group 1 capture buffer grows by one *B*. By the time we exit (?:A…)+, we have matched as many *A*s as the number of *B*s captured in Group 1. Later in the pattern a simple back-reference \1 to Group 1 matches these *B*s.

Alternate Solutions
Inside the self-referencing group ((?(1)\1)B), instead of using a conditional, we could use an optional (but possessive) back-reference to Group 1 \1?+. If Group 1 is set, it is matched. And the possessive + forbids the engine from backtracking and giving up the back-reference.

We've already looked at the recursive solution. Let's look at a beautiful solution in .NET.

(direct link)
Balancing Groups
In .NET, we can use balancing groups. This solution also uses a conditional, which is another example of a conditional to control failure.

As a reminder, the task is to match strings where the number of *A*s and *B*s is balanced, as in *AAA foo BBB*. We can use this:

```
^(?<Count_A>A)+[^AB]*(?<-Count_A>B)+(?(Count_A)(?!))$
```
✽ (?<Count_A>A)+ matches all the *A*s, adding each individual *A* to the CaptureCollection named *Count_A*. I gave the group that name because we use the group as a virtual counter.
✽ [^AB]*+ matches all the non-*A*, non-*B* characters.
✽ (?<-Count_A>B)+ matches all the *B* characters, popping individual *A* characters from the CaptureCollection as it does do ("decrementing the counter").
✽ (?(Count_A)(?!)) checks if the named capture Group *Count_A* is set, which can only be the case if we have not removed enough *A*s from the CaptureCollection. This would mean there are fewer *B*s then *A*s in the string. In that case, the engine matches the *THEN* branch of the conditional, which is the classic trick (?!) to force the regex engine to fail and attempt to backtrack.

For efficiency, each quantified group should be made atomic:
```
^(?>(?<Count_A>A)+)(?>[^AB]*)(?>(?<-Count_A>B)+)(?(Count_A)(?!))$
```
I know, the atomic version (which is far preferable for the engine) looks awful… Do you happen to know the people at Microsoft in charge of .NET regex? If so, please lobby them to support possessive quantifiers (and subroutines, and recursion). And if you don't mind, please shoot me a message as I'd love to know how to reach them.

Don't Miss The **Regex Style Guide**

and **The Best Regex Trick Ever!!!**

How to work with Recursive Regular Expressions

Ask Rex

Leave a Comment
1-2 of 2 Threads
YosheE – France
December 14, 2013 - 04:59
Subject: About : 3. Not so useful: checking if a lookaround is successful.

Hi,
Let me add my opinion about the 3rd point : I had a problem for which I found a solution with this syntax, and that not seems to work if I use only the lookaround. Please consider a matching test : "if the string ends with END, it should contain WORD, otherwise all is permitted" :
- with the conditional regex I write this :
R1 : ^(? (? =. *END$). *WORD. *END|. *)$
with this R1 regex, "abcd" matches, "theWORD is END" matches, but "only END" doesn't match because it ends with END but WORD is missing. That's what I need : presence of WORD is tested only if string ends with END. - without the conditional it becomes :
R2 : ^(? =. *END$). *WORD. *END|. *$
with R2 regex, the last test "only END" matches and that's not what I need
So I think that there are cases for which checking if a lookaround is successful is so useful. Otherwise, please give me another regex that works for my problem (maybe it exists one, I'm not a regex guru ^^). Regards,
Yosh
Reply to YosheE
Rex
December 22, 2013 - 20:57
Subject: RE: checking if a lookaround is successful

Hi Yoshe,
Sorry about the delay, I have been traveling then had to catch up on a million things. Finally looked at your message today.

Congratulations for building an interesting example!

With two lookarounds, there are several solutions.
Here's a simple solution with a single lookaround:
(?x)
^.*?WORD.*END$
|
^(?:(?!END$).)*$
In a majority of cases, your conditional implementation probably runs faster.
I've added that as an additional example for case 3. Wishing you a beautiful day, Rex
Ivandro Ismael – Guinea-Bissau
November 12, 2013 - 15:31
Subject: Hi!

Thi is really help full thank you