Subroutine Calls May or May Not Capture

This tutorial introduced regular expression subroutines with this example that we want to match accurately:

```
Name: John Doe
Born: 17-Jan-1964
Admitted: 30-Jul-2013
Released: 3-Aug-2013
```

In Ruby or PCRE, we can use this regular expression:

```
^Name:\ (.*)\n
Born:\ (?'date'(?:3[01]|[12][0-9]|[1-9])
                -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
                -(?:19|20)[0-9][0-9])\n
Admitted:\ \g'date'\n
Released:\ \g'date'$
```

Perl needs slightly different syntax, which also works in PCRE:

```
^Name:\ (.*)\n
Born:\ (?'date'(?:3[01]|[12][0-9]|[1-9])
                -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
                -(?:19|20)[0-9][0-9])\n
Admitted:\ (?&date)\n
Released:\ (?&date)$
```

Unfortunately, there are differences in how these three regex flavors treat subroutine calls beyond their syntax. First of all, in Ruby a subroutine call makes the capturing group store the text matched during the subroutine call. In Perl, PCRE, and Boost a subroutine call does not affect the group that is called.

When the Ruby solution matches the sample above, retrieving the contents of the capturing group “date” will get you `3-Aug-2013` which was matched by the last subroutine call to that group. When the Perl solution matches the same, retrieving `$+{date}` will get you `17-Jan-1964`. In Perl, the subroutine calls did not capture anything at all. But the “Born” date was matched with a normal named capturing group which stored the text that it matched normally. Any subroutine calls to the group don’t change that. PCRE behaves as Perl in this case, even when you use the Ruby syntax with PCRE.

JGsoft V2 behaves like Ruby when you use the first regular expression. You can remember this by the fact that the \g syntax is a Ruby invention, later copied by PCRE. JGsoft V2 behaves like Perl when you use the second regular expression. You can remember this by the fact that Perl uses ampersands for subroutine calls in procedural code too.

If you want to extract the dates from the match, the best solution is to add another capturing group for each date. Then you can ignore the text stored by the “date” group and this particular difference between these flavors. In Ruby or PCRE:

```
^Name:\ (.*)\n
Born:\ (?'born'(?'date'(?:3[01]|[12][0-9]|[1-9])
                        -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
                        -(?:19|20)[0-9][0-9]))\n
Admitted:\ (?'admitted'\g'date')\n
Released:\ (?'released'\g'date')$
```

Perl needs slightly different syntax, which also works in PCRE:

```
^Name:\ (.*)\n
Born:\ (?'born'(?'date'(?:3[01]|[12][0-9]|[1-9])
                       -(?:Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
                       -(?:19|20)[0-9][0-9]))\n
Admitted:\ (?'admitted'(?&date))\n
Released:\ (?'released'(?&date))$
```

Capturing Groups Inside Recursion or Subroutine Calls

There are further differences between Perl, PCRE, and Ruby when your regex makes a subroutine call or recursive call to a capturing group that contains other capturing groups. The same issues also affect recursion of the whole regular expression if it contains any capturing groups. For the remainder of this topic, the term “recursion” applies equally to recursion of the whole regex, recursion into a capturing group, or a subroutine call to a capturing group.

PCRE and Boost back up and restores capturing groups when entering and exiting recursion. When the regex engine enters recursion, it internally makes a copy of all capturing groups. This does not affect the capturing groups. Backreferences inside the recursion match text captured prior to the recursion unless and until the group they reference captures something during the recursion. After the recursion, all capturing groups are replaced with the internal copy that was made at the start of the recursion. Text captured during the recursion is discarded. This means you cannot use capturing groups to retrieve parts of the text that were matched during recursion.

Perl 5.10, the first version to have recursion, through version 5.18, isolated capturing groups between each level of recursion. When Perl 5.10’s regex engine enters recursion, all capturing groups appear as they have not participated in the match yet. Initially, all backreferences will fail. During the recursion, capturing groups capture as normal. Backreferences match text captured during the same recursion as normal. When the regex engine exits from the recursion, all capturing groups revert to the state they were in prior to the recursion. Perl 5.20 changed Perl’s behavior to back up and restore capturing groups the way PCRE does.

For most practical purposes, however, you’ll only use backreferences after their corresponding capturing groups. Then the difference between the way Perl 5.10 through 5.18 deal with capturing groups during recursion and the way PCRE and later versions of Perl do is academic.

Ruby’s behavior is completely different. When Ruby’s regex engine enters or exits recursion, it makes no changes to the text stored by capturing groups at all. Backreferences match the text stored by the capturing group during the group’s most recent match, irrespective of any recursion that may have happened. After an overall match is found, each capturing group still stores the text of its most recent match, even if that was during a recursion. This means you can use capturing groups to retrieve part of the text matched during the last recursion.

JGsoft V2 behaves like Ruby when you use the \g syntax borrowed from Ruby. It behaves like Perl 5.20 and PCRE when you use any other syntax.

Odd Length Palindromes in Perl and PCRE

In Perl and PCRE you can use `\b(?'word'(?'letter'[a-z])(?&word)\k'letter'|[a-z])\b` to match palindrome words such as `a`, `dad`, `radar`, `racecar`, and `redivider`. This regex only matches palindrome words that are an odd number of letters long. This covers most palindrome words in English. To extend the regex to also handle palindrome words that are an even number of characters long we have to worry about differences in how Perl and PCRE backtrack after a failed recursion attempt which is discussed later in this tutorial. We gloss over these differences here because they only come into play when the subject string is not a palindrome and no match can be found.

Let's see how this regex matches `radar`. The word boundary `\b` matches at the start of the string. The regex engine enters the two capturing groups. `[a-z]` matches `r` which is then stored in the capturing group "letter". Now the regex engine enters the first recursion of the group "word". At this point, Perl forgets that the "letter" group matched `r`. PCRE does not. But this does not matter. `(?'letter'[a-z])` matches and captures `a`. The regex enters the second recursion of the group "word". `(?'letter'[a-z])` captures `d`. During the next two recursions, the group captures `a` and `r`. The fifth recursion fails because there are no characters left in the string for `[a-z]` to match. The regex engine must backtrack.

Because `(?&word)` failed to match, `(?'letter'[a-z])` must give up its match. The group reverts to `a`, which was the text the group held at the start of the recursion. (It becomes empty in Perl 5.18 and prior.) Again, this does not matter because the regex engine must now try the second alternative inside the group "word", which contains no backreferences. The second `[a-z]` matches the final `r` in the string. The engine now exits from a successful recursion. The text stored by the group "letter" is restored to what it had captured prior to entering the fourth recursion, which is `a`.

After matching `(?&word)` the engine reaches `\k'letter'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more, making the capturing group give up the `a`. The second alternative now matches the `a`. The regex engine exits from the third recursion. The group "letter" is restored to the `d` matched during the second recursion.

The regex engine has again matched `(?&word)`. The backreference fails again because the group stores `d` while the next character in the string is `r`. Backtracking again, the second alternative matches `d` and the group is restored to the `a` matched during the first recursion.

Now, `\k'letter'` matches the second `a` in the string. That's because the regex engine has arrived back at the first recursion during which the capturing group matched the first `a`. The regex engine exits the first recursion. The capturing group is restored to the `r` which it matched prior to the first recursion.

Finally, the backreference matches the second `r`. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` matches at the end of the string. The end of the regex is reached and `radar` is returned as the overall match. If you query the groups "word" and "letter" after the match you'll get `radar` and `r`. That's the text matched by these groups outside of all recursion.

Why This Regex Does Not Work in Ruby

To match palindromes this way in Ruby, you need to use a special backreference that specifies a recursion level. If you use a normal backreference as in `\b(?'word'(?'letter'[a-z])\g'word'\k'letter'|[a-z])\b`, Ruby will not complain. But it will not match palindromes longer than three letters either. Instead this regex matches things like `a`, `dad`, `radaa`, `raceccc`, and `rediviiii`.

Let's see why this regex does not match `radar` in Ruby. Ruby starts out like Perl and PCRE, entering the recursions until there are no characters left in the string for `[a-z]` to match.

Because `\g'word'` failed to match, `(?'letter'[a-z])` must give up its match. Ruby reverts it to `a`, which was the text the group most recently matched. The second `[a-z]` matches the final `r` in the string. The engine now exits from a successful recursion. The group "letter" continues to hold its most recent match `a`.

After matching `\g'word'` the engine reaches `\k'letter'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more, reverting the group to the previously matched `d`. The second alternative now matches the `a`. The regex engine exits from the third recursion.

The regex engine has again matched `\g'word'`. The backreference fails again because the group stores `d` while the next character in the string is `r`. Backtracking again, the group reverts to `a` and the second alternative matches `d`.

Now, `\k'letter'` matches the second `a` in the string. The regex engine exits the first recursion which successfully matched `ada`. The capturing group continues to hold `a` which is its most recent match that wasn't backtracked.

The regex engine is now at the last character in the string. This character is `r`. The backreference fails because the group still holds `a`. The engine can backtrack once more, forcing `(?'letter'[a-z])\g'word'\k'letter'` to give up the `rada` it matched so far. The regex engine is now back at the start of the string. It can still try the second alternative in the group. This matches the first `r` in the string. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` fails to match after the first `r`. The regex engine has no further permutations to try. The match attempt has failed.

If the subject string is `radaa`, Ruby's engine goes through nearly the same matching process as described above. Only the events described in the last paragraph change. When the regex engine reaches the last character in the string, that character is now `a`. This time, the backreference matches. Since the engine is not inside any recursion any more, it proceeds with the remainder of the regex after the group. `\b` matches at the end of the string. The end of the regex is reached and `radaa` is returned as the overall match. If you query the groups "word" and "letter" after the match you'll get `radaa` and `a`. Those are the most recent matches of these groups that weren't backtracked.

Basically, in Ruby this regex matches any word that is an odd number of letters long and in which all the characters to the right of the middle letter are identical to the character just to the left of the middle letter. That's because Ruby only restores capturing groups when they backtrack, but not when it exits from recursion.

The solution, specific to Ruby, is to use a backreference that specifies a recursion level instead of the normal backreference used in the regex on this page.

Make a Donation

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