Recursion and Subroutine Calls May or May Not Be Atomic

Earlier topics in this tutorial explain regular expression recursion and regular expression subroutines. In this topic the word "recursion" refers to recursion of the whole regex, recursion of capturing groups, and subroutine calls to capturing groups.

Perl and Ruby backtrack into recursion if the remainder of the regex after the recursion fails. They try all permutations of the recursion as needed to allow the remainder of the regex to match. PCRE treats recursion as atomic. PCRE backtracks normally during the recursion, but once the recursion has matched, it does not try any further permutations of the recursion, even when the remainder of the regex fails to match. The result is that Perl and Ruby may find regex matches that PCRE cannot find, or that Perl and Ruby may find different regex matches.

Consider the regular expression `aa$|a(?R)a|a` in Perl or the equivalent `aa$|a\g'0'a|a` in Ruby 2.0. PCRE supports either syntax. Let's see how Perl, Ruby, and PCRE go through the matching process of this regex when `aaa` is the subject string.

The first alternative `aa$` fails because the anchor cannot be matched between the second and third `a` in the string. Attempting the second alternative at the start of the string, `a` matches `a`. Now the regex engine enters the first recursion.

Inside the recursion, the first alternative matches the second and third `a` in the string. The regex engine exits a successful recursion. But now, the `a` that follows `(?R)` or `\g'0'` in the regex fails to match because the regex engine has already reached the end of the string. Thus the regex engine must backtrack. Here is where PCRE behaves differently than Perl or Ruby.

Perl and Ruby remember that inside the recursion the regex matched the second alternative and that there are three possible alternatives. Perl and Ruby backtrack *into* the recursion. The second alternative inside the recursion is backtracked, reducing the match so far to the first `a` in the string. Now the third alternative is attempted. `a` matches the second `a` in the string. The regex engine again exits successfully from the same recursion. This time, the `a` that follows `(?R)` or `\g'0'` in the regex matches the third `a` in the string. `aaa` is found as the overall match.

PCRE, on the other hand, remembers nothing about the recursion other than that it matched `aa` at the end of the string. PCRE does backtrack *over* the recursion, reducing the match so far to the first `a` in the string. But this leaves the second alternative in the regex without any further permutations to try. Thus the `a` at the start of the second alternative is also backtracked, reducing the match so far to nothing. PCRE continues the match attempt at the start of the string with the third alternative and finds that `a` matches `a` at the start of the string. In PCRE, this is the overall match.

You can make recursion in Perl and Ruby atomic by adding an atomic group. `aa$|a(?>(?R))a|a` in Perl and `aa$|a(?>\g'0')a|a` in Ruby is the same as the original regexes in PCRE.

JGsoft V2 lets you choose whether recursion should be atomic or not. Atomic recursion gives better performance, but may exclude certain matches or find different matches as illustrated above. `aa$|a(?P>0)a|a` is atomic in JGsoft V2. You can remember this because this syntax for recursion uses an angle bracket just like an atomic group. You can use a number or a name instead of zero for an atomic subroutine call to a numbered or named capturing group. Any other syntax for recursion is not atomic in JGsoft V2.

Boost is of two minds. Recursion of the whole regex is atomic in Boost, like in PCRE. But Boost will backtrack into subroutine calls and into recursion of capturing groups, like Perl. So you can do non-atomic recursion in Boost by wrapping the whole regex into a capturing group and then calling that.

PCRE2 originally behaved like PCRE, treating all recursion and subroutine calls as atomic. PCRE2 10.30 changed this, trying to be more like Perl, but ending up like Boost. PCRE2 10.30 will backtrack into subroutine calls and recursion of capturing groups like Perl does. But PCRE2 is still not able to backtrack into recursion of the whole regex. In the examples below, “PCRE” means the original PCRE only. For PCRE2 10.22 and prior, follow the PCRE example. For PCRE2 10.30 and later, follow the Perl example.

Palindromes of Any Length in Perl and Ruby

The topic about recursion and capturing groups explains a regular expression to match palindromes that are an odd number of characters long. The solution seems trivial. `\b(?'word'(?'letter'[a-z])(?&word)\k'letter'|[a-z]?)\b` does the trick in Perl. The quantifier `?` makes the `[a-z]` that matches the letter in the middle of the palindrome optional. In Ruby we can use `\b(?'word'(?'letter'[a-z])\g'word'\k'letter+0'|[a-z]?)\b` which adds the same quantifier to the solution that specifies the recursion level for the backreference. In PCRE, the Perl solution still matches odd-length palindromes, but not even-length palindromes.

Let’s see how these regexes match or fail to match `deed`. PCRE starts off the same as Perl and Ruby, just as in the original regex. The group “letter” matches `d`. During three consecutive recursions, the group captures `e`, `e`, and `d`. The fourth recursion fails, because there are no characters left to match. Back in the third recursion, the first alternative is backtracked and the second alternative matches `d` at the end of the string. The engine exits the third recursion with a successful match. Back in the second recursion, the backreference fails because there are no characters left in the string.

Here the behavior diverges. Perl and Ruby backtrack *into* the third recursion and backtrack the quantifier `?` that makes the second alternative optional. In the third recursion, the second alternative gives up the `d` that it matched at the end of the string. The engine exits the third recursion again, this time with a successful zero-length match. Back in the second recursion, the backreference still fails because the group stored `e` for the second recursion but the next character in the string is `d`. Thus the first alternative is backtracked and the second alternative matches the second `e` in the string. The second recursion is exited with success.

In the first recursion, the backreference again fails. The group stored `e` for the first recursion but the next character in the string is `d`. Again, Perl and Ruby backtrack into the second recursion to try the permutation where the second alternative finds a zero-length match. Back in the first recursion again, the backreference now matches the second `e` in the string. The engine leaves the first recursion with success. Back in the overall match attempt, the backreference matches the final `d` in the string. The word boundary succeeds and an overall match is found.

PCRE, however, does not backtrack into the third recursion. It does backtrack *over* the third recursion when it backtracks the first alternative in the second recursion. Now, the second alternative in the second recursion matches the second `e` in the string. The second recursion is exited with success.

In the first recursion, the backreference again fails. The group stored `e` for the first recursion but the next character in the string is `d`. Again, PCRE does not backtrack into the second recursion, but immediately fails the first alternative in the first recursion. The second alternative in the first recursion now matches the first `e` in the string. PCRE exits the first recursion with success. Back in the overall match attempt, the backreference fails, because the group captured `d` prior to the recursion, and the next character is the second `e` in the string. Backtracking again, the second alternative in the overall regex match now matches the first `d` in the string. Then the word boundary fails. PCRE did not find any matches.

Palindromes of Any Length in PCRE

To match palindromes of any length in PCRE, we need a regex that matches words of an even number of characters and of an odd number of characters separately. Free-spacing mode makes this regex easier to read:

```
\b(?'word'
  (?'oddword' (?'oddletter' [a-z])(?P>oddword)  \k'oddletter' |[a-z])
| (?'evenword'(?'evenletter'[a-z])(?P>evenword)?\k'evenletter')
)\b
```

Basically, this is two copies of the original regex combined with alternation. The first alternatives has the groups “word” and “letter” renamed to “oddword” and “oddletter”. The second alternative has the groups “word” and “letter” renamed to “evenword” and “evenletter”. The call `(?P>evenword)` is now made optional with a question mark instead of the alternative `|[a-z]`. A new group “word” combines the two groups “oddword” and “evenword” so that the word boundaries still apply to the whole regex.

The first alternative “oddword” in this regex matches a palindrome of odd length like `radar` in exactly the same way as the regex discussed in the topic about recursion and capturing groups does. The second alternative in the new regex is never attempted.

When the string is a palindrome of even length like `deed`, the new regex first tries all permutations of the first alternative. The second alternative “evenword” is attempted only after the first alternative fails to find a match.

The second alternative starts off in the same way as the original regex. The group “evenletter” matches `d`. During three consecutive recursions, the group captures `e`, `e`, and `d`. The fourth recursion fails, because there are no characters left to match. Back in the third recursion, the regex engine notes that recursive call `(?P>evenword)?` is optional. It proceeds to the backreference `\k'evenletter'`. The backreference fails because there are no characters left in the string. Since the recursion has no further alternatives to try, it is backtracked. The group “evenletter” must give up its most recent match and PCRE exits from the failed third recursion.

In the second recursion, the backreference fails because the capturing group matched `e` during that recursion but the next character in the string is `d`. The group gives up another match and PCRE exits from the failed second recursion.

Back in the first recursion, the backreference succeeds. The group matched the first `e` in the string during that recursion and the backreference matches the second. PCRE exits from the successful first recursion.

Back in the overall match attempt, the backreference succeeds again. The group matched the `d` at the start of the string during the overall match attempt, and the backreference matches the final `d`. Exiting the groups “evenword” and “word”, the word boundary matches at the end of the string. `deed` is the overall match.

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!