## Backreferences That Specify a Recursion Level

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. The previous topic also explained that these features handle capturing groups differently in Ruby than they do in Perl and PCRE.

Perl, PCRE, and Boost restore capturing groups when they exit from recursion. This means that backreferences in Perl, PCRE, and Boost match the same text that was matched by the capturing group at the same recursion level. This makes it possible to do things like matching palindromes.

Ruby does not restore capturing groups when it exits from recursion. Normal backreferences match the text that is the same as the most recent match of the capturing group that was not backtracked, regardless of whether the capturing group found its match at the same or a different recursion level as the backreference. Basically, normal backreferences in Ruby don't pay any attention to recursion.

But while the normal capturing group storage in Ruby does not get any special treatment for recursion, Ruby actually stores a full stack of matches for each capturing groups at all recursion levels. This stack even includes recursion levels that the regex engine has already exited from.

Backreferences in Ruby can match the same text as was matched by a capturing group at any recursion level relative to the recursion level that the backreference is evaluated at. You can do this with the same syntax for named backreferences by adding a sign and a number after the name. In most situations you will use `+0` to specify that you want the backreference to reuse the text from the capturing group at the same recursion level. You can specify a positive number to reference the capturing group at a deeper level of recursion. This would be a recursion the regex engine has already exited from. You can specify a negative number to reference the capturing group a level that is less deep. This would be a recursion that is still in progress.

JGsoft V2 also supports backreferences that specify a recursion level using the same syntax as Ruby. To get the same behavior with JGsoft V2 as with Ruby, you have to use Ruby's \g syntax for your subroutine calls.

## Odd Length Palindromes in Ruby

In Ruby you can use `\b(?'word'(?'letter'[a-z])\g'word'\k'letter+0'|[a-z])\b` to match palindrome words such as `a`, `dad`, `radar`, `racecar`, and `redivider`. To keep this example simple, this regex only matches palindrome words that are an odd number of letters long.

Let's see how this regex matches `radar`. The word boundary `\b` matches at the start of the string. The regex engine enters the capturing group "word". `[a-z]` matches `r` which is then stored in the stack for the capturing group "letter" at recursion level zero. Now the regex engine enters the first recursion of the group "word". `(?'letter'[a-z])` matches and captures `a` at recursion level one. The regex enters the second recursion of the group "word". `(?'letter'[a-z])` captures `d` at recursion level two. During the next two recursions, the group captures `a` and `r` at levels three and four. The fifth recursion fails because there are no characters left in the string for `[a-z]` to match. The regex engine must backtrack.

The regex engine must now try the second alternative inside the group "word". The second `[a-z]` in the regex matches the final `r` in the string. The engine now exits from a successful recursion, going one level back up to the third recursion.

After matching `\g'word'` the engine reaches `\k'letter+0'`. The backreference fails because the regex engine has already reached the end of the subject string. So it backtracks once more. The second alternative now matches the `a`. The regex engine exits from the third recursion.

The regex engine has again matched `\g'word'` and needs to attempt the backreference again. The backreference specifies `+0` or the present level of recursion, which is 2. At this level, the capturing group matched `d`. The backreference fails because the next character in the string is `r`. Backtracking again, the second alternative matches `d`.

Now, `\k'letter+0'` 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 regex engine is now back outside all recursion. At this level, the capturing group stored `r`. The backreference can now match the final `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` matches at the end of the string. The end of the regex is reached and `radar` is returned as the overall match.

## Backreferences to Other Recursion Levels

Backreferences to other recursion levels can be easily understood if we modify our palindrome example. `abcdefedcba` is also a palindrome matched by the previous regular expression. Consider the regular expression `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter-1'|z)|[a-z])\b`. The backreference now wants a match the text one level less deep on the capturing group's stack. It is alternated with the letter `z` so that something can be matched when the backreference fails to match.

The new regex matches things like `abcdefdcbaz`. After a whole bunch of matching and backtracking, the second `[a-z]` matches `f`. The regex engine exits from a successful fifth recursion. The capturing group "letter" has stored the matches `a`, `b`, `c`, `d`, and `e` at recursion levels zero to four. Other matches by that group were backtracked and thus not retained.

Now the engine evaluates the backreference `\k'letter-1'`. The present level is 4 and the backreference specifies -1. Thus the engine attempts to match `d`, which succeeds. The engine exits from the fourth recursion.

The backreference continues to match `c`, `b`, and `a` until the regex engine has exited the first recursion. Now, outside all recursion, the regex engine again reaches `\k'letter-1'`. The present level is 0 and the backreference specifies -1. Since recursion level -1 never happened, the backreference fails to match. This is not an error but simply a backreference to a non-participating capturing group. But the backreference has an alternative. `z` matches `z` and `\b` matches at the end of the string. `abcdefdcbaz` was matched successfully.

You can take this as far as you like. The regular expression `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter-2'|z)|[a-z])\b` matches `abcdefcbazz`. `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter-99'|z)|[a-z])\b` matches `abcdefzzzzzz`.

Going in the opposite direction, `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter+1'|z)|[a-z])\b` matches `abcdefzedcb`. Again, after a whole bunch of matching and backtracking, the second `[a-z]` matches `f`, the regex engine is back at recursion level 4, and the group "letter" has `a`, `b`, `c`, `d`, and `e` at recursion levels zero to four on its stack.

Now the engine evaluates the backreference `\k'letter+1'`. The present level is 4 and the backreference specifies +1. The capturing group was backtracked at recursion level 5. This means we have a backreference to a non-participating group, which fails to match. The alternative `z` does match. The engine exits from the fourth recursion.

At recursion level 3, the backreference points to recursion level 4. Since the capturing group successfully matched at recursion level 4, it still has that match on its stack, even though the regex engine has already exited from that recursion. Thus `\k'letter+1'` matches `e`. Recursion level 3 is exited successfully.

The backreference continues to match `d` and `c` until the regex engine has exited the first recursion. Now, outside all recursion, the regex engine again reaches `\k'letter+1'`. The present level is 0 and the backreference specifies +1. The capturing group still retains all its previous successful recursion levels. So the backreference can still match the `b` that the group captured during the first recursion. Now `\b` matches at the end of the string. `abcdefzdcb` was matched successfully.

You can take this as far as you like in this direction too. The regular expression `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter+2'|z)|[a-z])\b` matches `abcdefzzedc`. `\b(?'word'(?'letter'[a-z])\g'word'(?:\k'letter+99'|z)|[a-z])\b` matches `abcdefzzzzzz`.

## 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!