Regular Expression Recursion

Perl 5.10, PCRE 4.0, Ruby 2.0, and all later versions of these three, support regular expression recursion. Perl uses the syntax `(?R)` with `(?0)` as a synonym. Ruby 2.0 uses `\g<0>`. PCRE supports all three as of version 7.7. Earlier versions supported only the Perl syntax (which Perl actually copied from PCRE). Recent versions of Delphi, PHP, and R also support all three, as their regex functions are based on PCRE. JGsoft V2 also supports all variations of regex recursion.

While Ruby 1.9 does not have any syntax for regex recursion, it does support capturing group recursion. So you could recurse the whole regex in Ruby 1.9 if you wrap the whole regex in a capturing group. .NET does not support recursion, but it supports balancing groups that can be used instead of recursion to match balanced constructs.

As we'll see later, there are differences in how Perl, PCRE, and Ruby deal with backreferences and backtracking during recursion. While they copied each other's syntax, they did not copy each other's behavior. JGsoft V2, however, copied their syntax and their behavior. So JGsoft V2 has three different ways of doing regex recursion, which you choose by using a different syntax. But these differences do not come into play in the basic example on this page.

Boost 1.42 copied the syntax from Perl. But its implementation is marred by bugs. Boost 1.60 attempted to fix the behavior of quantifiers on recursion, but it's still quite different from other flavors and incompatible with previous versions of Boost. Boost 1.64 finally stopped crashing upon infinite recursion. But recursion of the whole regex still attempts only the first alternative.

Simple Recursion

The regexes `a(?R)?z`, `a(?0)?z`, and `a\g<0>?z` all match one or more letters `a` followed by exactly the same number of letters `z`. Since these regexes are functionally identical, we'll use the syntax with R for recursion to see how this regex matches the string `aaazzz`.

First, `a` matches the first `a` in the string. Then the regex engine reaches `(?R)`. This tells the engine to attempt the whole regex again at the present position in the string. Now, `a` matches the second `a` in the string. The engine reaches `(?R)` again. On the second recursion, `a` matches the third `a`. On the third recursion, `a` fails to match the first `z` in the string. This causes `(?R)` to fail. But the regex uses a quantifier to make `(?R)` optional. So the engine continues with `z` which matches the first `z` in the string.

Now, the regex engine has reached the end of the regex. But since it's two levels deep in recursion, it hasn't found an overall match yet. It only has found a match for `(?R)`. Exiting the recursion after a successful match, the engine also reaches `z`. It now matches the second `z` in the string. The engine is still one level deep in recursion, from which it exits with a successful match. Finally, `z` matches the third `z` in the string. The engine is again at the end of the regex. This time, it's not inside any recursion. Thus, it returns `aaazzz` as the overall regex match.

Matching Balanced Constructs

The main purpose of recursion is to match balanced constructs or nested constructs. The generic regex is `b(?:m|(?R))*e` where `b` is what begins the construct, `m` is what can occur in the middle of the construct, and `e` is what can occur at the end of the construct. For correct results, no two of `b`, `m`, and `e` should be able to match the same text. You can use an atomic group instead of the non-capturing group for improved performance: `b(?>m|(?R))*e`.

A common real-world use is to match a balanced set of parentheses. `\((?>[^()]|(?R))*\)` matches a single pair of parentheses with any text in between, including an unlimited number of parentheses, as long as they are all properly paired. If the subject string contains unbalanced parentheses, then the first regex match is the leftmost pair of balanced parentheses, which may occur after unbalanced opening parentheses. If you want a regex that does not find any matches in a string that contains unbalanced parentheses, then you need to use a subroutine call instead of recursion. If you want to find a sequence of multiple pairs of balanced parentheses as a single match, then you also need a subroutine call.

Recursion with Alternation

If what may appear in the middle of the balanced construct may also appear on its own without the beginning and ending parts then the generic regex is `b(?R)*e|m`. Again, `b`, `m`, and `e` all need to be mutually exclusive. `\((?R)*\)|[^()]+` matches a pair of balanced parentheses like the regex in the previous section. But it also matches any text that does not contain any parentheses at all.

This regular expression does not work correctly in Boost. If a regex has alternation that is not inside a group then recursion of the whole regex in Boost only attempts the first alternative. So `\((?R)*\)|[^()]+` in Boost matches any number of balanced parentheses nested arbitrarily deep with no text in between, or any text that does not contain any parentheses at all. If you flip the alternatives then `[^()]+|\((?R)*\)` in Boost matches any text without any parentheses or a single pair of parentheses with any text without parentheses in between. In all other flavors these two regexes find the same matches.

The solution for Boost is to put the alternation inside a group. `(?:\((?R)*\)|[^()]+)` and `(?:[^()]+|\((?R)*\))` find the same matches in all flavors discussed in this tutorial that support recursion.

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!