Repetition with Star and Plus

One repetition operator or quantifier was already introduced: the question mark. It tells the engine to attempt to match the preceding token zero times or once, in effect making it optional.

The asterisk or star tells the engine to attempt to match the preceding token zero or more times. The plus tells the engine to attempt to match the preceding token once or more. `<[A-Za-z][A-Za-z0-9]*>` matches an HTML tag without any attributes. The angle brackets are literals. The first character class matches a letter. The second character class matches a letter or digit. The star repeats the second character class. Because we used the star, it's OK if the second character class matches nothing. So our regex will match a tag like `<B>`. When matching `<HTML>`, the first character class will match `H`. The star will cause the second character class to be repeated three times, matching `T`, `M` and `L` with each step.

I could also have used `<[A-Za-z0-9]+>`. I did not, because this regex would match `<1>`, which is not a valid HTML tag. But this regex may be sufficient if you know the string you are searching through does not contain any such invalid tags.

Limiting Repetition

There's an additional quantifier that allows you to specify how many times a token can be repeated. The syntax is `{min,max}`, where *min* is zero or a positive integer number indicating the minimum number of matches, and *max* is an integer equal to or greater than *min* indicating the maximum number of matches. If the comma is present but *max* is omitted, the maximum number of matches is infinite. So `{0,1}` is the same as `?`, `{0,}` is the same as `*`, and `{1,}` is the same as `+`. Omitting both the comma and *max* tells the engine to repeat the token exactly *min* times.

You could use `\b[1-9][0-9]{3}\b` to match a number between 1000 and 9999. `\b[1-9][0-9]{2,4}\b` matches a number between 100 and 99999. Notice the use of the word boundaries.

Watch Out for The Greediness!

Suppose you want to use a regex to match an HTML tag. You know that the input will be a valid HTML file, so the regular expression does not need to exclude any invalid use of sharp brackets. If it sits between sharp brackets, it is an HTML tag.

Most people new to regular expressions will attempt to use `<.+>`. They will be surprised when they test it on a string like `This is a <EM>first</EM> test`. You might expect the regex to match `<EM>` and when continuing after that match, `</EM>`.

But it does not. The regex will match `<EM>first</EM>`. Obviously not what we wanted. The reason is that the plus is *greedy*. That is, the plus causes the regex engine to repeat the preceding token as often as possible. Only if that causes the entire regex to fail, will the regex engine *backtrack*. That is, it will go back to the plus, make it give up the last iteration, and proceed with the remainder of the regex. Let's take a look inside the regex engine to see in detail how this works and why this causes our regex to fail. After that, I will present you with two possible solutions.

Like the plus, the star and the repetition using curly braces are greedy.

Looking Inside The Regex Engine

The first token in the regex is `<`. This is a literal. As we already know, the first place where it will match is the first `<` in the string. The next token is the dot, which matches any character except newlines. The dot is repeated by the plus.

The plus is *greedy*. Therefore, the engine will repeat the dot as many times as it can. The dot matches `E`, so the regex continues to try to match the dot with the next character. `M` is matched, and the dot is repeated once more. The next character is the `>`. You should see the problem by now. The dot matches the `>`, and the engine continues repeating the dot. The dot will match all remaining characters in the string. The dot fails when the engine has reached the void after the end of the string. Only at this point does the regex engine continue with the next token: `>`.

So far, `<.+` has matched `<EM>first</EM> test` and the engine has arrived at the end of the string. `>` cannot match here. The engine remembers that the plus has repeated the dot more often than is required. (Remember that the plus *requires* the dot to match only once.) Rather than admitting failure, the engine will *backtrack*. It will reduce the repetition of the plus by one, and then continue trying the remainder of the regex.

So the match of `.+` is reduced to `EM>first</EM> tes`. The next token in the regex is still `>`. But now the next character in the string is the last `t`. Again, these cannot match, causing the engine to backtrack further. The total match so far is reduced to `<EM>first</EM> te`. But `>` still cannot match. So the engine continues backtracking until the match of `.+` is reduced to `EM>first</EM`. Now, `>` can match the next character in the string. The last token in the regex has been matched. The engine reports that `<EM>first</EM>` has been successfully matched.

Remember that the regex engine is *eager* to return a match. It will not continue backtracking further to see if there is another possible match. It will report the first valid match it finds. Because of greediness, this is the leftmost longest match.

Laziness Instead of Greediness

The quick fix to this problem is to make the plus *lazy* instead of greedy. Lazy quantifiers are sometimes also called "ungreedy" or "reluctant". You can do that by putting a question mark after the plus in the regex. You can do the same with the star, the curly braces and the question mark itself. So our example becomes `<.+?>`. Let's have another look inside the regex engine.

Again, `<` matches the first `<` in the string. The next token is the dot, this time repeated by a lazy plus. This tells the regex engine to repeat the dot as few times as possible. The minimum is one. So the engine matches the dot with `E`. The requirement has been met, and the engine continues with `>` and `M`. This fails. Again, the engine will *backtrack*. But this time, the backtracking will force the lazy plus to expand rather than reduce its reach. So the match of `.+` is expanded to `EM`, and the engine tries again to continue with `>`. Now, `>` is matched successfully. The last token in the regex has been matched. The engine reports that `<EM>` has been successfully matched. That's more like it.

An Alternative to Laziness

In this case, there is a better option than making the plus lazy. We can use a greedy plus and a [negated character class](): `<[^>]+>`. The reason why this is better is because of the backtracking. When using the lazy plus, the engine has to backtrack for each character in the HTML tag that it is trying to match. When using the negated character class, no backtracking occurs at all when the string contains valid HTML code. Backtracking slows down the regex engine. You will not notice the difference when doing a single search in a text editor. But you will save plenty of CPU cycles when using such a regex repeatedly in a tight loop in a script that you are writing, or perhaps in a custom syntax coloring scheme for [EditPad Pro](). Hey

Only [regex-directed engines]() backtrack. Text-directed engines don't and thus do not get the speed penalty. But they also do not support lazy quantifiers.

Repeating \Q…\E Escape Sequences

The \Q…\E sequence escapes a string of characters, matching them as literal characters. The escaped characters are treated as individual characters. If you place a quantifier after the `\E`, it will only be applied to the last character. E.g. if you apply `\Q*\d+*\E+` to `*\d+**\d+*`, the match will be `*\d+**`. Only the asterisk is repeated. Java 4 and 5 have a bug that causes the whole \Q…E sequence to be repeated, yielding the whole subject string as the match. This was fixed in Java 6.

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!