Backtracking in Regular Expressions

A fundamental feature of regular expression matching involves the notion called backtracking. When the beginning of a pattern succeeds in a way that causes later parts in the pattern to fail, the matching engine backs up and recalculates the beginning part. Let's say you want to find the word following ''foo'':

```
$_ = "Food is on the foo table.";
if ( /\b(foo)\s+(\w+)/i ) {
    print "$2 follows $1.\n";
}
```

When the match runs, it first finds a possible match right at the beginning of the string, and loads up **$1** with ''foo''. However, as soon as the matching engine sees that there's no whitespace following the ''foo'' that it had saved in **$1**, it realizes its mistake and starts over again one character after where it had had the tentative match. This time it goes all the way until the next ''foo'', which works out, and you get the expected output of ''table follows foo.''

Sometimes minimal matching can help a lot. Imagine you'd like to match everything between ''foo'' and ''bar''. Initially, you write something like this:

```
$_ =  "The food is under the bar in the barn.";
if ( /foo(.*)bar/ ) {
    print "got <$1>\n";
}
```

Which perhaps unexpectedly yields:

```
got <d is under the bar in the >
```

That's because `.*` was greedy, so you get everything between the *first* ''foo'' and the *last* ''bar''. In this case, it's more effective to use minimal matching to make sure you get the text between a ''foo'' and the first ''bar'' thereafter.

```
  if ( /foo(.*?)bar/ ) { print "got <$1>\n" }
got <d is under the >
```

Here's another example: let's say you'd like to match a number at the end of a string, and you also want to keep the preceding part the match. So you write this:

```
$_ = "I have 2 numbers: 53147";
if ( /(.*)(\d*)/ ) {
    print "Beginning is <$1>, number is <$2>.\n";
}
```

That won't work at all, because **$1** was greedy and gobbled up everything, and also because \d* doesn't assert the existence of anything, since it can match the null string.

```
Beginning is <I have 2: 53147>, number is <>.
```

Here are some variants, most of which don't work:

```
$_ = "I have 2 numbers: 53147";
@pats = qw{
    (.*)(\d*)
    (.*)(\d+)
    (.*?)(\d*)
    (.*?)(\d+)
    (.*)(\d+)$
    (.*?)(\d+)$
    (.*)\b(\d+)$
    (.*\D)(\d+)$
};

for $pat (@pats) {
    printf "%-12s ", $pat;
    if ( /$pat/ ) {
        print "<$1> <$2>\n";
    } else {
        print "FAIL\n";
    }
}
```

That will print out:

```
(.*)(\d*)     <I have 2 numbers: 53147> <>
(.*)(\d+)     <I have 2 numbers: 5314> <7>
(.*?)(\d*)    <> <>
(.*?)(\d+)    <I have > <2>
(.*)(\d+)$    <I have 2 numbers: 5314> <7>
(.*?)(\d+)$   <I have 2 numbers: > <53147>
(.*)\b(\d+)$  <I have 2 numbers: > <53147>
(.*\D)(\d+)$  <I have 2 numbers: > <53147>
```

As you see, this can be a bit tricky. It's important to realize that a regular expression is merely a set of assertions that gives a definition of success. There may be 0, 1, or several different ways that the definition might succeed against a particular string. And if there are multiple ways it might succeed, you need to understand backtracking in order to know which variety of success you will achieve.

When using lookahead assertions and negations, this can all get even tricker. Imagine you'd like to find a sequence of nondigits not followed by ''123''. You might try to write that as

```
$_ = "ABC123";
if ( /^\D*(?!123)/ ) {
    print "Yup, no 123 in $_\n";
}
```

But that won't work at all; at least, not the way you're hoping: it claims that there is no 123 in the string. Here's a clearer picture of why not:

```
$x = 'ABC123' ;
$y = 'ABC445' ;
print "1: got $1\n" if $x =~ /^(ABC)(?!123)/ ;
```

```
print "2: got $1\n" if $y =~ /^(ABC)(?!123)/ ;
print "3: got $1\n" if $x =~ /^(\D*)(?!123)/ ;
print "4: got $1\n" if $y =~ /^(\D*)(?!123)/ ;
```

This prints

```
2: got ABC
3: got AB
4: got ABC
```

Many people wouldn't have expected test 3 to succeed. What's happening is that we've asked ''Is it true that at the start of **$x**, following 0 or more nondigits, we have something that's not 123?'' If the pattern matcher had let `\D*` expand to ''ABC'', this would have caused the whole pattern to fail. But the pattern really, *really* wants to succeed, so it uses the standard regexp backoff-and-retry and lets `\D*` expand to just ''AB'' this time. Now there's indeed something following ''AB'' that is not ''123''. It's in fact ''C123'', which suffices.

We can deal with this by using both an assertion and a negation. We'll say that the first part in **$1** must be followed by a digit, and in fact, it must also be followed by something that's not ''123''. Remember that the lookaheads are zero-width expressions--they only look, but don't consume any of the string in their match. So rewriting this way produces what you'd expect; that is, case 5 will fail, but case 6 succeeds:

```
print "5: got $1\n" if $x =~ /^(\D*)(?=\d)(?!123)/ ;
print "6: got $1\n" if $y =~ /^(\D*)(?=\d)(?!123)/ ;
6: got ABC
```

In other words, the two zero-width assertions next to each other work like they're ANDed together, just as you'd use any builtin assertions: `/^$/` matches only if you're at the beginning of the line AND the end of the line simultaneously. The deeper underlying truth is that juxtaposition in regular expressions always means AND, except when you write an explicit OR using the vertical bar. `/ab/` means match ''a'' AND (then) match ''b'', although the attempted matches are made at different positions because ''a'' is not a zero-width assertion, but a one-width assertion.