Regular Expression Tutorial

1. Introduction

Although there are plenty of *perl* hackers and other regular expression users, the amount of decent tutorials and guides on regular expressions on the 'net remains exceptionally low. Because I still find relatively many questions about regular expressions, and see how others struggle with them, I decided to write this tutorial. Bear in mind that this is still a work in-progress.

1.1. Purpose

The purpose of this tutorial is to help the reader on his or her way in the world of regular expressions. The basic concepts are explained and the largest pitfalls are covered (no pun intended. Well, maybe just a little).

1.2. Notation

All regular expressions in this tutorial are presented in a monospace font and on a lightgray background with a darkgray outline. Because it would be very difficult to clearly show spaces in a regular expression, and particularly on a web page, every space in every regex shown is represented by the `•`-symbol. The end result then looks like `regular•expression`.

1.3. Examples and Exercises

Most of the examples and all of the exercises were made using GNU *egrep*. Windows users can get a win32 port of GNU *egrep* here. Documentation can be found here. If you don't have GNU *egrep* on your UN*X box, you might try using your native implementation; it should be largely compatible.

If the directory where *egrep* is installed (like */usr/bin*, or *C:\windows\command*) is in your environment's *$PATH* variable (*%PATH%* in Windows), you should be able to invoke *egrep* simply by typing

```
$ egrep
Usage: egrep [OPTION]... PATTERN [FILE]...
Try `egrep --help' for more information.
```

Here, the dollar-sign represents the shell's prompt (similar to `C:\>` in Windows), and should not be typed. All command-line invocation examples show this prompt. Following is the text that should actually be typed, which is always shown in boldface. The remaining lines contain the command's output.

Basically you give *egrep* a regular expression and the name of a file. *egrep* then tries to match the regex against *each line* of the file. A line is only printed if it matches the regex.

1.4. Copyright and Distribution

Contact <kars@kde.nl> for more information.

2. What are they?

A regular expression, usually abbreviated to "regex" or "regexp", describes text patterns. Assume you're looking for a piece of text that starts with either two, three or four letters 'A', followed by exactly three letters 'C'. This pattern can be described with the regex `A{2,4}C{3}`.

From the above regex one can determine that not all characters are interpreted literally. The accolades (or curly braces, take your pick) clearly have a special meaning. Characters with such a special meaning are called **meta characters**. So, regular expressions have their own particular syntax, and so you could speak of a regex *language*.

As with most human languages the regex language has many dialects; regexes written for *perl* aren't automatically suited for *sed*, *awk* or *grep*, to name just a few standard UNIX tools.

I've chosen to write all the regexes in this tutorial in the POSIX dialect. This because POSIX is slowly winning terrain in the world of regexes, and because a fair amount of dialects are similar to it (well, actually it's the other way around). But this doesn't mean I'll be covering all the features of the POSIX 1003.2 regular expression standard. Another reason for using the POSIX dialect as opposed to the Perl dialect is because the Perl documentation does a much better job of explaining the Perl dialect than I ever will. Also, this way you won't be locked into any particular tool's regex extensions. In a way, the POSIX dialect can be considered the greatest common denominator.

2.1. Usage

A regex by itself does very little. Only by applying such a description of a text pattern to a piece of text does anything happen. The actual applying is done by a piece of software called a regex **engine**. The text is searched from the start until a piece of text is found that matches the pattern description (the regex), or until it runs out of text. Such a match is called a **pattern match**.

There are basically two ways of using regular expressions. One is by using special-purpose tools that were built specifically to apply regexes to text, like *grep*, *egrep* and *sed*. The other way is by using the regex capabilities built into a programming or scripting language. These days, most languages, like C, C++, Javascript, Python and PHP for example, provide functions or methods that can apply a regex to a piece of text. The code that actually applies regular expressions to text is called a regular expression **engine**.

awk and particularly *perl* don't quite fit either way. Once you get the hang of *perl*, you'll notice how tightly the concept of applying a regex to data is integrated into the whole design of *perl*.

3. Meta Characters

To be able to discuss meta characters, we first have to determine what "ordinary" characters mean to a regex. The regex `cat` does indeed find the "cat" in the text `The neighbour's cat pees on my lawn`, but also the "cat" in `the winter` **`cat`**`alog`. So, regular expressions work purely on text, and don't look at the semantics. It's important to realise that the above regex doesn't mean anything more to the regex engine than a 'c', followed by an 'a', followed by a 't', where ever it may be in the text to which the regex is applied.

To get you started, here's a simple example. fruits.txt contains a list of types of fruit, eight total, one per line. Once you've downloaded the example and saved it, open a console (or DOS box or whatever) and move to the directory where you saved the file. Once there, type the following:

```
$ egrep pear fruits.txt
pear
```

It might not be the most exciting demonstration of regular expressions at work, but if you get the same output, namely `pear`, it means you've successfully applied your first regex. Assuming this is your first time, that is.

Another example would be:

```
$ egrep ea fruits.txt
pear
```

```
peach
```

Slightly more interesting, the regex `ea` catches every line that contains `ea`, re-emphasising that regular expressions have no regard for semantics.

Another instructional example is:

```
$ egrep a fruits.txt
apple
orange
pear
peach
grape
banana
```

You might wonder what's so special about this example. The lesson lies in the last line of the result, `banana`. Recalling that *egrep* prints only those lines that match the pattern, you might wonder how *egrep* handles `banana` when applying `a`; perhaps you think it matches the line three times, which is more than once, and so the line is printed. The point is that *egrep* stops applying the regex as soon as it finds a match. Once it finds the first 'a', it stops searching, prints the line and moves on to the next.

This particular example illustrates that *egrep* doesn't care *what* it matches, or how often, but only wether it matches or not. Later we'll see examples that *do* care what or how often is matched. Obviously, these examples won't use *egrep*.

The last example for this section demonstrates a feature of *egrep*:

```
$  egrep -v a fruits.txt
blueberry
plum
```

The *-v* option tells *egrep* to invert the sense of the match. Now only lines that *don't* match the pattern are printed. And indeed, neither `blueberry` nor `plum` contains an 'a'.

3.1. Anchors

Using `^` and `$` you can force a regex to match only at the start or end of a line, respectively. So `^cat` matches only those lines that start with `cat`, and `cat$` only matches lines ending with `cat`.

A hands-on example that uses the same *fruits.txt* as in the previous section is the regex `^p`:

```
$ egrep ^p fruits.txt
pear
peach
plum
```

As you can see, this regex fails to match both `apple` and `grape`, since neither starts with a 'p'. The fact that they contain a 'p' elsewhere is irrelevant. Similarly, the regex `e$` only matches `apple`, `orange` and `grape`:

```
$ egrep 'e$' fruits.txt
apple
orange
grape
```

Mind the quotes though! In most shells, the dollar-sign has a special meaning. By putting the regex in single-quotes (not double-quotes or back-quotes), the regex will be protected from the shell, so to speak. It's generally a good idea to single-quote your regexes, so that's what I'll do in the examples from now on.

The Windows shell is an exception, mind you. You'll be better off using double quotes in that case.

Moving on, `^cat$` only matches lines that contain exactly `cat`. You can find empty lines in a similar way with `^$`. If

you're having trouble understanding that last one, just apply the definitions. The regex basically says: "Match a start-of-line, followed by an end-of-line". That's pretty much what an empty line means, right?

Mind you, a regex with only a start-of-line anchor `^` *always* matches, since every line has a start. The same obviously goes for the end-of-line anchor. If you don't believe me, just try it out on the fruit list:

```
$ egrep '^' fruits.txt
apple
orange
pear
peach
grape
banana
blueberry
plum
```

A lot of regex implementations offer the ability to use word anchors. As you saw, a regex like `cat` not only finds the *word* `cat`, but also all those cases where `cat` is "hidden" in other, longer words. In such cases you can use the start-of-word and end-of-word anchors, `\<` and `\>`, respectively. These meta characters don't match *on* characters, but *between* them.

So if you were looking only for occurrences of the word `cat`, you could use the regex `\<cat\>`.

For the next hands-on example you'll need the cats.txt file, which contains several words that contain `cat`. First, try the following:

```
$ egrep '\<cat' cats.txt
cat
cattle
catalog
scrawny cat
```

From this example it becomes clear that start-of-word boundaries not only work between words, but also catch words at the beginning of a line.

These word boundary anchors aren't supported by all regex implementations though. A number of implementations (including *perl*'s) offer is-word-boundary and not-a-word-boundary anchors instead, in which case the regex `\<cat\>` would have to be replaced with `\bcat\b`.

In this context, the term "word" should be taken lightly; every combination of letters, upper and lower case, the underscore (_) and digits counts as a word when dealing with word boundary anchors.

3.2. Character Classes

With the `[…]` construct you can indicate that on a certain position within the pattern one of several characters may appear. Suppose for instance that you're trying to find both `cake` and `coke`. In that case you can use the regex `c[ao]ke`.

Another example, to recognise hexadecimal digits, is `[0123456789abcdefABCDEF]`. This quickly becomes impractical though. Fortunately you can use a hyphen to specify a **range**: `[0-9]`. More than one range in a character class is also allowed: `[0-9a-fA-F]`.

Just make sure you don't write `[A-z]` when you mean `[A-Za-z]`. Though it might look convenient, the first regex also catches the six characters between 'Z' and 'a' (if you're using the ASCII character set, that is).

You can also specify a **negated character class** by placing a caret (^) directly after the opening bracket: `[^…]`. This inverts the sense of the character class: `[^0-9]` matches any character *but* digits.

Fine, but what if you want those brackets, hyphen and caret to appear as characters inside a character class? One way is to *escape* them with a backslash: `[\^\]]`. Another way is to put them in places in the character class where they're not valid. The regex engine will then treat the character as a literal. So, place the dash first or last within the character class, the caret

in any but the first place, and the closing bracket right after the opening bracket: `[]^[-]` is a valid character class containing four characters.

3.3. The Dot

The dot, `.`, can be considered a special case of a character class, in that it matches any character. `th.s` for instance matches both `this` and `thus`, but also `thqs`, `th#s`, etc.

This means that a regex to find an IP address, for example `209.204.146.22`, will *not* work. All three dots need to be escaped: `209\.204\.146\.22` does work. Well, there could be digits preceding or following the IP address. That can be solved by using word boundary anchors: `\<209\.204\.146\.22\>`.

Used inside a character class, the dot loses its meaning though. A character class to search for some punctuation characters, for example, might look like this: `[.,:;]`.

3.4. Quantifiers

Using quantifiers, you can specify how often a character, character class or group may or must be repeated in sequence. The general form is `{min,max}`.

An example is the regex `bo{1,2}t`, which matches both `bot` and `boot`. To match any sequence of three to five vowels, you can use `[aeiou]{3,5}`. Or you can use a quantifier to make something optional: `finds{0,1}` matches `find` and `finds`. This case occurs often enough to justify an abbreviation: the regex `finds?` is effectively identical to the previous.

Important to notice at this point is that a quantifier only applies to the item that precedes it. The question mark in the above regex only applies to `s`, *not* to the entire `finds`.

If you want to match something a certain number of times you can set the mimimum equal to the maximum: `^-{80,80}$` matches lines that consist of exactly eighty dashes. Some regex implementations allow this form to be shortened to `...{num}`. With this, the previous regex can be shortened to `^-{80}$`.

It's also allowed to leave out the upper bound: `a{5,}` will match any row of at least five letters 'a'. The case of "one or more" (eg. `a{1,}` occurs much more frequently, though. That's why this form has an abbreviation, the `+`: `a+` and `a{1,}` are effectively equivalent.

The case "zero or more" also has a short form: `*`. For instance, `e*` will match any number of letters 'e' in sequence, including zero. But be careful: a regex will always match as "early" as possible. So if you expect this regex, when applied to `b**ee**r`, to match the boldfaced text, you're wrong! That's because there's a sequence of e's at the beginning of the text, *before* the `b`. The fact that the sequence is zero characters long makes no difference to the regex. In such a case, `e+` might be more appropriate.

Important to know is that quantifiers are **greedy**. That means, that if you apply the regex `1*` to the text `11111`, it will consume all the 1's. Not until a quantifier's greediness would cause a pattern mis-match will the quantifier release some of the text it consumed.

Take the regex `[0-9]*25` for example, which matches numbers ending in `25`. If you apply it to the text `3425`, the quantifier will at first consume the entire text, because each of the characters can be matched by the character class `[0-9]`. But that prevents `25` from matching, causing the entire regex to fail.

In such cases the quantifier will release one character at a time. First the `5` is released, leaving the quantifier matching only `342`. When it turns out that that still isn't enough, the `2` is released as well, allowing the rest of the regex, `25`, to match.

This means that a regex that contains a lot of quantifiers will have many combinations to try before failing. So if the text it's applied to causes many near-matches, it might all of a sudden take a very long time to process the data.

3.5. Alternation

With the `|` meta character, the *or*, you can merge several regexes into a single regex. With this you supply the regex engine with *alternatives*. `Jack` and `Jill` are two seperate regexes, whereas `Jack|Jill` is one that will match either.

Further back I mentioned the regex `c[ao]ke`. Using alternation you can write it (less efficiently) as `c(a|o)ke`, where the parentheses (which therefor are meta characters too, more on this later) are used to limit the effect of the alternation.

Another, almost classic example is the regex `^(From|Subject|Date): •`, which can filter an email message's headers. In this particular example the parentheses are by no means optional; the regex `^From|Subject|Date: •` matches something else entirely. By pulling it apart you get three seperate regexes `^From`, `Subject` and `Date: •`, which clarifies (I hope) why the regex is wrong (as in, not fit for filtering email headers).

3.6. Grouping

In addition to the function of limiting the effect of alternation, parentheses `(…)` have another function, which is **grouping** for quantifiers. Everything about quantifiers that applies to characters and character classes also applies to groups.

An example is `(hurrah •){2,3}`, which matches `hurrah hurrah` as well as `hurrah hurrah hurrah •`.

A more complex example combines alternation and grouping with a quantifier: `(hurrah •|yahoo •){2,3}`. That gives twelve possible combinations, including for example `hurrah yahoo` and `yahoo hurrah yahoo •`.

3.7. Backreferences

The use of grouping has a very useful side-effect. That's because certain regex implementations "remember" the matched text in a grouping, and make this available both during and after the application of a regex.

Assume you have a piece of text you wish to search for double words, such as `…when • when • …`. Now, you could try to build a seperate regex for every word you can think of, but wouldn't it be convenient if you could say, "find something that matches this pattern, then match it again"?

You can. Provided your regex implementation supports it, parentheses (`(…)`) "remember" what they match. In that case you could search for double words with the regex `([a-zA-Z]+) • \1`. The meta character `\1` is called a **backreference**.

Using this regex also catches cases such as **`when when`**`ever` though, so in this case `([a-zA-Z]+) • \1\>` might be a better regex.

In this example the meta character `\1` refers to the first opening parenthesis. You can of course have several groups in a regex, but the maximum number of backreferences is limited to nine (\1 ... \9) in most regex implementations.

To determine which backreference corresponds to which group, you need to count the number of opening parentheses from the left. In the example above, we only had one group, so that's easy. But the next example is a bit more complicated.

`((the|a) (big( red)?|small( yellow)?) (car|bike))` contains six groups. The example file (contains five lines, four of which can be matched by the regex:

```
$ egrep '((the|a) (big( red)?|small( yellow)?) (car|bike))' car.txt
the big red car
a small bike
the small yellow car
a big red bike
```

To clarify which backreference corresponds to which group, I wrote a small perl-script. This gives the following output:

```
$ perl -n refs.pl car.txt
"the big red car"
\1 => the big red car
\2 => the
\3 => big red
```

```
\4 =>  red
\5 => (null)
\6 => car

"a small bike"
\1 => a small bike
\2 => a
\3 => small
\4 => (null)
\5 => (null)
\6 => bike

"the small yellow car"
\1 => the small yellow car
\2 => the
\3 => small yellow
\4 => (null)
\5 =>  yellow
\6 => car

"a big red bike"
\1 => a big red bike
\2 => a
\3 => big red
\4 =>  red
\5 => (null)
\6 => bike
```

The script applies the regex to every line of the example file, and prints the backreferences if it matches. In the output, `(null)` indicates that the group to which the backreference corresponds is not part of the match.

So, you can use multiple groups in a regex, but the maximum number of backreferences is, in most regex implementations, limited to nine (\1 ... \9).

A slightly larger example is the task of untangling the *query string* in a URL, for example `http://www.foobar.com/search?query=regex&view=detailed`.

Assume we want to extract the name and value of the *query* variable from this URL. This can be done with the regex `\?([a-zA-Z]+)=([^&]+)`. With this regex, we use `\?` to line up the regex with the query part, which starts after the question mark. Then we match the name of a variable using `[a-zA-Z]+`, and surround it with parentheses to save it for later processing, `([a-zA-Z]+)`. This should be followed by an equal sign, so we append `=` to the regex. Finally we need to capture the variable's value. This can be done with `[^&]+`, since the string that makes up the value goes on until the next `&`, which acts as a *name=value* delimiter. This also works if the value is not followed by an ampersand, in which case the variable's value takes up the rest of the URL. The value regex needs to be enclosed in parentheses since we want to save it for later, so we get `([^&]+)`.

Although there are two sets of parentheses in the regex, neither is used in the regex by a backreference. Then how do we get to the data? Well, this strongly depends on the tool in which the regex is used. Following are a few examples.

With *perl*, the content of both backreferences is available after the match in the variables *$1* and *$2*. The following snippet of code shows how this can be used.

```
$url = 'http://www.foobar.com/search?query=regex&view=detailed';
$url =~ /\?([a-zA-Z]+)=([^&]+)/;
print "$1 = $2\n";
```

In PHP you'd have to use the *ereg()* set of functions (see the manual), like this for example:

```
$url = 'http://www.foobar.com/search?query=regex&view=detailed';
ereg('\?([a-zA-Z]+)=([^&]+)', $url, $refs);
echo "$refs[1] = $refs[2]\n";
```

4. Pitfalls

Misconceptions or lack of understanding of quantifiers are the main cause of errors, although even the most hardened regex hackers make these mistakes every once in a while. Take, for example, the text `"Hey you", he said, "did you say something?"`. We'll try to match the first piece of quoted text, including the quotes. So we use the regex `".*"`, because we want to match a double-quote, followed by text, being an arbitrary character (`.`) matched an arbitrary number of times (`*`), followed by another double-quote.

But what we appear to match is not `"Hey you"`, but `"Hey you", he said, "did you say something?"`!

Whoops. Slght mistake. But where? The point is that quantifiers are so greedy, they don't even look at what the rest of the regex might want to match. `.*` devours everything from the first 'H' after the first quote to the end of the line, and is then coerced to release the last character to match the final double-quote in the regex.

Apparently, we need to be more precise about what we mean: we want a double-quote, followed by everything *but* a double-quote, followed by a double-quote. Or rather: `"[^"]*"`.

This regex does a much better job, but you need to realise that escaped quotes will ruin the fun: `"When he yelled, \"Come here!\", I left", she said.`. In this case we appear to match `"When he yelled, \"`. A solution for this problem is less trivial than might appear at first glance, and falls outside the scope of this tutorial.

5. More Information

Most of my knowledge of regular expressions comes from the book *Mastering Regular Expressions*, written by Jeffrey Friedl and published by O'Reilly. For more application-oriented information about regular expressions you could try O'Reilly's books on sed & awk or perl.

If you're a PHP programmer, be sure to read the manual page entries for the ereg (POSIX) and preg (Perl-compatible) regular expression set of functions.

Perl programmers can either check the manual entry for regular expressions at Perldoc.com, or you could try typing `man perlre` at the shell prompt (if you're running a UN*X-like OS, that is).
