

PET: Introduction to Regular Expressions - More Power to Your Elbow

Paul A. Marshall

1 Introduction

The UNIX shell environment (in its various forms) has a reputation for being a powerful tool. This is thanks, in part, to regular expressions (IMHO). Unfortunately, if you are not in the know, they tend to look a bit cryptic. You do not have to be a guru to enjoy this power. A basic understanding of the subject will do wonders for your CLI effectiveness. This tutorial aims to give you more power to your elbow.

1.1 The Reader

To derive the most benefit, you should be a reasonably confident shell user. Pipes and redirections will hold no fear, and you will not mind chucking a few commands into a file and calling it a script.

1.2 What is a Regular Expression?

I would describe a regular expression (AKA regexp) as a sequence of characters intended to match certain patterns of text. This can range from a literal such as “potato” to something more open ended like “any word that begins with a ‘c’ and ends with a ‘p’ or a ‘d’”.

1.3 The Tools of the Trade

We will be using the commands *egrep* and *sed* in this tutorial. However, this is not a tutorial for these commands, they are used to demonstrate regexps.

egrep (Extended General Regular Expression Parser) will be used to find patterns in text files. It will be used in the form:

```
egrep "<regular expression>" <filename>
```

sed (Stream EDitor) will be used to modify matching patterns. It will be used in the form:

```
sed 's/<regular expression>/<replacement text>/g' <filename>
```

These are not the only regexp friendly tools at your disposal. On a Linux system there are plenty of others. For example I am typing this with Vim. What? Vim? Why, of course it can! Perhaps the ultimate tool is the scripting language *perl*. I will leave the learning of *perl* as an exercise for the reader ;-)

2 Elements of Regular Expressions

2.1 The Simple Case - Literals

Let's get the ball rolling with some literal patterns. This will serve as an introduction for those unfamiliar with `grep` or `sed`. Say we have a text file called `mary.poem` whose contents are as follows:

```
Mary had a little lamb
Whose fleece was white as snow
And everywhere that Mary went
The lamb was sure to go
```

If we wanted to view those lines in the file containing the word "lamb", we could use `egrep`:

```
egrep "lamb" mary.poem
```

What if we wanted to try the poem with a different pet for Mary? We could see how this would read by using `sed`:

```
sed 's/lamb/dog/g' mary.poem
```

and that's when I discover why I should leave poetry to the experts ;-)

2.2 Escape Sequences

Using literal values will not work in all cases. This is because some characters have special meaning in regexps, these special characters are called *metacharacters*. For example, what if we wanted to use `sed` to search for the pattern `"/usr/local/src/"`? The `sed` syntax already uses forward slashes so how on earth can it know where our search pattern ends, and the next argument begins? This situation can be resolved by the use of *escape sequences*. In general, we can ensure that a metacharacter will be interpreted as a literal by inserting a backslash in front of it. ie:

```
sed 's/\/usr\/local\/src\/\/usr\/local\/mysrc\/\/g' myfile
```

Naturally, as backslash has a special meaning, it must also be represented by an escape sequence - ie `\"`.

2.3 The Anchors “^” and “\$”

These are very useful operators. “^” is used to match the beginning of a line whereas “\$” is used to match the end of a line. Taking our `mary.poem` example, the following commands will both display the first line only:

```
egrep "^Mary" mary.poem
egrep "lamb$" mary.poem
```

These operators are called *anchors* because they anchor the pattern to a particular part of the line.

2.4 Character Lists “[]”

We can use a pair of square brackets in our regexps to represent a number of possible characters that we would like to match. For example, “[bfs]unny” will match “bunny”, “funny” and “sunny”. Ranges of characters can be expressed with the help of a minus sign. For example, “[a-z]” will match any lower case letter, “[a-zA-Z0-9]” will match any alphanumeric character. A “^” after the opening square bracket reverses the meaning so as it will match every character *except* those in the character list. For example “[^A-Zbtj]oy” will *not* match “boy”, “toy” or “joy”, but will match any other non-upper case character followed by “oy”. Notice how the meaning of “^” changes depending on the context in which it is used.

2.5 “.” A Match for Anything

The “.” will match any character. For example, the regexp “pe.t” matches, among others, “pent”, “pest” and “pelt”.

2.6 Repeat After Me, “*”, “+”, “?” and “{ }”

- * Will match if the previous expression is repeated zero or more times. For example, the regexp “he1*o” will match “heo”, “helo”, “hello”, “hello”, “hellllllo” etc. This is *not* the same behaviour as the wildcard “*” character used in filename globbing. To replicate that behaviour “.*” would be required.
- + Will match if the previous expression is repeated one or more times. For example, the regexp “he1+o” will match “helo”, “helllllo” etc., but *not* “heo”.
- ? Will match if the previous expression is repeated zero times or once. For example, the regexp “hdd?” will match “hd” and “hdd” but nothing else.
- {n} Will match if the previous expression is repeated exactly *n* times. For example, the regexp “. {3}” will match any three character string.
- {n,} Will match if the previous expression is repeated *n* times or more. For example, the regexp “y{3,}abbadabbadoo” will match “yyyabbadabbadoo”, “yyyyabbadabbadoo”, “yyyyyabbadabbadoo” etc.
- {n,m} Will match if the previous expression is repeated at least *n* times but no more than *m* times. For example, the regexp “[0-9]{5,7}” will match any string with between five and seven digits.

2.7 Grouping “()”

Parentheses are used to group regular expressions. For example, the regexp “mua(ha){2,4}” will match “muahaha”, “muahahaha” and “muahahahaha”.

2.8 Alternation “|”

The alternation operator is used to specify alternatives. For example, the regexp “`hello|goodbye`” will match both “hello” and “goodbye”. The alternation applies to the largest possible regexp on either side. For that reason it is often used with parentheses to limit its scope. For example, the regexp “`(bel|yell)ow`” will match “below” and “yellow”.

2.9 Some sed Specific Notes

Some of the operators described here must be escaped if you wish to use them with the sed syntax given. Those effected become “`\?`”, “`\+`”, “`\{...\}`”, “`\|`”, and “`\(...\)`”. If you wish to use any of these symbols as literals, the easiest way is to enclose them in square brackets eg “`[|]`”. These rules also apply if you use ordinary GNU grep in preference to egrep.

2.10 Some More Useful Operators, “`\w`”, “`\W`” “`\b`” and “`\B`”

`\w` Matches any word character. In regexp terms, a word is made up of alphanumeric characters. It is equivalent to “`[A-Za-z0-9]`”.

`\W` Matches any non-word character.

`\b` Matches the beginning or end of a word, ie the word boundary. Why is this useful? Say we wanted to grep a file for all lines containing the word “and”. If we do simply “`egrep "and" myfile`”, it may well display lines containing words such as “band”, “sandy” etc. The way around this is to use “`egrep "\band\b" myfile`”. This is safer than using, say, “`egrep " and " myfile`” as we cannot be sure that the target word will be surrounded by spaces (eg there may be punctuation or a newline).

`\B` This is the opposite of “`\b`”. For example, “`\Band\B`” will match “handle” and “sandy” but *not* “and”, “hand”, or “android”.

3 Beware of Greed

Regexps have a tendency to match as much text as possible. Take the following example of an html line:

```
<IMG SRC="george.jpeg">My giraffe's name is <B>George</B>
```

One might expect the regexp “`<IMG.*>`” to match just the image tag from that line. In fact, it will match the *entire* line. This is nothing to worry about if we are merely doing a grep search. If we are doing a search and replace, however, it's a disaster as we will unintentionally obliterate the rest of the line. A regexp which matches a larger amount of text when a smaller section would have been a valid match is said to be *greedy*. One way around the problem is “`<IMG[>]*>`” ie we are matching as many non “`>`” characters as possible followed by “`>`”.

4 Practical Examples

By now you may be thinking “This is all very interesting, but is this really of any use to me?”. I give a few examples here to help you get the hang of how these various elements can be put together to achieve useful results. At the end of the day, you will need to use a bit of imagination. You need to be able to recognise when regexps could be useful, then you need to figure out how it can be done - and as with all things UNIX, there is always more than one way to do it!

4.1 Stripping “>” From Emails

Is it not a pain, my friends, when someone emails you a great joke and you want to forward it on whilst claiming it as your own (What, doesn't *everyone* do that? Oh well, just me then), but it's been forwarded many times and each line has about a dozen “>” signs in front of it? Fear not, here is one solution

```
sed 's/^>*/' email.original > email.modified
```

4.2 Validating Script Parameters

Say we have a shell script which requires a numerical value as it's first parameter. This could be validated by the script like this:

```
if echo $1 | egrep "[0-9]+$" >/dev/null
then
    echo "Yeah, I like that number, we can do business"
else
    echo "That's a duff number, sorry"
fi
```

This uses the fact that egrep will return a true status (from a shell point of view) if it finds any matches. The script pipes one line containing the parameter into egrep and acts on the result.

What if we decide we also want to allow up to three decimal places? My first attempt at a replacement egrep for this is:

```
egrep "[0-9]+\.[0-9]{0,3}$"
```

I'm happy with this until I realise it will accept whole numbers ending in a decimal point eg "12.". I'm not sure if this is technically correct but I don't like it anyway. I revise it as follows:

```
egrep "[0-9]+(\.[0-9]{1,3})?$"
```

4.3 Swapping “jpeg” and “gif” with “png”

At the time of writing, a company (whose name I can't remember) is claiming to hold a patent over some part of jpeg technology. The gif format is also the subject of a patent (held by Unisys, I believe). You decide enough is enough and you “regimp” all your website graphics to be all png format. Now all you need to do is update your html to reflect this. Here is one method:

```
sed 's/\.\.(jpeg\|jpg\|gif)\b/\.png/g' index.html > new_index.html
```

This should be reasonably foolproof but it is not 100% safe because no steps have been taken to ensure that the text substituted is within a valid image tag (maybe your website is a protest site about the issues involved and is called “I-Want-MY.jpeg”). Such a substitution is possible with sed, but it requires techniques that have been left out of this tutorial.

4.4 Checking Tarballs

How thoroughly do you check tarballs before you unpack them on your system? Do you list the contents and watch the zillions of files scroll past thinking that you will spot anything untoward?

Before unpacking a tarball onto your system, you will probably want to be sure that it will not create any files with absolute paths, or files in the current directory, or files with paths that use dots to place themselves elsewhere in your file tree. This boils down to a few rules; the path must not start with a forward slash, must contain a forward slash in its body somewhere, and must not contain “/./”. For good measure let’s also disallow paths that start with a dot which will help simplify the regexp. Here is one way to search for those “nasties”:

```
tar tfz suspect-package.tar.gz | egrep "^[./]|^[^/]*$|\./\.\\"
```

Notice how there is no need to escape the forward slash or dot inside the character list. Naturally, if you wanted to avoid severe RSI, it could be made into a small script.

5 The End\$

Well there you have it. Go forth and unleash your new powers. Just remember that the different commands that accept regexps have their own little quirks and you will need to check the documentation. There are many gotchas waiting to, um, get you, so always check that your regexps do exactly what you think they do. Don’t come a crying when your seven years in the making potentially best selling novel is ruined by some half baked command sequence (you know, the sequence that ends with “...; cp novel.modified novel.master”). ’Nuff said ;-)