Regular Expressions Revealed

Regular Expressions Revealed

David Totsch
Account Support Engineer
Hewlett-Packard Company
david_totsch@hp.com

Although regular expressions are renowned for their mystery and difficulty to master, they are a very valuable asset to have in your UNIX tool kit. Using regular expressions, anyone can perform powerful data queries, create sophisticated edits and more easily manipulate large amounts of data. Attend this session to allow the power and functionality of regular expressions to be revealed to you.

Regular Expressions

- **often feared by any normal(?) individual**
- **strange/complex as the name sounds**
- **name equivalent to idiomatic expression**
- **expression - not interpreted literally**
- **pattern matching**
- **“regular expression” is a term to describe a result**
- **describes a pattern or sequence of characters**

There is nothing “regular” about regular expressions. Undertaking the task of learning how to use them this is not a slight task; it will take much time and practice. However, you can learn enough about regular expressions in a short period of time to make it worthwhile.

If you read the manual page regexp(5), you may hurt your brain. However, most of us under the section “Pattern Matching” because we use it every day when we list or address files. So, you are starting beyond the beginning. Just keep in mind that regular expressions is a language for specifying what you want and you will do just fine. We are going to embark on a path to help you specify what you want more succinctly.

Why should I care about Regular Expressions?

- **It is a major strength of UNIX**
- **formulate powerful queries**
- **automate sophisticated text edits**
- **a valuable tool to include in your UNIX skill set**

You should care about Regular Expressions because you care about increasing your skill set. You should care about Regular Expressions because you are interested in doing the best job you can in the shortest amount of time. You should care about Regular Expressions because elegant solutions are valued over brute strength algorithms. You should care about Regular Expressions because they are fun and satisfying to create.

No other operating system offers the strength of something similar to Regular Expressions in the environment. Using Regular Expressions you can be very specific about what you are looking for (and avoid a bunch of noise to wade through). Since you can query very specifically, you can also edit what was found in a fine-grained manner. I am sure that you will find even more situations where minutes, even hours, of work can be avoided by using more and more complex Regular Expressions.

Where Would I Use Regular Expressions?

- **UNIX Editors**
  - **ed, ex**
  - **sed**
  - **vi**
- **Paging Commands**
  - **more**
  - **pg**
- **Pattern Matching Commands**
  - **grep (Global Regular Expression Print)**
  - **awk**
  - **expr**
  - **perl**

Above you see a list of commands that understand Regular Expressions. If you can master Regular Expressions just for use in your favorite UNIX editor and grep(1) you can realize benefits. If you already search for strings in the pagination commands, you will find Regular Expressions useful.

I would like to see you go on to using Regular Expressions in more sophisticated commands like awk(1) and perl(1). Awk(1) directives look for Regular Expressions and then take actions based upon what they find. You can say the awk(1) is Regular Expression driven. Perl is the same type of tool; it looks for patterns and takes actions. Both of these tools require a certain dedication to learning Regular Expressions. If you do not have a firm grasp on Regular Expressions, you can improve your chances of success with Perl by using awk(1) first. Perl uses Regular Expressions, but it allows an expanded set of operators that can be very imposing and difficult to master when you have no exposure.

Pattern Matching Notation

AKA File Name Generation (globbing)

? **Match any single character**

[ ] **Bracket notation**

***** **Anything or nothing**

Can be used within case statements!

If you take the time to read regexp(5), you will see it starts out with “Pattern Matching Notation”. You also know this concept as File Name Generation. If you have been around UNIX long enough (from the very beginning), you may also know the concept as “globbing”. In any event, this is how we specify files for commands to work with. Be very careful. This syntax is similar, yet very different, from the remainder of this discussion.

Basic Regular Expressions

[ ]	**collating sequence**
[^]	**non-collating sequence**
[a-z]	**collating sequence specified as range**
[: :]	**character class (eg. [:upper:])**
*****	**zero or more**
.	**any single character**
( )	**grouping (subexpression)**
^ $	**anchoring (^ = beginning of line, $ = end of line)**
\n	**n'th subexpression n = 0-9**
\{m,n\}	**repetition m = minimum, n=max m/n 0-255**

We are familiar with collating sequences -- that is, a list of characters that can be matched by a single character. You can think this syntax as "a character in this position can be any one of these". The non-collating sequence says "a character in this position can be any character except one of these".

When you use a character class, you may need to use double-square brackets. We will see some examples.

The repetition character "*" is slightly different than the meaning it has for file name generation. Now, it modifies the preceding pattern.

Pay attention. We will see some very interesting uses for grouping patterns.

One of the most handy features of Regular Expressions is that of anchoring. We can specify that what we are looking for is at the beginning of the line, end of the line, or is the entire line.

Still paying attention? We can ask for a specific sub-expression to be recalled (used again) without typing it all out again -- very handy when looking for something that should appear more than once on a line.

Besides asking for a patter to exist multiple times or not, we can ask for minimum, maximum and specific numbers of repetitions. Sounds sort of like recalling sub-expressions, but this syntax specifies the number of times we expect the preceding pattern to appear.

Extended Regular Expressions

+	one or more occurrences of preceding RE
?	zero or one occurrences of preceding RE
|	alternation

When you read regexp(5) you will see a section on Extended Regular Expressions. Not all commands that understand Regular Expressions understand Extended Regular Expressions. For instance, you have to call grep(1) with a specific option or use egrep(1) to get the extended capabilities.

The first two characters modify the preceding pattern. A plus sign says that a least one instance of the pattern must exist, but more are allowed. A question mark says that the preceding pattern does not have to exist, but, if it does, only one instance is allowed. Later, we will see that both of these can be emulated with Basic Regular Expressions.

Alternation is the additional capability gained with Extended Regular Expressions. What alternation allows us to do is specify more than one pattern to choose from in the given position. Better understand will come with examples.

GREP BRE Examples

$ grep [123] data
234
825.
Man-Year: 730 people working feverishly until noon.
157.26
+50.8992
The above line consists of two (2) spaces
.7257
187,472.66

Your handout titled “Regular Expressions Revealed: Data” is the contents of the data file we will be working on.

Lets start near the beginning and work with some Basic Regular Expression examples. Here we see a grep(1) with a simple bracket notation. Every line matched should have a “1”, “2” or “3” on it. I would like to point out what the grep(1) statement missed: lines 16 and 17. Neither of those two lines contain any of the number characters in the list.

grep [0-9] data

234
825.
Man-Year: 730 people working feverishly until noon.
157.26
-88.5
-66
+50
+50.8992
The above line consists of two (2) spaces
.7257
187,472.66

Here we wanted to match any line with a single number character on it.

grep th[aeiouAEIOU] data

Words are the voice of the heart
Every Program is part of some other program, and rarely fits.
Writing free verse is like playing tennis with the net down.
She sells C-Shells by the C-Shore.
Love the Sea? I dote upon it - from the beach.
Line printer paper is strongest at the perforations.
What was that?
Helping Others ==> we all benefit
She sells sea shells by the sea shore.
No one can feel as helpless as the owner of a sick goldfish.

Here we have a BRE that is looking for “th” followed by any of the vowels. Most of the lines have “the” in them somewhere. The second line of output is matching “the” within the word “other”. Beware of situations like that. The grep(1) is scanning for a character sequence. Unless you tell it to, it will not be looking for words.

grep th[^eiouEIOU] data

An Elephant is a mouse with an operating system.
Writing free verse is like playing tennis with the net down.
What was that?

Do you see why this BRE yields this output? That first two lines are very puzzling. OK, I have tricked you a bit. I left “a” out of the bracket notation, so the “tha” in “that” on the last line of output is matched. But, why do the first two lines match? Did it match “with” on both lines? Close, but not quite. It was the “th” in “with” that was matched. Since a space is not in the bracket notation, is is allowed as a match. Therefore, we matched “th” followed by a space. Is that what you expected? I will remind you that this is what we asked grep(1) to look for.

grep L.ve data

Love the Sea? I dote upon it - from the beach.
I Live! I Live!
The password is: L@vek[;'../bck

Lets take a look a saying “match any single character in this space”. Nope it is not a question mark like file name generation -- it is a period. In the above example, we have asked for “L”, any single character, then “ve”. Notice that the dot notation matched the at-sign in the last line of output.

grep ^The data

The future isn't what it used to be!
The above line is a single CR
The above line consists of two (2) spaces
The password is: L@vek[;'../bck
The End

Lets test out anchoring. Here we are looking for “The” at the beginning of the line. Why didn’t our BRE match line 15? It has “The” on it. Well, we asked for “The” to be the very first thing on the line by starting our BRE with a carat.

grep !$ data

Disk Crisis, please clean up!
The future isn't what it used to be!
I Live! I Live!

OK, another anchoring example. This time, we are looking for lines that end with an exclamation mark. Be very careful with syntax like this -- what if the line ends with an exclamation mark, then a space and finally an End-of-Line?

grep "[0-9][0-9]*\." data

825.
157.26
-88.5
+50.8992
187,472.66

Now it is time to get a bit more complicated. This time, lets look for lines that have numbers and a decimal point.

Again we will start with what was not matched: why didn't we get line 17 or 33 in the output? Line 17 is a miss because it lacks the required decimal point. Hey! Wait a minute! Doesn't a dot mean to match any character in this position? Well, we escaped the special meaning of the dot by preceding it with a back-slash (a universal UNIX convention). So, we are looking for a literal dot. Now, why doesn't line 33 match? It is number characters we are looking for are before the the dot, not after -- line 33 does not have any number characters before the dot.

grep "\(the\).*\1" data

Words are the voice of the heart

Love the Sea? I dote upon it - from the beach.

Getting a bit more complicated with BREs, lets explore sub-expressions. Notice that we escaped the parenthesis -- you don't have to do that for all commands. And, didn't I just get through saying that back-slashes escape the special meaning of the next character? Don't you just love UNIX consistency? Anyhow, what we are looking for are lines that have the sequence "the" on them twice. This BRE says "look for the string 'the' (and remember it as a sub-expression) followed by any number of various characters (not just the same character repeated) and then the first sub-expression again". This can be very handy when needing to match complicated expressions more than once.

grep "[0-9]\{3,4\}" data

```
234
825.
Man-Year: 730 people working feverishly until noon.
157.26
+50.8992
.7257
187,472.66
```


In contrast, we have syntax that is looking for multiple occurrences of characters that are number characters. In this case, at least three number characters, but no more than four. Just why was that last line of output matched? The comma and period interrupt the sequence of numbers.

egrep ERE Examples

```
$ egrep 8+ data
825.
-88.5
+50.8992
187,472.66
```


Now that we have explored BREs, lets take a look at some Extended Regular Expressions.

Here we see an ERE that is asking for at least one and possibly more eights in sequence. That was easy and fairly harmless.

egrep "(C-)?[Ss]hells" data

She sells C-Shells by the C-Shore.
She sells sea shells by the sea shore.

Lets take a leap and decompose a fairly complicated ERE. The sub-expression “C-” is modified by a question mark, yielding that “C-” appears only once or not at all. The bracket notation “[Ss]” is a simple way of requesting either the upper- or lower-case version of a single letter.

Is this syntax any different that just asking for “[Ss]hells”. No, it is not. However, consider the situation where we were looking for strings to perform edits upon. Then, pulling the “C-” into the ERE will be quite different.

egrep "[hH]elp(less|ing)" data

Helping Others ==> we all benefit

No one can feel as helpless as the owner of a sick goldfish.

Here is another complicated ERE. This time, alternation has been thrown into the mix. This matches “help” or “Help” following by either “less” or “ing”. We could have also used “[hH]elp(less|ing|full)”. This would have added “full” as an option to the alternation illustrating that alternation is not restricted to a two-way selection.

Spanning

\{M\}	**Exactly M occurrences**
\{M,\}	**Exactly M or more occurrences**
\{M,N\}	**At least M occurrences but no more than N**

*****	**Abbreviation for \{0,\}**
+	**Abbreviation for \{1,\}**
?	**Abbreviation for \{0,1\}**

It is time to make good on my promise to break down EREs into BREs. When using spanning, you can omit the maximum to mean unlimited. Of course, you can always ask for a minimum or maximum of zero. What I want to get at is that “{0,}” says zero or more (longhand for “*”). Similarly, “{1,}” says at least one but possibly more (longhand for “+”). Finally, we see that “{0,1}” says none or just one (longhand for “?”). One item to fully understand from this information is that BREs can do everything EREs can do (except alternation), you just have to do much more typing.

^(\+|-)?[0-9]+\.?[0-9]*$

Yikes! Is this really a Regular Expression, or is it supposed to mean that a cartoon character is cursing? Well, it is an ERE. Look at it for a moment and then we will break it down.

It starts and ends by anchoring to the front and end of the line. Whatever it is, this comprises the entire line.

After anchoring to the beginning of a line there is a sub-expression. This sub-expression is an alternation between either a literal plus-sign or a dash. This sub-expression is modified by a question mark -- either a plus sign, a minus sign, or nothing. This is followed by a bracket notation on the number characters. This bracket notation is modified by a plus sign, indicating at least one, but possibly more. After this, we have an escaped dot, so it requires a literal dot. Well, requires is too strong because our escaped dot is followed by a question mark (which means the dot is optional and only one may appear). This is followed by another bracket notation on number characters, but this time it is modified by an asterisk.

Care to guess what this ERE matches? How about signed and unsigned floating point numbers on a line by themselves. Well, almost. This ERE requires a number before the decimal point, but not after. For example, this ERE matches “25.” but not “.25”. Better syntax would be to switch the bracket notation modifications to require a number after the decimal point.

Removing Blank Lines Using Grep(1)

First Attempt

grep -v ^$

What does it mean to be a “blank line”?

Removing blank lines is a fairly common task. The grep(1) statement shown will remove all lines that consist of nothing but a end-of-line. Nifty. But, what about lines that might have a single space (or maybe several).

Removing Blank Lines (Continued)

grep -v "^[ ]*$"

There is a space inside the bracket notation.

OK, this grep(1) takes care of blank lines that might contain one or more spaces. Is that all? How about tabs?

Removing Blank Lines (Continued)

grep -v "^[ ]*$"

Uh, just what is different this time? There is a space and a tab inside the bracket notation.

Now we have a bit of improvement. The line can contain spaces and tabs. However, it is not very self-evident that there is a space and a tab when you look at the code.

Removing Blank Lines (Continued)

grep -v "^[[:space:]]*$"

Much better documentation!

Here are the other character classes you may find useful:

[:alpha:]	letters
[:upper:]	upper-case letters
[:lower:]	lower-case letters
[:digit:]	decimal digits
[:xdigit:]	hexadecimal digits
[:alnum:]	letters or decimal digits
[:space:]	characters producing white-space in displayed text
[:print:]	printing characters
[:punct:]	punctuation characters
[:graph:]	characters with a visible representation
[:cntrl:]	control characters

Lets Make A Simple Substitution

```
===
===
EQUAL=SIGN
===
===
```

sed s/=*/test/g

```
test
test
testEtestQtestUtestAtestLtesttestStestItestGtestNtest
test
test
```


Yikes! What went wrong here? One would have expected the lines of equal signs to be replaced, but our substitution was made between each character that was not an equal sign! Remember how the asterisk modifes -- is says several or none, which matches the gaps between characters. Lets try this again.

Second Attempt At A Simple Substitution

```
===
===
EQUAL=SIGN
===
===
```

sed "s/==*/test/g" equals

```
test
test
EQUALtestSIGN
test
test
```


Ahh, this is more like what we expected. The key was to specify at least one equal sign followed by any number of others.

Technique

- **DEVELOP AN ALGORITHM: Know what you want to match and how it might appear in the text.**
- **CODE: Write the pattern to describe what you want to match.**
- **TEST: Try the pattern out.**

Borrowing from “UNIX Power Tools”, you develop Regular Expressions from both ends. You have to stop and carefully consider what it is that you want to match and how it might appear in the input. Then, you create your Regular Expression. This Regular Expression is then tested against example data and fine tuned.

TESTING (HIT OR MISS)

- **HITS that should be MISSES**
- **HITS**
- **MISSES**
- **MISSES that should be HITS**

Work both ends to the middle!

Here we see more about what I meant by developing Regular Expressions from “both ends”. When you test, you will receive matches that should not have been matched. You will get some of what you want, too. You will miss some of the items you intended to match. Concentrate on the matches that should not have been and the data you did not match that you should have, thereby working from both ends.

Why should I care about REs?

```
$ cat sample
printf("ENTER YOUR NAME: ");
printf ( "OPENING FILE: %s\n", file );
printf ( "CAN'T OPEN FILE: %s\n", "/usr/data" );
fprintf(stderr,"CAN'T OPEN %s\n",errfile);
/* comment */ fprintf(stderr,"CAN'T OPEN %s\n",errfile);
value=2;
$ sed '1,$ s/\(.*[^f]\)printf.*"\(.*\)",\(.*\)/\1fprintf(stderr,"\2",\3/p' sample
printf("ENTER YOUR NAME: ");
printf ( "OPENING FILE: %s\n", file );
printf ( "CAN'T OPEN FILE: %s\n", "/usr/data" );
fprintf(stderr,"CAN'T OPEN %s\n",errfile);
/* comment */ fprintf(stderr,"CAN'T OPEN %s\n",errfile);
value=2;
```


Here is a quick example of why you might want to become familiar with Regular Expressions. Pretend with me for a minute that you have recently become the owner of several thousand (maybe hundreds of thousands) of lines of “c” code. The new boss, that knows nothing about “c” code (of course) has discovered that this code uses both “printf” and “fprintf”. This boss has decided to standardize on only “printf”. Pointy-Hair asks you to single-handedly change every line of source code, converting “printf” to “fprintf”.

After you panic (imagine days upon days with vi(1) and cc(1) -- gasp!), you realize that you could use a simple edit to convert all instances of “printf” to “fprintf”. Then, depression sets in because you realize that “fprintf” requires an additional argument in the list (the file to print to: e.g. stdout or stderr). After a few experiments, you also notice that your simple edit yields “ffprintf” when the original string is “fprintf” (you need to leave these alone).

After this short-lived frustration, you ask someone who knows Regular Expressions better than you for help. Together, you compose the sed(1) shown. This sed(1) leverages EREs to carefully edit all of the files of source code. After some time with cc(1) to make sure everything is OK, you realize that you accomplished what could have been months of droll, demeaning work, in just a few hours…then you faint. When you wake up, you inform the boss that the effort will take several weeks of hard effort (and spend those weeks on the golf course).

Resources

Aho, Alfred V., et al:	***The AWK Programming Language*** **Addison-Wesley Publishing Company**
Dougherty, Dale:	***sed & awk*** **O'Reilley & Associates, Inc.**
Parrett, William A.:	***UNIX For Application Developers*** **McGraw-Hill, Inc**
Peek, Jerry et al:	***UNIX Power Tools*** **O'Reilley & Associates, Inc.**
Jeffrey E.F. Friedl	**Mastering Regular Expressions** **O'Reilly & Associates, Inc.**

Make sure you get the second edition of "sed & awk" -- otherwise, your understanding of all the examples will be predicated on your understanding of nroff(1) and troff(1).

"Mastering Regular Expressions" is a very in-depth discussion that covers multiple versions of Regular Expressions. You may find yourself wading through some esoteric discussions to get at what you really need. However, the book is worth your money if you are serious about leveraging Regular Expressions.